MIT 6.851
Advanced Data Structures
Prof. Erik Demaine

Spring '12 Scribe Notes Collection

TA: Tom Morgan, Justin Zhang

Editing: Justin Zhang

Contents

List of Figures

Lecture 15

Lecture 16

Lecture 17

Lecture 18

Lecture 19

Lecture 20

Lecture 21

Lecture 1

Temporal data structure 1

Scribes: Oscar Moll (2012), Aston Motes (2007), Kevin Wang (2007)

1.1 Overview

In this first lecture we cover results on persistent data structures, which are data structures where we keep all information about past states. Persistent data structures are part of the larger class of temporal data structures. The other kind of temporal data structures, retroactive data structures, are the topic of lecture 2.

Usually we deal with data structure updates by mutating something in the existing data structure: either its data or the pointers that organize it. In the process we lose information previous data structures states. Persistent data structures do not lose any information.

For several cases of data structures and definitions of persistence it is possible to transform a plain data structure into a persistent one with asymptotically minimal extra work or space overhead.

A recurring theme in this area is that the model is crucial to the results.

Partial and full persistence correspond to time travel with a branching universe model such as the one in Terminator, and Deja Vu parts 1 and 2

1.2 Model and definitions

1.2.1 The pointer machine model of data structures

In this model we think of data structures as collections of nodes of a bounded size with entries for data. Each piece of data in the node can be either actual data, or a pointer to a node.

The primitive operations allowed in this model are:

1. `x = new Node()`

2. `x = y.field`
3. `x.field = y`
4. `x = y + z`, etc (i.e. data operations)
5. `destroy(x)` (if no other pointers to `x`)

Where `x`, `y`, `z` are names of nodes or fields in them.

Data structures implementable with these shape constraints and these operations includes linked lists and binary search trees, and in general corresponds to `struct`'s in C or objects in Java. An example of a data structure not in this group would be a structure of variable size such as an array.

1.2.2 Definitions of persistence

We have vaguely referred to persistence as the ability to answer queries about the past states of the structure. Here we give several definitions of what we might mean by persistence.

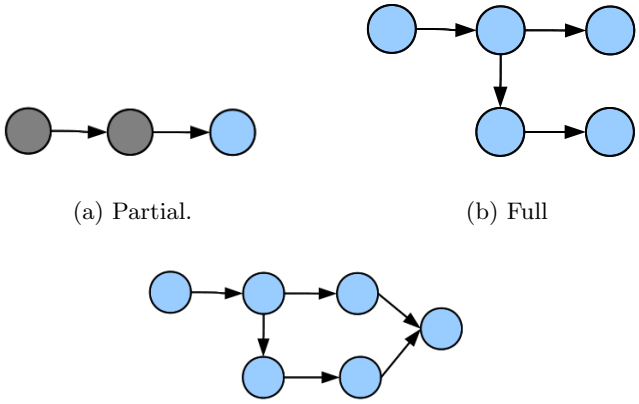
1. *Partial Persistence* – In this persistence model we may query any previous version of the data structure, but we may only update the latest version. We have operations `read(var, version)` and `newversion = write(var, val)`. This definition implies a linear ordering on the versions like in 1.1a.
2. *Full Persistence* – In this model, both updates and queries are allowed on any version of the data structure. We have operations `read(var, version)` and `newversion = write(var, version, val)`. The versions form a branching tree as in 1.1b.
3. *Confluent Persistence* – In this model, in addition to the previous operation, we allow combination operations to combine input of more than one previous versions to output a new single version. We have operations `read(var, version)`, `newversion = write(var, version, val)` and `newversion = combine(var, val, version1, version2)`. Rather than a branching tree, combinations of versions induce a DAG (direct acyclic graph) structure on the version graph, shown in 1.1c
4. *Functional Persistence* – This model takes its name from functional programming where objects are immutable. The nodes in this model are likewise immutable: revisions do not alter the existing nodes in the data structure but create new ones instead. Okasaki discusses these as well as other functional data structures in his book [10].

The difference between functional persistence and the rest is we have to keep all the structures related to previous versions intact: the only allowed internal operation is to add new nodes. In the previous three cases we were allowed anything as long as we were able to implement the interface.

Each of the succeeding levels of persistence is stronger than the preceding ones. Functional implies confluent, confluent implies full, and full implies partial.

Functional implies confluent because we are simply restricting ways on how we implement persistence. Confluent persistence becomes full persistence if we restrict ourselves to not use combinators. And full persistence becomes partial when we restrict ourselves to only write to the latest version.

The diagrams in 1.1 show what the version 'genealogies' can look like for each definition.



(a) Partial.

(b) Full

(c) Confluent/ Functional

Figure 1.1: Version diagrams. Gray means version is read only and blue means version is read-write

1.3 Partial persistence

Question: Is it possible to implement partial persistence efficiently?

Answer: Yes, assuming the pointer machine memory model and the restricting *in-degrees* of data nodes to be O(1). This result is due to Driscoll, Sarnak, Sleator, and Tarjan [6].

Proof idea: We will expand our data nodes to keep a modifications a.k.a. mods 'log'. When we have modified a node enough, we create a new node to take all further updates until it also fills.

For every node in our old data structure, the new data structure will have a collection of nodes: one current with the latest versions and potentially many old ones used only for reading the old versions. Every time we 'archive' a node we will also update all (versioned) pointers to to instead refer to the latest node.

1.3.1 Proof:

We extend our data nodes to contain the following information:

1. a read only area for data and pointers (corresponding to those in the original structure)
2. (new) a writeable area for back pointers. Node x has one backpointer to a node y if y has a pointer to x. This area has limited size, since we know ahead of time there are at most p pointers to our node.

3. (new) a writable modifications ('mods') area for entries of the form `(field, version, value)`. The size of this area also needs to be fixed, and it also has important consequences for write performance.

For an illustration of the construction check figures 1.2a and 1.2b

We implement read and write operations as follows:

1. `read(var, v)` search the mod log for the largest version w such that $w \leq v$. What if the value is in an 'old' node? then we would have gotten to it via an old version of a pointer (c.f. Figure 1.2c and Figure 1.3).
2. `write(var, val)`

   if n is not full, simply add to mod log. if n has no space for more mod logs,

   - $n' =$ new Node()
   - copy *latest* version of each field (data and forward pointers) to the static field section.
   - also copy back pointers to n'
   - for every node x such that n points to x, redirect its back pointers to n' (using our pointers to get to them) (at most d of them).
   - for every node x such that x points to n, call `write(x.p, n')` recursively (at most p recursive calls).

For some data structures such as lists or trees we often know what p is ahead of time, so we can implement the algorithm for these specific structures like in figure 1.3.

1.3.2 Analysis:

- Space:

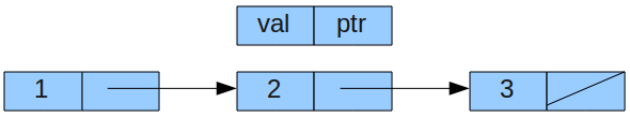
  If we choose the mod log to be bounded at size $2p$ then node has size $d + p + 2p$ which is also $O(1)$ because we assumed there were only $p \leq O(1)$ pointers into any node. The reasons for choosing such a mod log size are clarified in the following cost analysis.

- Time: A read is cheap, it requires constant time to check through a single node's mod log and pick the required version. A write is also cheap if we have space in the mod log. If not, a write can be expensive. Let $\text{cost}(n)$ denote the cost of writing to node n. In the worst case the algorithm makes recursive calls so we get:

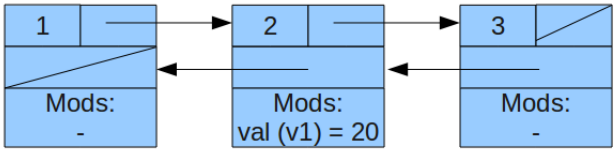
$$\text{cost}(n) = c + \sum_{x \to n} (\text{cost}(x))$$

  Where c represents the $O(1)$ cost of determining latest versions to copy into the new node, copying backpointers, etc. $x \to n$ stands for x points to n.

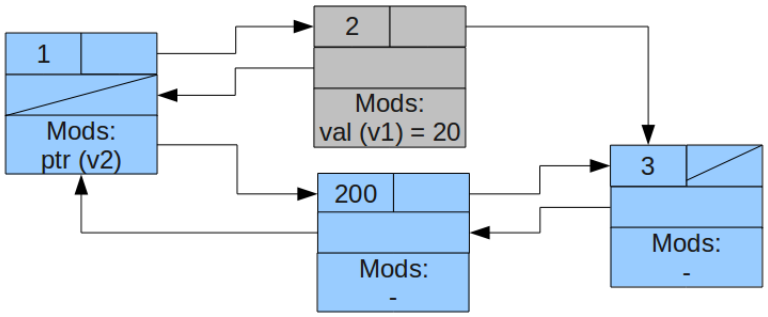
  Clearly the cost can be large because we could split many data nodes in the recursive steps. However, we know when a node becomes full then next operation will likely find a more empty



(a) Ephemeral linked structure. It has one data field and one pointer field



(b) The structure in 1.2a partially persistent. Showing one update: write(root.ptr.val = 20)



(c) The structure in 1.2b after a second update: `write(root.ptr.val = 200)`. Gray indicates the data node is read only

Figure 1.2: Constructing a partially persistent structure from an ephemeral one

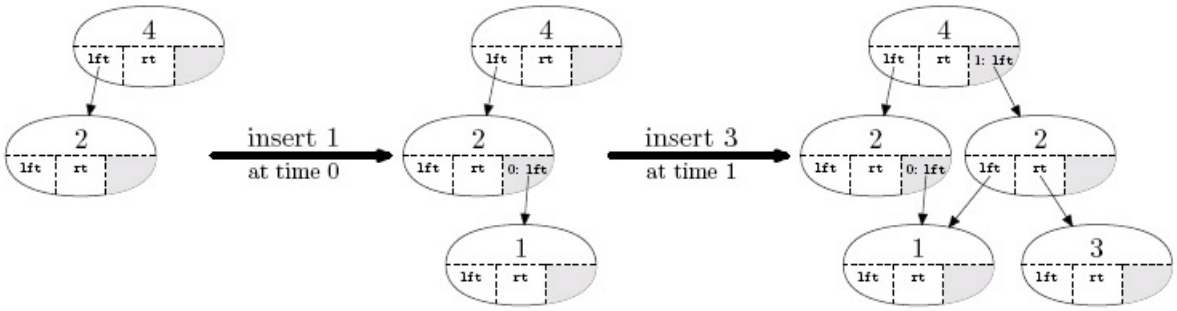


Figure 1.3: Persistence of a binary tree. We need a modification box the size of the in-degree of each data structure node (just one for trees).

mod log. For that reason amortized analysis more appropriate for this operation than worst case analysis.

Recall the potential method technique explained in [68]: if we know a potential function ϕ, then amortized_cost$(n) =$ cost$(n) + \Delta\phi$.

Consider the following potential function:

$$\phi = c * \#\text{ mod log entries in } \textit{current} \text{ data nodes}$$

Since the node was full and now it is empty, the change in potential associated with our new node is $-2cp$. So now we can write a recursive expression for our amortized cost:

$$\text{amortized\_cost}(n) \leq c + c - 2cp + p * \text{amortized\_cost}(x)$$

For some worst case node x. The second c covers the case where we find space in our mod log, and simply add an entry to it thus increasing potential by c.

By unfolding the recursion once we can see at each unfolding the $-2cp$ cancels out the extra cost from the recursion leaving only the $2c$ cost. Hence cost is $O(1)$ amortized. The recursion process is guaranteed to finish despite potential cycles in the graph, because splits decrease ϕ and ϕ is non-negative.

Further study by Brodal [1] has shown actual cost to also be $O(1)$ in the worst case.

1.4 Full persistence

The construction for partial persistence can be expanded to implement full persistence. This result is also due to [6]. We again assume a pointer machine with $p < O(1)$ incoming pointers per node.

We need to take care of a two new problems:

1) Now we need to represent versions in a way that lets us efficiently check which precedes which. Versions form a tree, but traversing it is $O(\#$ of versions).

2) Now we have to support writes to any version. This affects how we handle writes: we no longer have the concept of 'current' vs. 'old' nodes: every node needs to support writes.

1.4.1 Version representation

The version representation problem is solved by keeping a tree representing the version structure, as well as an efficient representation of this tree in a linearized way.

The linearized representation of the tree in the figure 1.4 is $ba, bb, bc, ec, eb, bd, ed, ea$. You can read ba as 'begin node a' and ea as 'end node a'. This representation losslessly encodes the tree, and we can directly answer queries about the tree using that encoded representation. For example we know c is nested within b, since $bb < bc$ and $ec < eb$.

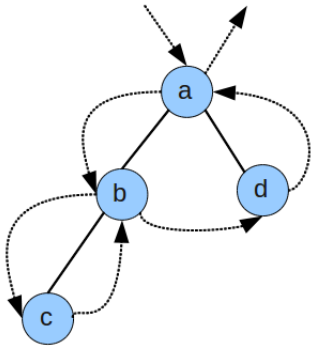


Figure 1.4: in-order tree traversal

This linearized representation can be implemented using an 'order maintenance' data structure. For now, it suffices to say an order maintenance data structure supports the following two operations, both in $O(1)$ time.

- insert an item before or after a specified element.
- check if item s precedes item t.

For example, a linked list supports insertions in $O(1)$, but tests for precedence take $O(n)$. Similarly, a balanced BST supports both operations but in $O(\log n)$ time. Deitz and Sleator show an $O(1)$ implementation for both operations in [4], which will be covered in lecture 8.

To implement version tree queries such as 'is version v an ancestor of version w' we can use two comparison queries $bv < bw$ and $ew < ev$ in $O(1)$. To implement updates like 'add version v as a child of version w' we can insert the two elements bv and ev after bw and before ew respectively, also in $O(1)$.

1.4.2 Construction and algorithm:

The nodes in our data structure will keep the same kinds of additional data per node as they did in the partially persistent case.

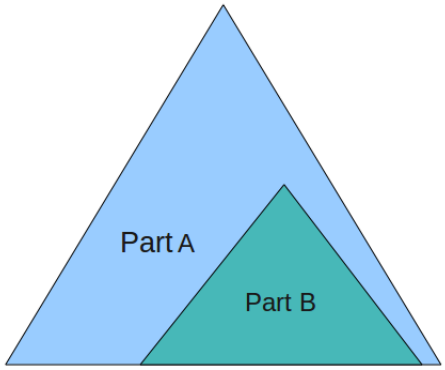


Figure 1.5: Splitting a tree-shaped version genealogy into two subtrees

For each node we store d data entries and p back pointers, but now allow up to $2(d + p + 1)$ modifications. The amount of data d is also a bound on *out-pointers* per node. Additionally we now also version back-pointers.

1. `read(n.field, version)`: By using the order-maintenance data structure we can pick the most recent ancestor of `version` from among the entries in the mod log and return that value.
2. `write(n.field, value, version)`: If there is space in node, just add mod entry. else:
   - m = new Node(). Split the contents of node n's mod log into *two* parts following the diagram figure 1.5. Partitioning into subtrees rather than arbitrarily is required.

     Now node m has some of the mods of the internal tree in Figure 1.5, and node n retains the 'older' half of the updates.
   - from the 'old' mod entries in node n, compute the latest values of each field and write them into the data and back pointer section of node m.
   - recursively update all (up to) $d + p + (d + p + 1)$ forward and backward pointers of neighbors.
   - insert new version to our version tree representation.

1.4.3 Analysis:

- space – 1 if we do not split or $d + p + 2(d + p + 1) = 3d + 3p + 2$ when we split a node, both $O(1)$
- time – `read(var, version)` is implemented in a similar way to the partial case. We use our auxiliary version tree data structure to find the largest ancestor of `version` from among a list of $O(1)$ elements in $O(1)$.

  Like with the partial case, writes are cheap when a node has space in its mods log and more expensive when nodes are full.

Consider $\phi = -c(\#\text{ empty slots})$, then when we split $\Delta\phi = -2c(d+p+1)$ and when we do not $\Delta\phi = c$. Hence, amortized_cost$(n) \leq c+c-2c(d+p+1)+(d+p+(d+p+1))*$amortized_cost(x)) for the worst possible choice of x from the neighbors. When we unfold the recursion once, we find the constants cancel out: $c - 2c(d+p+1) + (2p+2p+1)c = 0$.

OPEN: De-amortization of full persistence.
OPEN: Is there a matching lower bound for both full and partial persistence?

1.5 Confluent Persistence

Confluent persistence presents new challenges. Firstly, we again need to find a new representation of versions. Our tree traversal technique is does not extend to DAGs. Also, it is possible to have 2^u paths in version history after after u confluent updates. For example by concatenating a string with itself repeatedly we get a version diagram like that in figure 1.6.

Deques (double ended queues allowing stack and queue operations) with concatenation can be done in constant time per operation (Kaplan, Okasaki, and Tarjan [8]). Like with a string, we can create implicitly exponential deques in polynomial time by recursively concatenating a deque with itself.

The general transformation due to Fiat and Kaplan [9] is as follows:

- $e(v) = 1 + \log(\#\text{ of paths from root to } v)$. This measure is called the 'effective depth' of the version DAG: if we unravel the tree via a DFS (by making a copy of each path as if they didn't overlap) and rebalanced that tree this is the best we could hope to achieve.
- $d(v) =$ depth of node v in version DAG
- overhead: $\log(\#\text{ of updates}) + \max_v(e(v))$

This results reflects poor performance when $e(v) = 2^u$ where u is the number of updates. This is still exponentially better than the complete copy.

A lower bound also by Fiat and Kaplan is $\Omega(e(v))$ for update if queries are free. Their construction makes $e(v)$ queries per update.

OPEN: $O(1)$ or even $O(\log(n))$ space overhead per operation.

Collette, Iacono and Langerman consider the special case of 'disjoint operations': confluent operations performed only between versions with no shared data nodes. From there they show $O(\log(n))$ overhead is possible for that case.

If we only allow disjoint operations, then each data node's version history is a tree. When evaluating `read(node, field, version)` there are tree cases: when node modified at `version`, we simply read the new version. Where node along path between modifications, we first need to find the last modification. This problem can be solved with use of 'link-cut trees' (see lecture 19). Finally, when `version` is below a leaf the problem is more complicated. The proof makes use of techniques such as fractional cascading which will be covered in lecture 3. The full construction is explained in [11].

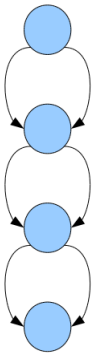


Figure 1.6: An example of $e(v)$ being linear on the number of updates.

1.6 Functional persistence

Functional persistence and data structures are explored in [10]. Simple examples of existing techniques include the following.

- *Functional balanced BSTs* – to persist BST's functionally, the main idea (a.k.a. 'Path copying') is to duplicate the modified node and propagate pointer changes by duplicating all ancestors. If there are no parent pointers, work top down. This technique has an overhead of $O(\log(n))$ per operation, assuming the tree is balanced. Demaine, Langerman, Price show this for link-cut trees as well [3].
- *Deques* – (double ended queues allowing stack and queue operations) with concatenation can be done in constant time per operation (Kaplan, Okasaki, and Tarjan [8]). Furthermore, Brodal, Makris and Tsichlas show in [14] it can be done with concatenation in constant time and update and search in $O(\log(n))$
- *Tries* – with local navigation and subtree copy and delete. Demaine, Langerman, Price show how to persist this structure optimally in [3].

Pippenger shows at most $\log()$ cost separation of the functional version from the regular data structure in [13].

OPEN: (for both functional and confluent) bigger separation? more general structure transformations?

OPEN: Lists with split and concatenate? General pointer machine?

OPEN: Array with cut and paste? Special DAGs?

Lecture 2

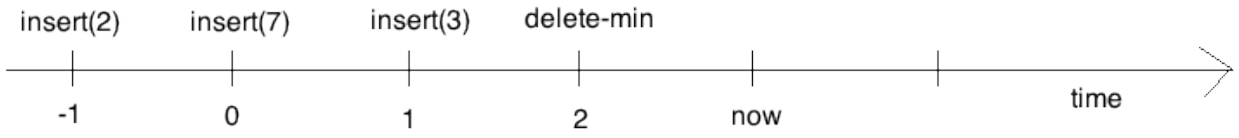
Temporal data structure 2

Scribes: Erek Speed (2012), Victor Jakubiuk (2012) Aston Motes (2007), Kevin Wang (2007)

2.1 Overview

The main idea of persistent data structures is that when a change is made in the past, an entirely new universe is obtained. A more science-fiction approach to time travel is that you can make a change in the past and see its results not only in the current state of the data structure, but also all the changes in between the past and now.

We maintain one timeline of updates and queries for a persistent data structure:



Usually, operations are appended at the end of the time line (present time). With retroactive data structures we can do that in the past too.

2.2 Retroactivity

The following operations are supported by retroactive DS:

- *Insert(t, update)* - inserts operation "update" at time t
- *Delete(t)* - deletes the operation at time t
- *Query(t, query)* - queries the DS with a "query" at time t

Uppercase Insert indicates an operation on retroactive DS, lowercase update is the operation on the actual DS.

You can think of time t as integers, but a better approach is to use an order-maintanance DS to avoid using non-integers (in case you want to insert an operation between times t and $t+1$), as mentioned in the first lecture.

There are three types of retroactivity:

- *Partial* - Query always done at $t = \infty$ (now)
- *Full* - Query at any time t (possibly in the past)
- *Nonoblivious* - Insert, Delete, Query at any time t, also if an operation modifies DS, we must say which future queries are changed.

2.2.1 Easy case with commutativity and inversions

Assume the following hold:

- *Commutative updates:* $x.y = y.x$ (x followed by y is the same as y followed by x); that is the updates can be reordered $\Rightarrow$ Insert(t, op) = Insert(now, op).
- *Invertible updates:* There exists an operation x^{-1}, such that $x.x^{-1} = \emptyset \Rightarrow$ Delete(t, op) = Insert(now, op^{-1})

Partial retroactivity

These two assumptions allow us to solve some retroactive problems easily, such as:

- *hashing*
- *array* with operation $A[i] += \Delta$ (but no direct assignment)

2.2.2 Full retroactivity

First, lets define the **search problem**: maintain set S of objects, subject to insert, delete, query(x, S).

Decomposable search problem [1980, 2007]: same as the search problem, with a restriction that the query must satisfy: query$(x, A \cup B) = f($query(x, A),query$(x, B))$, for some function f computed in $O(1)$ (sets A and B may overlap). Examples of problems with such a function include:

- *Dynamic nearest neighbor*
- *Successor on a line*
- *Point location*

Claim 1. *Full Retroactivity for decomposable search problems (with commutativity and inversions) can be done in $O(\lg m)$ factor overhead both in time and space (where m is the number of operations) using* **segment tree** *[1980, Bentley and Saxe [19]]*

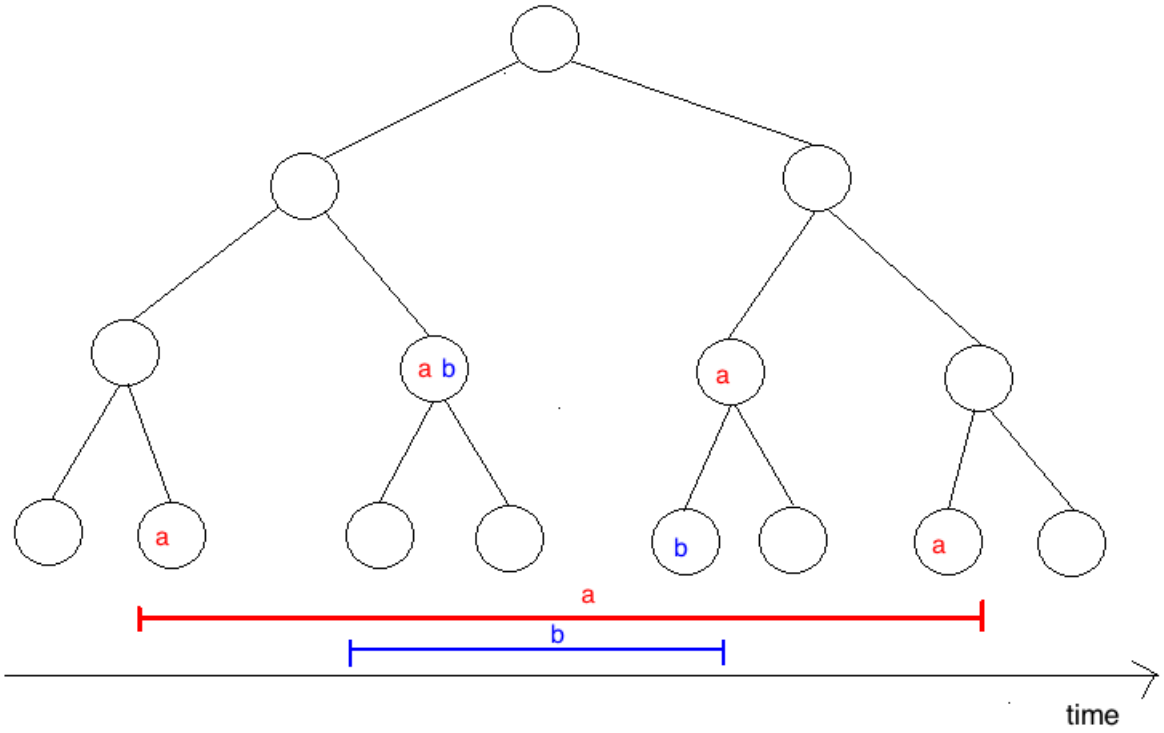


Figure 2.1: Segment Tree

We want to build a balanced search tree on time (leaves represent time). Every element "lives" in the data structure on the interval of time, corresponding to its insertion and deletion. Each element appears in $\lg n$ nodes.

To query on this tree at time t, we want to know what operations have been done on this tree from the beginning of time to t. Because the query is decomposable, we can look at $\lg n$ different nodes and combine the results (using the function f).

2.2.3 General case of full retroactivity

Roll back method:

- write down a (linear) chain of operations and queries
- change r time units in the past with factor $O(r)$ overhead.

That's the best we can do in general.

Lower bound: $\Omega(r)$ overhead necessary.

Proof: Data Structure maintains 2 values (registers): X and Y, initially $\emptyset$. The following operations are supported: $X = x$, $Y += \Delta$, $Y = X.Y$, query 'Y?'. Perform the following operations (Cramer's rule):

$$Y += a_n, X = X.Y, Y += a_{n-1}, X = X.Y, \ldots, Y += a_0$$

which is equivalent to computing

$$Y = a_n X^n + a_{n-1} X^{n-1} + ... + a_0$$

Now, execute Insert$(t = 0, X = x)$, which changes where the polynomial is evaluated. This cannot be done faster than re-evaluating the polynomial. In history-independent algebraic decision tree, for any field, independent of pre-processing of the coefficients, need $\Omega(n)$ field operations (result from 2001), where n is the degree of the polynomial.

2.2.4 Cell-probe problem

How many integers (words) of memory do you need to read to solve your problem? (gives a lower bound on running time).

Claim: $\Omega(\sqrt{\frac{r}{\lg r}})$.

OPEN problem: $\Omega(r)$.

Proof of the lower bound claim:

- DS maintans n words (integers of $w \geq \lg n$ bits).
- Arithmetic operations are allowed.
- Query = what is the value of an integer?
- Compute FFT.
- Retroactively change the words $\Rightarrow$ Dynamic FFT.
- Changing x_i requires $\Omega(\sqrt{n})$ cell probes.

2.2.5 Priority Queues

Now, let us move onto some more positive results. Priority queues represent a DS where retroactive operations potentially create chain reactions but we have still obtained some nice results for. The main operations are *insert* and *delete-min* which we would like to retroactively *Insert* and *Delete*.

Claim 2. *It is possible to implement a partially retroactive priority queue with only $O(\lg n)$ overhead per partially retroactive operation.*

Because of the presence of *delete-min*, the set of operations on priority queues is non-commutative. The order of updates now clearly matters, and *Insert*ing a *delete-min* retroactively has the potential to cause a chain reaction which changes everything that comes afterward. Partially retroactive priority queues are described in a paper by Demaine, Iacono, and Langerman [2].

To develop an intuition for how our DS changes given a retroactive operation, it is helpful to plot it on a two dimensional plane. The x-axis represents time and y-axis represents key value. Every $insert(t, k)$ operation creates a horizontal ray that starts at point (t, k) and shoots to the right (See Fig. 2.2).

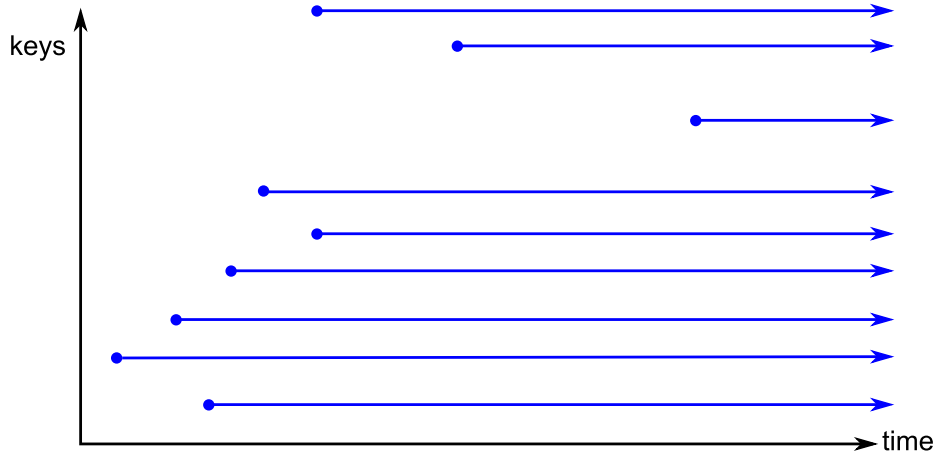


Figure 2.2: Graph of priority queue featuring only a set of *insert*s

Every *delete-min*() operation creates a vertical ray that starts at $(t, -\infty)$ and shoots upwards, stopping at the horizontal ray of the element it deletes. Thus, the horizontal ray becomes a line segment with end points (t, k) and (t_k, k), where t_k is the time of key k's deletion.

This combinations of inserts and deletes creates a graph of nonintersecting upside down "L" shapes, where each L corresponds to an *insert* and the *delete-min*() that deletes it. Elements which are never deleted remain rightward rays. Figure 2.3 demonstrates this by adding a few *delete-min*s to our previous graph of only *insert*s.

The rest of the discussion will focus on $Insert(t,$"$insert(k)$"$)$. It should be easy enough to convince yourself that $Delete(t,$"*delete-min*"$)$ has equivalent analysis if you think about Delete as inserting the deleted element at the time of deletion.

Consider the priority queue represented by figure 2.4. It's similar to the previous ones seen but with more inserts and deletes to better illustrate the chain-reactions of retroactive operations. Figure 2.5 shows what happens when elements are retroactively inserted. The retroactive operations and their chain reactions are shown in red. The cascading changes add up fast.

However, since we're only requiring partial retroactivity we only need to determine what element $Insert(t,$"$insert(k)$"$)$ inserts into Q_{now} where Q_{now} is the priority queue at the present time. Naively, it is easy to see that the element inserted at Q_{now} is: $\max\{k, k' \mid k' \text{ deleted at time} \geq t\}$.

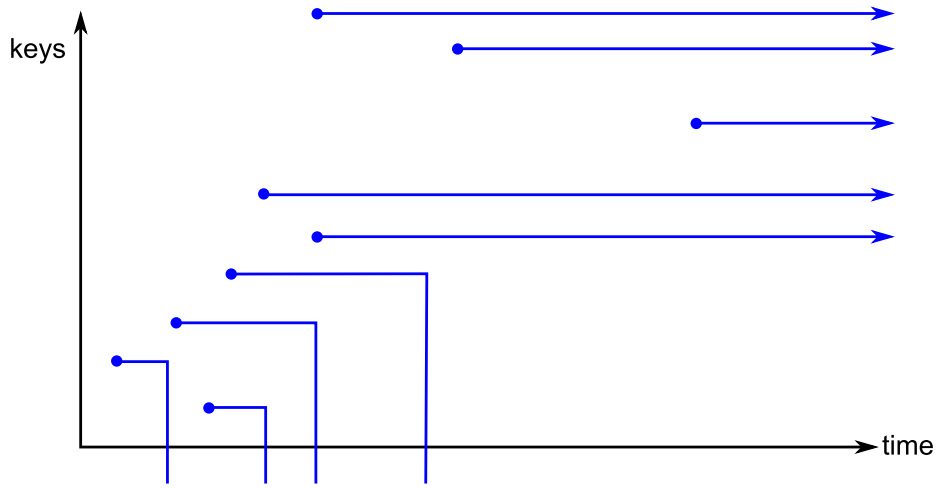


Figure 2.3: Adding *del-min*() operations leads to these upside down "L" shapes.

That is, the element that makes it to the "end" is the biggest element that was previously deleted (ie. the end of the chain-reaction shown in Figure 3) or simply k if it is bigger than those (ie. the insert caused no chain reactions).

Problem: Maintaining "deleted" elements is hard. It requires us to maintain the various chain-reactions which isn't efficient. Instead, we would like to simply keep track of inserts. Such a transformation is possible as long as we define the new concept of "bridges".

Definition 3. *We define time t to be a bridge if $Q_t \subseteq Q_{now}$.*

This simply means that all of the elements present at a bridge t' are also present at t_{now}. You can think of bridges as separating the chaotic chain-reactions that happen during retroactive operations as seen in figure 2.6.

If t' is the bridge preceding time t, then
$\max\{k' \mid k' \text{ deleted at time} \geq t\} = \max\{k' \notin Q_{now} \mid k' \text{ inserted at time} \geq t'\}$

With that transformation, we only need to maintain three data structures which will allow us to perform partially retroactive operations with only $O(\lg n)$ overhead.

- We will store Q_{now} as a balanced BST. It will be changed once per update.
- We will store a balanced BST where the leaves equal insertions, ordered by time, and augmented with $\forall$ node $x : \max\{k' \notin Q_{now} \mid k' \text{ inserted in } x\text{'s subtree}\}$.
- Finally, we will store a balanced BST where the leaves store all updates, ordered by time, and augmented by the following:

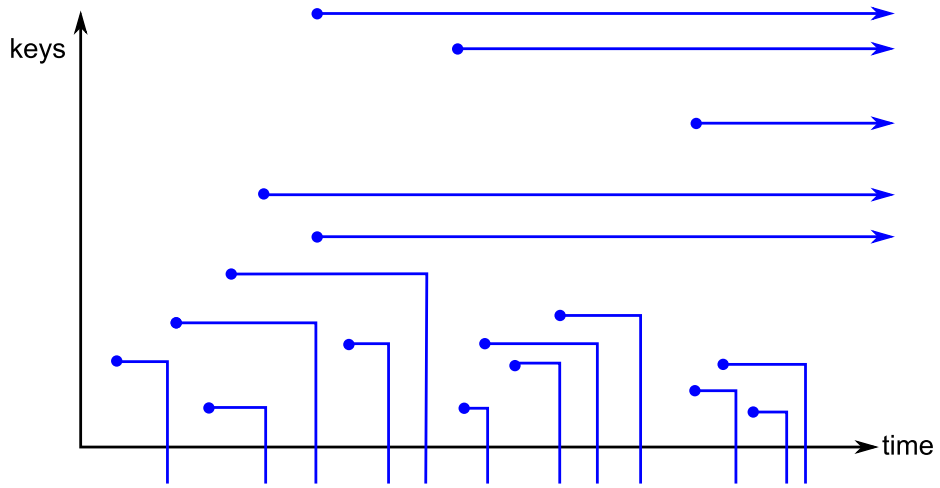


Figure 2.4: An L view representation of a priority queue with a more complicated set of updates

$$\begin{cases} 0 & \text{for inserts with } k \in Q_{now} \\ +1 & \text{for other inserts, } k \notin Q_{now} \\ -1 & \text{for } \textit{delete-mins} \end{cases}$$

as well as subtree sums.

Now, we have to uses these data structures to execute our $Insert$. This can be done in $O(\lg n)$ with the following steps.

- First, for an $Insert$ at time t we must find the preceding bridge at time t'. Using our BBST of all updates, we know that a bridge is a time(leaf) at which has a prefix of updates summing to 0 (with respect to the augmentation). This is possible in $O(\lg n)$ time due to the stored subtree sums. In brief, traverse the tree to find t, calculating the prefix sum as you descend. If t has a prefix sum of 0 then we're done. Otherwise, walk back up the tree to find the preceding update with a prefix sum of 0.
- Next, in our BBST of insertions we descend to t' and work back up the tree looking for the maximum node not already in Q_{now}. Because of the augmentation we store, this can be done in $O(\lg n)$ time.
- Finally, there are a collection of steps related to updating the data structures once we make our update, but given our data structures it should be easy to convince yourself that it can be done in $O(\lg n)$ time.

2.2.6 Other Structures

- *queue:* $O(1)$ partial, $O(\lg m)$ full

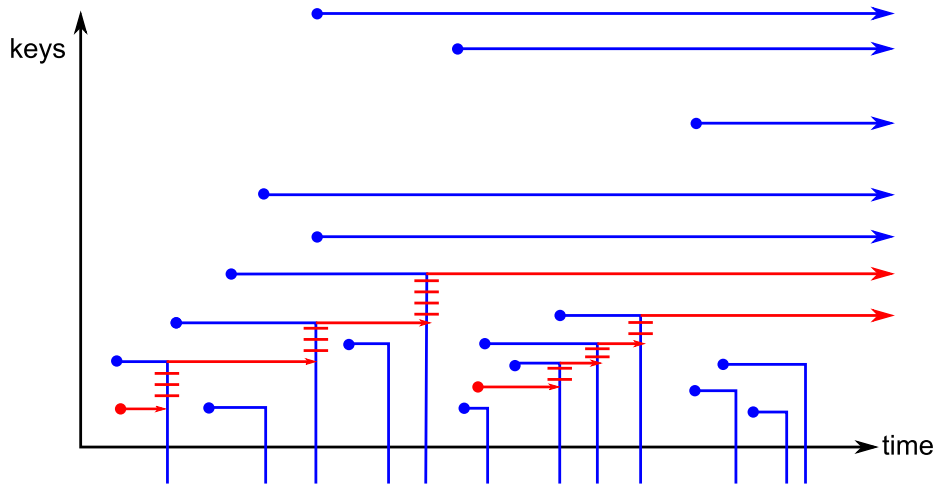


Figure 2.5: Retroactive inserts start at red dots and cause subsequent *delete-min* operations to effect different elements as shown.

- *deque:* $O(\lg m)$ full
- *union-find (incremental connectivity):* $O(\lg m)$ full
- *priority-queue:* $O(\sqrt{m}\lg m)$ full. This comes from the fact that any partially retroactive DS can be made fully retroactive with a $O(\sqrt{m})$ factor overhead. It's an **OPEN** problem whether or not we can do better.
- *successor:* This was actually the motivating problem for retroactivity. $O(\lg m)$ partial because it's a search problem. $O(\lg^2 m)$ full because it's also decomposable. However, Giora and Kaplan gave us a better solution of $O(\lg m)$ [16]! This new algorithm uses many data structures we'll learn about later; including fractional cascading (L3) and vam Emde Boas (L11).

2.2.7 Nonoblivious Retroactivity

Nonoblivious retroactivity was introduced in a paper by Acar, Blelloch, and Tangwongsan to answer the question, "What about my queries? [17]" Usually, when we use a data structure algorithmically (e.g. priority queue in Dijkstra) the updates we perform depend on the results of the queries.

To solve this problem we can simply add queries to our time line (fig. 2.7).

The notion of dependence depends upon the user, but we still want to do something reasonable. Now, when we make a retroactive update, we want our DS to tell us the location of the first query which changed as a result (e.g. the first error). Then we only have to re-run the algorithm from that first erroneous query. Our assumption is that the algorithm does this by *Delete*ing each wrong update and re-*Insert*ing the proper update in a strict left to right order such that each update is lower on the time-line than all errors.

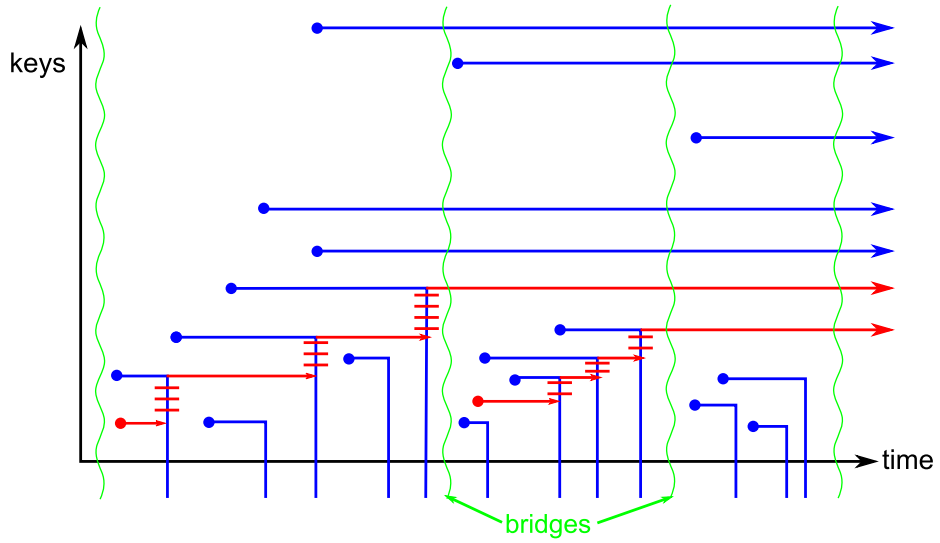


Figure 2.6: Bridges have been inserted as green wavy lines. Notice how they only cross elements present to the end of visible time.

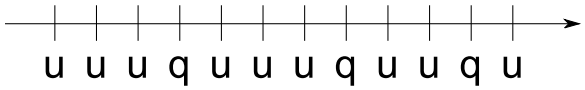


Figure 2.7: Time line of updates *and* queries.

After fixing each error, we can go back to using our retroactive DS in the usual way with no restrictions.

Priority-Queues: Our priority queues will support insert, delete, and min in $O(\lg m$ time per operation.

Because we have retroactive queries now, we separate the query and updates into different operations. Our matching retroactive queries will be Insert, Deleate, and Min. We will visualize our data structure using a similar 2D plane as before as seen in figure 2.8.

If we Insert an insert as shown in fig 2.8 then we get errors at each crossing. In our DS we will keep the 'incorrect' picture, but keep track of the errors. We will tell the algorithm where the first error (crossing) is and it will potentially Delete the query and re-Insert the operation thus getting the correct result.

However, we don't jump to the next error after this is done. Our assumption is that the algorithm will, based on the new query, do various updates which may even delete the new element before any further crossings.

Deleting a deletion is similar to Inserting an insertion as they both cause crossing errors.

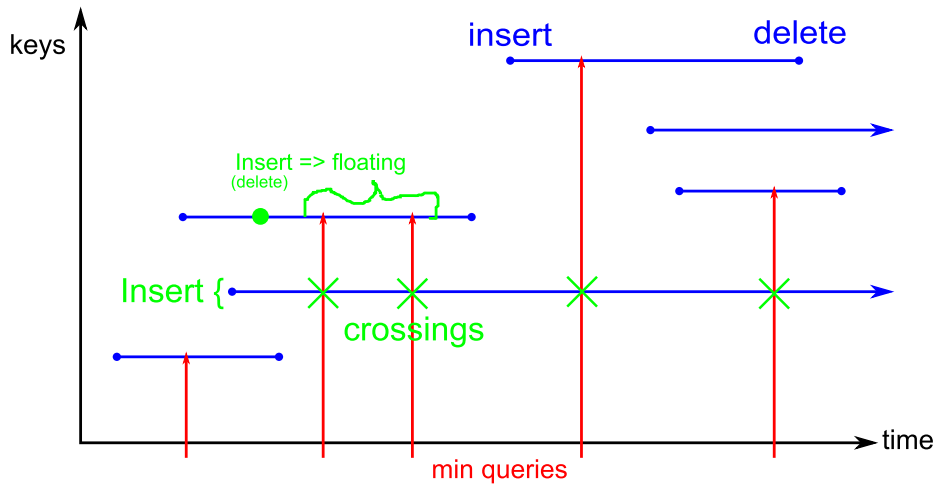


Figure 2.8: 2D representation of DS. *Insert*ing an *insert* leads to crossings as shown. *Insert*ing a *delete* leads to floating errors as shown.

The other two cases are Deleting an insertion and Inserting a deletion which cause floating errors. These are situations where queries return 'floating' values because they refer to a segment which is no longer there.

Both crossing and floating errors can be seen in figure 2.8.

We maintain the following two details

- First, we maintain the lowest leftmost crossing. This is equivalent to the leftmost lowest crossing. This is because of the invariant that all crossings involve horizontal segments with left endpoint left of *all* errors.
- Next, we maintain the left most floating error on each row separately.

Example: $Insert(t, \text{“min”})$. When adding a new query what is the first ray that I hit. This only involves the horizontal segments which are being inserted and deleting. This problem is called *upward ray shooting among dynamic segments*. We can solve it in $O(\lg m)$ per operation. This turns out to be the same as the fully retroactive successor problem mentioned earlier which was proved to be $O(\lg m)$ in [16].

The other cases are similar but you also need to use rightward ray shooting.

Lecture 3

Geometric data structure 1

Scribes: Brian Hamrick (2012) Ben Lerner (2012), Keshav Puranmalka (2012)

3.1 Overview

In the last lecture we saw the concepts of persistence and retroactivity as well as several data structures implementing these ideas.

In this lecture we are looking at data structures to solve the geometric problems of point location and orthogonal range queries. These problems encompass applications such as determining which GUI element a user clicked on, what city a set of GPS coordinates is in, and certain types of database queries.

3.2 Planar Point Location

Planar point location is a problem in which we are given a planar graph (with no crossings) defining a map, such as the boundary of GUI elements. Then we wish to support a query that takes a point given by its coordinates (x, y) and returns the face that contains it (see Figure 1 for an example). As is often the case with these problems, there is both a *static* and *dynamic* version of this problem. In the static version, we are given the entire map beforehand and just want to be able to answer queries. For the dynamic version, we also want to allow the addition and removal of edges.

3.2.1 Vertical Ray Shooting

A closely related problem to planar point location is **vertical ray shooting**. Just as in planar point location, we are interested in a planar graph defining a map. In this case, when we query a point (x, y) we are interested in the first line segment lying above it. Equivalently, if we imagine

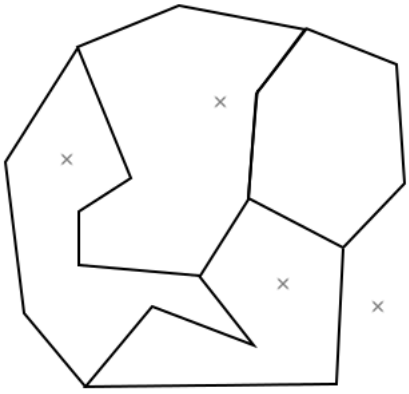


Figure 3.1: An example of a planar map and some query points

shooting a vertical ray from the query point, we want to return the first map segment that it intersects (see Figure 2).

We can use vertical ray shooting to solve the planar point location problem in the static case by precomputing, for each edge, what the face lying below it is. For the dynamic planar point location problem, we can again use this technique, but we need to maintain the face data dynamically, leading to an $O(\log n)$ additive overhead. The vertical ray shooting problem can be solved with a technique called a *line sweep*.

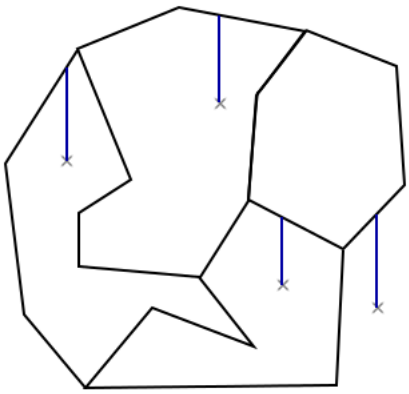


Figure 3.2: The same map and query points in the ray shooting view

3.2.2 Line Sweep

The **line sweep** is a relatively general technique for reducing the dimension of a geometric problem by 1. Given a geometric problem in d-dimensional space, we can consider a "vertical" $(d-1)$-dimensional crosssection, meaning a hyperplane perpendicular to the x_1 axis. Now we imagine our geometry problem within just that hyperplane and consider how it changes as we move the hyperplane along the x_1 axis. If this x_1 dimension has a suitable "timelike" interpretation, as it does in vertical ray shooting, then a persistent or fully retroactive data structure for the $(d-1)$-dimensional problem will allow us to solve the d-dimensional problem.

In the case of vertical ray shooting, in one dimension we can solve the problem with a balanced binary search tree. More specifically, we are doing *successor* queries. In the last lecture we saw how to make a partially persistent balanced binary search tree with $O(\log n)$ time queries, and we know that a fully retroactive successor query structure can be made with $O(\log n)$ queries as well. Note, however, that in this case the successor structure only allows us to handle horizontal segments, as the comparison function between general segments depends on the x-coordinate. What happens when we apply line sweep to solve the two dimensional problem?

As our vertical crosssection sweeps from left to right, we first note that no two line segments ever change order because of the assumption that there are no crossings in our map. This means that the tree structure only changes at a few discrete times. In particular, when we reach the left endpoint of a segment we are performing an insertion of that segment, and when we reach the right endpoint we are performing a deletion.

Supposing we implement this line sweep using a partially persistent balanced binary search tree, to make a vertical ray shooting query for the point (x, y), we find the update corresponding to the x-coordinate and make a query (using persistence) for (x, y) in that version of the data structure. It is also useful to note that this structure can be computed in $O(n \log n)$ preprocessing time, as shown by Dobkin and Lipton in [27].

Additionally, if we use the fully retroactive successor data structure, we can solve the dynamic vertical ray shooting problem with *horizontal* segments with $O(\log n)$ time queries. See [23] and [29].

Several variants of the vertical ray shooting problem are still open. Examples include:

- **OPEN:** Can we do $O(\log n)$ dynamic vertical ray shooting in a general planar graph?
- **OPEN:** Can we do $O(\log n)$ static ray shooting when the rays do not have to be vertical? Note that the three dimensional version of this problem is motivated by ray tracing.

3.2.3 Finding Intersections

The line sweep method can also be used to find intersections in a set of line segments, and this problem gives a good illustration of the line sweep method.

Given a set of line segments in the plane defined by their endpoints, we wish to find all of the intersections between them. See Figure 3 for an example of the problem.

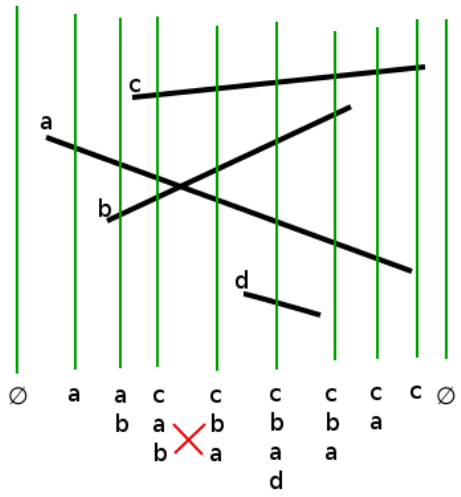


Figure 3.3: A line sweep for detecting intersections

To solve this problem, we use the line sweep technique where we store the line segments in a balanced binary search tree. The modifications to the search tree are as follows.

- The left endpoint of a segment causes us to insert that segment into the tree.
- The right endpoint of a segment causes us to delete that segment from the tree.
- Two segments crossing cause us to interchange their order within the tree.

We can use these ideas to solve the line segment intersection problem in $O(n \log n + k)$ when there are k intersections. To do this, we need to be able to efficiently determine the next time two segments would cross. We note that if a crossing would occur before we add or delete any more segments, it would have to involve two segments that are currently adjacent in the tree order. Then for each segment, we track when it would cross its successor in the tree and each internal node tracks the earliest crossing in its subtree. It is not difficult to maintain this extra data in $O(\log n)$ time per update, and when performing a swap can be done in constant time.

3.3 Orthogonal range searching

In this problem we're given n points in d dimensions, and the query is determining which points fall into a given box (a box is defined as the cross product of d intervals; in two dimensions, this is just a rectangle). See Figure 4 for an example. This is, in a sense, the inverse of the previous problem, where we were given a planar graph, and the query was in the form of a point.

In the static version of the problem we can preprocess the points, while in the dynamic version points are added and deleted. In both cases, we query the points dynamically. The query has different versions:

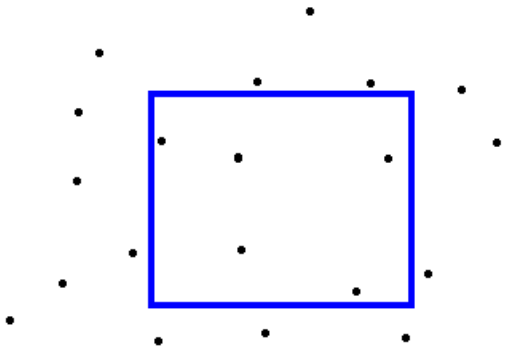


Figure 3.4: Orthogonal range searching

- Are there any points in the box? This is the "existence" version of the problem, and it's the easiest.
- How many points are in the box? This can solve existence as well.
- What are all the points in the box? Alternatively, what is a point, or what are ten points, in the box?

These questions are very similar, and we can solve them with the same efficiency. Once exception is the last question - for a given input, the answer may involve returning every point in the set, which would take $O(n)$ time. We're therefore looking for a solution something like $O(\log n + k)$, where k is the size of the output.

3.3.1 Range Trees

Let's start with the 1-dimensional case, $d = 1$. To solve it, we can just sort the points and use binary search. The query is an interval $[a, b]$; we can find the predecessor of a and the successor of b in the sorted list and use the results to figure out whether there are any points in the box; subtract the indices to determine the number of points; or directly print a list of points in the box. Unfortunately arrays don't generalize well, although we will be using them later.

We can achieve the same runtimes by using a structure called **Range Trees**. Range trees were invented by a number of people simultaneously in the late 70's [22], [21], [30], [31], [35].

We can build a range tree as follows. Consider a balanced binary search tree (BBST) with data stored in the leaves only. This will be convenient for higher dimensions. Each non-leaf node stores the min and max of the leaves in its subtrees; alternatively, we can store the max value in the left subtree if we want to store just one value per node.

Again, we search for PRED(a) and SUCC(b) (refer to Figure 5). As we search, we'll move down the tree and branch at a number of points. (As before, finding PRED(a) and SUCC(b) takes $O(\log n)$ time.) Once the paths to PRED(a) and SUCC(b) diverge, any time we turn left at a node while searching for PRED(a), we know that all the leaves of the right subtree of that node are in the given interval. The same thing is true for left subtrees of the right branch. If the left tree branches right

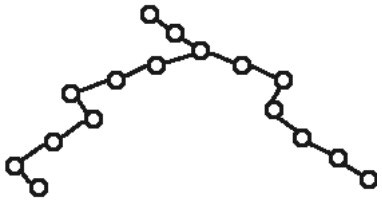


Figure 3.5: Branching path to PRED(a) and SUCC(b)

or the right tree branches left, we don't care about the subtree of the other child; those leaves are outside the interval. The answer is then implicitly represented as follows: If we store the size of

Figure 3.6: The subtrees of all the elements between a and b

each node's subtree in the node, we can compute the number of elements in our list in $O(\log n)$ time. To find the first k numbers, we can visit the first k elements after PRED(a) in $O(k)$ time (the operations in the previous two sentences might be easier to visualize with reference to Figure 6). These are the same results we received with arrays, but range trees are easier to generalize to higher d.

Let's look at the $2-d$ case, searching for points between a_1 and b_1 in x and a_2 and b_2 in y. We can build a range tree using only the x coordinate of each point, and then repeat the above procedure to figure out which points fall between a_1 and b_1.

We now want to sort the points by y coordinate. We can't use a range tree over all the points, because we're only interested in the points which we know are between a_1 and b_1, rather than every point in the set. We can restrict our attention to these points by creating, for each x subtree, to a corresponding one dimensional y range tree, consisting of all the points in that x subtree, but now sorted by their y-coordinate. We'll also store a pointer in each node of the x range tree to the corresponding y tree (see Figure 7 for an example). For example, α_x, β_x, and γ_x point to corresponding subtrees α_y, β_y, and γ_y. γ_x is a subtree of β_x in x, but γ_y is disjoint from β_y (even though γ_y and γ_x store the same set of points).

This structure takes up a lot of space: every point is stored in $\log n$ y range trees, so we're using $\theta(nlogn)$ space. However, we can now do search efficiently - we can first filter our point set by x coordinate, and then search the corresponding y range trees to filter by the y coordinate. We're searching through $O(\log n)$ y subtrees, so the query runs in $O(\log^2 n)$ time.

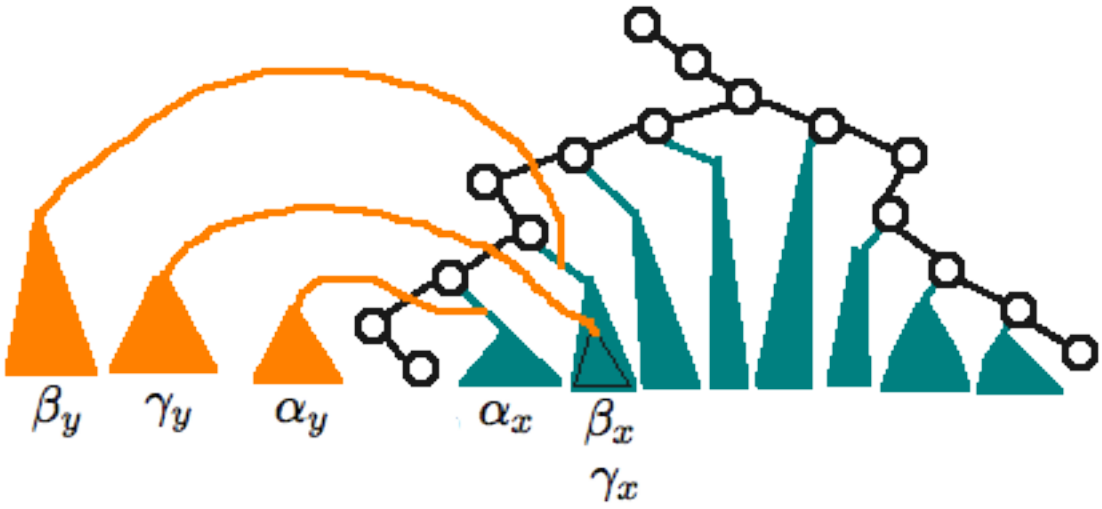


Figure 3.7: Each of the nodes at the x level have a pointer to all of the children of that node sorted in the y dimension, which is denoted in orange.

We can extend this idea to d dimensions, storing a y subtree for each x subtree, a z subtree for each y subtree, and so on. This results in a query time of $O(\log^d n)$. Each dimension adds a factor of $O(\log n)$ in space; we can store the tree in $O(n \log^{d-1} n)$ space in total. If the set of points is static, we can preprocess them in $O(n \log^{d-1})$ time for $d > 1$; sorting takes $O(n \log n)$ time for $d = 1$. Building the range trees in this time bound is nontrivial, but it can be done.

3.3.2 Layered Range Trees

We can improve on this data structure by a factor of $\log n$ using an idea called **layered range trees** (See [28], [36], [34]). This is also known as **fractional cascading**, or **cross linking**, which we'll cover in more detail later. The general idea is to reuse our searches.

In the $2 - d$ case, we're repeatedly performing a search for the same subset of y-coordinates, such as when we search both α_2 and γ_2 for points between a_2 and b_2.

To avoid this, we do the following. Instead of a tree, store an array of all the points, sorted by y-coordinate (refer to Figure 8). As before, have each node in the x tree point to a corresponding subarray consisting of those points in the tree, also sorted by their y-coordinate.

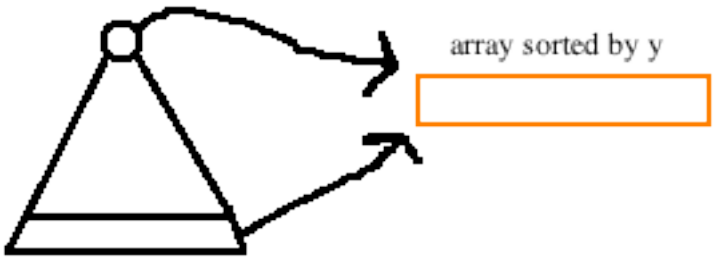


Figure 3.8: Storing arrays instead of range trees

We can do the same thing as before, taking $O(\log n)$ time to search through each array by y; but we can also do better: we can search through just one array in y, the array corresponding to the root of the tree, which contains every element, sorted by y.

We want to maintain this information as we go down the x range tree. As we move from a node v in the x tree to one of its children, say v_r, v_r will point to an array, sorted in y, that contains a subset of the y array v points to. This subset is arbitrary, as it depends on the x-coordinate of the points.

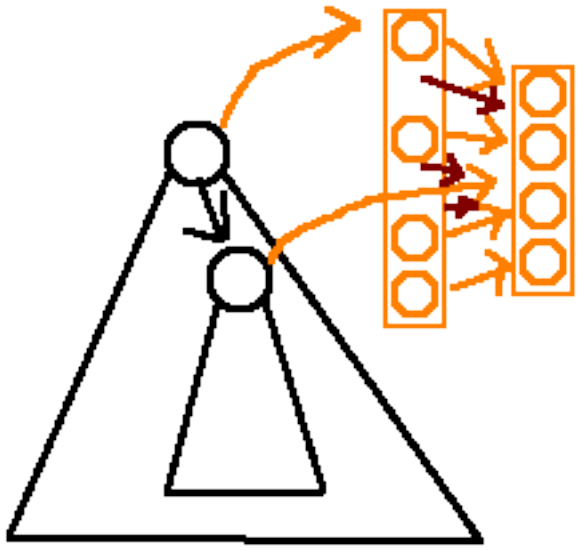


Figure 3.9: Here is an example of cascading arrays in the final dimension. The large array contains all the points, sorted on the last dimensions. The smaller arrays only contain points in a relevant subtree (the small subtree has a pointer to the small array). Finally, the big elements in the bih array has pointers to its "position" in the small array.

Let's store a pointer in each element of v's array to an element in v_r's array. If the element is in both arrays, it should point to itself; otherwise, let it point to its successor in v_r's array. (Each element will actually store two pointers, one to v_r's array and one to v_ℓ's array.) Then, in particular, we can find the predecessor and successor of a_2 and b_2 in any subtree by just following pointers. This lets you query the y-coordinate in $O(\log n + k)$ time, so we avoid the extra log factor from the previous strategy, allowing us to solve the $2 - d$ problem in $O(\log n)$ time in total. For arbitrary d, we can use this technique for the last dimension; we can thereby improve the general query to $O(\log^{d-1} n + k)$ time for $d > 1$.

3.3.3 Dynamic Point Sets

We can make the previous scheme dynamic using amortization. In general, when we update a tree, we only change a few things, usually near the leaves. Updating a tree near its leaves takes constant time, as we're only changing a few things. Occasionally, we'll need to update a large section of the tree, which will take a longer time, but this happens infrequently.

It turns out that if we have $O(n \log^{d-1} n)$ space and preprocessing time, we can make the structure

dynamic for free using weight balanced trees.

3.3.4 Weight Balanced Trees

There are different kinds of weight balanced trees; we'll look at the oldest and simplest version, $BB[\alpha]$ trees [33]. We've seen examples of height-balanced trees: AVL trees, where the left and right subtrees have heights within an additive constant of each other, and Red-Black trees, where the heights of the subtrees are within a multiplicative factor of each other.

In a weight balanced tree, we want to keep the size of the left subtree and the right subtree roughly the same. Formally, for each node v we want

$$\textsc{size}(\textsc{left}(v)) \geq \alpha \cdot \textsc{size}(v)$$

$$\textsc{size}(\textsc{right}(v)) \geq \alpha \cdot \textsc{size}(v)$$

We haven't defined size: we can use the number of nodes in the subtree, the number of leaves in the subtree, or some other reasonable definition. We also haven't selected α. If $\alpha = \frac{1}{2}$, we have a problem: the tree must be perfectly balanced at all times. Taking a small α, however, (say, $\alpha = \frac{1}{10}$), works well. Weight balancing is a stronger property than height balancing: a weight balanced tree will have height at most $\log_{1/\alpha} n$.

We can apply these trees to our layered range tree. [31][34] Updates on a weight balanced tree can be done very quickly. Usually, when we add or delete a node, it will only affect the nodes nearby. Occasionally, it will unbalance a large part of the tree; in this case, we can destroy that part of the tree and rebuild it. When we do so, we can rebuild it as a perfectly balanced tree.

Our data structure only has pointers in one direction - each parent points to its children nodes, but children don't point to their parents, or up the tree in general. As a result, we're free to rebuild an entire subtree whenever it's unbalanced. And once we rebuild a subtree, we can make at least $\theta(k)$ insertions or deletions before it becomes unbalanced again, where k is the size of the subtree.

When we do need to rebuild a subtree, we can charge the process of rebuilding to the $\theta(k)$ updates we've made. Since each node we add can potentially unbalance every subtree it's a part of (a total of $\log(n)$ trees), we can update the tree in $\log(n)$ amortized time (assuming that a tree can be rebuilt in $\theta(k)$ time, which is easy).

So, for layered range trees, we have $O(\log^d n)$ amortized update, and we still have a $O(\log^{d-1} n)$ query.

3.3.5 Further results

For static orthogonal range searching, we can achieve a $O(\log^{d-1} n)$ query for $d > 1$ using less space: $O\left(n\frac{\log^{d-1}(n)}{\log\log n}\right)$ [24]. This is optimal in some models.

We can also achieve a $O(\log^{d-2} n)$ query for $d > 2$ using $O(n \log^d(n))$ space [25], [26]. A more recent result uses $O(n \log^{d+1-\epsilon}(n))$ space [20]. This is conjectured to be an optimal result for queries.

There are also non-orthogonal versions of this problem - we can query with triangles or general simplices, as well as boxes where one or more intervals start or end at infinity.

3.4 Fractional Cascading

Fractional cascading is a technique from Chazelle and Guibas in [25] and [26], and the dynamic version is discussed by Mehlhorn and Näher in [32]. It is essentially the idea from layered range trees put into a general setting that allows one to eliminate a log factor in runtime.

3.4.1 Multiple List Queries

To illustrate the technique of fractional cascading, we will use the following problem. Suppose you are given k sorted lists $L_1, \ldots, L_k$ each of length n, and you want to find the successor of x in each of them. One could trivially solve this problem with k separate binary searches, resulting in a runtime of $O(k \log n)$. Fractional cascading allows this problem to be solved in $O(k + \log n)$.

To motivate this solution we can look at the layered range trees above and think about how we can retain information when moving from one list to the next. In particular, if we were at a certain element of L_1, we could store where that element lies in L_2, and then quickly walk from one list to the next. Unfortunately, it might be the case that *all* of the elements of L_2 lie between two of the elements of L_1, in which case our position in L_1 doesn't tell us anything. So we should add some more information to L_1. This leads to the idea of fractional cascading.

Define new lists $L'_1, \ldots, L'_k$ by $L'_k = L_k$, and for $i < k$, let L'_i be the result of merging L_i with every other element of L'_{i+1}. Note that $|L'_i| = |L_i| + \frac{1}{2}|L'_{i+1}|$, so $|L_i| \leq 2n = O(n)$.

For each $i < k$, keep two pointers from each element. If the element came from L_i, keep a pointer to the two neighboring elements from L'_{i+1}, and vice versa. These pointers allow us to take information of our placement in L'_i and in $O(1)$ turn it into information about our placement in L_i and our placement in half of L'_{i+1}.

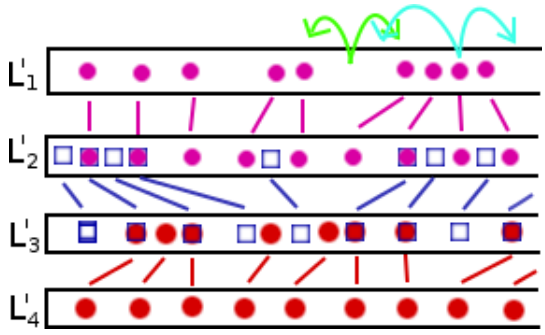


Figure 3.10: An illustration of fractional cascading

Now to make a query for x in all k lists is quite straightforward. First, query L'_1 for the location of x with a binary search. Now to find the location of x in L'_{i+1} from the location of x in L'_i, find the two neighboring elements in L'_{i+1} that came from L'_i using the extra pointers. Then these elements have exactly one element between them in L'_{i+1}. To find our actual location in L'_{i+1}, we simply do a comparison with that intermediate element. This allows us to turn the information about x's location in L'_i into information about x's location in L'_{i+1} in $O(1)$ time, and we can retrieve x's location in L_i from its location in L'_i in $O(1)$, and thus we can query for x in all k lists in $O(k + \log n)$ time.

3.4.2 General Fractional Cascading

To generalize the above technique, we first note that we can replace "every other element" with "α of the elements, distributed uniformly", for any small α. In this case, we used $\alpha = \frac{1}{2}$. However, by using smaller α, we can do fractional cascading on any graph, rather than just the single path that we had here. To do this, we just (modulo some details) cascade α of the set from each vertex along each of its outgoing edges. When we do this, cycles may cause a vertex to cascade into itself, but if we choose α small enough, we can ensure that the sizes of the sets stays linearly bounded.

In general, fractional cascading allows us to do the following. Given a graph where

- Each vertex contains a set of elements
- Each edge is labeled with a range $[a, b]$
- The graph has *locally bounded in-degree*, meaning for each $x \in \mathbb{R}$, the number of incoming edges to a vertex whose range contains x is bounded by a constant.

We can support a search query that finds x in each of k vertices, where the vertices are reachable within themselves from a single node of the graph so the edges used form a tree with k vertices. As before, we could do the search query in $O(k \log n)$ with k separate binary searches and fractional cascading improves this to $O(k + \log n)$, where n is the largest size of a vertex set.

Lecture 4

Geometric data structure 2

Scribes: Brandon Tran (2012), Nathan Pinsker (2012), Ishaan Chugh (2012), David Stein (2010), Jacob Steinhardt (2010)

4.1 Overview- Geometry II

We first show how to further utilize fractional cascading to remove yet another lg factor from d-dimensional orthogonal range searches, so that we can query in $\mathcal{O}(n \lg^{d-2} n)$. Recall that fractional cascading allow for:

- searching for x in k lists of length n in $\mathcal{O}(\lg n + k)$ time.
- this works even if the lists are found online by navigating a bounded-degree graph

Then we move on to the topic of kinetic data structures. These are data structures that contain information about objects in motion. They support the following three types of queries: (i) change the trajectory of an object; (ii) move forward to a specified point in time (advance); (iii) return information about the state of the objects in the current time. Examples we will go over are kinetic predecessor/successor and kinetic heaps.

4.2 3D Orthogonal Range Search in $\mathcal{O}(\lg n)$ Query Time

The work here is due to Chazell and Guibas [CG86].

In general we want to query on the section of our points in space defined by: $[a_1, b_1] \times [a_2, b_2] \times [a_3, b_3]$.

4.2.1 Step 1

We want to query $(-\infty, b_2) \times (-\infty, b_3)$, which we call the restricted two dimensional case. There are no left endpoints in our query, so the area we're querying basically looks like:

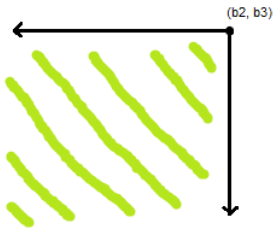


Figure 4.1: Query Area

We want to call points dominated by (b_2, b_3) in the yz-plane in time: $\mathcal{O}(k)$ + the time needed to search for b_3 in the list of z-coordinates. We transform this to a ray-stabbing query. The basic idea is to use horizontal rays to stab vertical rays, where the horizontal ray is our ray of query.

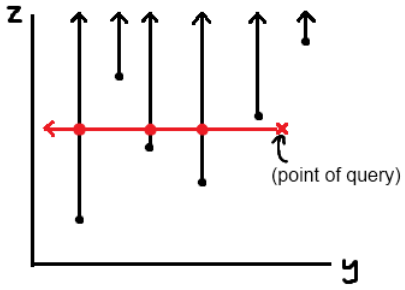


Figure 4.2: Stabbing Vertical Rays

A crossing indicates that the point whose vertical ray caused the crossing lies in the query range. What we want to do is walk along our horizontal ray and spend constant time per crossing. For each vertical ray, we keep a horizontal line at the bottom point of the ray. This line is extended left and right until it hits another vertical ray. (These lines are blue in the figure below.)

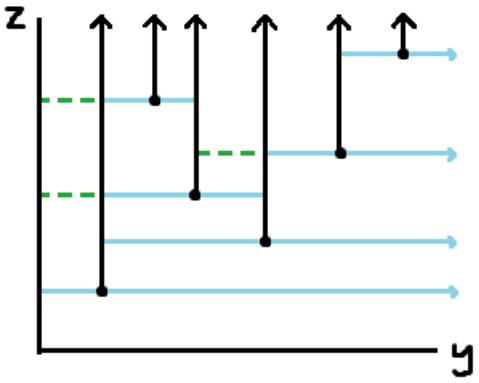


Figure 4.3: Extending Lines

When traversing, we binary search for the z value and then go from "face" to "face" in the picture. However, we want to bound the degree of the faces (the number of faces bordering it) to make the query more efficient. Thus we look at all the horizontal lines that end at some vertical ray and

extend approximately half of them into the face to create more faces with lower degree (the green dotted lines in the figure above). Since we only extend half, the number of lines decreases by a factor of 2 for every vertical ray. Since $\sum_{i=0}^{\infty} \frac{1}{2^i} = 2$, we only have an $\mathcal{O}(n)$ increase in space. This is due to Chazelle [C86].

4.2.2 Step 2

Now we want to query $[a_1, b_1] \times (-\infty, b_2) \times (-\infty, b_3)$ in $\mathcal{O}(k) + \mathcal{O}(\lg n)$ searches. To do this, we use a one dimensional range tree on the x-coordinates, where each node stores the structure in Step 1 on points in its subtree.

4.2.3 Step 3

Now we query $[a_1, b_1] \times [a_2, b_2] \times (-\infty, b_3)$. We do something similar to a range tree on the y-coordinate, where each node v stores a key $= \max(\text{left}(v))$, the structure in Step 2 on points in its right subtree, and the structure in Step 2, but inverted, on points in its left subtree. The inverted structure allows queries of $[a_1, b_1] \times (a_2, \infty) \times (-\infty, b_3)$. This can easily be done in the same way as before. At this point, we know that we can only afford a constant number of calls to the structures in Step 2.

We start by walking down the tree. At each node v, if $\text{key}(v) < a_2 < b_2$ we go to its right child. If $\text{key}(v) > b_2 > a_2$, we go to its left child. Finally, if $a_2 \leq \text{key}(v) \leq b_2$, we query on the Step 2 structure and the inverted Step 2 structure at this node. The reason this gives us all the points we want is that the structure in the left subtree allows for queries from a_2 to ∞. However, since this only stores points with y-coordinate less than or equal to $\text{key}(v)$, we get the points between a_2 and $\text{key}(v)$. Similarly, the structure in the right subtree allows for leftward queries, so we get the points between $\text{key}(v)$ and b_2. Thus we only needed two calls to the Step 2 structures and one $\lg n$ search (walking down the tree). We note that we could have used this idea for all three coordinates, but we didn't need to since range trees give us the x-coordinate for free (this requires a lg factor of extra space).

4.2.4 Step 4

Finally, we query $[a_1, b_1] \times [a_2, b_2] \times [a_3, b_3]$. We do the exact same thing as in Step 3, except now on the z-coordinates, and at each node we store the structure in Step 3.

4.2.5 Total Time and Space

This whole process required $\mathcal{O}(\lg n + k)$ time and $\mathcal{O}(n \lg^3 n)$ space. This means that in general, for $d \geq 3$ dimensions, orthogonal range search costs $\mathcal{O}(\lg^{d-2}(n) + k)$. The basic idea was that if using more space is not a problem, one-sided intervals are equivalent to two-sided intervals. The reason why this doesn't further reduce the runtime is that if we were to try this on four dimensions, the

initial structure in Step 1 would need to account for more than two dimensions, which it cannot do quickly.

4.3 Kinetic Data Structures

The work here is due to Basch, Guibas, and Hershberger [BGH99].

The idea behind kinetic data structures is that we have objects moving with some velocity, and we want to know where they will be at some future time. Also, we want to be able to account for the objects' change in trajectories. The operations are:

- advance(t) : now $= t$
- change($x, f(t)$) : changes trajectory of x to $f(t)$

We will only be dealing with the following models of trajectories:

- affine: $f(t) = a + bt$
- bounded-degree arithmetic: $f(t) = \sum_{i=0}^{n} a_i t^i$
- pseudo-algebraic: any certificate of interest flips between true and false $\mathcal{O}(1)$ times

We define a certificate as a boolean function about the points.

4.3.1 Approach

1. Store the data structure that is accurate now. Then queries about the present are easy.
2. Augment the data structures with certificates which store conditions under which the data structures is accurate and which are true now.
3. Compute the failure times when each certificate fails to hold (takes $\mathcal{O}(1)$ time).
4. Place the failure times into a priority queue. When we advance time, we look for certificates that fail and "fix" them.

4.3.2 Metrics

There are four metrics we generally use to measure the performance of a kinetic data structure:

- responsiveness — when an event happens (e.g. a certificate failing), how quickly can the data structure be fixed?
- locality — what is the most number of certificates any object participates in?
- compactness — what is the total number of certificates?

- efficiency What is the ratio of the worst-case number of data structure events (disregarding modify) to the worst case number of "necessary" changes? (The notion of a "necessary change" is somewhat slippery. In practice we will define it on a per-problem basis.)

4.3.3 Kinetic Predecessor

The first kinetic data structure problem we will look at is *kinetic predecessor* (also called kinetic sorting). We want to support queries asking for the predecessor of an element (assuming the elements are sorted by value). We take the following approach:

1. Maintain a balanced binary search tree.
2. Let $x_1, \ldots, x_n$ be the in-order traversal of the BST. Keep the certificates $\{(x_i < x_{i+1}) \mid i = 1, \ldots, n-1\}$.
3. Compute the failure time of each certificate as failure_time$_i := \inf\{t \geq \text{now} \mid x_i(t) > x_{i+1}(t)\}$. For example, if the motion of each object is linear, compute the first time after the current point at which two lines intersect.
4. Implement *advance*(t) as follows:

```
while t >= Q.min:
  now = Q.min
  event(Q.delete-min)
now = t

define event (certificate (x[i] < x[i+1])):
  swap(x[i], x[i+1]) in BST
  add certificate (x[i+1] <= x[i])
  replace certificate (x[i-1] <= x[i]) with (x[i-1] <= x[i+1])
  replace certificate (x[i+1] <= x[i+2]) with (x[i] <= x[i+2])
```

Each time a certificate fails, we remove it from the priority queue and replace any certificates that are no longer applicable. It takes $\mathcal{O}(\lg n)$ time to update the priority queue per event, but the problem is that there may be $\mathcal{O}(n^2)$ events. We now analyze according to our metrics.

Efficiency If we need to "know" the sorted order of points, then we will need an event for every order change. Each pair of points can swap a constant number of times, yielding $\mathcal{O}(n^2)$ possible events. Therefore the efficiency is $\mathcal{O}(1)$.

Responsiveness Because we used a BBST, we can fix the constant number of certificate failures in $\mathcal{O}(\lg n)$ time.

Local Each data object participates in $\mathcal{O}(1)$ certificates. In fact, each participates in at most 2 certificates.

Compact There were $\mathcal{O}(n)$ certificates total.

4.3.4 Kinetic Heap

The work here is due to de Fonseca and de Figueiredo [FF03]

We next consider the kinetic heap problem. For this problem, the data structure operation we want to implement is $find_{min}$. We do this by maintaining a heap (for now, just a regular heap, no need to worry about Fibonacci heaps). Our certificates check whether each node is smaller than its two children in the heap. Whenever a certificate breaks, we simply apply the heap-up operation to fix it, and then modify the certificates of the surrounding nodes in the heap appropriately. In this case, our data structure has $\mathcal{O}(\lg n)$ responsiveness, $\mathcal{O}(1)$ locality, and $\mathcal{O}(n)$ compactness. The only non-obvious metric is efficiency. We show below that the efficiency is in fact $\mathcal{O}(\lg n)$ (by showing that the total number of events is $\mathcal{O}(n \lg n)$).

Analyzing the number of events in kinetic heap. We will show that there are at most $\mathcal{O}(n \lg n)$ events in a kinetic heap using amortized analysis. For simplicity, we will carry through our analysis in the case that all functions are linear, although the general case works the same.

Define $\Phi(t, x)$ as the number of descendents of x that overtake x at some time after t.

Define $\Phi(t, x, y)$ as the number of descendants of y (including y) that overtake x at some time greater than t. Clearly, $\Phi(t, x) = \Phi(t, x, y) + \Phi(t, x, z)$, where y and z are the children of x.

Finally, define $\Phi(t) = \sum_x (\Phi(t, x))$. Φ will be the potential function we use in our amortized analysis. Observe that $\Phi(t)$ is $\mathcal{O}(n \lg n)$ for any value of t, since it is at most the total number of descendants of all nodes, which is the same as the total number of ancestors of all nodes, which is $\mathcal{O}(n \lg n)$. We will show that $\Phi(t)$ decreases by at least 1 each time a certificate fails, meaning that certificates can fail at most $\mathcal{O}(n \lg n)$ times in total.

Consider the event at time t^+ when a node y overtakes its parent x, and define z to be the other child of x. The nodes x and y exchange places, but no other nodes do. This means that the only changes to any of the potential functions between t and t^+ are:

$$\Phi(t^+, x) = \Phi(t, x, y) - 1$$

and

$$\Phi(t^+, y) = \Phi(t, y) + \Phi(t, y, z).$$

Since $y > x$ now, we also see that

$$\Phi(t, y, z) \leq \Phi(t, x, z).$$

Finally, we need that

$$\Phi(t, x, z) = \Phi(t, x) - \Phi(t, x, y).$$

Then

$$\begin{aligned}
\Phi(t^+, x) + \Phi(t^+, y) &= \Phi(t, x, y) - 1 + \Phi(t, y) + \Phi(t, y, z) \\
&\leq \Phi(t, x, y) - 1 + \Phi(t, y) + \Phi(t, x, z) \\
&= \Phi(t, x, y) - 1 + \Phi(t, y) + \Phi(t, x) - \Phi(t, x, y) \\
&= \Phi(t, y) + \Phi(t, x) - 1
\end{aligned}$$

From these, it follows that:

$$\Phi(t^+) \leq \Phi(t) - 1,$$

which completes the analysis. We conclude that the total number of data structure modifications is $\mathcal{O}(n \lg n)$.

4.3.5 Other Results

We now survey the results in the field of kinetic data structures. For a more comprehensive survey, see [Gui04].

2D convex hull. (Also diameter, with, and minimum area/perimeter rectangle.) Efficiency: $\mathcal{O}(n^{2+\epsilon})$, $\Omega(n^2)$ [BGH99]. **OPEN:** 3D case.

Smallest enclosing disk. $\mathcal{O}(n^3)$ events. **OPEN:** $\mathcal{O}(n^{2+\epsilon})$?

Approximatie results. We can $(1+\epsilon)$-approximate the diameter and the smallest disc/rectangle in $\frac{1}{\epsilon}^{\mathcal{O}(1)}$ events [AHP01].

Delaunay triangulations. $\mathcal{O}(1)$ efficiency [AGMR98]. **OPEN:** how many total certificate changes are there? It is known to be $\mathcal{O}(n^3)$ and $\Omega(n^2)$.

***Any* triangulation.** $\Omega(n^2)$ changes even with Steiner points [ABdB$^+$99]. $\mathcal{O}(n^{2+\frac{1}{3}})$ events [ABG$^+$02]. **OPEN:** $\mathcal{O}(n^2)$ events? We can achieve $\mathcal{O}(n^2)$ for pseudo-triangulations.

Collision detection. See the work done in [KSS00], [ABG$^+$02], and [GXZ01].

Minimal spanning tree. $\mathcal{O}(m^2)$ easy. **OPEN:** $o(m^2)$? $\mathcal{O}(n^{2-\frac{1}{6}})$ for H-minor-free graphs (e.g. planar) [AEGH98].

Lecture 5

Dynamic optimality 1

Scribes: Brian Basham (2012), Travis Hance (2012), Jayson Lynch (2012)

5.1 Overview

In the next two lectures we study the question of dynamic optimality, or whether there exists a binary search tree algorithm that performs "as well" as all other algorithms on any input string. In this lecture we will define a binary search tree as a formal model of computation, show some analytic bounds that a dynamically optimal binary search tree needs to satisfy, and show two search trees that are conjectured to be dynamically optimal. The first is the splay tree, which we will cover only briefly. The second will require us to build up a geometric view of a sequence of binary search tree queries.

5.2 Binary Search Trees

The question we will explore in this lecture is whether or not there is a "best" binary search tree. We know that there are self-balancing binary search trees that take $O(\log n)$ per query. Can we do better? In order to explore this question, we need a more rigorous definition of binary search tree.

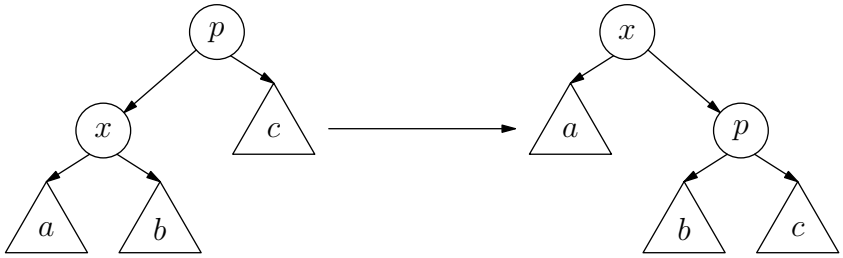
In this lecture we will treat the binary search tree (BST) as a model of computation that is a subset of the pointer machine model of computation.

5.2.1 Model of Computation

A BST can be viewed is a model of computation where data must be stored as keys in a binary search tree. Each key has a pointer to its parent (unless it is the root) and a pointer to its left and right children, or a null pointer if they do not exist. The value of the key stored in the left child of a node must be less than or equal to the value of the key stored at the node, which is in turn less

than or equal to the value of the key stored at the right child. The model supports the following unit-cost operations:

- Walk to left child.
- Walk to right child.
- Walk to parent.
- Rotate node x.



The first three operations only change a pointer into the tree. The last operation updates the tree itself.

These models support the query *Search*(x), which starts with a pointer to the root and can use the above operations, but must at some point visit the node with key x. In these two lectures we will assume for convenience that queries are always successful. We will also ignore insertions and deletions, for ease of definitions and proofs.

5.2.2 Is There a Best BST?

We already know of a number of self-balancing BSTs that take $O(\log n)$ per search. These include AVL trees and red-black trees.

Question: Is $O(\log n)$ the best possible?

Answer: Yes – in the worst case. Any tree on n items must have depth $\Omega(\log n)$, and an adversary could choose to search for the key located at the largest depth every round. However, in some cases we can do better. In general, we will consider a sequence of searches, and consider the best possible total time to complete the entire sequence.

5.2.3 Search Sequences

We will assume the n keys stored in the tree have values $\{1, 2, \ldots n\}$. We will consider sequences of search operations $x_1, x_2, \ldots x_m$, ordered by time. Some sequences are intuitively "easier" than other sequences. For example, If $x_1 = x_2 = \cdots = x_m = X$, then any search tree with X at the root can achieve constant time access per search.

We will investigate some possible properties of BST algorithms that guarantee certain access time bounds for specific input sequences.

5.2.4 Sequential Access Property

A BST algorithm has the *Sequential Access Property* if the search sequence $\{1, 2 \dots n\}$ takes an amortized $O(1)$ time per operation.

This property seems easy to achieve, as it constitutes performing an in-order tree traversal in $O(n)$ time. It is slightly more complicated in this model of computation, as all searches must start at the root. However, starting at the last key can be simulated by rotating to the root, which we will not prove in this lecture.

5.2.5 Dynamic Finger Property

A BST algorithm has the *Dynamic Finger Property* if, for any sequence of operation $x_1, x_2, \dots x_m$, the amortized access time for x_k is $O(|(x_k - x_{k-1})|)$.

This is a generalization of the Sequential Access Property. The Dynamic Finger Property tells me that as long as my queries remain close in space, the time needed will be small.

The Dynamic Finger Property can be achieved by a BST with some difficulty. In a more general pointer machine model, this is easy to achieve with a level-linked tree, a BST with pointers between adjacent nodes at every level.

5.2.6 Static Optimality/Entropy Bound

A BST algorithm is *Statically Optimal* if, given an input sequence where element k appears a p_k fraction of the time, the amortized access time per search is

$$O\left(\sum_{k=1}^{n} p_k \log \frac{1}{p_k}\right)$$

This is the information theoretic lower bound for the amortized access time for a static tree, hence the name static optimality.

5.2.7 Working Set Property

A BST algorithm has the *Working Set Property* if for a given search for x_i, if t_i distinct elements were accessed since the last access of x_i, the search takes an amortized $O(\log t_i)$.

The Working Set Property implies Static Optimality (although we will not prove it). If a few items are accessed often in some subsequence, the Working Set Property guarantees that these accesses are fast.

The Working Set Property says that keys accessed recently are easy to access again. The Dynamic Finger Property says that keys close in space to a key recently accessed are also easy to access. The following property will combine these.

5.2.8 Unified Property

A BST algorithm has the *Unified Property* [40] if, given that $t_{i,j}$ unique keys were accessed between x_i and x_j, then search x_j costs an amortized

$$O(\log(\min_{i<j}(|x_i - x_j| + t_{i,j} + 2)))$$

This is a generalization of both the Working Set Property and the Dynamic Finger Property, and implies both. If there is a key that was accessed somewhat recently and is somewhat close in space, this access will be cheap.

It is unknown whether or not there is any BST that achieves the Unified Property. This property can be achieved by a pointer machine data structure [40]. The best upper bound known is a BST that achieves an additive $O(\log \log n)$ factor on top of every operation.

5.2.9 Dynamic Optimality

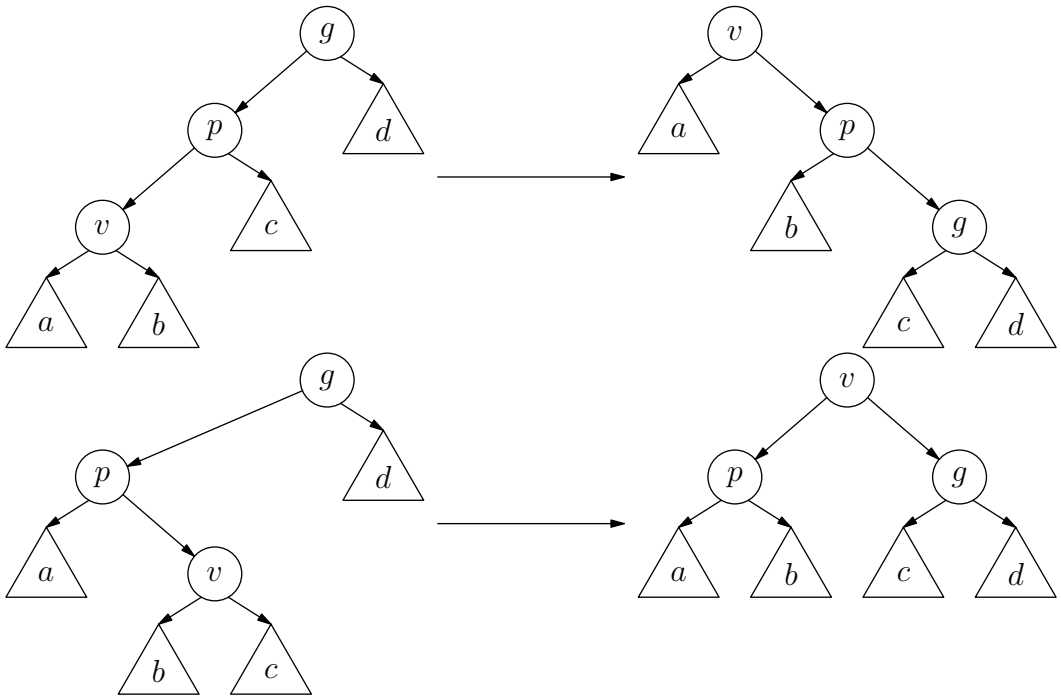
A BST algorithm is *Dynamically Optimal* if the total cost of a sequence of searches is within a multiplicative $O(1)$ factor of the optimal BST algorithm for that sequence. The optimal is the offline optimal, the min cost over all BST algorithms once a particular sequence is known. Dynamic Optimality implies that a particular online BST algorithm can achieve a constant factor approximation of the cost of the offline optimal algorithm.

Open Questions: Is there an online dynamically optimal BST algorithm? Is there an online dynamically optimal pointer machine algorithm (either competitive with the best offline BST or offline pointer machine?)

Answer: We don't know any of the above. The best known is an online $O(\log \log n)$-competitive algorithm, which we will study next lecture. However, there are some candidates that are conjectured to be dynamically optimal, although a proof is not known.

5.3 Splay Trees

Splay trees were introduced by Sleator and Tarjan [41] in 1985. In a splay tree, a search for key x will start at the root and go down the tree until the key x is reached. Then, x is moved to the root using the following two "splay" operations, the *zig-zig* and the *zig-zag*:



In the zig-zig, we rotate y and then rotate x. In the zig-zag step, we rotate x twice. Splay trees differ from the "Move-to-root" algorithm because of the zig-zig operation, instead of simply rotating x twice. We perform these operations until x is either at the root of the tree, or it is a child of the root, in which case we perform one single rotation on x.

5.3.1 Analytic properties

Splay trees have been shown to have some of the properties discussed above. In particular, splay trees:

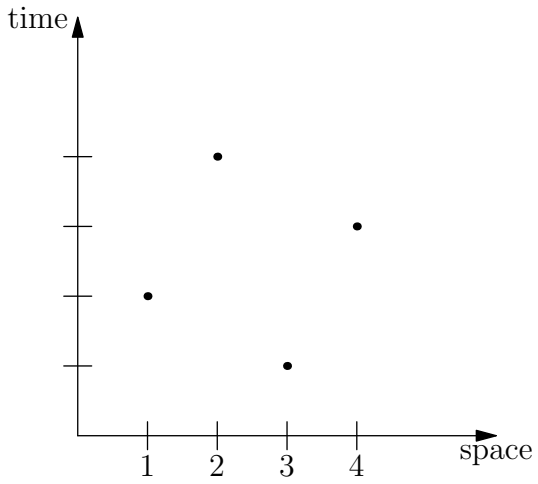
- are statically optimal, [41]
- have the Working Set Property, [41]
- have the Dynamic Finger Property. This was proven in 2000 by Cole et. al. [37][38]

It is not known whether splay trees are dynamically optimal, or even if they satisfy the Unified Property. They were conjectured to be dynamically optimal in the original paper [41], but this conjecture is still open.

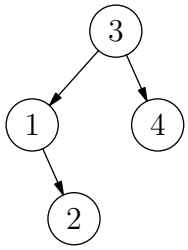
5.4 Geometric View

It turns out that there is a very neat geometric view of binary search tree algorithms, from [39]. Suppose on the key set $\{1, 2, \ldots, n\}$, we have an access sequence $x_1, \ldots, x_m$. We can represent this using the points (x_i, i).

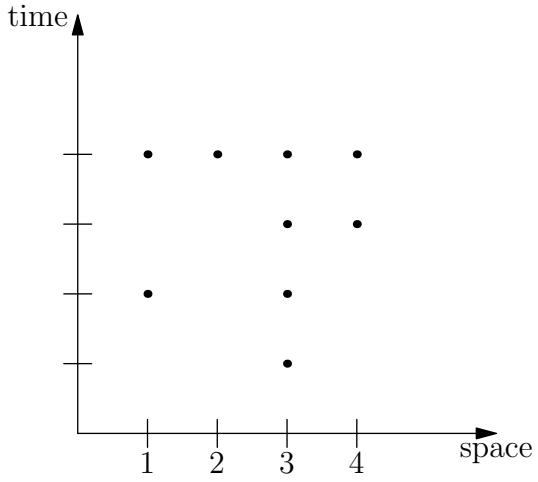
Example: If our key set is $\{1, 2, 3, 4\}$ and our access sequence is $3, 1, 4, 2$, we would draw this picture:



We could perform the access with this static binary tree:



In the first access, we only have to touch the 3. In the second access, we have to touch the 3 and the 1. In the third access, we have to touch the 3 and the 4. In the fourth access, we have to touch the 3, the 1, and the 2. We also represent these touches geometrically. If we touch item x and time t, then we draw a point at (x, t).



In general, we can represent a BST algorithm for a given input sequence by drawing a point for each item that gets touched. We consider the cost of the algorithm to be the total number of points; when running the BST, there is never any reason to touch a node more than a constant number of times per search. Thus, the number of points is within a constant factor of the cost of the BST.

Definition. We say that a point set is *arborally satisfied* if the following property holds: for any pair of points that do not both lie on the same horizontal or vertical line, there exists a third point which lie in the rectangle spanned by the first two points (either inside or on the boundary).

Note that the point set for the BST given in the example is arborally satisfied. In general, the following holds:

Theorem. A point set containing the points (x_i, i) is arborally satisfied if and only if it corresponds to a valid BST for the input sequence $x_1, \ldots, x_m$.

Corollary. The optimum binary search tree execution is equivalent to the smallest arborally satisfied set containing the input.

OPEN: What is the computational complexity of finding the smallest arborally satisfied set? Is there an $O(1)$-approximation?

Proof. First, we prove that the point set for any valid BST algorithm is arborally satisfied. Consider points (x, i) and (y, j), where x is touched at time i and y is touched at time j. Assume by symmetry that $x < y$ and $i < j$. We need to show that there exists a third point in the rectangle with corners as (x, i) and (y, j). Also let $\text{lca}_t(a, b)$ denote the least common ancestor of nodes a and b right before time t. We have a few cases:

- If $\text{lca}_i(x, y) \neq x$, then we can use the point $(\text{lca}_i(x, y), i)$, since $\text{lca}_i(x, y)$ must have been touched if x was.
- If $\text{lca}_j(x, y) \neq y$, then we can use the point $(\text{lca}_j(x, y), j)$.
- If neither of the above two cases hold, then we must have x be an ancestor of y right before time i and y be an ancestor of x right before time j. Then at some time k ($i \leq k < j$), y must have been rotated above x, so we can use the point (y, k).

Next, we show the other direction: given an arborally satisfied point set, we can construct a valid BST corresponding to that point set. Now we will organize our BST into a treap which is organized in heap-order by next-touch-time. Note that next-touch-time has ties and is thus not uniquely defined, but this isn't a problem as long as we pick a way to break ties. When we reach time i, the nodes touched form a connected subtree at the top, by the heap ordering property. We can now take this subtree, assign new next-touch-times and rearrange into a new local treap. Now, if a pair of nodes, x and y, stradle the boundary between the touched and untouched part of the treap, then if y is to be touched sooner than x then $(x, now) \rightarrow (y, next - touch(y))$ is an unsatisfied rectangle because the leftmost such point would be the right child of x, not y. ■

5.4.1 Greedy Algorithm

There is a simple greedy algorithm to construct arborally satisfiable sets. We consider the point set one row at a time. Now add any points to that row that would be necessary to make the current subset satisfiable. This is repeated until all rows of points are satisfied. It is conjectured that this greedy algorithm is $O(Opt)$ or event $Opt + O(m)$.

Theorem. The online arborally satisfiable set algorithm implies an online BST algorithm with $O(1)$ slowdown.

Corollary. If the greedy algorithm is $O(Opt)$, then we have an algorithm for dynamic optimality.

Proof. First, store the touched nodes from an access in a split tree. Split trees can move a node to the root, and then delete that node leaving two trees in amortized $O(1)$ time. This allows us to perform the reordering and separation based on touched nodes for all n nodes in only $O(n)$ time. Now we can essentially construct a BST which is essentially a treap of split trees ordered by the previously touched node. These trees allow us to efficently touch the predecessor and successor nodes in the parent tree when touching a node in the split tree. Thus we are able to simulate the decisions from the arborally satisfiable set algorithm with only only a constant factor slowdown.

Lecture 6

Dynamic optimality 2

Scribes: Aakanksha Sarda (2012), David Field (2012), Leonardo Urbina (2012), Prasant Gopal (2010), Hui Tang (2007), Mike Ebersol (2005)

6.1 Overview

In the last lecture we discussed representing executions of binary search trees as point sets. A BST execution is represented as the point set of all nodes touched during the execution of the algorithm. A point set represents a valid BST execution if and only if it is arborally satistified. A point set is arborally satisfied iff any rectange spanned by two points not on a horizonal or vertical line contains another point. This point set representation of binary searches suggested an offline greedy algorithm for generating valid point sets from a set of queries, by adding the points required to make the point set arborally satisfied row by row. This greedy algorithm is conjectured to be $O(\text{optimal})$. The offline greedy algorithm can be simulated online.

In this lecture we are going to focus on lower bounds. We will be using the method of independent rectangles to establish lower bounds. We will discuss three applications of the method of independent rectanges Wilber 1, Wilber 2, and signed greedy. The signed greedy algorithm is almost the same as greedy, but it is a lower bound. It is conjectured, but not proven that these two algorithms are within a constant factor of each other. We will also discuss tango trees which are $O(\lg \lg n)$-competitive.

6.2 Independent Rectangle Bounds

Now we establish lower bounds on the performance of binary search trees using the method of independent rectangles. A pair of rectangles are called independent if the rectangles are not arborally satisfied and no corner of either rectangle is strictly inside the other rectangle. The Independent Rectangle bounds state that the number of additional points required to make a point set arborally satisfied is greater than or equal to the maximum number of independent rectangles in that point

set. Due to the equivalence between arborally satisfied point sets and binary search trees, this puts a lower bound on the performance of binary search trees.

Given any two points a and b, there is a unique rectangle with a as one of its vertices, and b as the opposite vertex. Let us refer to this rectangle as rectangle ab. If the line $\overline{ab}$ has positive slope, let us call this rectangle a S-rectangle. S stands for the slash, '/', formed by the line. If the line $\overline{ab}$ has negative slope, let us call this rectangle a B-rectangle. B stands for the backslash, '\', formed by the line.

Definition 4. *A pair of rectangles ab and cd are called independent if the rectangles are not arborally satisfied and no corner of either rectangle is strictly inside of the other rectangle.*

Definition 5. *Given a point set P, let OPT be the cardinality of the smallest arborally satisfied superset of P. (OPT may also be used to refer to the minimal superset)*

Definition 6. *Given a point set P, let $MAXIND_{\square}$ be the cardinality of the largest set of independent rectangles that is a subset of the rectangles defined by P. Let $MAXIND_{\boxslash}$ and $MAXIND_{\boxbslash}$ be defined in the same way but for independent S-rectangles and independent B-rectangles, respectively.*

Theorem 7. *Given a point set S, $OPT \geq |input| + 1/2\, MAXIND_{\square}$*

From now on, all lemmas, definitions, and theorems will be stated using S-rectangles; but they can symetrically applied to B-rectangles as well.

Definition 8 (S-satisfied)**.** *We call a point set S-satisfied if all S-rectangles in it have another point in them.*

Definition 9 ($OPT_{\boxslash}$)**.** *Given a point set P, $OPT_{\boxslash}$=the cardinality of the smallest S-satisfied superset of P.*

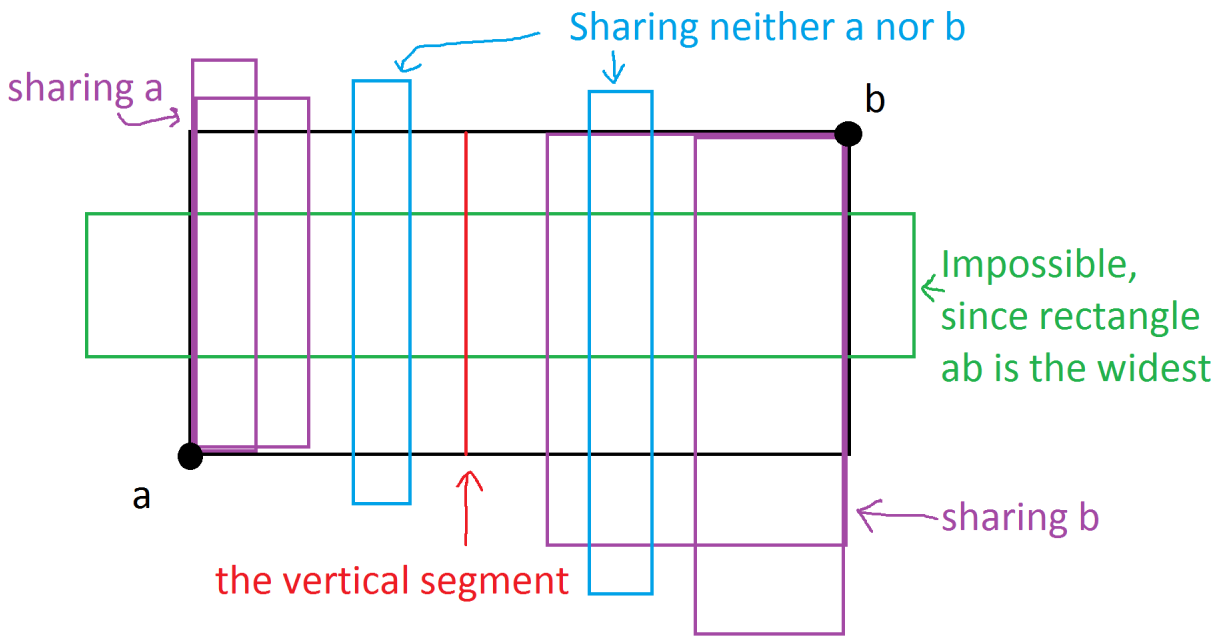
Lemma 10. $OPT_{\boxslash} \geq |input| + MAXIND_{\boxslash}$

6.2.1 Proof of Lemma

Proof. Assume the points in the input point set P have distinct x and y. If this is not the case, the input can be skewed so that the input x and y are all distinct. Consider the largest set of independent S-rectangles that is a subset of the S-rectangles defined by P. (or any one of the largest sets if it is not unique) We will show by a charging argument that $OPT_{\boxslash} \geq |\text{input}| + MAXIND_{\boxslash}$.

Existance of vertical segment: Consider the widest rectangle in the independent set, shown in the figure below. Call the point at its lower left vertex a and the point at its upper right vertex b. There must exist a vertical segment going from the botton edge to the top of edge of this rectangle which has no points in other rectangles. This is because all rectangles who's interiors intersect the widest rectangle's interior must share vertex a with the widest rectangle, share vertex b with the widest rectangle, or share neither. The sharing a rectangles must be left of the sharing b rectangles by independence (they cannot just share edges due to the distinct x coordinates). The sharing neithers cannot have one vertical edge intersecting a particular a or b sharing rectangle, and one not, because the rectangles must be independent. Since the a and b sharing rectangles

cannot intersect, and the sharing neither rectangles must fit in between the sharing b and sharing a rectangles, there will be room for a vertical segment v.



Selection of p and q: Take p to be the topmost rightmost point from $OPT_{\boxslash}$ in S-rectangle ab to the left of the vertical line segment. Let q be the bottommost leftmost point from $OPT_{\boxslash}$ in S-rectangle ab to the right of the line segment and not below p.
Assume these points are not horizontal. If there was any point from $OPT_{\boxslash}$ in S-rectangle pq then that point would either to the left of the vertical segment and farther right and up than p or to the right of the vertical segment and father down and right than q, which cannot be the case by the definition of p and q. Then S-rectangle pq is not satisfied, which is a contradiction. Thus segment pq is horizontal.

Charging: Remove the S-rectangle ab from the set of independent S-rectangles and charge it to the horizontal line containing p and q. (Note that a and b are not being removed, just the S-rectange ab). The pair of points p and q will not be charged together again since the vertical segment that intersects the horizontal line between them in not intersected or contained by any other rectangles, so no other rectangle contains segment pq. Any set of n charges to a horizonal line must charge $n+1$ distinct points, since no pair of points can be charged twice, and all pairs of points charged must be consequtive points on the line. At most 1 of these points can be from the input. Thus the number of added points on a horizontal line is greater than or equal to the number of charges to that line. Thus $OPT_{\boxslash} \geq |\text{input}| + MAXIND_{\boxslash}$ as desired. □

6.2.2 Proof of Theorem

Proof. Using the lemma:

$$\begin{aligned} OPT &\geq \max(OPT_{\boxslash}, OPT_{\boxbslash}) && (6.1)\\ &\geq 1/2(OPT_{\boxslash} + OPT_{\boxbslash}) && (6.2)\\ &\geq |input| + 1/2\, MAXIND_{\boxslash} + 1/2\, MAXIND_{\boxbslash} && (6.3)\\ &\geq input| + 1/2\, MAXIND_{\square} && (6.4) \end{aligned}$$

□

6.3 Lower Bounds

6.3.1 Wilber's second lower bound [42]:

Wilber's second bound can be introduced as follows. Given the input (access) point set:

1. For each point p:
   (a) Look at all of the orthogonally visible points below p
   (b) Count the number of alternations between left/right of p
2. Sum over all p

Proof: Add an independent rectangle of different sign for each alternation.

OPEN: OPT = ΘWilber2

6.3.2 Key-independent Optimality [43]:

Suppose now that that the key values are "meaningless". In this case we could just permute them randomly. With this mind we can make the following claim:

$$\mathbf{E}[\text{OPT}] = \text{working-set bound}$$

Hence, splay trees are key-independent optimal.

Sketch of Proof: The expected number of changes to to max in random permutation is

$$\mathbf{E}[\text{Wilber2}(x_i)] = \Theta(\log t_i)$$

Wilber's First Lower Bound [42]: We can arrive at Wilber's first lower bound as follows. Fix a lower-bound tree P having the input as its keys. Then:

1. For each node y of P: Count the number of alternations in the access sequence $x_1, x_2, \ldots, x_n$ between accesses in left and right subtrees of y (ignoring accesses to y or outside of y's subtree.
2. Sum over all y

Proof: Add an independent rectangle for each alternation.

Example: bit-reversal sequence Consider the sequence of numbers that results from the list of all non-negative integers, and reading their binary representations backwards. Namely, start with

$$0, 1, 2, 3, 4, 5, 6, 7, \ldots$$

Which written in binary is,

$$000_1, 001_2, 010_2, 011_2, 100_2, 101_2, 110_2, 111_2$$

Reversing these yields:

$$000_1, 100_2, 010_2, 110_2, 001_2, 101_2, 011_2, 111_2$$

In other words

$$0, 4, 2, 6, 1, 5, 3, 7$$

If we use this as the access sequence in a perfect binary tree we see that the number of alternations at y equals the size of the subtree rooted at y. From this it follows that:

$$\begin{aligned} & \text{Wilber1} = \Theta(n \log n) \\ \Rightarrow \quad & \text{OPT} = \Theta(n \log n) \end{aligned}$$

OPEN: Is it true that for every access sequence there exist a tree P such that

$$\text{OPT} = \Theta(\text{Wilber1})$$

6.4 Tango Trees

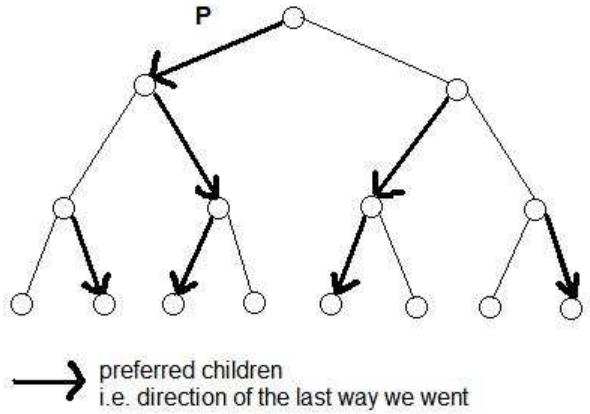
Tango trees [DHIP04] are an $O(\lg \lg n)$-competitive BST. They represent an important step forward from the previous competitive ratio of $O(\lg n)$, which is achieved by standard balanced trees. The running time of Tango trees is $O(\lg \lg n)$ higher than Wilber's first bound, so we also obtain a bound on how close Wilber is to OPT. It is easy to see that if the lower bound tree is fixed without knowing the sequence (as any online algorithm must do), Wilber's first bound can be $\Omega(\lg \lg n)$ away from OPT, so one cannot achieve a better bound using this technique.

To achieve this improved performance, we divide a BST up into smaller auxiliary trees, which are balanced trees of size $O(\lg n)$. If we must operate on k auxiliary trees, we can achieve $O(k \lg \lg n)$ time. We will achieve $k =$ (1+ the increase in the Wilber bound given by the current access), from which the competitiveness follows.

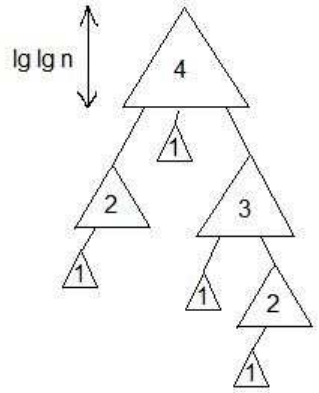
Let us again take a perfect binary tree P and select a node y in P. We define the preferred child of y to be the root of the subtree with the most recent access i.e. the preferred child is the left one

iff the last access under y was to the left subtree. As in the figure below, each node has at most one preferred child, and "preferred paths" through the tree are formed.

If y has no children or its children have not been accessed, it has no preferred child. An interleave is equivalent to changing the preferred child of a node, which means that the Wilber bound is the total number of times a preferred child was changed.



Now we define a preferred path as a chain of preferred child pointers. We store each preferred path from P in a balanced auxiliary tree that is conceptually separate from T, such that the leaves link to the roots of the auxiliary trees of "children" paths.



Because the height of P is $\lg n$, each auxiliary tree will store $\leq \lg n$ nodes. A search on an auxiliary tree will therefore take $O(\lg \lg n)$ time, so the search cost for k auxiliary trees $= O(k \lg \lg n)$.

A preferred path is not stored by depth (that would be impossible in the BST model), but in the sorted order of the keys.

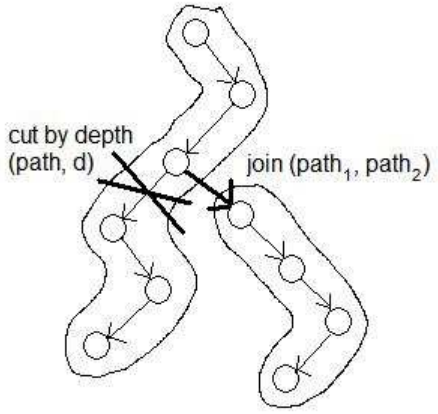
6.4.1 Searching Tango trees

To search this data structure for node x, we start at the root node of the topmost auxiliary tree (which contains the root of P). We then traverse the tree looking for x. It is likely that we will jump

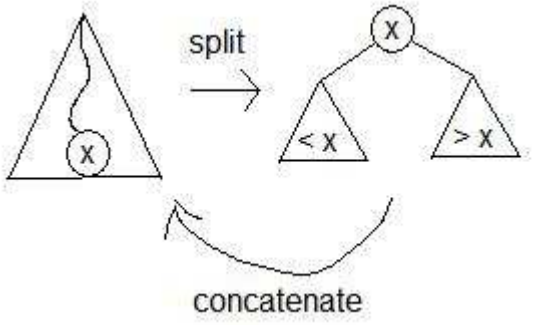
between several auxiliary trees – say we visit k trees. We search each auxiliary tree in $O(\lg \lg n)$ time, meaning our entire search takes place in $O(k \lg \lg n)$ time. This assumes that we can update our data structure as fast as we can search, because we will be forced to change $k - 1$ preferred children (except for startup costs if a node has no preferred children).

6.4.2 Balancing Auxiliary Trees

The auxiliary trees must be updated whenever preferred paths change. When a preferred path changes, we must cut the path from a certain point down, and insert another preferred path there.



We note that cutting and joining resemble the *split* and *concatenate* operations in balanced BST's. However, because auxiliary trees are ordered by key rather than depth, the operations are slightly more complex than the typical *split* and *concatenate*.



Luckily, we note that a *depth* $> d$ corresponds to an interval of keys - from min(path below depth d) to max (path below depth d). Thus, cutting from a point down becomes equivalent to cutting a segment in the sorted order of the keys. In this way, we can change preferred paths by cutting a subtree out of an auxiliary tree using two split operations and adding a subtree using a concatenate operation. To perform these cut and join operations, we label the roots of the path subtrees with a special color and pretend it's not there. This means we only need to worry about doing the *split* and *concatenate* on a subtree of height $\lg n$ rather than trying to figure out what to do with all the

auxiliary subtrees hanging off the path subtree we are interested in. We know that balanced BSTs can support *split* and *concatenate* in $O(\lg size)$ time, meaning they all operate in $O(\lg \lg n)$ time. Thus, we remain $O(\lg \lg n)$-competitive.

6.5 Signed greedy algorithm

This section will build on the Greedy algorithm from last lecture, to present a modified "Signed Greedy" algorithm whose run-time serves as a lower bound.

6.5.1 Algorithm

Conduct the greedy-algorithm sweep as before, considering one point row at a time. However, for signed-greedy, we consider only unsatisfied ◪-rectangles (or only ◧-rectangles) when deciding whether to add points.

Thus, only the ◪-rectangles (or only the ◧-rectangles) will be satisfied. Also, similarly to the original greedy algorithm, every added point will correspond to an independent ◪-rectangle.

Definition 11. *Define $OPT_{\boxtimes}$ to be the smallest union of the ◪-satisfying and ◧-satisfying supersets of the points.*

6.5.2 Signed Greedy provides a lower bound

Theorem 12. $\max\{$◪*-greedy*, ◧*-greedy*$\} = \theta($*biggest independent-rectangle LB*$)$

Proof.

$$\begin{aligned}
\mathrm{OPT}_{\boxtimes} &\geq |input| + \frac{1}{2}\max(\text{number of independent rectangles}) && (6.5)\\
&\geq \frac{1}{2}\max(\text{◪-greedy}, \text{◧-greedy}) && (6.6)\\
&\geq \frac{1}{2}\max(\mathrm{OPT}_{◪}, \mathrm{OPT}_{◧}) && (6.7)\\
&\geq \frac{1}{4}(\mathrm{OPT}_{◪} + \mathrm{OPT}_{◧}) && (6.8)\\
&\geq \frac{1}{4}\mathrm{OPT}_{\boxtimes} && (6.9)
\end{aligned}$$

□

Thus, a constant factor 'sandwich' is created, and all the expressions (1) to (5) must be within a constant factor of each other. In particular, (1) - which is the independent rectangle lower bound - and (2) must be within a constant factor of each other, and the theorem follows.

6.5.3 Greedy vs. ◪-greedy

Greedy corresponds to a valid BST, and provides and upper bound on the run-time of OPT. Signed-greedy ($\text{OPT}_{\boxtimes}$) is not a valid BST, but it provides a lower bound on the run-time of every valid BST. The key difference between the point-sets obtained from Greedy and from Signed-Greedy is that Signed-Greedy fails to take into account the interactions between the ◪-rectangles and the ◹-rectangles, i.e. the points added to satisfy ◪-rectangles may introduce new unsatisfied ◹-rectangles or vice versa.

Project idea: Compare upper-bound and lower-bound for many point sets.

Lecture 7

Memory hierarchy 1

Scribes: Claudio A Andreoni (2012), Sebastien Dabdoub (2012), Usman Masood (2012), Eric Liu (2010), Aditya Rathnam (2007)

7.1 Memory Hierarchies and Models of Them

So far in class, we have worked with models of computation like the word RAM or cell probe models. These models account for communication with memory one word at a time: if we need to read 10 words, it costs 10 units.

On modern computers, this is virtually never the case. Modern computers have a memory hierarchy to attempt to speed up memory operations. The typical levels in the memory hiearchy are:

Memory Level	Size	Response Time
CPU registers	$\approx$ 100B	$\approx$ 0.5ns
L1 Cache	$\approx$ 64KB	$\approx$ 1ns
L2 Cache	$\approx$ 1MB	$\approx$ 10ns
Main Memory	$\approx$ 2GB	$\approx$ 150ns
Hard Disk	$\approx$ 1TB	$\approx$ 10ms

It is clear that the fastest memory levels are substantially smaller than the slowest ones. Generally, each level has a direct connection to only the level directly below it in the hierarchy. In addition, the faster, smaller levels are substantially more expensive to produce, so do not expect 1GB of register space any time soon. Many of the levels communicate in blocks. For example, asking Main Memory to read one integer will typically also transmit a "block" of nearby data. So processing the other block members requires no additional memory transfers. This issue is exacerbated when communicating with the disk: the 10ms is dominated by the time needed to find the data (move the read head over the disk). Modern disks are circular, spinning at 7200rpm, so once the head is in position, reading all of the data on that "ring" is much faster.

This speaks to a need for algorithms that are designed to deal with "blocks" of data. Algorithms that properly take advantage of the memory hiearchy will be much faster in practice; and memory models which correctly describe the hiearchy will be more useful for analysis. We will see some

fundamental models with some associated results today.

7.2 External Memory Model

The external memory model was introduced by Aggarwal and Vitter in 1988 [44]; it is also called the "I/O Model" or the "Disk Access Model" (DAM). The external memory model simplifies the memory hierachy to just two levels. The CPU is connected to a fast cache of size M; this cache in turn is connected to a much slower disk of effectively infinite size. Both cache and disk are divided into blocks of size B, so there are $\frac{M}{B}$ blocks in the cache. Transfering one block from cache to disk (or vice versa) costs 1 unit. Memory operations on blocks resident in the cache are free. Thus, the natural goal is to minimize the number of transfers between cache and disk.

Clearly any algorithm from say the word RAM model with running time $T(N)$ requires no worse than $T(N)$ memory transfers in the external memory model (at most one memory transfer per operation). The lower bound, which is usually harder to obtain, is $\frac{T(N)}{B}$, where we take perfect advantage of cache locality; i.e., each block is only read/written a constant number of times.

Note that, the external memory model is a good first approximation to the slowest connection in the memory hiearchy. For a large database, "cache" could be system RAM and "disk" could be the hard disk. For a small simulation, "cache" might be L2 and "disk" could be system RAM.

7.2.1 Scanning

Scanning N items trivially costs $O(\lceil \frac{N}{B} \rceil)$ memory transfers. The ceiling is important, because if $N = o(B)$ then we end up reading more items than necessary.

7.2.2 Searching

Searching is accomplished with a B-Tree using a branching factor that is $\Theta(B)$. In practice, we would want it to be exactly $B + 1$ so that a single node fits in one memory block and we always have a branching factor ≥ 2. Insert, delete, and predecessor/successor searches are then handled with $O(\log_{B+1} N)$ memory transfers. This will require $O(\log N)$ time in the comparision[1] model; so there is an improvement by a factor of $O(\log B)$.

The $O(\log_{B+1} N)$ bound is in fact **optimal** for searches; we can see this from an information theoretic argument. We want to figure out where our items fits amongst all the N items. There are $N + 1$ positions where our item can fit and we need $\Theta(\log N + 1)$ bits of information to specify one of those positions. Each read from cache (one block) tells us where the item fits among B items, yielding $O(\log(B + 1))$ bits of information. Thus we need at least $\Theta(\frac{\log N}{\log(B+1)})$ or $\Omega(\log_{B+1} N)$ memory transfers to reveal all $\Theta(\log(N + 1))$ bits.

For insert/delete, however, this bound is **not optimal**.

[1] This is not the standard comparison model; here we mean that the only permissible operation on elements is to compare them pairwise.

7.2.3 Sorting

In the word RAM model, a B-Tree can sort in optimal time: just insert all elements and then do an inorder traversal. However, the same technique yields $O(N \log_{B+1} N)$ (amortized) memory transfers in the external memory model, which is **not optimal**.

An optimal algorithm is a $\frac{M}{B}$-way version of mergesort. It obtains performance by solving subproblems that fit in cache, leading to a total of $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ memory transfers. This bound is actually **optimal** in the comparison model [44].

7.2.4 Permutation

The permutation problem is: given N elements in some order and a new ordering, rearrange the elements to appear in the new order. Naively, this takes $O(N)$ operations: just swap each element into its new position. It may be faster to assign each item a key equal to its permutation ordering and then apply the aforementioned optimal sort algorithm. This gives us a bound of $O(\min\{N, \frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}\})$ (amortized). Note that this result only holds in the "indivisible model," where words cannot be cut up and re-packed into other words.

7.2.5 Buffer Trees

Buffer trees are essentially a dynamic version of sorting. Buffer trees achieve $O(\frac{1}{B} \log_{\frac{M}{B}} \frac{N}{B})$ (amortized) memory transfers per operation. Bound has to be amortized because it is typically $o(1)$. They also achieve the optimal sorting bound if all elements are inserted then the minimum deleted sequentially. The operations are batched updates (insert, delete-min) and delayed queries. If we do a query, we don't get the answer right then; it's delayed. There's a new operation called flush, which returns answers to all unanswered queries so far in $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ time. So if we do $O(N)$ operations and then do flush, it doesn't take any extra time. Find-min queries are free; the min item is maintained all the time. They can be used as efficient external memory priority queues.

7.3 Cache Oblivious Model

The cache-oblivious model is a variation of the external-memory model introduced by Frigo, Leiserson, Prokop, and Ramachandran in 1999 [53, 54]. In this model, the algorithm does not know the block size, B, or the total memory size, M. In particular, our algorithms will look like normal RAM algorithms, but we will analyze them differently.

For modeling assumptions, we will assume that the caching is done automatically. We will also assume that the caching mechanism is optimal (in the offline sense). In practice, this can be achieved with either LRU (Least Recently Used) or FIFO (First In First Out) since they are $O(1)$-competitive with the offline optimal algorithm if they have a cache with twice the size. Since none of our algorithms change by replacing the cache size M with $2M$, this does not change our bounds.

Good algorithms for this model give us good algorithms for all values of B and M. They are especially useful for multi-level caches and for caches with changing values of B and M.

7.3.1 Scanning

The bound is identical to external memory: $O(\lceil \frac{N}{B} \rceil)$ memory transfers. We can use the same algorithm as before, since we only depend on B in the analysis.

7.3.2 Search Trees

A cache-oblivious variant of the B-tree [47, 48, 52] provides the `INSERT`, `DELETE`, and `SEARCH` operations with $O\log_{B+1} N$ (amortized) memory transfers, as in the external-memory model. The latter half of this lecture concentrates on cache-oblivious B-Trees.

7.3.3 Sorting

As in the external-memory model, sorting N elements can be performed cache-obliviously using $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ memory transfers [53, 50]. This will be done in the next lecture.

7.3.4 Permuting

The min{} is no longer possible[53, 50], since that depends on knowing M and B. Both component bounds (sorting or linear) from the external memory model are still valid, we just don't which gives the minimum.

7.3.5 Priority Queues

A priority queue can be implemented that executes the `INSERT`, `DELETE`, and `DELETE-MIN` operations in $O\left(\frac{1}{B} \log_{\frac{M}{B}} \frac{N}{B}\right)$ (amortized) memory transfers [46, 49]. This assumes a tall-cache which states: $M = \Omega(B^{1+\epsilon})$.

7.4 Cache Oblivious B-Trees

Now we will discuss and analyze the data structure leading to the previously stated cache-oblivious search tree result. We will use a data structure that shares many features with the standard B-tree. It will require modification since we do not know B, unlike in the external memory model. To start, we will build a static structure supporting searches in $O(\log_{B+1} N)$ time.

7.4.1 Static Search Trees

First, we will construct a *complete* binary search tree over all N elements. To achieve the $\log_{B+1}$ complexity on memory transfers, the tree will be represented on disk in the *van Emde Boas* layout[54]. The vEB layout is defined recursively. The tree will be split in half by height; the upper subtree has height $\frac{1}{2} \log N$ and it holds $O(\sqrt{N})$ elements. The top subtree in turn links to

$O(\sqrt{N})$ subtrees each with $O(\sqrt{N})$ elements. Each of the $\sqrt{N}+1$ subtrees is in turn divided up according to the vEB layout. On disk, the upper subtree is stored first, with the bottom subtrees laid out sequentially after it. This layout can be generalized to trees where the height is not a power of 2 with a $O(1)$ branching factor (≥ 2)[47].

Note that when the recursion reaches a subtree that is small enough (size less than B), we can stop. Smaller working sets would not gain or lose anything since they would not require any additional memory transfers.

Claim 13. *Performing a search on a search tree in the van Emde Boas layout requires* $O\log_{B+1} N$ *memory transfers.*

Proof. Consider the level of detail that "straddles" B. That is, continue cutting the tree in half until the height of *each subtree* first becomes $\leq \log B$. At this point, the height must also be greater than $\frac{1}{2}\log B$. Note that the size of the subtrees is at least $\sqrt{B}$ and at most B, giving between B and B^2 elements at this "straddling" level.

In the (search) walk from root to leaf, we will access no more than $\frac{\log N}{\frac{1}{2}\log B}$ subtrees[2]. Each subtree at this level then requires at most 2 memory transfers to access[3]. Thus the entire search requires $O(4\log_B N)$ memory transfers.

□

7.4.2 Dynamic Search Trees

Note that the following description is modeled after the work of [48], which is a simplification of [47].

Ordered File Maintenance

First, we will need an additional supporting data structure that solves the Ordered File Maintenance (OFM) problem. For now, treat it as a black box; the details will be given in the next lecture.

The OFM problem involves storing N elements in an array of size $O(N)$ with a specified ordering. Note that this implies gaps of size $O(1)$ are permissible in the structure. The OFM data structure then supports `INSERT` (between two given consecutive items) and `DELETE` operations. It accomplishes each by moving elements in an interval of size $O(\log^2 N)$ (amortized) via $O(1)$ interleaved scans. It follows that the operations require $O\left(\frac{\log^2 N}{B}\right)$ transfers.

OPEN It is conjectured that $O(\log^2 N)$ (amortized) time is optimal.

Back to Search Trees: Linking vEB layout and OFM

Now, we want to also enable fast `SEARCH` operations.

[2]The tree height is $O(\log N)$; the subtree heights are $\Omega(\log B)$.

[3]Although the subtrees each have size at most B, they may not align to cache boundaries; e.g., half in cache-line i and half in line $i+1$.

To do so, we construct an OFM over the N keys, plus the necessary gaps. Then, we create a vEB-arranged tree over the OFM items, adding pointers between the leaves of the tree and the corresponding items in the OFM. The result is illustrated in Figure 7.1.

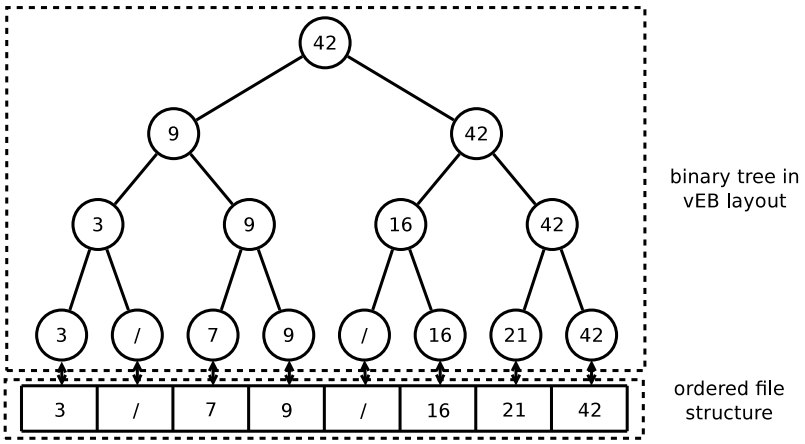


Figure 7.1: A tree with vEB layout to search in an OFM. The / symbol represents gaps.

Notice that the keys are all at the leaves. Inner nodes contain the maximum of the subtrees rooted in their children (gaps count as $-\infty$).

With this structure, `SEARCH` is done by checking each node's left child to decide whether to branch left or right. It takes $O(\log_{B+1} N)$.

`INSERT` requires first searching for the successor of the item to be inserted. Then, we insert the item in the OFM, and finally we update the tree, from the leaves that were changed up to the root. Note that updates must be performed in *post-order*, so that each node can compute the new maximum of its children. The process is illustrated in Figure 7.2. `DELETE` behaves analogously: we search in the tree, delete from OFM, and update the tree.

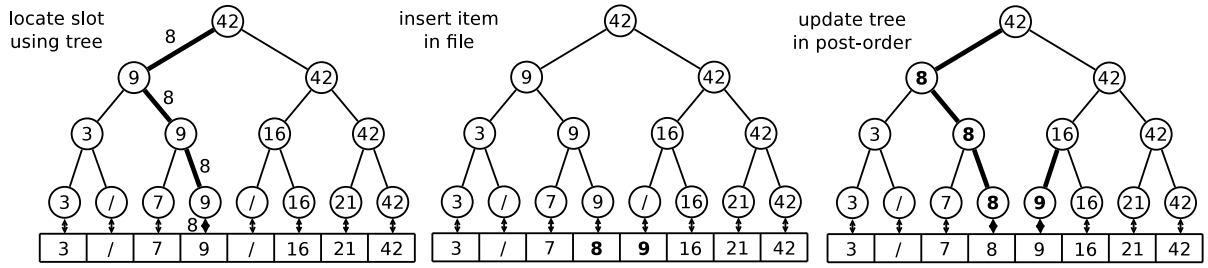


Figure 7.2: Inserting the key '8' in the data structure.

Now, we want to analyze the update operations.

Claim 14. *`INSERT` and `DELETE` take* $O\left(\log_{B+1} N + \frac{\log^2 N}{B}\right)$ *block reads.*

Proof. We notice immediately that finding the item to update takes $O(\log_{B+1} N)$ block reads, and updating the OFM $O\left(\frac{\log^2 N}{B}\right)$.

To see how long it takes to update the tree, we need to consider three separate tree layers independently. For the first layer, we define small vEB triangles to be subtrees in the tree which 1) have size smaller (or equal) to B and 2) are in one of the last two levels of the tree. Also, we are going to consider large vEB triangle, each of them having 1) size larger than B and 2) contain two levels of small vEB triangles. For the second layer, we consider the subtree whose leaves are the roots of the larger vEB blocks. For the third layer, we consider the path from the root of the subtree to the root of the whole tree. This division is illustrated in Figure 7.3.

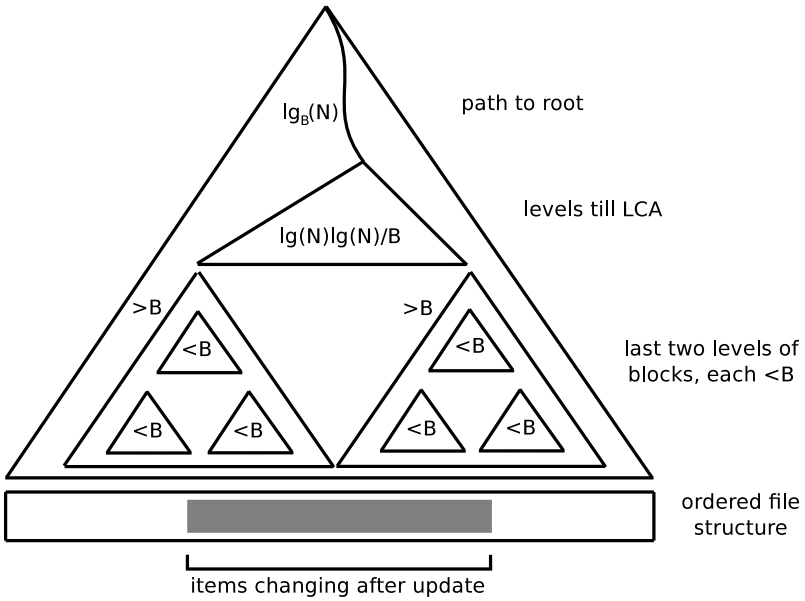


Figure 7.3: The structure division to analyze updates.

We start by noticing that a chunk of size less (or equal) to B can span at most two blocks in memory (when it is between the two). So, reading a small vEB triangle requires at most reading two blocks. Because we are updating the structure in post-order, when we are in the last two levels we only need to keep 4 blocks in cache, two for the child small vEB triangle and two for the parent small vEB triangle being evaluated. As long as we have at least 4 blocks of cache available, we can then update all the lowest two levels with an amortized constant amount of block reads per vEB triangles to update. Since there are $O\left(1 + \frac{\log^2 N}{B}\right)$ such triangles in need for an update in the lowest two levels, updating them takes $O\left(\frac{\log^2 N}{B}\right)$ (amortized) block reads.

Proceeding upwards, we notice that the $O\left(\frac{\log^2 N}{B}\right)$ roots of large vEB triangles are the leaves of a $O\left(\frac{\log^2 N}{B}\right)$-sized tree τ. Notice that the root of τ is the LCA of all items that were modified at the leaves. The size of τ is equal (asymptotically) to the amount of reads we already did for the lowest two layers. Thus, we can afford reading an entire block to update each item of τ without adding to the runtime. So, any simple algorithm can be used to update τ.

Finally, to update the path from the root of τ to the root of the whole structure, we do $O\left(\log B + 1N\right)$ more block reads. Therefore, we can update the entire tree in $O\left(\log_{B+1} N + \frac{\log^2 N}{B}\right)$. □

At this point, we would still like to improve the bound even further. What we would like to

achieve is $O\left(\log_{B+1} N\right)$ (what we can obtain with B-trees), and what we have is too costly if $B = o(\log N \log\log N)$. Fortunately, we can rid ourselves of the $\frac{\log^2 N}{B}$ component using the technique of *indirection*.

Wrapping Up: Adding Indirection

Indirection involves grouping the elements into $\Theta\left(\frac{N}{\log N}\right)$ groups of size $\Theta(\log N)$ elements each. Now we will create the structure we just described over the *minimum* of each $O(\log N)$ group. As a result, the vEB-style BST over the OFM array will act over $\Theta\left(\frac{N}{\log N}\right)$ leaves instad of $\Theta(N)$ leaves.

The vEB storage allows us to search the top structure in $O(\log_{B+1} N)$; we will also have to scan one lower group at cost $O\left(\frac{\log N}{B}\right)$ for a total search cost of $O(\log_{B+1} N)$ memory transfers.

Now `INSERT` and `DELETE` will require us to reform an entire group at a time, but this costs $O\left(\frac{\log N}{B}\right) = O(\log_B N)$ memory transfers, which is sufficiently cheap. As with y-fast trees, we will also want to manage the size of the groups: they should be between 25% and 100% full. Groups that are too small or too full can be merged then split or merged (respectively) as necessary by destroying and/or forming new groups. We will need $\Omega(\log N)$ updates to cause a merge or split. Thus the merge and split costs can be charged to the updates, so their amortized cost is $O(1)$ The minimum element only needs to be updated when a merge or split occurs. So, expensive updates to the vEB structure only occur every $O(\log N)$ updates at cost $O\left(\frac{\log_{B+1} N + \frac{\log^2 N}{B}}{\log N}\right) = O\left(\frac{\log N}{B}\right) = O\left(\log_{B+1} N\right)$. Thus all `SEARCH`, `INSERT`, and `DELETE` operations cost $O(\log_{B+1} N)$.

Lecture 8

Memory hierarchy 2

Scribes: Pedram Razavi (2012), Thomas Georgiou (2012), Tom Morgan (2010)

8.1 From Last Lectures...

In the previous lecture, we discussed the External Memory and Cache Oblivious memory models. We additionally discussed some substantial results from data structures under each model; the most complex of these was the Cache Oblivious B-Tree which can give us $O(\log_B N)$ runtime for insert, delete, and search. In that construction, we used the Ordered File Maintenance (OFM) data structure as a black box to maintain an ordered array with $O(\log^2 N)$ updates. Now we will fill in the details of the OFM. We will also come back to the List Labeling problem which was introduced in Lecture 1 as a black box for the full persistence data structures, specifically for linearizing the version tree.

8.2 Ordered File Maintenance [55] [56]

The OFM problem is to store N elements in an array of size $O(N)$, in a specified order. Additionally, the gaps between elements must be $O(1)$ elements wide, so that scanning k elements costs $O(\lceil \frac{k}{B} \rceil)$ memory transfers. The data structure must support deletion and insertion (between two existing elements). These updates are accomplished by re-arranging a contiguous block of $O(\log^2 N)$ elements using $O(1)$ interleaved scans. Thus the cost in memory transfers is $O(\frac{\log^2 N}{B})$; note that these bounds are amortized.

The OFM structure obtains its performance by guaranteeing that no part of the array becomes too densely or too sparsely populated. When a density threshold is violated, rebalancing (uniformly redistribute elements) occurs. To motivate the discussion, imagine that the array (size $O(N)$) is split into pieces of size $\Theta(\log N)$ each.

Now **imagine** a complete binary tree (depth: $O(\log N) - \Theta(\log \log N) = O(\log N)$) over these subsets. Each tree-node tracks the number of elements and the number of total array slots in its

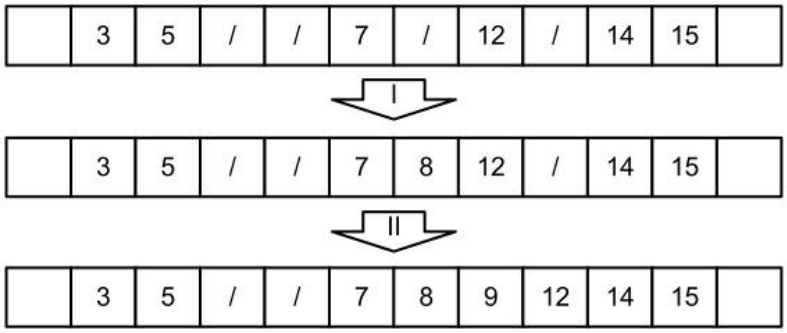


Figure 8.1: Example of two inserts in OFM. "/ " represents an empty gap, each of these gaps are $O(1)$ elements wide. I. insert element 8 after 7 II. insert element 9 after 8. 12 needs to get shifted first in this case.

range. Density of an interval is then defined as the ratio of the number of elements stored in that interval to the number of total array slots in that interval.

8.2.1 Updates

To update element X,

- Update a leaf chunk of size $\Theta(\log N)$ containing X.
- Walk up the tree to the first node **within the density threshold**.
- Uniformly redistribute the elements in this node's interval.

The density of a node, represented by ρ, is a measure to see how much of an interval is occupied, therefore we can define it as the **number of elements/array slots in the interval**. A density of 1 means that all the slots are full (too dense) and a density of 0 means that all slots are empty (too sparse). We want to avoid both of these extreme cases and maintain a density threshold. The density threshold is depth-dependent. The depth, represented by d, is defined such that the tree root has depth 0, and tree leaves have depth $h = \Theta(\log N)$. We require:

$$\rho \geq \frac{1}{2} - \frac{1}{4}\frac{d}{h} \in [\frac{1}{4}, \frac{1}{2}] : \text{ not too sparse}$$
$$\rho \leq \frac{3}{4} + \frac{1}{4}\frac{d}{h} \in [\frac{3}{4}, 1] : \text{ not too dense}$$

Notice that the density constraints are highest at the shallowest node. Intuitively, saving work (i.e., having tight constraints) at the deepest nodes gains relatively little performance because the working sets are comparatively small.

Keep in mind that the BST is never physically constructed. It is only a conceptual tool useful for understanding and analyzing the OFM structure. To perform the tree search operations, we

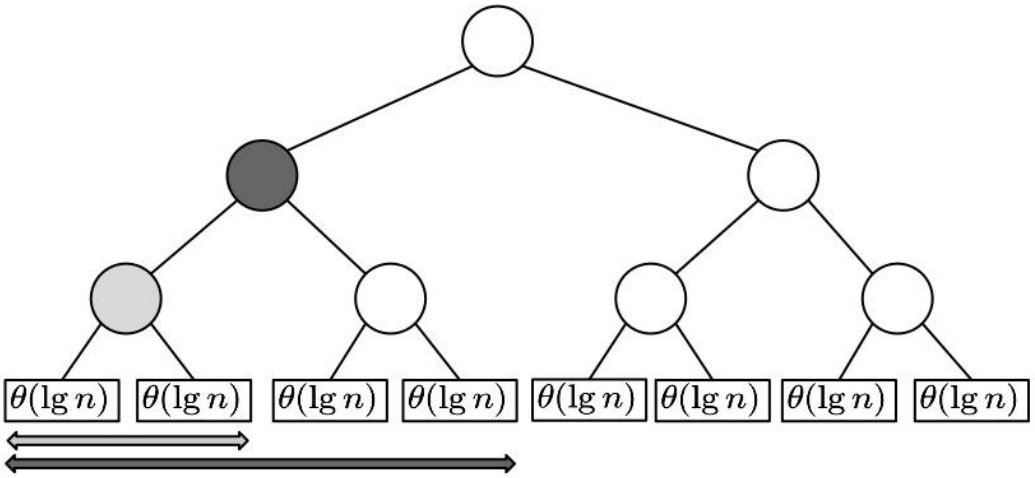


Figure 8.2: Conceptual binary tree showing how to build the intervals. If we insert a new element in a leaf and there is not enough room for it and interval is too dense (showed in light gray), we are going to walk up the tree and look at a bigger interval (showed in dark gray) until we find the right interval. In the end we at most get to the root which means redistribution of the the entire array

can instead examine the binary representation of the left/right edges of a "node" to determine whether this node is a left/right child. Then the scan can proceed left/right accordingly; this is the "interleaved scan."

8.2.2 Analysis

Consider a node D at depth d and its left child and right child at depth $d+1$; say node D is within its threshold, so we need to rebalance its interval. Observe that post-balancing, all descendants of node D will have the same density as node D. Since density constraints become stricter at shallower depths, this means that D's descendants will be (relatively) well-within their thresholds. Specifically, the two children of D require around $\frac{m}{4h}$ or $\Omega(\frac{m}{\log N})$ (where m is the size of the child's interval) for their densities to violate their thresholds. So we can charge the cost of rebalancing the parent interval, which has size $\leq 2m$, to those $\Omega(\frac{m}{\log N})$ updates. Each update is charged $O(\log N)$ times each because there are only $\frac{m}{4 \log N}$ updates (until the child violates a threshold) but m charges in the parent interval. Now we deal with the fact that we have only considered depth d and $d+1$. Each leaf is below $O(\log N)$ ancestors, so each update may be charged at each tree level. This makes for a total of $O(\log^2 N)$ (amortized) memory moves.

This analysis is an amortized result. It can be made worst-case[57], but we did not discuss it in class. It also conjectured that $\Omega \log^2 N$ is an appropriate lower bound for this problem.

8.3 List Labeling

List Labeling is an easier problem than OFM. It involves maintaining explicit (e.g., integer) labels in each node of a linked list. These labels should increase monotonically over the list. The data structure needs to support insertion between two labels and deletion of a label. Depending on the size of the label space, we are aware of a spread of time-bounds:

Label Space	Best Update Query Time	Comments
$(1+\epsilon)N...N \lg N$	$O(\lg^2 N)$	Use OFM (linear space).
$N^{1+\epsilon}...N^{O(1)}$	$\Theta(\lg N)$	OFM method with thresholds at $\approx \frac{1}{\alpha^d}$, 1
2^N	$\Theta(1)$	Simple: have N bits for N items, so insert by bisecting intervals

8.3.1 List Ordered Maintenance

Here we need to maintain an ordered linked list subject to insertion and deletion. Additionally, it should answer "ordering queries": does node x come before node y? $O(1)$ updates and queries are possible using indirection [58].

Consider a "top" summary structure covering a set of "bottom" structures. The bottom structures are each size $\Theta(\log N)$. To perform the labeling, we can use the simple exponential label space list labeling algorithm. The required label space is $2^n = 2^{\log N} = N$, which is affordable. The summary structure is size $O(\frac{N}{\log N})$; each element points to one of the bottom members. For this list, use the $\Theta(\log N)$ list labeling algorithm. The cost is $O(1)$ amortized since it costs $\Theta(\log N)$ per update, but updates only happen once every $\Theta(\log N)$ updates to the bottom members.

Then we can form an implicit label for every bottom member element in the form of (top label, bottom label). This requires $O(\log N)$ bits to store, so we can compare two labels in $O(1)$ time. The key point here is that changing one top label updates numerous bottom labels simultaneously. This is the reason we are able to obtain better performance than the optimal result quoted in the above table. Lastly, worst-case bounds are also obtainable [58].

8.4 Cache-Oblivious Priority Queues [59]

Our objective with the cache-oblivious priority queue is to support all operations with $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ amortized memory transfers per operation in the cache-oblivious model. The data structure is divided up into $\Theta(\log\log n)$ levels of sizes $N, N^{2/3}, N^{4/9}, ..., O(1)$. Each level consists of an up buffer, and many down buffers which store elements moving up and down in the data structure as linked lists. For a level of size $X^{3/2}$, the up buffer is of size $X^{3/2}$ and there are at most $X^{1/2}$ down buffers. Each down buffers has $\Theta(X)$ elements, except for the first one which may be smaller.

We maintain the following invariants on the order of the elements:

1. In a given level, all of the elements in the down buffers must be smaller than all of the elements in the up buffer.

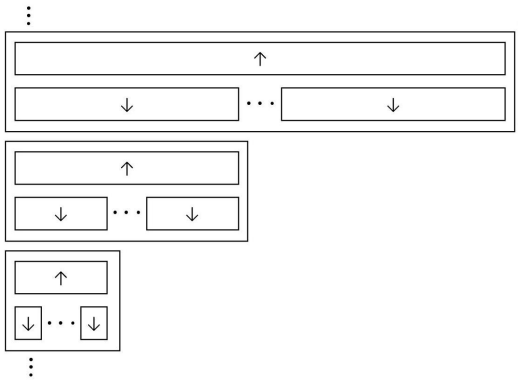


Figure 8.3: A cache-oblivious priority queue

2. In a given level, the down buffers must be in increasing order - all elements in the ith down buffer must be smaller than all of the elements in the $(i+1)$th down buffer.

3. The down buffers are ordered globally - all elements in a down buffer in a smaller level must be smaller than all elements in any down buffer in a larger level.

8.4.1 Find-min

The minimum element in the priority queue should always be in the first down buffer at the smallest level, so we simply find it there.

8.4.2 Delete-min

The minimum element in the priority queue should always be in the first down buffer at the smallest level, so we simply go there and remove it. However, this may cause the down buffers to under flow, in which case we perform a pull operation.

8.4.3 Insertion

Insertion a new element is done in three steps:

1. Append the element to the up buffer at the smallest level.

2. If the new element is smaller than an element in a down buffer, swap it into the down buffers as necessary.

3. If the up buffer overflows, perform a push operation on it.

8.4.4 Push

The objective of pushing is to empty out the up buffer at level X by moving all of its elements into level $X^{3/2}$. To do this, we first sort all of these elements, and then distribute them among the down and up buffers in level $X^{3/2}$. This distribution is performed by scanning through the buffers in increasing order (first down buffers, then up buffer) and inserting our elements where they belong. If one of the down buffers overflows we simply split it in half, and if this causes the number of down buffers to grow too large, we move the last one into the up buffer. If up buffer overflows, we simply perform a push on it.

8.4.5 Pull

Our goal here is to refill the down buffers at level X by bringing down the smallest X elements from level $X^{3/2}$. Once we get these elements, we will sort them together with the up buffer. We will leave the largest elements in the up buffer (the same number of elements as were there before the pull) and then split the remaining elements between the down buffers.

We recall that each of the down buffers at level $X^{3/2}$ except for perhaps the first one should have $\Theta(X)$ elements, so we simply sort the first two down buffers in level $X^{3/2}$ and bring the smallest X elements from them to be put into level X's down buffers. However, if there are fewer than X elements here, then we recursively pull $X^{3/2}$ elements from level $X^{9/4}$.

8.4.6 Analysis

We will first argue that ignoring recursion, the number of memory transfers during a push or pull operation on level $X^{3/2}$ is $O(\frac{X}{B}\log_{M/B}\frac{X}{B})$. First we assume that any level of size less than M will be entirely in cache, so those are free. In the remaining cases, we use the tall cache assumption to say $X^{3/2} > M \geq B^2$, thus $X > B^{4/3}$.

First consider a push at level $X^{3/2}$. Sorting costs $O(\frac{X}{B}\log_{M/B}\frac{X}{B})$, and distribution costs $O(\frac{X}{B} + X^{1/2})$. $\frac{X}{B}$ is the cost for scanning, and $X^{1/2}$ is the cost for loading the first part of each down buffer. If $X \geq B^2$ then this distribution costs $O(X/B)$ which is dwarfed by sorting. There is only one level remaining in which $B^{4/3} \leq X \leq B^2$ and for this level we will simply assume that our ideal cache always holds on to1 line buffer down buffer, so we don't have to pay the $X^{1/2}$ start up cost. We can do this because $X \leq B^2$ so by the tall cache assumption $X^{1/2} \leq B \leq M/B$. Therefore sorting is always the dominant cost. The analysis for pulling is essentially identical.

Thus, X elements involved in pushing or pulling cost $O(\frac{X}{B}\log_{M/B}\frac{X}{B})$. One can prove that each element can only be charged for a constant number of pushes and pulls per level. The intuition is basically that each element goes up and then down. Thus the cost per element is $O(\frac{1}{B}\sum_X \log_{M/B}\frac{X}{B})$. Since X grows doubly exponentially, the $\log_{M/B}\frac{X}{B}$ term grows exponentially. Therefore, the last term, $\log_{M/B}\frac{N}{B}$ dominates the sum and we are left with $O(\frac{1}{B}\log_{M/B}\frac{N}{B})$ per operation as desired.

Lecture 9

Memory hierarchy 3

Scribes: Tim Kaler(2012) Jenny Li(2012) Elena Tatarchenko(2012)

9.1 Overview

This is the last lecture on memory hierarchies. Today's lecture is a crossover between cache-oblivious data structures and geometric data structures.

First, we describe an optimal cache-oblivious sorting algorithm called Lazy Funnelsort. We'll then see how to combine Lazy Funnelsort with the sweepline geometric technique to solve batched geometric problems. Using this sweepline method, we discuss how to solve batched orthogonal 2D range searching. Finally, we'll discuss online orthogonal 2D range searching, including a linear-space cache-oblivious data structure for 2-sided range series, as well as saving a log log factor from the normal 4-sided range search.

9.2 Lazy Funnelsort

A funnel merges several sorted lists into one sorted list in an output buffer. Suppose we'd like to merge K sorted lists of total size K^3. We can merge them in $O(\frac{K^3}{B} \lg_{M/B}(\frac{K}{B}) + K)$ memory transfers. *Note: The $+K$ term may be important in certain implementations, but generally the first term will dominate.*

A K-funnel is a complete binary tree with K leaves. Each of the subtrees in a K-funnel is recursively a $\sqrt{K}$-funnel. The edges of a $\sqrt{K}$-funnel point to buffers, each of which is size $K^{3/2}$. Because there are $\sqrt{K}$ buffers, the total size of the buffers in the K-funnel is K^2. See Figure 9.1 for a visual representation.

When the k-funnel is initialized, its buffers are all empty. At the very bottom, the leaves point to input lists.

The actual funnelsort algorithm is an $N^{1/3}$-way mergesort with $N^{1/3}$-funnel merger. *Note: We can*

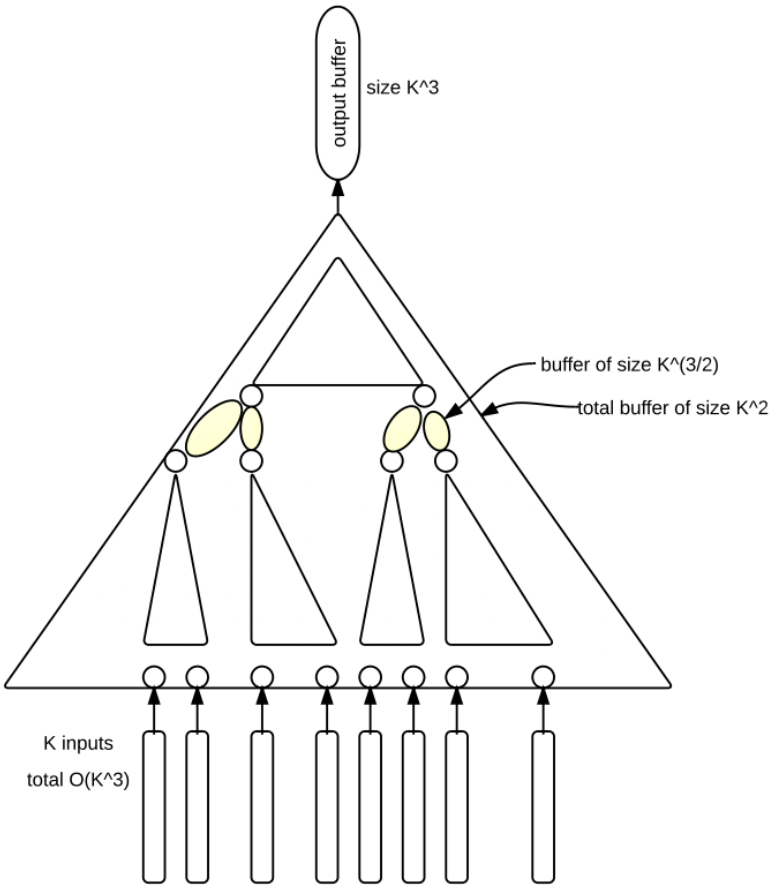


Figure 9.1: Memory layout for K-funnel sort.

only afford $N^{1/3}$-funnel as a merger because of the K^3 multiplier from above.

9.2.1 Filling buffers

When filling a buffer, we'd like the end result to be the output buffer being completely full.

In order to fill a buffer, we must look to the two child buffers to supply the data. See Figure 9.2. We will merge the elements of the child buffers into the parent buffer, as long as both child buffers still contain items. We merge by picking putting in the smaller element of the two child buffers (regular binary merge). Whenever one of the child buffers becomes empty, recursively fill it. We are only considering two input buffers and one resulting, merged buffer at a time. As described above, each leaf in the funnelsort tree has an input list which supplies the original data.

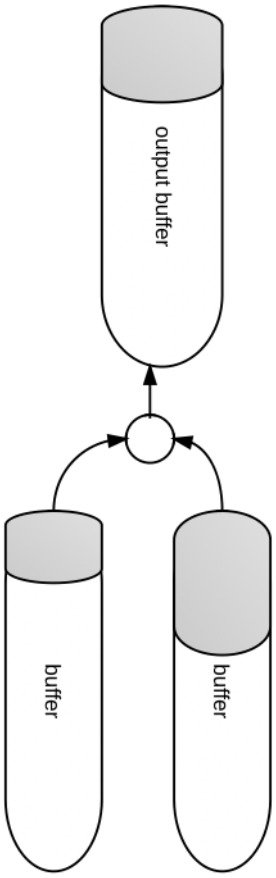


Figure 9.2: Filling a buffer.

9.2.2 Distribution Sweeping via Lazy Funnelsort

The idea is that we can use funnelsort to sort, but it can also do a divide and conquer on the key value.

We can actually augment the binary merge of the filling algorithm to maintain auxilliary information about the coordinates. For example, we can given a point, use this funnelsort to figure out its nearest neighbor.

9.3 Orthogonal 2D Range Searching

Given N points and some rectangles, we'd like to return which points are in which rectangles.

9.3.1 Batched

The batched problem involves getting all N points and N rectangles first. We have all the information upfront, and we're also given a batch of queries and want to solve them all.

The optimal time is going to be $O(\frac{N}{B} \lg_{M/B}(\frac{N}{B}) + \frac{out}{B})$

Algorithm

First, count the number of rectangles containing each point. We need this to figure out what our buffer size needs to be in the funnel.

1. Sort the points and rectangle corners by x using lazy funnelsort.
2. Divide and conquer on x, where we mergesort by y and perform an upward sweepline algorithm. We have slabs L and R, as seen in Figure 9.3.

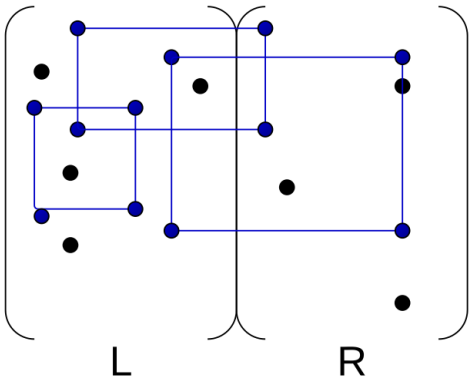


Figure 9.3: Left and Right slabs containing points and rectangles.

3. The problem of which points are in which rectangles are from when the rectangle completely spans one of the slabs, the rectangles in green in Figure 9.7, but not when the points of the rectangle are within that slab.
4. Maintain the number of active rectangles (sliced by the sweep line) with a left corner in L and completely span R. Increment C_L when we encounter a lower-left corner of a rectangle, and decrement when we encounter an upper-left corner.
5. Symmetrically maintain C_R on rectangles which span slab L.
6. When we encounter a point in L, add C_R to its counter. Similarly, add C_L to the counter of any point in R.

At this point, we can create buffers of the correct size. However, this is not optimal.

The optimal solution is to carve the binary tree into linear size subtrees, and make each run in linear time.

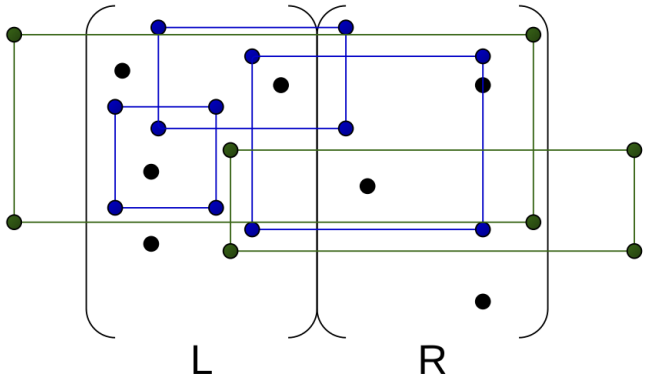


Figure 9.4: Slabs that also contain slabs that span either L or R.

9.3.2 Online Orthogonal 2D range search [61] [62]

It is possible to preprocess a set of N points in order to support range queries while incurring only $O(\log_B N + \frac{out}{B})$ external memory transfers.

We will consider three different kinds of range queries:

- 2-sided range queries : $(\leq x_q, \leq y_q)$
- 3-sided range queries : $([x_{q_{min}}, x_{q_{max}}], \leq y_q)$
- 4-sided range queries : $([x_{q_{min}}, x_{q_{max}}], [y_{q_{min}}, y_{q_{max}}])$

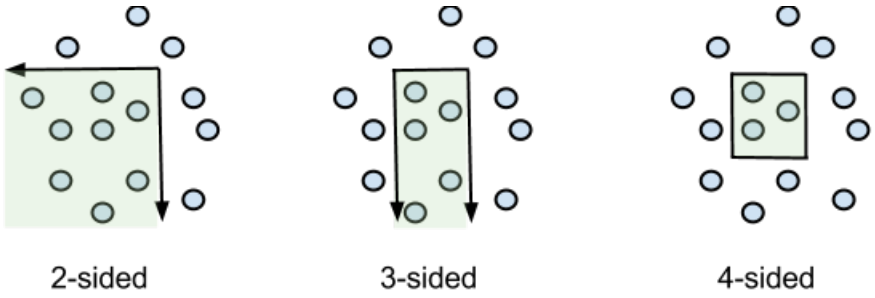


Figure 9.5: Depicts the three different types of range queries.

Only a small amount of extra space is needed to perform cache oblivious range queries. The below table compares the space required to perform range queries in the random access model, and the cache oblivious model.

Query Type	RAM	Cache Oblivious
2-sided	$O(N)$	$O(N)$
3-sided	$O(N)$	$O(N \lg N)$
4-sided	$O(N \frac{\lg N}{\lg \lg N})$	$O(N \frac{\lg^2 N}{\lg \lg N})$

2-sided range queries

First we will show how to construct a data-structure that is capable of performing 2-sided range queries while using only linear space. Using this data-structure, it will be possible to construct data-structures for the 3-sided and 4-sided cases which have the desired space bounds.

At a high level, our datastructure consists of a vEB BST containing our N points sorted on their y coordinate. Each point in the tree contains a pointer into a single array which has $O(N)$ size. This array contains one or more copies of each of the N points, and is organized in a special way. To report the points satisfying a 2-sided range query $(\leq x_q, \leq y_q)$ we find the point in the tree and follow a pointer into the array, and then scan the array until we encounter a point whose x coordinate is greater than x_q. The array's structure guarantees that we will only scan $O(out)$ points and that the distinct points scanned are precisely those satisfying the range query $(\leq x_q, \leq y_q)$.

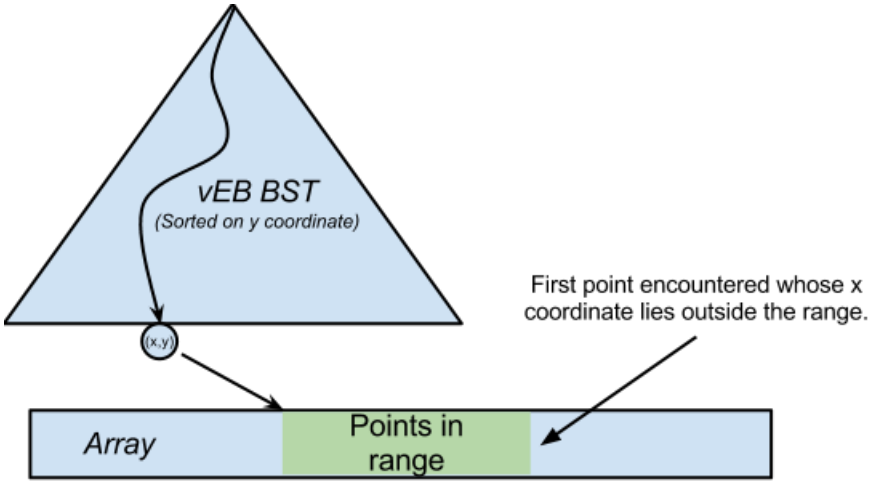


Figure 9.6: A high level description of the datastructure which supports 2 sided range queries using only linear space.

We will proceed to describe the structure of the array and prove the following claims:

1. The high level procedure we described will find all points in $(\leq x_q, \leq y_q)$.
2. The number of scanned points is $O(out)$.
3. The array has size $O(N)$.

First Attempt

We will begin by describing a simple structure for the array which will satisfy claims 1 and 2, but fail to have linear size. This attempt will introduce concepts which will be useful in our second (and successful) attempt.

Definition 15. *A range* $(\leq x_q, \leq y_q)$ *is* ***dense*** *in an array* S *if*

$$|\{(x, y) \in S : x < x_q\}| \leq 2 \cdot \#\ \textit{points in}\ (\leq x_q, \leq y_q)$$

Definition 16. *A range is* ***sparse*** *with respect to* S *if it is not dense in* S*.*

Note that if a range is dense in an array S and S is sorted on x, then we can report all points within that range by scanning through S. Since the range is dense in S, we will scan no more than twice the number of points reported.

Our strategy will be to construct an array $S_0 S_1 \dots S_k$ so that for any query $(\leq x_q, \leq y_q)$ there exists an i for which that query is dense in S_i.

Consider the following structure:

Let $S_0 =$ all points (sorted by x coordinate)

Let $y_i =$ largest y where some query $(\leq x_q, \leq y)$ is sparse in S_{i-1}.

Let $S_i = S_{i-1} \cap (*, \leq y_i)$.

Then let our array be $S_0 S_1 \dots S_i$.

Now consider the query $(\leq x_q, \leq y_q)$. There is some i for which $y_i < y_q$, and for this i the query will be dense in S_{i-1}. For otherwise, y_q would be a larger value of y for which some query $(\leq x_q, \leq y)$ is sparse in S_{i-1}, contradicting the definition of y_i. Now we can construct our vEB BST and have each node point to the start of the subarray for which the corresponding query is dense. This data structure will allow us to find all points in a range while scanning only $O(out)$ points (satisfying claims 1 and 2). However, the array $S_0 S_1 \dots S_k$ may not have $O(N)$ size. Figure 7 shows an example in which the array's size will be quadratic in N.

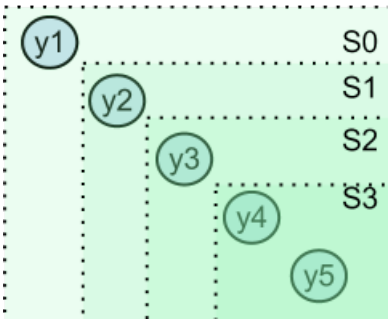


Figure 9.7: An example showing how our first attempt may construct an array with size quadratic in N.

Second (correct) attempt

Let $x_i = \max x$ coordinate where $(\leq x, \leq y_i)$ is sparse in S_{i-1}.

Let $y_i =$ largest y where some query $(\leq x_q, \leq y)$ is sparse in S_{i-1}.

Let $P_{i-1} = S_{i-1} \cap (\leq x_i, *)$.

Let $S_i = S_{i-1} \cap ((*, \leq y_i) \cup (\geq x_i, *))$. (Note that this definition differs from that in our first attempt.)

Our array will now be $P_0 P_1 \dots P_{i-1} S_i \dots S_k$ (where S_j has $O(1)$ size for j between i and k.)

First, we show that the array is $O(N)$ in size. Notice that $|P_{i-1} \cap S_i| \leq \frac{1}{2}|P_{i-1}|$ since $(\leq x_i, \leq y_i)$ is sparse in S_{i-1}. Charge storing P_{i-1} to $(P_{i-1} - S_i)$. This results in each point only being charged once by a factor of $\frac{1}{1-\frac{1}{2}} = 2$. Which implies that the total space used is $\leq 2N$.

Next, notice that when an element is repeated in the array its x coordinate must be less than the x coordinate of the last seen point in the query. Therefore, by focusing on a monotone sequence of x coordinates we can avoid duplicates.

Finally, the total time spent scanning the array will be $O(out)$ because each point is repeated only $O(1)$ times. Therefore, this data structure can support $O(\log_B N + \frac{out}{B})$ 2-sided range queries while using only $O(N)$ space.

3-sided range queries

Maintain a vEB search tree in which the leaves are points keyed by x. Each internal node stores two 2-sided range query datastructures containing the points in that node's subtree. Since each point appears in the 2-sided datastructures of $O(\log N)$ internal nodes, the resulting structure uses $O(N \log N)$ space.

To perform the 3-sided range query $([x_{q_{min}}, x_{q_{max}}], \leq y_q)$, we first find the least common ancestor of $x_{q_{min}}$ and $x_{q_{max}}$ in the tree. We then perform the query $(\geq x_{q_{min}}, \leq y_q)$ in the left child's structure and the query $(\leq x_{q_{max}}, \leq y_q)$ in the right child's structure. The union of the points reported will be the correct answer to the 3-sided range query.

OPEN: 3-sided range queries with $O(\log_B N + \frac{out}{B})$ queries and $O(N)$ space.

4-sided range queries

Notice that we can easily achieve $O(\log_B N + \frac{out}{B})$ queries if we allow ourselves to use $O(N \log^2 N)$ space by constructing a vEB tree keyed on y which contains 3-sided range query structures.

However, it is possible to use only $O(N \frac{\log^2 N}{\log \log N})$ space by storing each point in only $O(\frac{\log N}{\log \log N})$ 3-sided range query structures.

Conceptually, we contract each $\frac{1}{2} \lg \lg N$ height subtrees into $\sqrt{\lg N}$ degree nodes. This results in a tree of height $O(\frac{\lg N}{\lg \lg N})$.

Now, as before, we store two 3-sided range query structure in each internal node containing the

points in their subtree. Further, we store $\lg N$ static search trees keyed by x on points in each interval of the node's $\sqrt{\lg N}$ children.

To perform the query $([x_{q_{min}}, x_{q_{max}}], [y_{q_{min}}, y_{q_{max}}])$ we first find the least common ancestor of $y_{q_{min}}$ and $y_{q_{max}}$ in the tree. Then we perform the query $([x_{q_{min}}, q_{q_{max}}], \geq y_{q_{min}})$ in the child node containing $y_{q_{min}}$, and the query $([x_{q_{min}}, q_{q_{max}}], \leq y_{q_{min}})$ in the child containing $y_{q_{max}}$. Finally, use the search tree keyed on x associated with the interval between the child containing $y_{q_{min}}$ and the child containing $y_{q_{max}}$ to perform the query$([x_{q_{min}}, x_{q_{max}}], *)$ for all the in-between children at once. This data structure supports $O(\log_B N + \frac{out}{B})$ 4-sided range queries, while using only $O(N \frac{\log^2 N}{\log \log N})$. space.

Lecture 10

Dictionaries

Scribes: Scribe: Edward Z. Yang (2012), Katherine Fang (2012), Benjamin Y. Lee (2012), David Wilson (2010), Rishi Gupta (2010)

10.1 Overview

In the last lecture, we finished up talking about memory hierarchies and linked cache-oblivious data structures with geometric data structures. In this lecture we talk about different approaches to hashing. First, we talk about different hash functions and their properties, from basic universality to k-wise independence to a simple but effective hash function called simple tabulation. Then, we talk about different approaches to using these hash functions in a data structure. The approaches we cover are basic chaining, perfect hashing, linear probing, and cuckoo hashing.

The goal of hashing is to provide a solution that is faster than binary trees. We want to be able to store our information in less than $O(u \lg u)$ space and perform operations in less than $O(\lg u)$ time.

In particular, **FKS hashing** achieves $O(1)$ worst-case query time with $O(n)$ expected space and takes $O(n)$ construction time for the static dictionary problem. **Cuckoo Hashing** achieves $O(n)$ space, $O(1)$ worst-case query and deletion time, and $O(1)$ amortized insertion time for the dynamic dictionary problem.

10.2 Hash Function

In order to hash, we need a *hash function*. The hash function allows us to map a universe $\mathcal{U}$ of u keys to a slot in a table of size m. We define three different four different hash functions: totally random, universal, k-wise independent, and simple tabulation hashing.

Definition 17. *A* hash function *is a map h such that*

$$h : \{0, 1, \ldots, u-1\} \to \{0, 1, \ldots, m-1\}.$$

10.2.1 Totally Random Hash Functions

Definition 18. *A hash function h is* totally random *if for all $x \in \mathcal{U}$, independent of all y for all $y \neq x \in \mathcal{U}$,*

$$\Pr_h\{h(x) = t\} = \frac{1}{m}$$

Totally random hash functions are the same thing as the simple uniform hashing of CLRS [141]. However, with the given defintion, a hash function must take $\Theta(\lg m)$ to store the hash of one key $x \in \mathcal{U}$ in order for it to be totally random. There are u keys, which mean in total, it requires $\Theta(u \lg m)$ bits of information to store a totally random hash function.

Given that it takes $\Theta(u \lg u)$ to store all the keys in a binary tree, $\Theta(u \lg m)$ essentially gives us no benefits. As a result, we consider some hash functions with weaker guarantees.

10.2.2 Universal Hash Functions

The first such hash function worth considering is the universal families and the strong unversal families of hash functions.

Definition 19. *A family of hash functions $\mathcal{H}$ is* universal *if for every $h \in \mathcal{H}$, and for all $x \neq y \in \mathcal{U}$,*

$$\Pr_h\{h(x) = h(y)\} = O(\frac{1}{m}).$$

Definition 20. *A set $\mathcal{H}$ of hash functions is said to be a* strong universal family *if for all $x, y \in \mathcal{U}$ such that $x \neq y$,*

$$\Pr_h\{h(x) = h(y)\} \leq \frac{1}{m}$$

There are two relatively simple universal families of hash functions.

Example: $h(x) = [(ax) \mod p] \mod m$ for $0 < a < p$

In this family of hash functions, p is a prime with $p \geq u$. And ax can be done by multiplication of by vector dot product. The idea here is to multiple the key x by some number a, take it modulo a prime p and then slot it into the table of size m.

The downside of this method is that the higher slots of the table may be unused if $p < m$ or more generally, if $ax \mod p$ is evenly distributed, than table slots greater than $p \mod m$ will have fewer entries mapped to them.

The upside is that this a hash function belonging to this universal family only needs to store a and p, which takes $O(\lg a + \lg p)$ bits.

Example: $h(x) = (a \cdot x) >> (\lg u - \lg m)$

This hash function works if m and u are powers of 2. If we assume a computer is doing these computations, than m and u being powers of 2 is reasonable. Here, the idea is to multiply a by x

and then rightshift the resulting word. By doing this, the hash function uses the $\lg m$ high bits of $a \cdot x$. These results come from Dietzfelbinger, Hagerup, Katajainen, and Penttonen [142].

10.2.3 k-Wise Independent

Definition 21. *A family $\mathcal{H}$ of hash functions is* k-wise independent *if for every $h \in \mathcal{H}$, and for all distinct $x_1, x_2, \ldots, x_k \in \mathcal{U}$,*

$$\Pr\{h(x_1) = t_1 \& \cdots \& h(x_k) = t_k\} = O(\frac{1}{m^k}).$$

Even pairwise independent ($k = 2$) is already stronger than universal. A simple example of a pairwise independent hash is $h(x) = [(ax + b) \mod p] \mod m$ for $0 < a < p$ and for $0 \leq b < p$. Here, again, p is a prime greater than u.

There are other interesting k-wise independent hash functions if we allow $O(n^\epsilon)$ space. One such hash function presented by Thorup and Zhang has query time as a function of k [144]. Another hash function that takes up $O(n^\epsilon)$ space is presnted by Siegel [145]. These hash functions have $O(1)$ query when $k = \Theta(\lg n)$.

Example: Another example of a k-wise independent hash function presented by Wegman and Carter [143] is

$$h(x) = [(\sum_{i=0}^{k-1} a_i x^i) \mod p] \mod m.$$

In this hash function, the a_is satisfy $0 \leq a_i < p$ and $0 < a_{k-1} < p$. p is still a prime greater than u.

10.2.4 Simple Tabulation Hashing [143]

The last hash function presented is simple tabulation hasing. If we view a key x as a vector $x_1, x_2, \ldots, x_c$ of characters, we can create a totally random hash table T_i on each character. This takes $O(cu^{1/c})$ words of space to store and takes $O(c)$ time to compute. In addition, simple tabulation hashing is 3-independent. It is defined as

$$h(x) = T_1(x_1) \oplus T_2(x_2) \oplus \cdots \oplus T_c(x_c).$$

10.3 Basic Chaining

Hashing with chaining is the first implementation of hashing we usually see. We start with a hash table, with a hash function h which maps keys into slots. In the event that multiple keys would be hashed to the same slot, we store the keys in a linked list on that slot instead.

For slot t, let c_t denote the length of the linked list chain corresponding to slot t. We can prove results concerning expected chain length, assuming universality of the hash function.

Claim 22. *The expected chain length $E[C_t]$ is constant, since $E[C_t] = \sum_i [Pr[h(x_i) = t]] = \sum_i [O(1/m)] = O(n/m)$.*

where $\frac{n}{m}$ is frequently called the load factor. The load factor is constant when we assume that $m = \Theta(n)$. This assumption can be kept satisfied by doubling the hash table size as needed.

However, even though we have a constant expected chain length, it turns out that this is not a very strong bound, and soon we will look at chain length bounds w.h.p. (with high probability). We can look at the variance of chain length, analyzed here for totally random hash functions, but in general we just need a bit of symmetric in the hash function:

Claim 23. *The expected chain length variance is constant, since we know $E[C_t^2] = \frac{1}{m}\sum_s E[C_s^2] = \frac{1}{m}\sum_{i \neq j} Pr[h(x_i) = h(x_j)] = \frac{1}{m} m^2 O(\frac{1}{m}) = O(1)$.*

Therefore, $Var[C_t] = E[C_t^2] - E[C_t]^2 = O(1)$.

where again we have assumed our usual hash function properties.

10.3.1 High Probability Bounds

We start by defining what high probability (w.h.p.) implies:

Definition 24. *An event E occurs with high probability if $Pr[E] \geq 1 - 1/n^c$ for any constant c.*

We can now prove our first high probability result, for totally random hash functions, with the help of Chernoff bounds:

Theorem 25 (Expected chain length with Chernoff bounds)**.** *$Pr[C_t > c\mu] \leq \frac{\exp(c-1)\mu}{(c\mu)^{c\mu}}$, where μ is the mean.*

We now get our expected high probability chain length, when the constant $c = \frac{\lg n}{\lg \lg n}$

Claim 26 (Expected chain length $C_t = O(\frac{\lg n}{\lg \lg n})$)**.** *For the chosen value of c, $Pr[C_t > c\mu]$ is dominated by the term in the denominator, becoming $1/(\frac{\lg n}{\lg \lg n})^{\frac{\lg n}{\lg \lg n}} = \frac{1}{2^{\frac{\lg n}{\lg \lg n} \lg \lg n}} \approx 1/n^c$*

so for chains up to this length we are satisfied, but unfortunately chains can become longer! This bound even stays the same when we replace the totally random hash function assumption with either $\Theta(\frac{\lg n}{\lg \lg n})$-wise independent hash functions (which is a lot of required independence!), as found by Schmidt, Siegel and Srinivasan (1995) [151], or simple tabulation hashing [150]. Thus, the bound serves as the motivation for moving onto perfect hashing, but in the meantime the outlook for basic chaining is not as bad as it first seems.

The major problems of accessing a long chain can be eased by supposing a cache of the most recent $\Omega(\log n)$ searches, a recent result posted on Pătraşcu's blog (2011) [152]. Thus, the idea behind the cache is that if you are unlucky enough to hash into a big chain, then caching it for later will amortize the huge cost associated with the chain.

Claim 27 (Constant amortized access with cache, amortized over $\Theta(\lg n)$ searches)**.** *For these $\Theta(\lg n)$ searches, the number of keys that collide with these searches is $\Theta(\lg n)$ w.h.p. Applying Chernoff again, for $\mu = \lg n$ and $c = 2$, we get $Pr[C_t \geq c\mu] > 1/n^{\epsilon}$ for some ϵ.*

So by caching, we can see that the expected chain length bounds of basic chaining is still decent, to some extent.

10.4 FKS Perfect Hashing – Fredman, Komlós, Szemerédi (1984) [157]

Perfect hashing changes the idea of chaining by turning the linked list of collisions into a separate collision-free hash table. *FKS hashing* is a two-layered hashing solution to the static dictionary problem that achieves $O(1)$ worst-case query time in $O(n)$ expected space, and takes $O(n)$ time to build.

The main idea is to hash to a small table T with collisions, and have every cell T_t of T be a collision-free hash table on the elements that map to T_t. Using perfect hashing, we can find a collision-free hash function h_i from C_t to a table of size $O(C_t^2)$ in constant time. To make a query then we compute $h(x) = t$ and then $h_t(x)$.

Claim 28 (Expected linear time and space for FKS perfect hashing)**.** $E[space] = E\left(\sum C_t^2\right) = n^2 * O(1/m) = O\left(\frac{n^2}{m}\right)$.

If we let $m = O(n)$, we have expected space $O(n)$ as desired, and since the creation of each T_t takes constant time, the total construction time is $O(n)$.

Claim 29 (Expected # of collisions in C_t)**.** $E[\#collisions] = C_t^2 * O(1/C_t^2) = O(1) \leq \frac{1}{2}$

where the inequality can be satisfied by setting constants. Then for perfect hashing, $Pr[\#Collisions = 0] \geq \frac{1}{2}$. If on the first try we do get a collision, we can try another hash function and do it again, just like flipping a coin until you get heads.

The perfect hashing query is $O(1)$ deterministic and expected linear construction time and space, as we can see from the above construction. Updates, which would make the structure dynamic, are randomized.

10.4.1 Dynamic FKS – Dietzfelbinger, Karlin, Mehlhorn, Heide, Rohnert, and Tarjan (1994) [153]

The translation to dynamic perfect hashing is smooth and obvious. To insert a key is essentially two-level hashing, unless we get a collision in the C_t hash table, in which case we need to rebuilt the table. Fortunately, the probability of collision is small, but to absorb this, if the chain length C_t grows by a factor of two, then we can rebuild the C_t hash table, but with a size multiplied by a factor of 4 larger, due to the C_t^2 size of the second hash table. Thus, we will still have $O(1)$ deterministic query, but additionally we will have $O(1)$ expected update. A result due to Dietzfelbinger and Heide in [154] allows Dynamic FKS to be performed w.h.p. with $O(1)$ expected update.

10.5 Linear probing

Linear probing is perhaps one of the first algorithms for handling hash collisions that you learn when initially learning hash tables. It is very simple: given hash function h and table size m, an insertion for x places it in the first available slot $h(x) + i \mod m$. If $h(x)$ is full, we try $h(x) + 1$, and $h(x) + 2$, and so forth. It is also well known that linear probing is a terrible idea, because "the rich get richer, and the poorer get poorer"; that is to say, when long runs of adjacent elements develop, they are even more likely to result in collisions which increase their size.

However, there are a lot of reasons to like linear probing in practice. When the runs are not too large, it takes good advantage of cache locality on real machines (the loaded cache line will contain the other elements we are likely to probe). There is some experimental evidence that linear probing imposes only a 10% overhead compared to normal memory access. If linear probing has really bad performance with a universal hash function, perhaps we can do better with a hash function with better independence guarantees.

In fact, it is an old result that with a totally random hash function h, we only pay $O(1/\epsilon^2)$ expected time per operation while using $O((1+\epsilon)n)$ space [146]. If $\epsilon = 1$, this is $O(1)$ expected time with only double the space (a luxury in Knuth's time, but reasonable now!) In 1990, it was shown that $O(\lg n)$-wise independent hash functions also resulted in constant expected time per operation [147].

The breakthrough result in 2007 was that we in fact only needed 5-independence to get constant expected time, in [148] (updated in 2009). This was a heavily practical paper, emphasizing machine implementation, and it resulted in a large focus on k-independence in the case that $k = 5$.

At this time it was also shown that 2-independent hash functions could only achieve a really bad lower bound of $\Omega(\lg n)$ expected time per operation; this bound was improved in 2010 by [149] showing that there existed some 4-independent hash functions that also had $\Omega(\lg n)$ expected time (thus making the 5-independence bound tight!)

The most recent result is [150] showing that simple tabulation hashing achieves $O(1/\epsilon^2)$ expected time per operation; this is just as good as totally random hash functions.

OPEN: In practical settings such as dictionaries like Python, does linear probing with simple tabulation hashing beat the current implementation of quadratic probing?

10.5.1 Linear probing and totally random hashing

It turns out the proof that given a totally random hash function h, we can do queries in $O(1)$ expected time, is reasonably simple, so we will cover it here [148]. The main difficulty for carrying out this proof is the fact that the location some key x is mapped to in the table, $h(x)$, does not necessarily correspond to where the key eventually is mapped to due to linear probing. In general, it's easier to reason about the distribution of $h(x)$ (which is very simple in the case of a totally random hash function) and the distribution of where the keys actually reside on the table (which has a complex dependence on what keys were stored previously). We'll work around this difficulty by defining a notion of a "dangerous" interval, which will let us relate hash values and where the keys actually land.

Theorem 30. *Given a totally random hash function h, a hash table implementing linear probing*

will perform queries in $O(1)$ expected time

Proof. Assume a totally random hash function h over a domain of size n, and furthermore assume that the size of the table $m = 3n$ (although this proof generalizes for arbitrary $m = (1+\epsilon)n$; we do this simplification in order to simplify the proof). For our analysis, we will refer to an imaginary perfect binary tree where the leaves correspond to slots in our hash table (similar to the analysis we did for ordered file maintenance.) Nodes correspond to ranges of slots in the table.

Now, define a node of height h (i.e. interval of size 2^h) to be "dangerous" if the number of keys in the table which hash to this node is greater than $\frac{2}{3}2^h$. A dangerous node is one for which the "density" of filled slots is high enough for us to be worried about super-clustering. Note that "dangerous" only talks about the hash function, and not where a key ends up living; a key which maps to a node may end up living outside of the node. (However, also note that you need at most 2^{h+1} keys mapping to a node in order to fill up a node; the saturated node will either be this node, or the adjacent one.)

Consider the probability that a node is dangerous. By assumption that $m = 3n$, so the expected number of keys which hash to a single slot is $1/3$, and thus the expected number of keys which hash to slots within a node at height h, denoted as X_h, is $E[X_h] = 2^h/3$. Denote this value by μ, and note that the threshold for "dangerous" is 2μ. Using a Chernoff bound we can see $\Pr[X_h > 2\mu] \leq e^\mu/2^{2\mu} = (e/4)^{2^h/3}$. The key property about this probability is that it is double exponential.

At last, we now relate the presence of run in tables (clustering) to the existence of dangerous nodes. Consider a run in table of length $\in [2^l, 2^{l+1})$ for arbitrary l. Look at the nodes of height $h = l - 3$ spanning the run; there are at least 8 and at most 17. (It is 17 rather than 16 because we may need an extra node to finish off the range.) Consider the first four nodes: they span $> 3 \cdot 2^h$ slots of the run (only the first node could be partially filled.) Furthermore, the keys occupying the slots in these nodes must have hashed within the nodes as well (they could not have landed in the left, since this would contradict our assumption that these are the first four nodes of the run.) We now see that at least one node must be dangerous, as if all the nodes were not dangerous, there would be less than $< 4 \cdot \frac{2}{3} \cdot 2^h = \frac{8}{3} \cdot 2^h$ occupied slots, which is less than the number of slots of the run we cover ($\frac{9}{3} \cdot 2^h$).

Using this fact, we can now calculate an upper bound on the probability that given x, a run containing x has length $\in [2^l, 2^{l+1}]$. For any such run, there exists at least one dangerous node. By the union bound over the maximum number of nodes in the run, this probability is $\leq 17\Pr[\text{node of height } l - 3 \text{ is dangerous}] \leq 17 \cdot (e/4)^{2^h/3}$ So the expected length of the run containing x is $\Theta(\sum_l 2^l \Pr[\text{length is } \in [2^l, 2^{l+1})]) = \Theta(1)$, as desired (taking advantage of the fact that the inner probability is one over a doubly exponential quantity). □

If we add a cache of $\lg^{n+1} n$ size, we can achieve $O(1)$ amortized with high probability [150]; the proof is a simple generalization of the argument we gave, except that now we check per batch whether or not something is in a run.

10.6 Cuckoo Hashing – Pagh and Rodler (2004) [155]

Cuckoo hashing is similar to double hashing and perfect hashing. *Cuckoo hashing* is inspired by the Cuckoo bird, which lays its eggs in other birds' nests, bumping out the eggs that are originally there. Cuckoo hashing solves the dynamic dictionary problem, achieving $O(1)$ worst-case time for queries and deletes, and $O(1)$ expected time for inserts.

Let f and g be $(c, 6\log n)$-universal hash functions. As usual, f and g map to a table T with m rows. But now, we will state that f and g hash to two separate hash tables. So $T[f(x)]$ and $T[g(x)]$ refer to hash entries in two adjacent hash tables. The cuckoo part of Cuckoo hashing thus refers to bumping out a keys of one table in the event of collision, and hashing them into the other table, repeatedly until the collision is resolved.

We implement the functions as follows:

- *Query(x)* – Check $T[f(x)]$ and $T[g(x)]$ for x.
- *Delete(x)* – Query x and delete if found.
- *Insert(x)* – If $T[f(x)]$ is empty, we put x in $T[f(x)]$ and are done.

  Otherwise say y is originally in $T[f(x)]$. We put x in $T[f(x)]$ as before, and bump y to whichever of $T[f(y)]$ and $T[g(y)]$ it didn't just get bumped from. If that new location is empty, we are done. Otherwise, we place y there anyway and repeat the process, moving the newly bumped element z to whichever of $T[f(z)]$ and $T[g(z)]$ doesn't now contain y.

  We continue in this manner until we're either done or reach a hard cap of bumping $6\log n$ elements. Once we've bumped $6\log n$ elements we pick a new pair of hash functions f and g and rehash every element in the table.

Note that at all times we maintain the invariant that each element x is either at $T[f(x)]$ or $T[g(x)]$, which makes it easy to show correctness. The time analysis is harder.

It is clear that query and delete are $O(1)$ operations. The reason Insert(x) is not horribly slow is that the number of items that get bumped is generally very small, and we rehash the entire table very rarely when m is large enough. We take $m = 4n$.

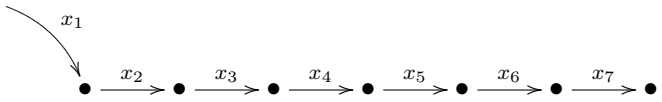
Since we only ever look at at most $6\log n$ elements, we can treat f and g as random functions. Let $x = x_1$ be the inserted element, and $x_2, x_3, \ldots$ be the sequence of bumped elements in order. It is convenient to visualize the process on the *cuckoo graph*, which has verticies $1, 2, \ldots, m$ and edges $(f(x), g(x))$ for all $x \in S$. Inserting a new element can then be visualized as a walk on this graph. There are 3 patterns in which the elements can be bumped.

- *Case 1* Items $x_1, x_2, \ldots, x_k$ are all distinct. The bump pattern looks something like[1]

  The probability that at least one item (ie. x_2) gets bumped is

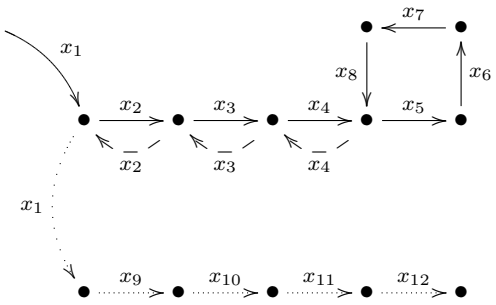
$$Pr(T[f(x)]\text{is occupied}) = Pr(\exists\, y : f(x) = g(y) \vee f(x) = f(y)) < \frac{2n}{m} = \frac{1}{2}.$$

[1]Diagrams courtesy of Pramook Khungurn, Lec 1 scribe notes from when the class was taught (as 6.897) in 2005



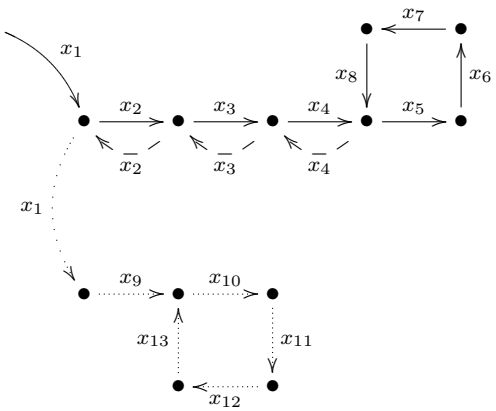
The probability that at least 2 items get bumped is the probability the first item gets bumped ($< 1/2$, from above) times the probability the second item gets bumped (also $< 1/2$, by the same logic). By induction, we can show that the probability that at least t elements get bumped is $< 2^{-t}$, so the expected running time ignoring rehashing is $< \sum_t t2^{-t} = O(1)$. The probability of a full rehash in this case is $< 2^{-6 \log n} = O(n^{-6})$.

- *Case 2* The sequence $x_1, x_2, \ldots, x_k$ at some point bumps an element x_i that has already been bumped, and $x_i, x_{i-1}, \ldots, x_1$ get bumped in turn after which the sequence of bumping continues as in the diagram below. In this case we assume that after x_1 gets bumped all the bumped elements are new and distinct.



The length of the sequence (k) is at most 3 times max{#solid arrows, #dashed arrows} in the diagram above, which is expected to be $O(1)$ by Case 1. Similarly, the probability of a full rehash is $O(2^{\frac{-6 \log n}{3}}) = O(n^{-2})$.

- *Case 3* Same as Case 2, except that the dotted lines again bump something that has been bumped before (diagram on next page).



In this case, the cost is $O(\log n)$ bumps plus the cost of a rehash. We compute the probability Case 3 happens via a counting argument. The number of Case 3 configurations involving t

distinct x_i given some x_1 is ($\leq n^{t-1}$ choices for the other x_i) $\cdot$ ($< t^3$ choices for the index of the first loop, where the first loop hits the existing path, and where the second loop hits the existing path) $\cdot$ (m^{t-1} choices for the hash values to associate with the x_i) $= O(n^{t-1}t^3m^{t-1})$.

The total number of configurations we are choosing from is $\binom{m}{2}^t = O(2^{-t}m^{2t})$, since each x_i corresponds to a possible edge in the cuckoo graph. So the total probability of a Case 3 configuration (after plugging in $m = 4n$) is

$$\sum_t \frac{O(n^{t-1}t^3m^{t-1})}{O(2^{-t}m^{2t})} = O(n^{-2})\sum_t \frac{t^3}{2^{3t}} = O(n^{-2}).$$

If there is no rehash, the cost of insertion is $O(1)$ from Cases 1 and 2. The probability of a rehash is $O(n^{-2})$. So, we have an insertion taking time $Pr(\text{Rehash})\cdot(O(\log n)+n\cdot\text{Insert})+(1-Pr(\text{Rehash}))\cdot O(1) = O(1/n)\cdot\text{Insert} + O(1)$, so overall the cost of an insertion is $O(1)$ in expectation as desired. Thus, with Cuckoo hashing we have $(2+\epsilon)n$ space, and 2 deterministic probes per query.

10.6.1 Claims

With either totally random or $O(\lg n)$-wise independence, we get $O(1)$ amortized expected update, and $O(1/n)$ build failure probability, which is the chance that your keyset will be completely unsustainable with the current Cuckoo hash table, at which point you would have to start over and rebuild [155].

6-wise independence is insufficient for Cuckoo hashing to get $O(1)$ expected update, with a build failure probability of $1 - 1/n$, which is quite bad. This result is shown by Cohen and Kane (2009) in [156].

With simple tabulation hashing, the build failure probability becomes $\Theta(1/n^(1/3))$, which can be found in [150].

Theorem 31 (Constant expected update, for totally random hash functions)**.** *Pr[Insert follows bump path of length k] $\leq 1/2^k$*

For two hash functions g and h, where each has n values, and each of these n values has $\lg m$ *bits. Thus we need* $2n\lg n$ *bits to encode g and h.*

Claim 32 (Encoding hash functions in $2n\lg(n) - k$ time)**.**

Lecture 11

Integer 1

Scribes: Sam Fingeret (2012), Shravas Rao (2012), Paul Christiano (2010)

11.1 Overview

In this lecture, we will be discussing integer data structures that can be used to solve the predecessor problem. We will start by describing the various models we can use, and then introduce two integer data structures, Van Emde Boas and Y-fast trees.

11.2 Integer Data Structures

Before presenting any integer data structures, we present some models for computations with integers (both to precisely state what we are looking for, and as a framework to prove lower bounds).

In each model, memory is arranged as a collection of words of some size (and potentially a small amount of working memory), and a fixed cost is charged for each operation. We also always make the *fixed universe assumption*. We assume that each data structure needs to answer queries with respect to a fixed set of w-bit integers $\mathcal{U} = \{0, 1, 2, \ldots, u-1\}$.

11.2.1 Trandichotomous RAM

- Memory is divided into an array of size S made up of cells of size w. It is practical to require w to be greater than $\log(S)$, as otherwise, you won't be able to access the whole array. If we let n be the problem size, then this also implies that $w \geq log(n)$.
- There is some fixed set of operations, each of which modifies $O(1)$ memory cells.
- Words serve as pointer, so cells can be addressed arbitrarily.

11.2.2 word-RAM

In word-RAM, the processor may use the $O(1)$ fixed operations $+, -, *, /, \%, \&, |, ^, >>, <<, <, >$, you would expect as primitives in a high-level language such as C.

11.2.3 The Cell-Probe Model

- Memory is divided into cells of size w, where w is a parameter of the model.
- Reading to or writing from a memory cell costs 1 unit.
- Other operations are free

This model does not realistically reflect a modern processor, but its simplicity and generality make it appropriate for proving lower bounds.

These models vary in how powerful they are. In particular, the cell probe model is stronger than the trandichotomous RAM model which is in turn stronger than the word RAM model, followed by the pointer machine and the BST.

11.3 Successor / Predecessor Queries

We are interested in data structures which maintain a set of integers S and which are able to insert an integer into S, delete an integer from S, report the smallest element of S greater than some query x, or report the largest element of S less than some query x.

We sill start by presenting the current known bounds for such a data structure on these various models.

11.3.1 BST Model

In the BST model, this problem is known to require $\Theta(\log(n))$ time per query.

11.3.2 Word RAM model

In the word RAM model, we have a few data structures. Using the Van Emde Boas data structure, we can achieve $O(\log(w))$ time per query, and $\Theta(u)$ space. However, we can decrease the amount of space using modified version of the Van Emde Boas data structure, $O(\log(w))$ time per query with high probability, and $\Theta(n)$ space. Y-fast trees also achieve the same bounds.

There are also fusion trees, which can achieve $O\left(\log_w(n)\right)$ time with high probability, and $\Theta(u)$ space. We will be covering these in lecture 12.

Note that the query time of a fusion tree may or may not be better than that of a Y-fast tree, depending on our values of w and n. If we chose the optimal data structure depending on these values, then it turns out that the query time is $O\left(\sqrt{\log(n)}\right)$ time with high probability, with $\Theta(u)$ space. This can be shown by a small calculation.

11.3.3 Cell Probe Model

The lower bounds with the cell probe model are $O(n\text{poly}\log(n))$ space, and $\Omega\left(\min\{\log_w(n), \frac{\log(w)}{\log(\frac{\log(w)}{\log\log(n)})}\right)$ time. The latter bound takes the optimal of van Emde Boad and fusion trees. Specifically, van Emde Boas is optimal when $w = O(\text{poly}\log(n))$, while fusion trees are optimal for when $w = 2^{\Omega\left(\sqrt{\log(n)}\log\log(n)\right)}$

11.3.4 Pointer machine model

Here, our word is specified by a pointer. Using van Emde Boas we can use $O(\log(\log(u)))$ time per operation, and $\Theta(u)$ space. There also exists a lower bound of $O(\log(\log(u)))$ time per operation, and $\Omega(u)$ space

11.4 Van Emde Boas Trees

Now we will describe the structure of a Van Emde Boas. The goal of this data structure is to be able to bound the running time of all operations by $T(u) \leq T(\sqrt{u}) + O(1)$. This way, we get using Master's theorem that $T(u) = O(\log\log u)$.

To start, if we are given a binary string x write high(x) for the first $|x|/2$ digits and low(x) for the last $|x|/2$ digits. When x is an integer with binary representation bin(x), we write c for the number whose binary representation is high(bin(x)) and i for the number whose binary representation is low(bin(x)). Then given an integer x, it is stored in block V.cluster[c] and within that block it is at position i.

A VEB, V, of size u with word size w consists of $\sqrt{(u)}$ VEB's of size $\sqrt{(u)}$ and word size $w/2$, each denoted by V.cluster[c]. If $x = \langle c, i\rangle$ is contained in V (and x is not the minimum), then i is contained in V.cluster[c]. In addition, we maintain a summary VEB V.summary of size $\sqrt{(u)}$ and word size $w/2$, which contains all integers c such that V.cluster[c] is not empty. Finally, each structure contains the minimum element, V.min (or None if it is empty), and a copy of the maximum element, V.max.

To answer a query Successor($V, x = \langle c, i\rangle$), we first check if x is less than or greater the minimum and maximum elements respectively. In this case, the minimum element is the successor, or there is no successor. Otherwise, we check to see if i is less than the maximum of V.cluster[c]. In this case, there exists an element greater than x in the same cluser as x, so we can just recursively search for the sucessor in that cluster. Otherwise, the successor of x is the min of V.cluster[c'], where c' is the successor of c in V.summary. In this case, we recursively find the successor of c in V.summary. Note that in both cases, we require a single search in a VEB of size $O(\sqrt{(u)})$. The query to find the predecessor works in a very similar fashion.

For the query Insert($V, x = \langle c, i\rangle$), we again start by checking the minimum and the maximum. If V is empty, (V.min is None), then we let both the min and the max be x, and we are done. If x is less than V.min, then we let V.min be x, and insert the old minimum. If x is greater than V.max then we replace this with a copy of x in V.max. Now, we check to see if V.cluster[c] is empty, or if there is no minimum. In this case, we recursively insert c into V.summary, and then insert i into

V.cluster[c]. Inserting i will take constant time in this case because V.cluster[c] is empty at this point. Otherwise, we just recursively insert i into V.cluster[c].

Finally, to preform deletions, Delete($V, x = \langle c, i \rangle$), we start with the case in which x is the minimum element. If the max is also x, then we can just delete this element by setting V.min and V.max to None. Otherwise, we have to find the new min, which is $\langle$V.summary.min, V.cluster[V.summary.min].min$\rangle$ (note that the minimum is not stored elsewhere), delete it, and set it as the new V.min. To delete x if x is not V.min, we first recursively delete i from V.cluster[c].i. If V.cluster[c] is now empty, then this would have taken constant time, and now we must recursively delete c from V.summary. Finally, we must update V.max. If V.summary is empty, then there is only one element in V, and V.max must be set to V.min. Otherwise, V.max is now $\langle$V.summary.max, V.cluster[V.summary.max].max$\rangle$).

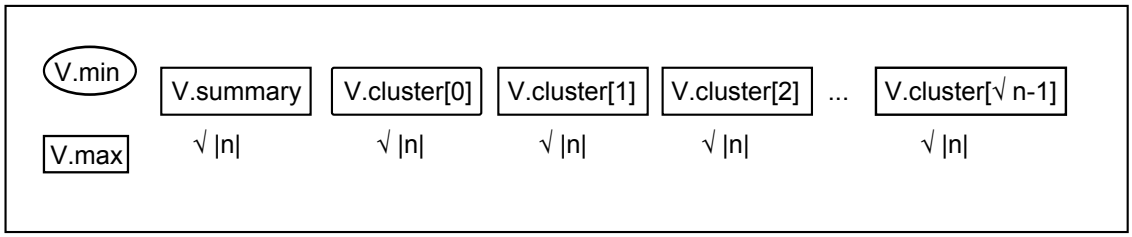


Figure 11.1: A visual representation of the recursive structure of a VEB. Note that at each level, the minimum is stored directly, and not recursively.

Theorem 33. *A VEB preforms the operations sucessor, predecessor, insertion and deletion in* $O(\log \log u)$ *time.*

Proof. The running time T in all cases satisfies $T(u) \leq T(\sqrt{u}) + O(1)$. Note that specifically, we only recurse once. It follows by the Master's theorem, that $T(u) = O(\log \log u)$. □

11.5 Binary tree view

One can also interpret the van Emde Boas structure as a binary tree built over the bit vector v where $v[i] = 1$ if i is in the structure and $v[i] = 0$ otherwise. This is how van Emde Boas presented the structure in his original paper [64]. At a given node, we store the OR of its two children, thus the value of a node determines if there exists in the structure some integer in the interval of that node. We will also store in every node the minimum and maximum of all integers stored in its interval. In addition, we can compute the k^{th} level node in the path to any given integer x in $O(1)$ time because that node corresponds to the k high-order bits of x.

This view gives us an algorithm to quickly compute the predecessor or successor of any given integer using the tree structure, assuming that our set is static. The key observation is that if we look at the values of the nodes along the path to a given integer x, the values form a monotone sequence of contiguous ones followed by zeros. Thus, we can do a binary search to find the $1 \rightarrow 0$ transition in the sequence, so let the final 1-value node be u and the first 0-value node be v. In the case that

no transition exists, then we either have no integers in the structure or x is in the structure, both of which are trivial cases. If v is the left child of u, then we know that the successsor of x must lie in the right child of u and that the right child of u contains at least one integer, thus we can simply return the minimum stored at the right child of u. Similarly, if v is the right child of u, then we can obtain the predecessor of x by looking at the maximum stored in the left child of u. Finally, to obtain the successor from the predecessor or vice versa, we can store all elements of the structure in a sorted linked list.

11.5.1 Analysis

Because the binary tree has height w and the query time is dominated by the binary search on the path, this gives us an $O(\lg w)$ or $O(\lg \lg u)$ query time bound. However, to implement the binary search on a pointer machine, a stratified tree is required, thus storing at each node pointers to its ancestor of height 2^i for $i = 0, 1, ..., \lg w$, thus because we have $O(u)$ nodes we obtain space bound $O(u \lg w)$. In addition, updating this structure is very expensive, as we have to update all nodes along the path, for $O(\lg u)$ update time. However, as van Emde Boas notes [64], the same trick of not storing the minimum recursively can reduce the update time to $O(\lg \lg u)$, and this bound is optimal on a pointer machine.

11.5.2 Indirection

We now present a second way to reduce the update time on this tree structure through indrection. Suppose that instead of having our tree structure be over an set of integers, suppose we instead use a set of binary search trees, each of size $\Theta(w)$. In other words, we partition the set of integers in the structure into $\Theta(\frac{n}{w})$ parts and store each part in a binary tree. For a given BST, we can arbitrarily choose any integer stored in the tree as its representative element in the larger tree structure.

We allow the BST structures to have size in $[1/2w, 2w]$. Once a BST gets too large, we split it into two BSTs. Once a BST gets too small, we merge it with one of the adjacent BSTs and then potentially split the result. These splits and merges, as well as updating the top-level structure can all be done in $O(w)$ time, so we just need to show that this happens in at most one out of every $O(w)$ updates. However, after a split we can verify that a binary search tree has between $2/3w$ and $3/2w$ vertices, so does not need to split or merged again for another $O(w)$ operations. When we merge two trees, we pay in advance for the next time we will have to split the merged tree. Thus the total amortized time is $O(1)$ for each modification, so the time bound is dominated by the bottom-level update time, which is $O(\lg w) = O(\lg \lg u)$, as desired.

11.6 Reducing space

All structures given so far require $\Omega(u)$ space, but we wish to achieve the optimal $\Theta(n)$ space bound. We present two methods for achieving this bound as well as $O(\lg \lg u)$ update time with high probability through the use of perfect hashing.

11.6.1 Hash tables in vEB

Suppose in our original van Emde Boas structure, we do not store empty clusters and replace child arrays by dynamic perfect hash tables. By charging every hash table entry to the minimum in that cluster, every integer in the structure is charged at most twice by either its unique parent or entry in a summary structure, thus we obtain the optimal space bound of $\Theta(n)$. In addition, if we use a hash table with $O(1)$ query and update with high probability, this allows us to obtain query and update bounds of $O(\lg \lg u)$ with high probability.

11.6.2 X-fast trees

Both x-fast trees and y-fast trees are due to Willard [65]. While X-fast trees do not achieve the desired bounds, they are an intermediate step towards y-fast trees which do. The main idea is to only store the 1-value nodes in the simple tree view structure. For a given 1-value node in the tree, we can compute a corresponding bitstring by looking at the path of left-child and right-child steps from the root to the given node, with a left-child step corresponding to a 0 in the string and a right-child step corresponding to a 1. An x-fast tree consists a perfect hash table of all such bitstrings for all 1-value nodes in the tree, or all prefixes of bitstrings of integers in the structure. This table allows us to map a bitstring to the minimum and maximum values stored at that node in $O(1)$ time.

Note that we can look up the value of any given node in the tree by checking whether its bitstring is in the hashtable. Thus, we can still perform the binary search in $O(\lg w)$ time with high probability and achieve the same query bound. However, to perform an update, we still need to update all nodes along the path from the root, thus we have an $O(w)$ update bound. In addition, we store information for all w prefixes of every integer in the structure, for a worst case $O(nw)$ space bound.

11.6.3 Y-fast trees

To obtain Y-fast trees, we use the same indirection trick. We maintain a top-level x-fast tree structure of size $\Theta(\frac{n}{w})$ with the bottom level structures being binary search trees of size $\Theta(w)$. The $O(\lg w)$ query bound is maintained, but the top-level update can now be charged to the $\Theta(w)$ bottom-level updates, thus the update time is dominated by the $O(\lg w)$ time for the bottom-level BST structures. In addition, the space required for the top level structure is now $O(n)$, thus we achieve total space bound $O(n)$, as desired.

Lecture 12

Integer 2

Scribes: Kent Huynh (2012), Shoshana Klerman (2012), Eric Shyu (2012)

12.1 Introduction

We continue our analysis of integer data structures, focusing this lecture on *fusion trees*. This structure employs some neat bit tricks and word-level parallelism. In particular, we discuss the following techniques necessary to understand the workings of a fusion tree: *sketching*, which allows certain w-bit words to be compressed to less than w bits, *parallel comparison*, where multiple words can be compared for the cost of a single word, and finally the computation of the most significant set bit of a word in constant-time.

12.2 Overview of Fusion Trees

We first describe the key results of fusion trees, as well as the model we will be assuming for the majority of the exposition. As with the van Emde Boas trees described in the previous lecture, we will be working under the word RAM model (transdichotomous RAM with C-style operations) and we are looking to store n w-bit integers statically. Under these assumptions, the fusion tree covered here and detailed by Fredman & Willard in their original papers ([158], [159]) performs predecessor/successor operations in $O(\log_w n)$ time, and require $O(n)$ space. Other models and variants of interest include:

- AC^0 RAM version [160]: the model is restricted to operations with constant-depth (but unbounded fan-in and fan-out) circuits. In particular, multiplication is not permitted, since it requires a circuit of logarithmic depth in the number of bits (this model was more relevant in the past when multiplication was very costly relative to other operations like additions; this is no longer the case due to optimizations such as pipelining);
- Dynamic version via exponential trees [161]: this version achieves $O(\log_w n + \lg\lg n)$ *deterministic* update time, i.e. a $\lg\lg n$ overhead over the static version;

- Dynamic version via hashing [162]: this version achieves $O(\log_w n)$ *expected* update time. This is based on performing sketching 'more like' hashing. **OPEN:** Can this bound be achieved *with high probability*?

12.3 The General Idea

The underlying structure of a fusion tree is a B-tree, with branching factor $w^{1/5}$; actually, any small constant power suffices, since the height of the tree will be $\Theta(\log_w n)$. The main issue arises when searching a node during a predecessor search: we would like to achieve $O(1)$ time for this operation, which appears impossible since it seems to require at least reading in $O(w^{1/5} \cdot w) = O(w^{6/5})$ bits. However, this (and predecessor/successor) can actually be done in the desired time bound with $k^{O(1)}$ preprocessing. The main idea is to distinguish the set of keys in a node with less than w bits, which is the basis behind the next section.

12.4 Sketching

Since for each node in a fusion tree there are at most $k = w^{1/5}$ keys, it seems reasonable that these keys can be represented by only $w^{1/5}$ bits and still be comparable. Indeed, this can be accomplished as follows. Let the keys be $x_0 \leq x_1 \leq \cdots \leq x_{k-1}$; each key x_i can be represented as a path in a binary tree of depth w, where the left branch at the i-th node from the root is taken if the i-th most significant bit of x_i is 0, and otherwise the right branch is taken. Then if all k keys are overlaid on the same tree, then it is evident that the resulting paths branch out at at most $k-1$ nodes (this is easily formalized by induction). In essence, at most $k - 1 = w^{1/5} - 1$ bits matter in ordering the x_i. See figure 12.1 for an example.

In particular, let the bits of the corresponding nodes be in positions $b_0, b_1, \ldots, b_{r-1}$ (where $r \leq w^{1/5}$). Then the *perfect sketch* of x (denoted by $sketch(x)$) is the r-bit string where the i-th bit is the b_i-th bit of x. Clearly, the sketch operation preserves order among the x_i, since each sketch keeps the bits that distinguish all the x_i in the right order. Sketching also allows all keys to be read in constant time, since each sketch has $O(w^{1/5})$ bits so the total size of all sketches is $O(kw^{1/5}) = O(w^{2/5}) = o(w)$ bits. Under some models, such as the AC^0 model, the perfect sketch operation is a single operation [160]. Later in this lecture we will see how to perform a sufficient approximation using multiplication and standard C-style operations.

However, this raises another problem. The search for a given query q may be such that q is not equal to any of the x_i (since there are no restrictions on the values of the arguments to predecessor/successor). Hence, the path of q in the binary tree may diverge from the other paths of x_i at a node which does not correspond to one of the bits $b_0, \ldots, b_{r-1}$; in that case, the location of $sketch(q)$ among the $sketch(x_i)$ will not necessarily be equivalent to the location of q among the x_i. This can be resolved by the technique of *desketchifying* as discussed next.

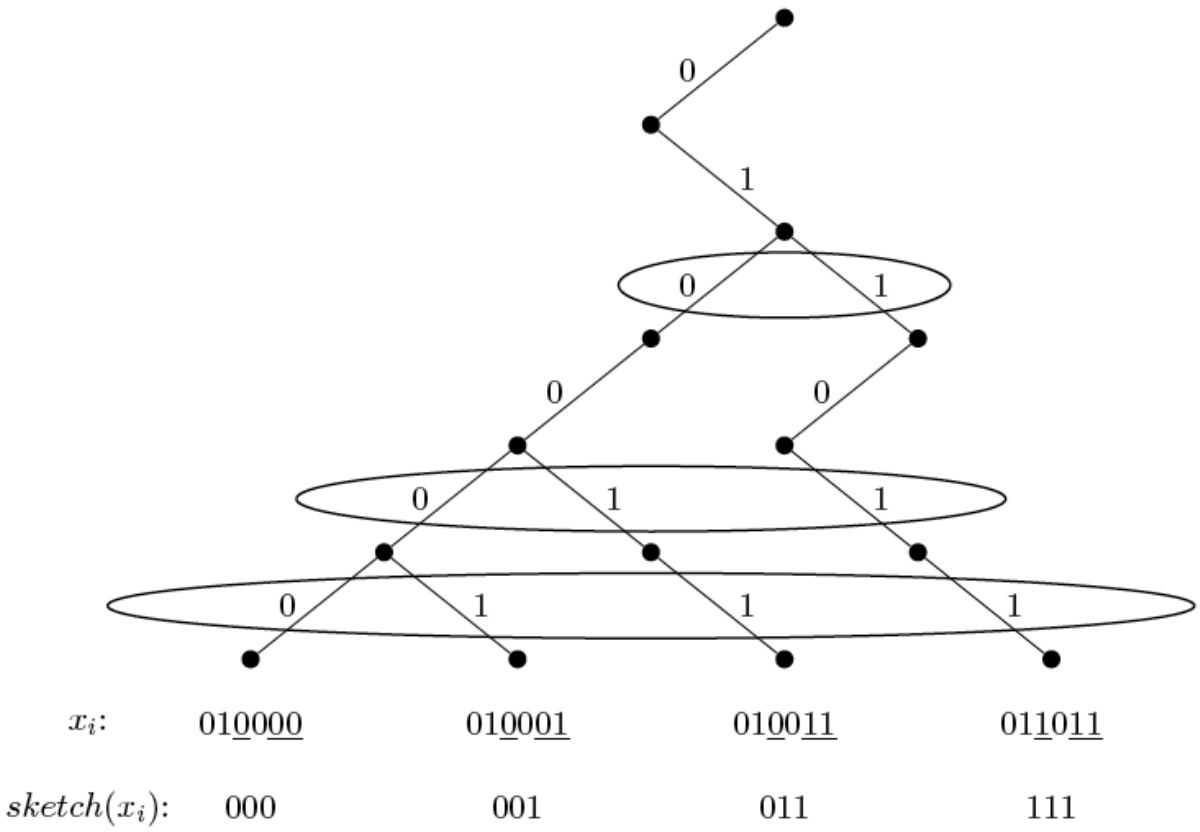


Figure 12.1: An example of the *sketch* function with 4 keys. The levels corresponding to the 3 bits sketched are circled.

12.5 Desketchifying

By modifying the search for q, we can still obtain the predecessor or successor of q without any additional (asymptotic) runtime overhead. Let x_i and x_{i+1} be the sketch neighbours of q, i.e. $sketch(x_i) \leq sketch(q) \leq sketch(x_{i+1})$. Then we determine the longest common prefix (equivalently, the lowest common ancestor) of the actual elements between either q and x_i, or q and x_{i+1}. Suppose this prefix p has length y; then the node n corresponding to this prefix is the highest such that the path for q diverges from the path of every key in the fusion node. In particular, there are no keys in the child subtree of n which contains the path of q. Since the other child subtree of n contains a key of the fusion node (either x_i or x_{i+1}) it must contain either the predecessor or successor of q. This can be determined as follows:

- If the $(y+1)$-st bit of q is 1, then q's predecessor belongs in the $p0$ subtree, so we search for the predecessor of $e = p011\cdots1$.
- If the $(y+1)$-st bit of q is 0, then q's predecessor belongs in the $p1$ subtree, so we search for the successor of $e = p100\cdots0$.

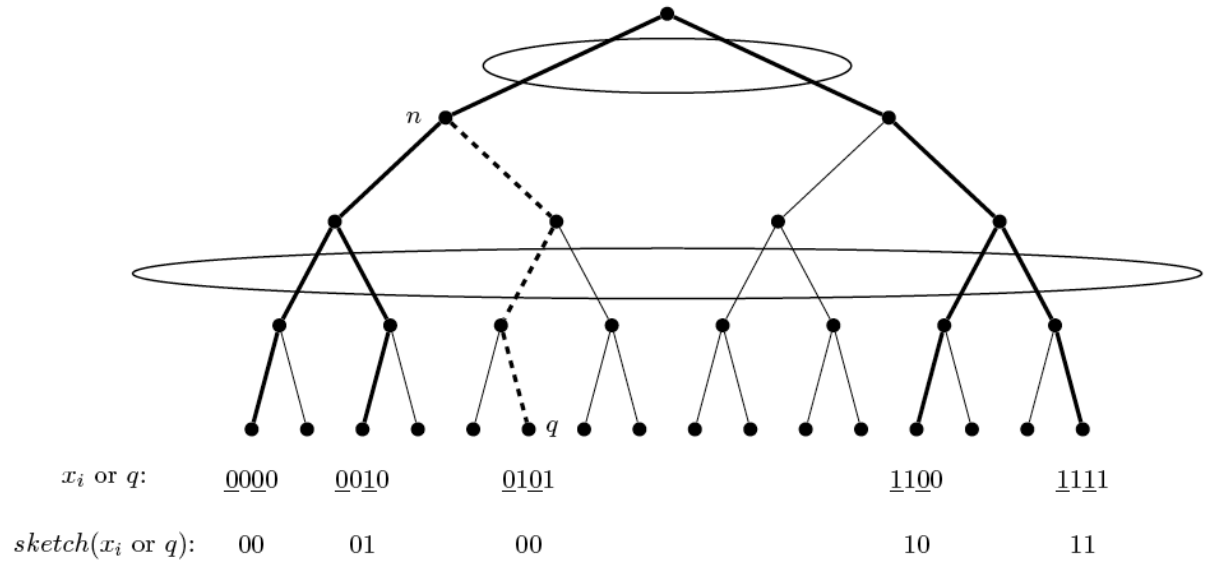


Figure 12.2: An example when the search query is not among the keys of the fusion node. The paths to the keys are bolded, whereas the path to the query q is dashed; the levels corresponding to the bits sketched are circled as before. Here, the sketch neighbours of q are x_0 and x_1, but x_0 is neither a predecessor nor successor of q.

In both cases, the search will successfully find the requisite key because all the sketch bits in the prefix of e and the target will match, and all the sketch bits in the suffix of e (following the first y bits) will be either the highest (when searching for predecessor) or lowest (when searching for successor). Once one of the predecessor/successor is found, the other can be determined simply by checking the appropriate adjacent sketch word in the fusion node. See figure 12.2 for an example.

There are still several issues remaining before our fusion tree will run with the desired time bounds under the word RAM model. First, we demonstrate how to perform an approximation of the perfect sketch in reasonable time. Then we show how to achieve constant runtime of two particular subroutines: finding the location of a $w^{1/5}$-bit integer among $w^{1/5}$ such integers, encoded as a $w^{2/5}$-bit word; and determining the most significant set bit of a w-bit word (this can be used to determine the length of the longest common prefix of two strings by XORing them together). This will conclude our description of the fusion tree's operation.

12.6 Approximating Sketch

Although a perfect sketch is computable in $O(1)$ time as an AC^0 operation, we want a way to compute an *approximate* sketch on a Word RAM using just multiplication and other standard operations. The hard part about computing a sketch is getting all the bits we care about consecutive and succinct. So this approximate sketch will have all the important bits, spread out in some predictable pattern (independent of the word x we are sketching) of length $O(w^{4/5})$, with some additional garbage between them. But we will be able to apply masks to get just the bits we care about.

Let x' be x masked so it just has the important bits. So

$$x' = x \ \ AND \ \ \sum_{i=0}^{r-1} 2^{b_i}$$

Now multiply x' by some mask m (that will have set bits in positions m_j) to get

$$x' \cdot m = (\sum_{i=0}^{r-1} x_{b_i} 2^{b_i})(\sum_{j=0}^{r-1} 2^{m_j}) = \sum_{i=0}^{r-1}\sum_{j=0}^{r-1} x_{b_i} 2^{b_j+m_j}$$

Claim 34. *For any important bits $b_0, b_1, \ldots b_{r-1}$, we can choose $m_0, m_1, \ldots m_{r-1}$ such that*

1. *$b_j + m_j$ are distinct for all j. This means that there are no collisions when we add up all the bits.*
2. *$b_0 + m_0 < b_1 + m_1 < \ldots b_{r-1} + m_{r-1}$. This means that order of our important bits in x is preserved in $x' \cdot m$.*
3. *$(b_{r-1} + m_{r-1}) - (b_0 + m_0) = O(w^{4/5})$. Thus the span of the bits will be small.*

Proof. We'll choose some $m'_0, m'_1, \ldots m'_{r-1} < r_3$ such that $b_j + m'_j$ are all distinct modulo r^3. We'll prove this by induction. Suppose we have picked $m'_0, m'_1, \ldots m'_{t-1}$. Then m'_t must be different than $m'_i + b'_j - b_k \forall i, j, k$. There are t choices for i (since i can be any of the previous choices), and r choices for j, k. Thus, there are a total of $tr^2 < r^3$ things for m'_t to avoid, and we have r^3 choices, so we can always choose m'_t to avoid collisions. So this satisfies property (1).

To satisfy (2) and (3) we intuitively want to spread out $m_i + b_i$ by intervals of r^3. To do this we let

$$m_i = m'_i + (w - b_i + ir^3 \text{ rounded down to nearest multiple of } r^3) \equiv m'_i \pmod{r^3}$$

We claim without proof that spacing will make

$$m_0 + b_0 < \ldots < m_{r-1} + b_{r-1}$$

and also since $m_0 + b_0 \approx w$ and $m_{r-1} + b_{r-1} \approx w + r^4$ we will have $(b_{r-1} + m_{r-1}) - (b_0 + m_0) \approx r^4 = O(w^{4/5})$. So properties (2) and (3) will be satisfied. □

12.7 Parallel Comparison

We need to be able to find where $sketch(q)$ lies among the sketches of keys $sketch(x_0) < sketch(x_1) ... < sketch(x_{k-1})$ at a given node in constant time. We can do this parallel comparison with standard operations. We will use something called the "node sketch."

Node Sketch: We store all the sketches of the x_i's at a node in a single word by prepending a `1` to each and concatenating them. The result will look like this: $\texttt{1}sketch(x_0)\ldots\texttt{1}sketch(x_{k-1})$.

In order to compare $sketch(q)$ to all the key sketches with one subtraction, we take $sketch(q)$ and make k copies of it in a word $\texttt{0}sketch(q)\ldots\texttt{0}sketch(q)$. We denote this by $sketch(q)^k$. If the sketches were 5 bits long, we would multiply $sketch(q)$ by `000001...000001`.

Then we subtract this value from the node sketch. This lets us subtract $\texttt{0}sketch(q)$ from each $\texttt{1}sketch(x_i)$ with a single operation: since $\texttt{1}sketch(x_i)$ is always bigger than $\texttt{0}sketch(q)$, carrying will not cause the subtractions to interfere with each other. In fact, the first bit of each block will be `1` if and only if $sketch(q) \leq sketch(x_i)$. After subtracting, we AND the result with `100000...100000` to mask all but the first bit of each block.

The $sketch(x_i)$'s are sorted in each node, so for some index k we have $sketch(q) > sketch(x_i)$ when $i < k$ and $sketch(q) \leq sketch(x_i)$ otherwise. We need to find this index k. Equivalently, we need to find the number of bits which are equal to 1 in the above result. This is a special case of finding the index of the most significant 1 bit. To do this, we can multiply by `000001...000001`: all the bits which were set to 1 will collide in the first block of the result, so we can find their sum by looking at that block. We can AND the result with 1111 and shift right to get the total number of 1s.

In summary, we have:

1. Compute the node sketch.
2. Compute $sketch(q)^k$.
3. Subtract $sketch(q)^k$ from the node sketch.
4. AND the difference with `100000...100000`.
5. Find the most significant bit / number of 1 bits of the result. This is the index of the 0 to 1 transition and the rank of the sketch.

12.8 Most Significant Set Bit

We conclude with the computation of the index of the most significant set bit of a w-bit word in $O(1)$ time, under the word RAM model. The solution is particularily messy, but it will use all the techniques that we have just seen for fusion trees. The first insight is to split the word x into $\sqrt{w}$ clusters of $\sqrt{w}$ bits. Our strategy is to identify the first non-empty cluster (this is the hardest part), and then the index of the first 1-bit within that cluster.

To illustrate the the following procedures, let $\sqrt{w} = 4$ and

$$x = \underbrace{0101}_{\sqrt{w}} \quad \underbrace{0000}_{\sqrt{w}} \quad \underbrace{1000}_{\sqrt{w}} \quad \underbrace{1101}_{\sqrt{w}}$$

1. Identifying non-empty clusters. This is done in $O(1)$ time with a series of bit tricks.

(a) Identify which clusters have the first bit set. Compute bitwise AND between x and a constant F to get t_1

$$\begin{array}{rcccc} x = & 0101 & 0000 & 1000 & 1101 \\ F = & 1000 & 1000 & 1000 & 1000 \\ \hline t_1 = & \underline{0}000 & \underline{0}000 & \underline{1}000 & \underline{1}000 \end{array}$$

(b) Identify if the remaining bits (not first bit of a cluster) are set. Compute bitwise XOR between the previous result and x to get t_2.

$$\begin{array}{rcccc} x = & 0101 & 0000 & 1000 & 1101 \\ t_1 = & \underline{0}000 & \underline{0}000 & \underline{1}000 & \underline{1}000 \\ \hline t_2 = & 0\underline{000} & 0\underline{000} & 0\underline{000} & 0\underline{000} \end{array}$$

Now we subtract t_2 from F, and if the 1-bit in a cluster of F end up getting borrowed (so that it becomes a 0), then we know that there was something in the corresponding cluster

$$\begin{array}{rcccc} F = & 1000 & 1000 & 1000 & 1000 \\ t_2 = & 0\underline{000} & 0\underline{000} & 0\underline{000} & 0\underline{000} \\ \hline t_3 = & \underline{0}xxx & \underline{1}000 & \underline{1}000 & \underline{0}xxx \end{array}$$

Finally XOR this result with F, to indicate that the remaining bits for a particular cluster are set

$$\begin{array}{rcccc} F = & 1000 & 1000 & 1000 & 1000 \\ t_3 = & \underline{0}xxx & \underline{1}000 & \underline{1}000 & \underline{0}xxx \\ \hline t_4 = & \underline{1}000 & \underline{0}000 & \underline{0}000 & \underline{1}000 \end{array}$$

(c) Now just OR the results from the previous steps, and this will tell us which clusters have set bits in them.

$$\begin{array}{rcccc} t_1 = & \underline{0}000 & \underline{0}000 & \underline{1}000 & \underline{1}000 \\ t_4 = & \underline{1}000 & \underline{0}000 & \underline{0}000 & \underline{1}000 \\ \hline y = & \underline{1}000 & \underline{0}000 & \underline{1}000 & \underline{1}000 \end{array}$$

We can view y as the summary vector of all the $\sqrt{w}$ clusters.

2. Compute perfect sketch of y. We will need to do this for the next step, where we perform a parallel comparison and need multiple copies of $sketch(y)$ in a single word. Above we computed y which tells us which clusters have bits in them. Unfortunately these bits are spread, but we can compress them into a $\sqrt{w}$ word by using a perfect sketch. Fortunately,

we know exactly how the b_is (the bits that we care about for the sketch) are spaced in this case. We care about the first bit of each $\sqrt{w}$ cluster, which is every other $\sqrt{w}$ bit. So

$$b_i = \sqrt{w} - 1 + i\sqrt{w}$$

To compute the sketch, we claim (without exact proof) that we can use

$$m_j = w - (\sqrt{w} - 1) - j\sqrt{w} + j$$

If we do this, then

$$b_j + m_j = w + (i - j)\sqrt{w} + j$$

will be distinct (no collisions) for $0 \leq i, j < \sqrt{w}$ and also conveniently

$$b_i + m_i = w + i$$

So to get our perfect sketch of $sketch(y)$, we just need to multiply $y \cdot m$ and shift it right by w.

3. Find first 1-bit in $sketch(y)$. This will tell us the first non-empty cluster of x. We use perform a parallel comparison of $sketch(y)$ to all of the $\sqrt{w}$ powers of 2. In our example these are

   0001
   0010
   0100
   1000

   This will tell us the first power of 2 that is greater than $sketch(y)$, which tells us the first set bit in $sketch(y)$. Because we reduced y to $sketch(y)$ which is $\sqrt{w}$ bits, the words generated for parallel comparison take up $\sqrt{w}(\sqrt{w} + 1) < 2w$ bits, less than two words, so we can do this parallel comparison in $O(1)$ time.

4. Now that we know the first cluster c of x that has a set bit, we will find the first set bit d of c. To do this, first shift x right by $c \cdot \sqrt{w}$, bitwise AND the result with $\underbrace{11\ldots11}_{\sqrt{w}\text{ bits}}$ to get just the bits in that cluster. Now we perform the exact same type of parallel comparison as in the previous step, to find the first set bit d.

5. Finally, we compute the index of the most signficant set bit to be $c\sqrt{w} + d$.

Each step along the way takes $O(1)$ time, which makes this take $O(1)$ time overall.

Lecture 13

Integer 3

Scribes: Karan Sagar (2012), Jacob Hurwitz (2012), Jingjing Liu (2010), Meshkat Farrokhzadi (2007), Yoyo Zhou (2005)

13.1 Overview

In the last two lectures, we discussed several data structures for solving predecessor and successor queries in the word RAM model: van Emde Boas trees, y-fast trees, and fusion trees. This establishes an upper bound on the predecessor problem.

In this lecture we discuss lower bounds on the cell-probe complexity of the static predecessor problem with constrained space. In particular, if we are constrained to polynomial space, then we obtain a lower bound of

$$\Omega\left(\min\left\{\frac{\lg w}{\lg\lg n}, \log_w n\right\}\right)$$

using the round elimination technique in a communication model. This shows that the minimum of van Emde Boas trees and fusion trees solves the static predecessor problem optimally, up to a $\lg\lg$ factor.

The lower bound is particularly elegant because it only relies only on information theory. On the other hand, the upper bounds we've seen have relied on the computer technology available, since they use specific bit manipulations and C-style operations.

13.2 Survey of predecessor lower bound results

13.2.1 The problem

Given a data structure (a set S of n w-bit integers) and a query (an element x, possibly not in S), the goal is to find the predecessor of x in S as efficiently as possible. Observe that with $O(2^w)$ space

we can precompute and store all the results to achieve constant query time; to avoid trivializing the problem, we assume $O(n^{O(1)})$ space for our data structures.

The results we are about to discuss are actually for an easier problem, colored predecessor. In colored predecessor, each element of S is colored red or blue, and the goal is to return the *color* of the predecessor of x in S. Since we can solve the colored predecessor problem using a solution to the predecessor problem, this gives a stronger lower bound for our original problem.

13.2.2 Results

- Ajtai [Ajt88] proved the first $\omega(1)$ (that is, superconstant) lower bound, claiming that $\forall w, \exists n$ that gives $\Omega(\sqrt{\lg w})$ query time.
- Miltersen [Mil94] rephrased the same proof ideas in terms of communication complexity and then showed that $\forall w$, $\exists n$ that gives $\Omega(\sqrt{\lg w})$ query time, and $\forall n$, $\exists w$ that gives $\Omega(\sqrt[3]{\lg n})$ query time.
- Miltersen, Nisan, Safra, and Widgerson [MNSW95, MNSW98] introduced the round elimination technique and used it to give a clean proof of the same lower bound.
- Beame and Fich [BF99, BF02] proved two strong bounds. They showed that $\forall w, \exists n$ that gives $\Omega\left(\frac{\lg w}{\lg\lg w}\right)$ query time, and $\forall n, \exists w$ that gives $\Omega\left(\sqrt{\frac{\lg n}{\lg\lg n}}\right)$ query time. They also gave a static data structure achieving $O\left(\min\left\{\frac{\lg w}{\lg\lg w}, \sqrt{\frac{\lg n}{\lg\lg n}}\right\}\right)$, which shows that these bounds are optimal if we insist on pure bounds (bounds dependent only on n or only on w).
- Xiao [Xia92] independently proved the same lower bound as Beame and Fich.
- Sen and Venkatesh [Sen03, SV08] gave a stronger version of the round elimination lemma that we about to introduce in this lecture, which gives a cleaner proof of the same bounds.
- Patrascu and Thorup [PT06, PT07] gave a tight trade-off between n, w, and space for the static problem. Letting the space be $n \cdot 2^a$, they found a lower bound of

$$\Theta\left(min\left\{\log_w n, \lg\left(\frac{w-\lg n}{a}\right), \frac{\lg\frac{w}{a}}{\lg\left(\frac{a}{\lg n}\lg\frac{w}{a}\right)}, \frac{\lg\frac{w}{a}}{\lg\left(\lg\frac{w}{a}/\lg\frac{\lg n}{a}\right)}\right\}\right).$$

  The first term looks like fusion trees, the second like van Emde Boas, and the other two terms don't have good intuition but show that we can do a little better for certain space values. Given $O(n \operatorname{polylg} n)$ space, we have $a = O(\lg\lg n)$, implying a bound of $\Theta\left(\min\left\{\log_w n, \frac{\lg w}{\lg\frac{\lg w}{\lg\lg n}}\right\}\right)$. This shows that van Emde Boas trees are optimal if $w = O(\operatorname{polylg} n)$, and fusion trees are optimal if $\lg w = \Omega(\sqrt{\lg n}\lg\lg n)$.

13.3 Communication Complexity

13.3.1 Communication complexity view point

We consider the problem in the communication complexity model. In this model, Alice knows a value x and Bob knows a value y. Collectively, they would like to compute some function $f(x, y)$. However, they are limited to a protocol in which they alternate sending messages to each other; furthermore, Alice's messages can only be a bits long, and Bob's messages can only be b bits long.

Applying this model to the colored predecessor problem, we can think of Alice as the query algorithm, and she knows the query x. Bob is the memory/RAM, and he knows the data structure y. Together, they want to find $f(x, y)$, which is the answer to the colored predecessor query. Intuitively Alice wants to "ask" Bob for values from the data structure so she can compute the answer. Thus, Alice's messages will be address bits, so $a = O(\lg(\text{space}))$. Bob will return the values stored in the data structure, so $b = w$, the word size. Each "round" of communication consists of two messages and corresponds to one probe in the cell probe model. (This is a static problem, so we only have cell probe loads, not writes.) Thus, the number of messages equals twice the number of cell probes. If we can establish a lower bound in the cell probe model, that will imply a lower bound in the word RAM model.

13.3.2 Predecessor lower bound

Claim: The number of messages (and therefore number of cell probes) needed in the communication model is $\Omega(\min\{\lg_a w, \lg_b n\})$.

Corollary: This implies the Beame-Fich-Xiao lower bound of $\Omega\left(\min\left\{\frac{\lg w}{\lg\lg w}, \sqrt{\frac{\lg n}{\lg\lg n}}\right\}\right)$.

Assuming polynomial space, $a = \Theta(\lg n)$. The lower bound is largest when $\log_a w \stackrel{\Theta}{=} \log_b n$, where $\stackrel{\Theta}{=}$ denotes equality up to a constant factor. Solving, we find:

$$\frac{\lg w}{\lg\lg n} \stackrel{\Theta}{=} \frac{\lg n}{\lg w} \Leftrightarrow \lg w \stackrel{\Theta}{=} \sqrt{\lg n \lg\lg n} \Leftrightarrow \lg\lg w \stackrel{\Theta}{=} \lg\lg n$$

Using this equivalence, we now rewrite $\Omega(\min\{\lg_a w, \lg_b n\})$ as

$$\Omega\left(\min\left\{\frac{\lg w}{\lg a}, \frac{\lg n}{\lg w}\right\}\right) = \Omega\left(\min\left\{\frac{\lg w}{\lg\lg n}, \frac{\lg n}{\sqrt{\lg n \lg\lg n}}\right\}\right) = \Omega\left(\min\left\{\frac{\lg w}{\lg\lg w}, \sqrt{\frac{\lg n}{\lg\lg n}}\right\}\right).$$

This representation makes it easiest to see the Beame-Fich-Xiao lower bound. However, using some of the intermediate steps in the above calculation, we can also rewrite the bound as

$$\Omega\left(\min\left\{\frac{\lg w}{\lg\lg n}, \log_w n\right\}\right),$$

which is the format we presented in the overview at the beginning of lecture. This representation makes it easy to see that the first argument to the min function looks (up to a $\lg\lg$ factor) like van Emde Boas complexity, and the second argument looks like fusion tree complexity.

13.4 Round Elimination

Let's return to the abstract communication model (not necessarily related to the predecessor problem) to discuss round elimination. Round elimination gives some conditions under which the first round of communication can be eliminated. To do this, we consider the "k-fold" of an arbitrary function f:

Definition 35. *Let $f^{(k)}$ be a variation on f, in which Alice has the k inputs $x_1, \ldots, x_k$, and Bob has inputs: y, $i \in 1, \ldots k$, and $x_1, \ldots, x_{i-1}$ (note that this overlaps with Alice's inputs). The goal is to compute $f(x_i, y)$.*

Now assume Alice must send the first message. Observe that she must send this message even though she doesn't know i yet. Intuitively, if $a \ll k$, she is unlikely to send anything useful about x_i, which is the only part of her input that matters. Thus, we can treat the communication protocol as starting from the second message.

Lemma 36 (Round Elimination Lemma)**.** *Assume there is a protocol for $f^{(k)}$ where Alice speaks first that uses m messages and has error probability δ. Then there is a protocol for f where Bob speaks first that uses $m-1$ messages and has error probability $\delta + O(\sqrt{a/k})$.*

Intuition: If i were chosen uniformly at random (which is the worst case), then in Alice's first message the expected number of bits "about x_i" is a/k. Bob can guess these bits at random; the probability of guessing all bits correctly is $1/2^{a/k}$, so the probability of failure is $1 - 2^{-a/k}$. Because we are interested in small a/k, a series expansion shows that $1 - 2^{-a/k} \approx a/k$. Thus, by eliminating Alice's message, the error probability should increase by about a/k. In reality, this intuition is not entirely correct, and we can only bound the increase in the error by $\sqrt{a/k}$, which is often acceptable depending on the application.

13.5 Proof of Predecessor Bound

Proof sketch: Let t be the number of cell probes, or equivalently, the number of rounds of communication. Our goal is to apply the Round Elimination Lemma $2t$ times to eliminate all the messages. At this point, the color of the predecessor must be guessed (assuming $n' \geq 2$, which we'll define soon), and so the probability of success is at most $\frac{1}{2}$.

If we can bound the increase in error probability by at most $1/6t$ at each step, then $2t$ applications of the Round Elimination Lemma will increase the error probability from 0 to at most $\frac{1}{3}$. However, this yields a probability of success of at least $\frac{2}{3}$, which is a contradiction.

In other words, *no* algorithm with t cell probes can solve this problem, so t is a lower bound on the expected performance of any static randomized colored predecessor data structure.

13.5.1 Eliminating Alice→Bob

Alice's input has w' bits (initially, $w' = w$). Divide it into $k = \Theta(at^2)$ equal-size chunks $x_1, \ldots, x_k$. Each chunk is of w'/k bits. If we can bound the error increase by $O(1/t)$, then tweaking the constants will give $1/6t$ as desired.

We construct a tree with branching factor $2^{w'/k}$ on the w'-bit strings corresponding to the Alice's possible inputs, which are the elements of the data structure. The tree then has height k. In the worst case, we can constrain the n' (initially, $n' = n$) elements to all differ in ith chunk. Alice and Bob know the structure of the inputs, so Bob knows i and the common prefix of all elements $x_1, \ldots, x_{i-1}$. Thus, when Alice's message is eliminated, the goal changes to query x_i in data structure for ith chunk, and w' is reduced to $w'/k = \Theta(w'/at^2)$.

An analogy of this data structure is van Emde Boas tree since vEB binary searches on levels to find longest prefix match, reducing w' as it goes. Using the lemma, the error probability increases by $O(\sqrt{a/at^2}) = O(1/t)$, which is exactly what we can afford per elimination.

13.5.2 Eliminating Bob→Alice

Now that Alice's message is eliminated, Bob is speaking first, so he doesn't know the query's value. Bob's input is n' integers each of size w' bits. Divide the integers into $k = \Theta(bt^2)$ equal chunks of n'/k integers each. Remember that fusion trees could recurse in a set of size $n/w^{1/5}$ after $O(1)$ cell probes. Here, we are proving that after one probe, you can only recurse into a set of size $n/w^{O(1)}$, which gives the same bound for error increase, which is $O(1/t)$.

To get lower bound, constrain input such that ith chunk x_i starts with prefix i in binary. Alice's query starts with some random $\lg k$ bits, which decides which chunk is interesting. If Bob speaks first, he cannot know which chunk is interesting,

So using the lemma, the elimination rises error probability by $O(1/t)$; reduces n' to $n'/k = \Theta(n'/bt^2)$ and w' to $w' - \lg k = w' - \Theta(\lg bt^2)$. As long as w' does not get too small, $w = \Omega(\lg(bt^2))$, this last term is negligible (say, it reduces w' by a factor of at most 2).

13.5.3 Stopping

Thus, each round of eliminations reduces n' to $\Theta(n'/bt^2)$ and w' to $\Theta(w'/at^2)$. Further, the probability of error at the end can be made to be at most $\frac{1}{3}$ by choosing appropriate constants.

We stop the elimination when $w' = O(\lg(bt^2))$ or $n' = 2$. If these stop conditions are met, we have proven our lower bound: there were many rounds initially, so we could do enough eliminations to reduce n and w to these small values. Otherwise, we have a protocol which gives an answer with zero messages, and the error probability is at most $\frac{1}{3}$, which is impossible. So we must be in the first case (the stop conditions are met).

Hence, we have established a lower bound $t = \Omega(\min\{\lg_{at^2} w, \lg_{bt^2} n\})$. However, because $t = O(\lg n)$ and $a \geq \lg n$, we have $a \leq at^2 \leq a^3$. Likewise, because $t = O(\lg w)$ and $b = w$, we have $b \leq bt^3 \leq b^3$. Thus, we can conclude that $t = \Omega(\min\{\lg_a w, \lg_b n\})$.

13.6 Sketch of the Proof for the Round Elimination Lemma

We ran out of time to discuss this section in class, so it does not appear in the lecture video.

13.6.1 Some Information Theory Basics

Definition 37. *$H(x)$, called the entropy of x, is the number of bits needed on average to represent a sample from a distribution of the random variable x. Formally,*

$$H(x) = \sum_{x_0} \Pr[x = x_0] \cdot \lg \frac{1}{\Pr[x = x_0]}$$

Definition 38. *$H(x \mid y)$ is the conditional entropy of x given y: the entropy of x, if y is known:*

$$H(x \mid y) = E_{y_0}[H(x|y = y_0)]$$

Definition 39. *$I(x : y)$ is the mutual or shared information between x and y:*

$$I(x : y) = H(x) + H(y) - H((x, y)) = H(x) - H(x \mid y)$$

$I(x : y \mid z)$ is defined in a manner similar to that of $H(x \mid y)$.

13.6.2 The Round Elimination Lemma

Call Alice's first message $m = m(x_1, \ldots, x_k)$. Next, we use a neat theorem from information theory to rewrite entropy as a sum:

$$a = |m| \geq H(m) = \sum_{i=1}^{k} I(x_i : m \mid x_1, \ldots, x_{i-1})$$

If i is distributed uniformly in $\{1, \ldots, k\}$, then $E_i[I(x : m \mid x_1, \ldots, x_k)] = \frac{H(m)}{k} \leq \frac{a}{k}$. This is why $\frac{a}{k}$ was an estimate for how many bits of information Bob could learn from the message about Alice's message. Note that we bounded $I(x_i : m \mid x_1, \ldots, x_{i-1})$, so even if Bob already knows $x_1, \ldots, x_{i-1}$ and receives m, he still learns at most $\frac{a}{k}$ bits about x_i.

The prove the lemma, we must build a protocol for f given the assumed protocol for $f^{(k)}$. We can build a protocol $f(x, y)$ as follows:

1. Fix $x_1, \ldots, x_{i-1}$ and i a priori (known to both players) at random.
2. Alice pretends $x_i = x$.
3. Run the $f^{(k)}$ protocol, starting at the second message, by assuming the first message is $m = m(x_1, \ldots, x_{i-1}, \tilde{x}_i, \ldots, \tilde{x}_k)$, where $\tilde{x}_j$ is a random variable drawn from the distribution of x_j. Now the first message does not depend on $x_i = x$ (even x_i is chosen randomly), so Bob can generate it by himself, without any initial message from Alice.
4. Now Alice has some actual x, which she must use as x_i, and almost certainly $\tilde{x}_i \neq x$. But we know that $I(x_i : m)$ is very small, so the message doesn't really depend on x_i in a crucial way. This means that a random message was probably good: Alice can now fix $x_{i+1}, \ldots, x_k$, so that $m(x_1, \ldots, x_{i-1}, \tilde{x}_i, \ldots, \tilde{x}_k) = m(x_1, \ldots, x_{i-1}, x, \ldots, x_k)$, for the desired $x_i = x$.

The last step is the crucial one which also introduces an error probability of $O(\sqrt{a/k})$. This can be proved based on the "Average Encoding Theorem" from information theory. There also exists a more subtle problem that this theorem solves: not only must $x_{i+1}, \ldots, x_k$ exist, so that a Bob's random guess for a message is made valid, but their distributions are close the the original distributions, so the error probability δ does not increase too much.

Lecture 14

Integer 4

Scribes: Leon Bergen (2012), Andrea Lincoln (2012), Tana Wattanawaroon (2012), Haitao Mao (2010), Eric Price (2007)

14.1 Overview

Today we are going to go over:

- The reduction between sorting and priority queues
- A survey of sorts
- Bitonic Sequences
- Logarithmic Merge Operation
- Packed Sorting
- Signature Sort Algorithm

The reduction and survey will serve as motivation for signature sort. Bitonic Sequences will be used to build Logarithmic Merge which will in turn be used to build Packed Sorting which will in finally be used to build Signature Sort.

14.2 Sorting reduced to Priority Queues

Thorup [72] showed that if we can sort n w-bit integers in $O(nS(n, w))$, then we have a priority queue that can support the insertion, deletion, and find minimum operations in $O(S(n, w))$. To get a constant time priority queue, we need linear time sorting.
OPEN: linear time sorting

14.3 Sorting reduced to Priority Queues

Following is a list of results outlining the current progress on this problem.

- Comparison model: $O(n \lg n)$
- Counting sort: $O(n + 2^w)$
- Radix sort: $O(n \cdot \frac{w}{\lg n})$
- van Emde Boas: $O(n \lg w)$, improved to $O(n \lg \frac{w}{\lg n})$ (see [71]).
- Signature sort: linear when $w = \Omega(\lg^{2+\varepsilon} n)$ (see [67]).
- Han [69]: $O(n \lg \lg n)$ deterministic, AC^0 RAM.
- Han and Thorup: $O(n\sqrt{\lg \lg n})$ randomized, improved to $O(n\sqrt{\lg \frac{w}{\lg n}})$ (see [70] and [71])).

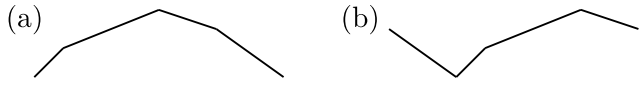
Today, we will focus entirely on the details of the signature sort. This algorithm works whenever $w = \Omega(\log^{2+\varepsilon} n)$. Radix sort, which we should already know, works for smaller values of w, namely when $w = O(\log n)$. For all other values of w and n, it is open whether we can sort in linear time. We previously covered the van Emde Boas tree, which allows for $O(n \log \log n)$ sorting whenever $w = \log^{O(1)} n$. The best we have done in the general case is a randomized algorithm in $O(n\sqrt{\log \frac{w}{\log n}})$ time by Han, Thorup, Kirkpatrick, and Reisch.
OPEN: $O(n)$ time sort $\forall w$

14.4 Sorting for $w = \Omega(\log^{2+\varepsilon} n)$

The signature sort was developed in 1998 by Andersson, Hagerup, Nilsson, and Raman [67]. It sorts n w-bit integers in $O(n)$ time when $w = \Omega(\log^{2+\varepsilon} n)$ for some $\varepsilon > 0$. This is a pretty complicated sort, so we will build the algorithm from the ground up. First, we give an algorithm for sorting bitonic sequences using methods from parallel computing. Second, we show how to merge two words of $k \leq \log n \log \log n$ elements in $O(\log k)$ time. Third, using this merge algorithm, we create a variant of MERGESORT called *packed sorting*, which sorts n b-bit integers in $O(n)$ time when $w \geq 2(b+1) \log n \log \log n$. Fourth, we use our packed sorting algorithm to build signature sort.

14.4.1 Bitonic Sequences

A *bitonic sequence* is a cyclic shift of a monotonically increasing sequence followed by a monotonically decreasing sequence. When examined cyclically, it has only one local minimum and one local maximum.



Cyclic shifts preserve the binoticity of a sequence.

To sort a bitonic sequence `btcseq`, we run the algorithm shown below in the figure below. Assume $n = \text{len(btcseq)}$ is even.

Algorithm 1 Bitonic sequence sorting algorithm

```
btcsort(btcseq):
  for i from 0 to n/2-1:
    if btcseq[i]>btcseq[i+n/2]:
      swap(btcseq[i], btcseq[i+n/2])
  btcsort(btcseq[0:n/2-1])
  btcsort(btcseq[n/2:n-1])
```

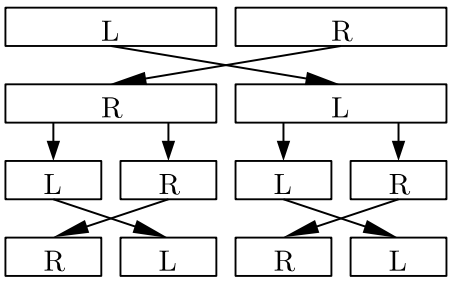
Bitonic sequence sorting maintains two invariants: (a) the sequence `btcseq` contains multiple sections of bitonic sequences (each level of `btcsort` splits a bitonic sequence into two bitonic sequences), and (b) when considering two sections of bitonic sequences, an element in the left section is always smaller than an element in the right section.

After $O(\log n)$ rounds, `btcseq` is broken into n sections. Since the left section has elements smaller than the right section, sorting is complete.

For more information about sorting bitonic sequences, including a proof of the correctness of this algorithm, see [68, Section 27.3].

14.4.2 Logarithmic Merge Operation

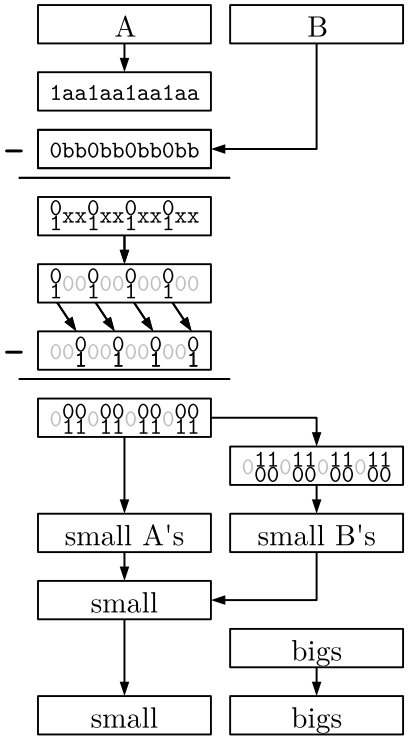
The next step is to merge two sorted words, each containing k b-bit elements. First, we concatenate the first word with the reverse of the second word, getting a bitonic sequence. To efficiently reverse a word, we mask out the leftmost $\frac{k}{2}$ elements and shift them right by $\frac{k}{2}b$, then mask out the rightmost $\frac{k}{2}$ elements and shift them left by $\frac{k}{2}b$. Taking the OR of the two resulting words leaves us with the original word with the left and right halves swapped. We can now recurse on the left and right halves of the word, giving us the recursion $T(n) = T(\frac{k}{2}) + O(1)$, so the whole algorithm takes $T(k) = O(\log k)$ time. The two words may now be concatenated by shifting the first word left by kb and taking its OR with the second word. The key here is to perform each level of the recursion in parallel, so that each level takes the same amount of time. This list reversal is illustrated in the figure below.



The first two steps in the recursion for reversing a list.

All that remains is to run the bitonic sorting algorithm on the elements in our new word. To do so, we must divide the elements in two halves and swap corresponding pairs of elements which are out of order. Then we can recurse on the first and second halves in parallel, once again giving us the recursion $T(k) = 2T(\frac{k}{2}) + O(1) \Rightarrow T(k) = O(\log k)$ time. Thus we need a constant-time operation which will perform the desired swapping.

Assume we have an extra 0 bit before each element packed into the word. We use this spare bit to help bitmask our word. We will mask the left half of the elements and set this extra bit to 1 for each element, then mask the right half of the elements and shift them left by $\frac{k}{2}b$. After we subtract the second word from the first, a 1 will appear in the extra bit iff the element in the corresponding position of the left half is greater than the element in the right half. Thus we can mask the extra bits, shift the word right by $b - 1$ bits, and subtract it from itself, resulting in a word which will mask all the elements of the right half which belong in the left half and vice versa. From this, we use bit shifts, OR, and negation to get our sorted list. See the diagram below for clarification; the process is fairly straightforward.



Parallel swap operation in bitonic sorting.

Using these bit tricks, we end up with our desired constant time all-swap operation. This leads to the following theorem:

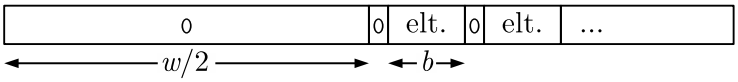
Theorem 40. *Suppose two words each contain k b-bit elements. Then we can merge these words in $O(\log k)$ time.*

Proof. As already noted, we can concatenate the first word with the reverse of the second word in $O(\log k)$ time. We can then apply the bitonic sorting algorithm to these concatenated words.

Since this algorithm has a recursion depth of $O(\log k)$, and we can perform the required swaps in constant time, our merge operation ends up with time complexity $O(\log k)$. □

14.4.3 Packed Sorting

In this section we will present packed sorting [66], which sorts n b-bit integers in $O(n)$ time for a word size of $w \geq 2(b+1)\log n \log\log n$. This bound for w allows us to pack $k = \log n \log\log n$ elements into one word, leaving a zero bit in front of each integer, and $\frac{w}{2}$ zero bits at the beginning of the word.



Structure for packing b-bit integers into a w-bit word.

To sort these elements using packed sorting, we build up merge operations as follows:

1. Merge two sorted words in $O(\lg k)$ time, as shown with bitonic sorting in the previous section.
2. Merge sort on k elements with (1) as the merge operation. This sort has the recursion $T(k) = 2T(\frac{k}{2}) + O(\lg k) \Rightarrow T(k) = O(k)$.
3. Merge two sorted lists of r sorted words into one sorted list of $2r$ sorted words. This is done in the same manner as in a standard merge sort, except we use (1) to speed up the operation. We start by merging the left-most words in the two lists using (1). The first word following this merger must consist of the smallest k elements overall, so we output this word. However, we cannot be sure about the relative position of the elements in the second word. We therefore examine the maximum element in the second word, and add the second word to the beginning of the list which contained this maximum element. We continue merging the lists in this manner, resulting in $O(r \log k)$ overall.
4. Merge sort all of the words with (3) as the merge operation and (2) as the base case.

Theorem 41. *Let word size $w \geq 2(b+1)\log n \log\log n$. Then packed sorting sorts n b-bit integers in $O(n)$ time.*

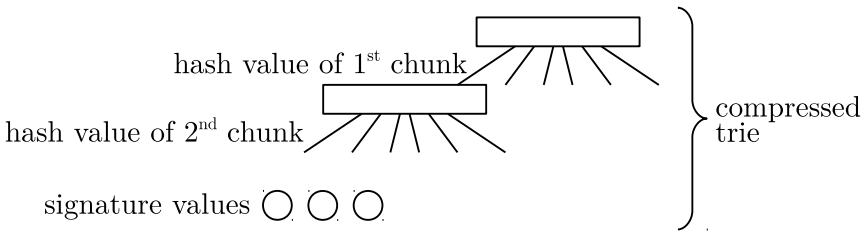
Proof. The sort defined in Step 4 of the algorithm above has the recursion $T(n) = 2T(\frac{n}{2}) + O(\frac{n}{k}\log k)$ and the base case $T(k) = O(k)$. The recursion depth is $O(\log \frac{n}{k})$, and each level takes $O(\frac{n}{k}\lg k) = O(\frac{n}{\lg n})$ time, so altogether they contribute a cost of $O(n)$. There are also $\frac{n}{k}$ base cases, each taking $O(k)$ time, giving a total runtime of $O(n)$, as desired. □

14.4.4 Signature Sort Algorithm

We use our packed sorting algorithm to build our seven step signature sort. We assume that $w \geq \log^{2+\varepsilon} n \log\log n$. The signature sort will sort n w-bit integers in $O(n)$ time. We will start by breaking each integer into $\lg^{\varepsilon} n$ equal-size chunks, which will then be replaced by $O(\lg n)$-bit

signatures via hashing. These smaller signatures can be sorted with packed sorting in linear time. Because hashing does not preserve order, we will build a compressed trie of these sorted signatures, which we will then sort according to the values of the original chunks. This will allow us to recover the sorted order of the integers. The signature sort will proceed as follows:

1. Break each integer into $\lg^{\varepsilon} n$ equal-size chunks. (Note the distinguishment from a fusion tree, which has chunks of size $\lg^{\varepsilon} n$)
2. Replace each chunk by a $O(\lg n)$-bit hash (static perfect hashing is fine). By doing this, we end up with n $O(\lg^{1+\varepsilon} n)$-bit *signatures*. One way we can hash is to multiply by some random value x, and then mask out the hash keys. This will allow us to hash in linear time. Now, our hash does not preserve order, but the important thing is that it does preserve identity.
3. Sort the signatures in linear time with packed sorting, shown above.
4. Now we want to rescue the identities of the signatures. Build a compressed trie over the signatures, so that an inorder traversal of the trie gives us our signatures in sorted order. The compressed trie only uses $O(n)$ space.



A trie for storing signatures. Edge represent hash values of chunks; leaves represent possible signature values.

   To do this in linear time, we add the signatures in order from left to right. Since we are in the word RAM, we can compute the LCP with $(i-1)^{\text{st}}$ signature by taking the most significant 1 bit of the XOR. Then, we walk up the tree to the appropriate node, and charge the walk to the decrease in the rightmost path length of the trie. The creation of the new branch is constant time, so we get linear time overall. This process is similar to the creation of a Cartesian tree.

5. Recursively sort the edges of each node in the trie based on their actual values. This is a recursion on (node ID, actual chunk, edge index), which takes up $(O(\log n), O(\frac{w}{\log^{\varepsilon} n}), O(\log n))$ space. The edge indices are in there to keep track of the permutation. After a constant $\frac{1}{\varepsilon}+1$ levels of recursion, we will have $b = O(\log n + \frac{w}{\log^{1+\varepsilon} n}) = O(\frac{w}{\log n \log\log n})$, so we can use packed sorting as the base case of the recursion.
6. Permute the child edges.
7. Do an inorder traversal of the trie to get the desired sorted list from the leaves.

Putting these steps together, we get:

Theorem 42. *Let $w \geq \log^{2+\varepsilon} n \log\log n$. Then signature sort will sort n w-bit integers in $O(n)$ time.*

Proof. Breaking the integers into chunks and hashing them (Steps 1 and 2) takes linear time. Sorting these hashed chunks using packed sort (Step 3) also takes linear time. Building the compressed trie over signatures takes linear time (Step 4), and sorting the edges of each node (Step 5) takes constant time per node for a linear number of nodes. Finally, scanning and permuting the nodes (Step 6) and the in-order traversal of the leaves of the trie (Step 7) will each take linear time, completing the proof. □

Lecture 15

Static Trees

Scribes: Jelle van den Hooff(2012), Yuri Lin(2012) Andrew Winslow (2010)

15.1 Overview

In this lecture, we look at various data structures for static trees, in which, given a static tree, we perform some preprocessing and then answer queries on the data structure. The three problems we look at in this lecture are range minimum queries (RMQ), least common ancestors (LCA), and level ancestors (LA); we will support all these queries in constant time per operation and linear space.

15.1.1 Range Minimum Query (RMQ)

In the range minimum query problem, we are given (and we preprocess) an array A of n numbers. In a query, the goal is to find the minimum element in a range spanned by $A[i]$ and $A[j]$:

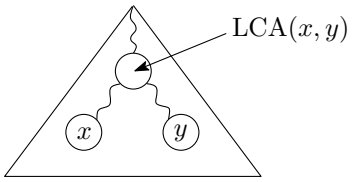
$$\begin{aligned} \text{RMQ}(i,j) &= (\arg)\min\{A[i], A[i+1], \ldots, A[j]\} \\ &= k, \text{ where } i \leq k \leq j \text{ and } A[k] \text{ is minimized} \end{aligned}$$

We care not only about the value of the minimum element, but also about the index k of the minimum element between $A[i]$ and $A[j]$; given the index, it is easy to look up the actual value of the minimum element, so it is a more interesting problem to find the index of the minimum element between $A[i]$ and $A[j]$.

The range minimum query problem is not a tree problem, but it closely related to a tree problem (least common ancestor).

15.1.2 Lowest Common Ancestor (LCA)

In the lowest common ancestor problem, we want to preprocess a rooted tree T with n nodes. In a query, we are given two nodes x and y and the goal is to find their lowest common ancestor in T:

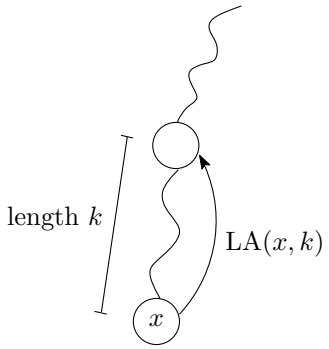


15.1.3 Level Ancestor (LA)

Finally, we will also solve the level ancestor problem, in which we are again given a rooted tree T. Given a node x and an integer k, the query goal is to find the k^{th} ancestor of node x:

$$\text{LA}(x, k) = \text{parent}^k(x)$$

Of course, k cannot be larger than the depth of x.



All of these problems will be solved in the word RAM model, though the use of model is not as essential as it has been in the integer data structures we have discussed over the previous lectures. Although lowest common ancestor and level ancestor seem like similar problems, fairly different techniques are necessary to solve them as far as anyone knows. The range minimum query problem, however, is basically identical to that of finding the lowest common ancestor.

15.2 Reductions between RMQ and LCA

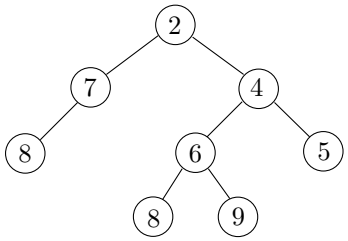
15.2.1 Cartesian Trees: Reduction from RMQ to LCA

A Cartesian tree is a nice reduction mechanism from an array A to a binary tree T, dating back to a 1984 paper by Gabow, Bentley, and Tarjan [73], and provides an equivalence between RMQ and LCA.

To construct a Cartesian tree, we begin with the minimum element of the array A, which we can call $A[i]$. This element becomes the root of the Cartesian tree T. Then the left subtree of T is a Cartesian tree on all elements to the left of $A[i]$ (which we can write as $A[< i]$), and the right subtree of T is likewise a Cartesian tree on the elements $A[> i]$.

An example is shown below for the array $A = [8, 7, 2, 8, 6, 9, 4, 5]$. The minimum of the array is 2, which gets promoted to the root. This decomposes the problem into two halves, one for the left subarray $[8, 7]$ and one for the right subarray $[8, 6, 9, 4, 5]$. 7 is the minimum element in the left subarray and becomes the left child of the root; 4 is the minimum element of the right subarray and is the right child of the root. This procedure continues until we get the binary tree in the diagram below.

$[8, 7, 2, 8, 6, 9, 4, 5]$



The resulting tree T is a min heap. More interestingly, the result range minimum query for a given range in the array A is the lowest common ancestor of those two endpoints in the corresponding Cartesian tree T.

In the case of ties between multiple equal minimum elements, two options are:

1. Break ties arbitrarily, picking one of the equal elements as the minimum.
2. Consider all of the equal elements to be one "node", making a non-binary tree.

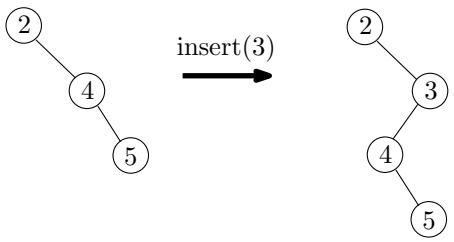
Because the second option is silghtly messier, in this class we will break ties arbitrarily, though either option will not affect the answer.

Construction in linear time

Construction of the Cartesian tree according to the naive recursive algorithm will take at least $n \lg n$ time, and may even take quadratic time. In fact, Cartesian trees can be computed in linear time, using a method that is basically the same as the method seen in the last lecture for building a compressed trie in linear time.

Walk through the array from left to right, inserting each element into the tree by walking up the right spine of the tree (starting from the leaf), and inserting the element in the appropriate place. Because we are building the tree from left to right, each inserted element will by definition be the rightmost element of the tree created so far.

For example, if we have a tree for which the subarray $[2, 4, 5]$ has been inserted, and the next element is 3, then insertion has the following result:



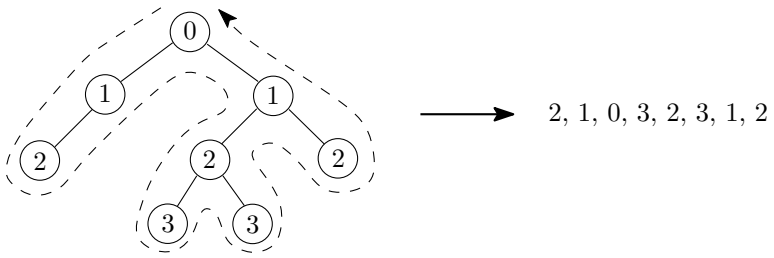
We walk up the right spine of the tree starting at 5, and continue until we reach 2 (the root), which is the first element smaller than 3. The edge between 2 and 4 is replaced with a new edge from 2 to 3, and the previous right subtree of 2 becomes the left subtree of 3.

Each such insertion takes constant amortized time; although sometimes paths may be long, each insertion only touches nodes along the right spine of the tree, and any node along the right spine that has been touched ends up in the left subtree of the inserted node. Any node along the right spine is touched at most once, and we can charge the expensive inserts to the decrease in length of the right spine.

Therefore, construction of Cartesian trees can be done in linear time, even in the comparison model.

15.2.2 Reduction from LCA to RMQ

We can also reduce in the other direction, reducing from LCA to RMQ by reconstructing an array A when we are given a binary tree T. To do this, we do an in-order traversal of the nodes in the tree. However, we must have numbers to use as the values of the array; to this end, we label each node with its depth in the tree.



This sequence behaves exactly like the original array $A = [8, 7, 2, 8, 6, 9, 4, 5]$, from which this tree was constructed. The result for $\text{RMQ}(i, j)$ on the resulting array A is the same as calling $\text{LCA}(i, j)$ on the input tree for the corresponding nodes.

15.2.3 RMQ universe reduction

An interesting consequence of the reductions between RMQ and LCA is that they allow us to do universe reductions for RMQ problems. There are no guarantees on the bounds of any numbers in the original array given for range minimum queries, and in general the elements may be in any arbitrary ordered universe. By chaining the two reductions above, first by building a Cartesian tree

from the elements and then by converting back from the tree to an array of depths, we can reduce the range to the set of integers $\{0, 1, \ldots, n-1\}$.

This universe reduction is handy; the algorithms described above assume a comparison model, but after the pair of reductions we can now assume that all of the inputs are small integers, which allows us to solve things in constant time in the word RAM model.

15.3 Constant time LCA and RMQ

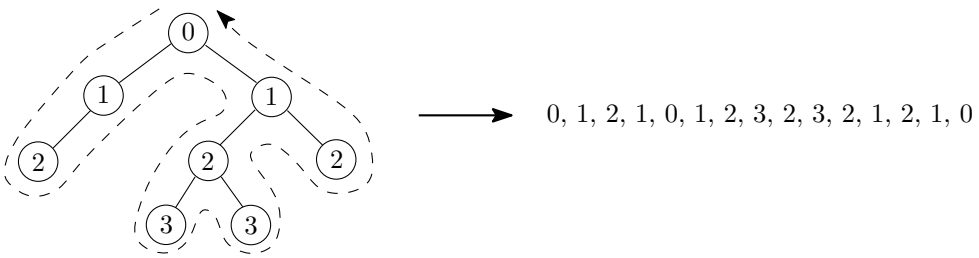
15.3.1 Results

The LCA and RMQ problems can both be optimally solved with constant query time and linear storage space. The first known technique is documented in a 1984 paper by Harel and Tarjan, [74]. In lecture we looked at an algorithm based on a 2004 paper by Bender and Colton, [75].

15.3.2 Reduction from LCA to ± 1 RMQ

The algorithm by Bender and Colton solves a special case of the RMQ problem where adjacent values differ by either $+1$ or -1 called ± 1 RMQ.

First, we will take a look at a reduction from LCA to ± 1 RMQ. The earlier reduction does not work as differences might have absolute values larger than 1. For our new approach, we perform an Eulerian tour based on the in-order traversal and write down every visit in our LCA array. At every step of the tour we either go down a level or up a level, so the difference between two adjacent values in our array is either $+1$ or -1.



Since every edge is visited twice, we have more entries in the array than in the original in-order traversal, but still only $O(N)$. To answer $\text{LCA}(x, y)$, calculate $\text{RMQ}(\text{in-order}(x), \text{in-order}(y))$ where $\text{in-order}(x)$ is the in-order occurrence of the node x in the array. These occurrences can be stored while creating the array. Observe that this new array can also be created by filling in the gaps in the array from the original algorithm. Thus the minimum between any two original values does not change, and this new algorithm also works.

15.3.3 Constant time, $n \lg n$ space RMQ

A simple datastructure than can answer RMQ queries in constant time but uses only $n \lg n$ space can be created by precomputing the RMQ for all intervals with lengths that are powers of 2. There

are a total of $O(n \lg n)$ such intervals, as there are $\lg n$ intervals with lengths that are powers of 2 no longer than n, with n possible start locations.

We claim that any queried interval is the (non-disjoint) union of two power of 2 intervals. Say the query has length k. Then the query can be covered by the two intervals of length $2^{\lfloor \lg k \rfloor}$ that touch the beginning and ending of the query. The query can be answered by taking the min of the two precomputed answers. Observe that this works because we can take the min of an element twice without any problems. Furthermore, this also enables us to store the location of the minimum element.

15.3.4 Indirection to remove log factors

We have seen indirection used to remove log factors of time, but here we will apply indirection to achieve a $O(n)$ space bound. Divide the array into bottom groups of size $\frac{1}{2} \lg n$ (this specific constant will be used later). Then, store a parent array of size $2n/\lg n$ that stores the min of every group.

Now a query can be answered by finding the RMQ of a sequence in the parent array, and at most two RMQ queries in bottom groups. Note that we might have to answer two sided queries in a bottom group for sufficiently small queries. For the parent array we can use the $n \lg n$ space algorithm as the logarithms cancel out.

15.3.5 RMQ on very small arrays

The only remaining question is how to solve the RMQ problem on arrays of size $n' = \frac{1}{2} \lg n$. The idea is to use lookup tables, since the total number of different possible arrays is very small.

Observe that we only need to look at the relative values in a group to find the location of the minimum element. This means that we can shift all the values so that the first element in the group is 0. Then, once we know the location, we can look in the original array to find the value of the minimum element.

Now we will use the fact the array elements differ by either $+1$ or -1. After shifting the first value to 0, there are now only $2^{\frac{1}{2} \lg n} = \sqrt{n}$ different possible bottom groups since every group is completely defined by its n' long sequence of $+1$ and -1s. This total of $\sqrt{n}$ is far smaller than the actual number of groups!

In fact, it is small enough so that we can store a lookup table for any of the n'^2 possible queries for any of the $\sqrt{n}$ groups in $O(\sqrt{n}(\frac{1}{2} \lg n)^2 \lg \lg n)$ bits, which easily fits in $O(n)$ space. Now every group can store a pointer into the lookup table, and all queries can be answered in constant time with linear space for the parent array, bottom groups, and tables.

15.3.6 Generalized RMQ

We have LCA using ± 1 RMQ in linear space and constant time. Since general RMQ can be reduced LCA using universe reduction, we also have a linear space, constant time RMQ algorithm.

15.4 Level Ancestor Queries (LAQ)

First we introduce notation. Let $h(v)$ be the height of a node v in a tree. Given a node v and level l, $LAQ(v, l)$ is the ancestor a of v such that $h(a) - h(v) = l$. Today we will study a variety of data structures with various preprocessing and query times which answer $LAQ(v, l)$ queries. For a data structure which requires $f(n)$ query time and $g(n)$ preprocessing time, we will denote its running time as $\langle g(n), f(n) \rangle$. The following algorithms are taken from the set found in a paper from Bender and Farach-Colton[76].

15.4.1 Algorithm A: $\langle O(n^2), O(1) \rangle$

Basic idea is to use a look-up table with one axis corresponding to nodes and the other to levels. Fill in the table using dynamic programming by increasing level. This is the *brute force* approach.

15.4.2 Algorithm B: $\langle O(n \log n), O(\log n) \rangle$

The basic idea is to use **jump pointers**. These are pointers at a node which reference one of the node's ancestors. For each node create jump pointers to ancestors at levels $1, 2, 4, \ldots, 2^k$. Queries are answered by repeatedly jumping from node to node, each time jumping more than half of the remaining levels between the current ancestor and goal ancestor. So worst-case number of jumps is $O(\log n)$. Preprocessing is done by filling in jump pointers using dynamic programming.

15.4.3 Algorithm C: $\langle O(n), O(\sqrt{n}) \rangle$

The basic idea is to use a **longest path decomposition** where a tree is split recursively by removing the longest path it contains and iterating on the remaining connected subtrees. Each path removed is stored as an array in top-to-bottom path order, and each array has a pointer from its first element (the root of the path) to it parent in the tree (an element of the path-array from the previous recursive level). A query is answered by moving upwards in this *tree of arrays*, traversing each array in $O(1)$ time. In the worst case the longest path decomposition may result in longest paths of sizes $k, k-1, \ldots, 2, 1$ each of which has only one child, resulting in a tree of arrays with height $O(\sqrt{n})$. Building the decomposition can be done in linear by by precomputing node heights once and reusing them to find the longest paths quickly.

15.4.4 Algorithm D: $\langle O(n), O(\log n) \rangle$

The basic idea is to use **ladder decomposition**. The idea is similar to longest path decomposition, but each path is extended by a factor of two backwards (up the tree past the root of the longest path). If the extended path reaches the root, it stops. From the ladder property, we know that node v lies on a longest path of size at least $h(v)$. As a result, one does at most $O(\log n)$ ladder jumps before reaching the root, so queries are done in $O(\log n)$ time. Preprocessing is done similarly to Algorithm C.

15.4.5 Algorithm E: $\langle O(n \log n), O(1) \rangle$

The idea is to combine jump pointers (Algorithm B) and ladders (Algorithm D). Each query will use one jump pointer and one ladder to reach the desired node. First a jump is performed to get at least halfway to the ancestor. The node jumped to is contained in a ladder which also contains the goal ancestor.

15.4.6 Algorithm F: $\langle O(n), O(1) \rangle$

An algorithm developed by Dietz[77] also solves LCA queries in $\langle O(n), O(1) \rangle$ but is more complicated. Here we combine Algorithm E with a reduction in the number of nodes for which jump pointers are calculated. The motivation is that if one knows the level ancestor of v at level l, one knows the level ancestor of a descendant of v at level l'. So we compute jump pointers only for leaves, guaranteeing every node has a descendant in this set. So far, preprocessing time is $O(n + L \log n)$ where L is the number of leaves. Unforunately, for an arbitrary tree, $L = O(n)$.

Building a tree with $O(\frac{n}{\log n})$ leaves

Split the tree structure into two components: a **macro-tree** at the root, and a set of **micro-trees** (of maximal size $\frac{1}{4} \log n$) rooted at the leaves of the macro-tree. Consider a depth-first search, keeping track of the orientation of of the ith edge, using 0 for downwards and 1 for upwards. A micro-tree can be described by a binary sequence, e.g. $W = 001001011$ where for a tree of size n, $|W| = 2n - 1$. So an upper bound the number of micro-trees possible is $2^{2n-1} = 2^{2(\frac{1}{4} \log n)-1} = O(\sqrt{n})$. But this is a loose upper bound, as not all binary sequences are possible, e.g. $00000 \ldots$. A valid micro-tree sequences has an equal number of zeros and ones and any prefix of a valid sequence as at least as many zeros as ones.

Use macro/micro-tree for a $\langle O(n), O(1) \rangle$ solution to LAQ

We will use macro/micro-trees to build a tree with $O(\frac{n}{\log n})$ leaves and compute jump pointers only for its leaves ($O(n)$ time). We also compute all microtrees and their look-up tables (see Algorithm A) in $O(\sqrt{n} \log n)$ time. So total preprocessing time is $O(n)$. A query $LAQ(v, l)$ is performed in the following way: If v is in the macro-tree, jump to the leaf descendant of v, then jump from the leaf and climb a ladder. If v is in a micro-tree and $LAQ(v, l)$ is in the micro-tree, use the look-up table for the leaf. If v is in a micro-tree and $LAQ(v, l)$ is not in the micro-tree, then jump to the leaf descendant of v, then jump from the leaf and climb a ladder.

Lecture 16

Strings

Scribes: Cosmin Gheorghe (2012), Nils Molina (2012), Kate Rudolph (2012), Mark Chen (2010), Aston Motes (2007), Kah Keng Tay (2007), Igor Ganichev (2005), Michael Walfish (2003)

16.1 Overview

In this lecture, we consider the string matching problem - finding some or all places in a text where the query string occurs as a substring. From the perspective of a one-shot approach, we can solve string matching in $O(|T|)$ time, where $|T|$ is the size of our text. This purely algorithmic approach has been studied extensively in the papers by Knuth-Morris-Pratt [83], Boyer-Moore [78], and Rabin-Karp [81].

However, we entertain the possibility that multiple queries will be made to the same text. This motivates the development of data structures that preprocess the text to allow for more efficient queries. We will show how to construct, use, and analyze these string data structures.

16.2 Predecessor Problem

Before we get to string matching we will solve an easier problem which is necessary to solve string matching. This is the predecessor problem among strings. Assume we have k strings, $T_1, T_2, ..., T_k$ and the query is: given some pattern P, we would like to know where P fits among the k strings in lexicographical order. More specifically we would like to know the predecessor of P among the k strings, according to the lexicographical order of the strings.

Using our already developed data structures for this problem is not efficient as strings can be quite long and comparing strings is slow. So we need to use some other data structure that takes into account this fact.

16.2.1 Tries and Compressed Tries

To solve the predecessor problem we will use a structure called a **trie**. A **trie** is a rooted tree where each child branch is labeled with letters in the alphabet Σ. We will consider any node v to store a string which represents the concatenation of all brach labels on the path from the root r to the v. More specifically given a path of increasing depth $p = r, v_1, v_2, ..., v$ from the root r to a node v, the string stored at node v_i is the concatenation of the string stored in v_{i-1} with the letter stored on the branch $v_{i-1}v_i$. We will denote the strings stored in the leaves of the trie as words, and the strings stored in all other nodes as prefixes. The root r represents the empty string.

We order the child branches of every node alphabetically from left to right. Note that the fan-out of any node is at most $|\Sigma|$. Also an inorder traversal of the trie outputs the stored strings in sorted order.

It is common practice to terminate strings with a special character $\$ \notin \Sigma$, so that we can distinguish a prefix from a word. The example trie in Figure 1 stores the four words ana$, ann$, anna$, and anne$.

If we assign only one letter per edge, we are not taking full advantage of the trie's tree structure. It is more useful to consider **compact** or **compressed** tries, tries where we remove the one letter per edge constraint, and contract non-branching paths by concatenating the letters on these paths. In this way, every node branches out, and every node traversed represents a choice between two different words. The compressed trie that corresponds to our example trie is also shown in Figure 1.

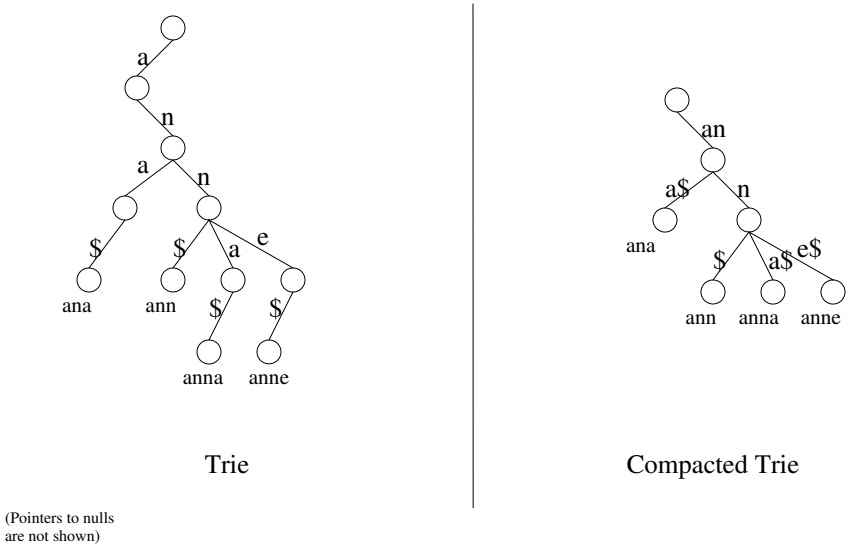


Figure 1: Trie and Conmpacted Trie Examples

16.2.2 Solving the Predecessor Problem

We will store our k strings $T_1, T_2, ..., T_k$ in a trie data structure as presented above. Given a query string P, in order to find it's predecessor we walk down the trie from the root, following the child branches. At some point we either fall off the tree or reach a leaf. If we reach a leaf then we are done, because the pattern P matches one of our words exactly. If we fall of the three then the predecessor is the maximum string stored in the tree imediately to the left of where we fell off. For example if we search for the word anb in the tree if Figure 1, we fall of the tree after we traverse branches a and n downward. Then b is between branches a and n of the node where we fell of, and the predecessor is the maximum string stored in the subtree following branch a.

For every node v of the trie we will store the maximum string in the subtree defined by v. Then, to allow fast predecessor queries we need a way to store every node of the trie, that allows both fast traversals of edges (going down the tree) and fast predecessor queries within the node.

16.2.3 Trie Node Representation

Depending on the way in which we choose to store a node in the trie, we will get various query time and space bounds.

Array

We can store every node of the trie as an array of size $|\Sigma|$, where each cell in the array represents one branch. In this case following a down branch takes $O(1)$ time. To find the predecessor within an array, we simply precompute all possible $|\Sigma|$ queries and store them. Thus, predecessor queries within a node also take $O(1)$ time. Unfortunately, the space used for every node is $O(|\Sigma|)$, so the lexicographical total space used is $O(|T||\Sigma|)$, where $|T|$ is the number of nodes in the trie $|T_1| + |T_2| + ... + |T_k|$. The total query time is $O(|P|)$.

Binary Search Tree

We can store a trie node as a binary search tree. More specifically, we store the branches of a node in a binary search tree. This will reduce the total space of the trie to $O(|T|)$, but will increase the query time to $O(|P| \log \Sigma)$.

Hash Table

Using a hash table for each node gives us an $O(|T|)$ space trie with $O(P)$ query time. Unfortunately with hashing we can only do exact searches, as hashes don't support predecessor.

Van Emde Boas

To support predecessor queries we can use a Van Emde Boas data structure to store each node of the trie. With this total space used to store the trie is $O(|T|)$ and query time is $O(P \log \log \Sigma)$.

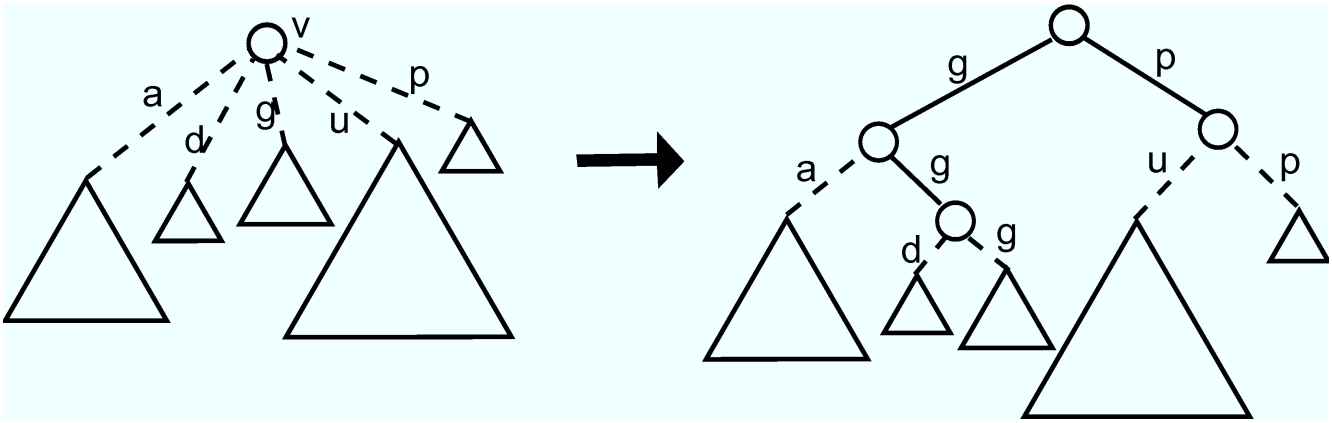


Figure 16.1: On the left we have node v with 5 children. Each triangle represents a subtree, and the size of the triangle represents the weight of the subtree (the number of descendand leaves of the subtree). On the right we can see the weight balanced BST. The solid lines are the edges of the weight balanced BST and the dashed lines represent the initial edges to the children of v.

Weight Balanced BST

We will store the children of the node v in a weight balanced binary search tree, where the weight of each child x is equal to the total number of descendand leaves in subtree x.

To construct such a tree we split the children of node v in two, such that the number of descendant leaves of the children on the left is as close as possible to the number of descendant leaves of the children to the right. We then recurse on the children on the left and on the children on the right. To support predecessor searches in this new tree, we can label each edge e of the constructed tree with the maximum edge label corresponding to some child x of v, in the subtree determined by following e. To better understand this construction look at Figure 2.

Lemma 43. *In a trie where we store each node as a weight balanced BST as above, a search takes* $O(P + \log k)$ *time.*

Proof. We will prove this by showing that at every edge followed in the tree, we either advance one letter in P or we decrease the number of candidate T_i's by 1/3. Assume that in our traversal we are at node y in the weight balanced BST representing node x in the trie. Consider all trie children $c_1, c_2, ..., c_m$ of node x present in y's subtree. Let D be the total number of descendant leaves of $c_1, c_2, ..., c_m$. If the number of descendant leaves of any child c_i is at most $D/3$ then following the next edge from y reduces the number of candidate leaves decreases by at least 1/3. Otherwise, there is some child c_i that has at least $D/3$ descendants. Then from the construction of the weight balanced BST, c_i has to be at most a grandchild of y. Then, by following the next two edges, we either move to node c_i in the trie, in which case we advance one letter in P, or we don't move do node c_i in which case we reduce the number of potential leaves by 1/3. □

Note that k is the number of strings stored in the trie. Even though our data structure achieves $O(P + \log k)$ query time, we would however like to achieve $O(P + \log \Sigma)$ time, to make query time independent of the number of strings in the trie. We can do this by using indirection and leaf trimming, similar to what we used to solve the level ancestor problem in constant time and linear space.

We will trim the tree by cutting edges below all maximally deep nodes that have at least $|\Sigma|$ descendant leaves. We will thus get a top tree, which has at most $|T|/|\Sigma|$ leaves and thus at most $|T|/|\Sigma|$ branching nodes. On this tree we can store trie nodes using arrays, to get linear query and $O(T)$ space. Note that we can only store arrays for the branching nodes and leaves, as there may be more nodes than $|T|/|\Sigma|$. But for the non-branching nodes we can simply store a pointer to the next node, or use the compressed trie from the start instead.

The bottom trees have less than $|\Sigma|$ descendant leaves, which means we can store trie nodes using weight balanced BSTs, which use linear space, but now have $O(P + \log \Sigma)$ query time since there are less than Σ leaves.

In total we get a data structure with $O(T)$ space and $O(P + \log \Sigma)$ query time. Another data structure that obtains these bounds is the suffix trays [86].

16.3 Suffix Trees

A **suffix tree** is a compressed trie built on all $|T|$ suffixes of T, with \$ appended. For example, if our text is the string $T = banana\$$, then our suffix tree will be built on $\{banana\$, anana\$, nana\$, ana\$, na\$, a\$\}$. The suffix starting at the ith index is denoted $T[i:]$. For a non-leaf node of the suffix tree, define the letter depth of the node as the length of the prefix stored in the node.

Storing strings on the edges and in the nodes is potentially very costly in terms of space. For example, if all of the characters in our text are different, storage space is quadratic in the size of the text. To decrease storage space of the suffix tree to $O(|T|)$, we can replace the strings on each edge by the indices of its first and last character, and omit the strings stored in each node. We lose no information, as we are just removing some redundancies.

16.3.1 Applications

Suffix trees are versatile data structures that have myriad applications to string matching and related problems:

String Matching

To solve the string matching problem, note that a substring of T is simply a prefix of a suffix of T. Therefore, to find a pattern P, we walk down the tree, following the edge that corresponds to the next set of characters in P. Eventually, if the pattern matches, we will reach a node v that stores P. Finally, report all the leaves beneath v, as each leaf represents a different occurrence. There is a unique way to walk down the tree, since every edge in the fan out of some node must have a distinct first letter. The runtime of a search, however, depends on how the nodes are stored. If

the nodes are stored as a hash table, this method achieves $O(P)$ time. Using trays, we can achieve $O(P + \lg \Sigma)$, and using a hash table and a van Emde Boas structure, we can achieve $O(P + \lg \lg \Sigma)$ time. These last two results are useful because they preserve the sorted order of the nodes, where a hash table does not preserve sorting.

First k Occurrences

Instead of reporting all matching subsequences of T, we can report the first k occurrences of P in only $O(k)$ additional time. To do this, add a pointer from each node to its leftmost descendant leaf, and then connect all the leaves via a linked list. Then, to determine the first k occurrences, perform a search as above, and then simply follow pointers to find the firsk k leaves that match that pattern.

Counting Occurrences

In this variant of the string matching problem, we must find the number of times the pattern P appears in the text. However, we can simply store in every node the size of the subtree at that node.

Longest Repeating Substring

To find the longest repeated substring, we look for the branching node with maximum letter depth. We can do this in $O(T)$ time. Suppose instead we are given two indices i and j, and we want to know the longest common prefix of $T[i:]$ and $T[j:]$. This is simply an LCA query, which from L15 can be accomplished in $O(1)$ time.

All occurences of $T[i:j]$

When the pattern P is a known substring of the text T, we can interpret the problem as a LA query: We'd like to find the $(j-i)$th ancestor of the leaf for $T[i:]$. We must be careful because the edges do not have unit length, so we must interpret the edges as weighted, by the number of characters that are compressed onto that edge.

We must make a few modifications to the LA data structure from L15 to account for the weighted edges. We store the nodes in the long path/ladder DS of L15 in a van Emde Boas predecessor DS, which uses $O(\lg \lg T)$ space. Additionally, we can't afford the lookup tables at the bottom of the DS from L15. Instead, we answer queries on the bottom trees in the indirection by using a ladder decomposition. This requires $O(\lg \lg n)$ ladders, to reach height $O(\lg n)$. Then, we only need to run a predecessor query on the last ladder. This gives us $O(\lg \lg T)$ query and $O(T)$ space. This result is dues to Abbott, Baran, Demaine and others in Spring 2005's 6.897 course.

Multiple Documents

When there are multiple texts in question, we can concatenate them with $\$_1, \$_2, ..., \$_n$. Whenever we see a $\$_i$, we trim below it to make it a leaf and save space. Then to find the longest common substring, we look for the node with the maximum letter depth with greater than one distinct \$ below. To count the number of documents containing a pattern P, simply store at each node the number of distinct \$ signs as a descendant of that node.

Document Retrieval

We can find d distinct documents T_i containing a pattern P in only an additional $O(d)$ time, for a total of $O(P + d)$. This is ideal, because if P occurs an astronomical number of times in one document, and only once or twice in a second document, a more naive algorithm could spend time finding every occurrence of P when the result will only be 2 documents. This DS, due to Muthukrishnan [85], will avoid that problem.

To do this, augment the data structure by having each $\$_i$ store the leaf number of the previous $\$_i$. Then, suppose we have searched for the pattern P and come to the node v. Suppose the leaf descendants of v are numbered l through n. So, we want the first occurrence of $\$_i$ in the range $[l, n]$, for each i. With the augmentation, this is equivalent to looking for the $\$_i$ whose stored value is $< l$, because this indicates that the previous $\$_i$ is outside of the interval $[l, n]$.

We can solve this problem with an RMQ query from L15. Find the minimum in $O(1)$ time. Suppose the minimum is found at position m. Then, if the stored value of that leaf is $< l$, that is an answer, so output that the pattern occurs in document i. Then, recurse on the remaining intervals, $[l, m-1]$ and $[m+1, n]$. Thus, each additional output we want can be found in $O(1)$ time, and we can stop any time we want.

16.4 Suffix Arrays

Suffix trees are powerful data structures with applications in fields such as computational biology, data compression, and text editing. However, a **suffix array**, which contains most of the information in a suffix tree, is a simpler and more compact data structure for many applications. The only drawback of a suffix array is that is that it is less intuitive and less natural as a representation. In this section, we define suffix arrays, show that suffix arrays are in some sense equivalent to suffix trees, and provide a fast algorithm for building suffix arrays.

16.4.1 Example

Let us store the suffixes of a text T in lexicographical order in an intermediate array. Then the suffix array is the array that stores the index corresponding to each suffix. For example, if our text is *banana*\$, the intermediate array is $[\$, a\$, ana\$, anana\$, banana\$, na\$, nana\$]$ and the suffix array is $[6, 5, 3, 1, 0, 4, 2]$, as in Figure 2. Since suffixes are ordered lexicographically, we can use binary search to search for a pattern P in $O(|P| \log |T|)$ time.

We can compute the length of the longest common prefix between neighboring entries of the intermediate array. If we store these lengths in an array, we get what is called the **LCP array**. This is illustrated in Figure 3. These LCP arrays can be constructed in $O(|T|)$ time, a result due to Kasai et al [82]. In the example above, the LCP array constructed from the intermediate array is $[0, 1, 3, 0, 0, 2]$. Using LCP arrays, we can improve pattern searching in suffix arrays to $O(|P| + \log |T|)$.

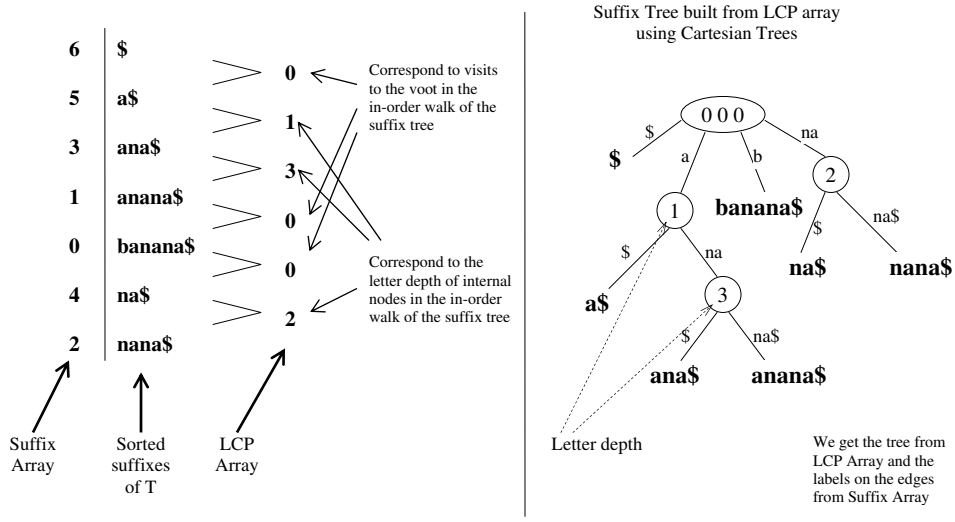


Figure 2: Suffix Array and LCP Array examples and their relation to Suffix Tree

16.4.2 Suffix Arrays and Suffix Trees

To motivate the construction of a suffix array and a LCP array, note that a suffix array stores the leaves of the corresponding suffix tree, and the LCP array provides information about the height of internal nodes, as seen in Figure 2. Our aim now is to show that suffix trees can be transformed into suffix arrays in linear time and vice versa.

To formalize this intuition, we use **Cartesian Trees** from L15. To build a Cartesian tree, store the minimum over all LCP array entries at the root of the Cartesian tree, and recurse on the remaining array pieces. For a concrete example, see Figure 3 below.

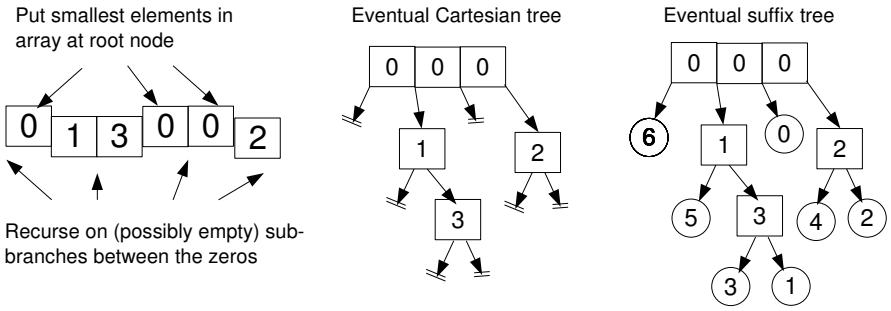


Figure 3: Constructing the Cartesian tree and suffix tree from LCP array

From Suffix Trees to Suffix Arrays

To transform a suffix tree into a suffix array, we simply run an in-order traversal of the suffix tree. As noted earlier, this process returns all suffixes in alphabetical order.

From Suffix Arrays to Suffix Trees

To transform a suffix array back into a suffix tree, create a Cartesian tree from the LCP array associated to the suffix array. Unlike in L15, use all minima at the root. The numbers stored in the nodes of the Cartesian tree are the letter depths of the internal nodes! Hence, if we insert the entries of the suffix array in order into the Cartesian tree as leaves, we recover the suffix tree. In fact, we are rewarded with an augmented suffix tree with informationa bout the letter depths. From L15, this construction is possible in linear time.

16.5 DC3 Algorithm for Building Suffix Arrays

Here, we give a description of the DC3 (Difference Cover 3) divide and conquer algorithm for building a suffix array in $O(|T| + sort(\Sigma))$ time. We closely follow the exposition of the paper by Karkkainen-Sanders-Burkhardt [80] that originally proposed the DC3 algorithm. Because we can create suffix trees from suffix arrays in linear time, a consequence of DC3 is that we can create suffix trees in linear time, a result shown independently of DC3 by Farach [79], McCreight [84], Ukkonen [87], and Weiner [88]

1. Sort the alphabet Σ. We can use any sorting algorithm, leading to the $O(\text{sort}(\Sigma))$ term.

2. Replace each letter in the text with its rank among the letters in the text. Note that the rank of the letter depends on the text. For example, if the text contains only one letter, no matter what letter it is, it will be replaced by 1. This operation is safe, because it does not change any relations we are interested in. We also guarantee that the size of the alphabet being used is no larger than the size of the text (in cases where the alphabet is excessively large), by ignoring unused alphabets.

3. Divide the text T into 3 parts and package triples of letters into *megaletters*. More formally,

form T_0, T_1, and T_2 as follows:

$$\begin{aligned} T_0 = &< (T[3i], T[3i+1], T[3i+2]) \quad \text{for} \quad i = 0, 1, 2, \ldots > \\ T_1 = &< (T[3i+1], T[3i+2], T[3i+3]) \quad \text{for} \quad i = 0, 1, 2, \ldots > \\ T_2 = &< (T[3i+2], T[3i+3], T[3i+4]) \quad \text{for} \quad i = 0, 1, 2, \ldots > \end{aligned}$$

Note that T_i's are just texts with $n/3$ letters of a new alphabet Σ^3. Our text size has become a third of the original, while the alphabet size has cubed.

4. Recurse on $< T_0, T_1 >$, the concatenation of T_0 and T_1. Since our new alphabet is of cubic size, and our original alphabet is pre-sorted, radix-sorting the new alphabet only takes linear time. When this recursive call returns, we have all the suffixes of T_0 and T_1 sorted in a suffix array. Then all we need is to sort the suffixes of T_2, and to merge them with the old suffixes to get suffixes of T, because

$$\texttt{Suffixes}(T) \cong \texttt{Suffixes}(T_0) \cup \texttt{Suffixes}(T_1) \cup \texttt{Suffixes}(T_2)$$

If we can do this sorting and merging in linear time, we get a recursion formula $T(n) = T(2/3n) + O(n)$, which gives linear time.

5. Sort suffixes of T_2 using radix sort. This is straight forward to do once we note that

$$T_2[i:] \cong T[3i+2:] \cong (T[3i+2], T[3i+3:]) \cong (T[3i+2], T_0[i+1:]).$$

The logic here is that once we rewrite $T_2[i:]$ in terms of T, we can pull off the first letter of the suffix and pair it with the remainder. We end up with something where the index $3i+3$ corresponds with the start of a triplet in T_0, specifically, $T_0[i+1]$, which we already have in sorted order from our recursive call.

Thus, we can radix sort on two coordinates, the triplet $T_0[i+1]$ and then the single alphabet $T[3i+2]$, both of which we know the sorted orders of. This way, we get $T_2[i:]$ in sorted order. Specifically, the radix sort is just on two coordinates, where the second coordinate is already sorted.

6. Merge the sorted suffixes of T_0, T_1, and T_2 using standard linear merging. The only problem is finding a way to compare suffixes in constant time. Remember that suffixes of T_0 and T_1 are already sorted together, so comparing a suffix from T_0 and a suffix from T_1 takes constant time. To compare against a suffix from T_2, we will once again decompose it to get a suffix from either T_0 or T_1. There are two cases:

- Comparing T_0 against T_2:

$$\begin{aligned} & & T_0[i:] \quad & \texttt{vs} \quad & T_2[j:] \\ & \cong & T[3i:] \quad & \texttt{vs} \quad & T[3j+2:] \\ \cong & & (T[3i], T[3i+1:]) \quad & \texttt{vs} \quad & (T[3j+2], T[3j+3:]) \\ & \cong & (T[3i], T_1[i:]) \quad & \texttt{vs} \quad & (T[3j+2], T_0[j+1:]) \end{aligned}$$

  So we just compare the first letter and then, if needed, compare already sorted suffixes of T_0 and T_1.

- Comparing T_1 against T_2:

$$\begin{array}{rrcl} & T_1[i:] & \texttt{vs} & T_2[j:] \\ \cong & T[3i+1:] & \texttt{vs} & T[3j+2:] \\ \cong & (T[3i+1], T[3i+2], T[3i+3:]) & \texttt{vs} & (T[3j+2], T[3j+3], T[3j+4:]) \\ \cong & (T[3i+1], T[3i+2], T_0[i+1:]) & \texttt{vs} & (T[3j+2], T[3j+3], T_1[j+1:]) \end{array}$$

  So we can do likewise by first comparing the two letters in front, and then comparing already sorted suffixes of T_0 and T_1 if necessary.

Lecture 17

Succinct 1

Scribes: David Benjamin(2012), Lin Fei(2012), Yuzhi Zheng(2012),Morteza Zadimoghaddam(2010), Aaron Bernstein(2007)

17.1 Overview

Up until now, we have mainly studied how to decrease the query time, and the preprocessing time of our data structures. In this lecture, we will focus on maintaining the data as compactly as possible. The for this section is to get very "small" space, that is often for static data structure. Going to look a binary tries, using binary alphabets. Another way is to use bit strings. It is easy to do linear space, it it is not optimal. Our goal will be to get as close the information theoretic optimum as possible. We will refer to this optimum as OPT. Note that most "linear space" data structures we have seen are still far from the information theoretic optimum because they typically use O(n) *words* of space, whereas OPT usually uses O(n) bits. This strict space limitation makes it really hard to have dynamic data structures, so most space-efficient data structures are static.

Here are some possible goals we can strive for:

- *Implicit Data Structures* – Space = information-theoretic-OPT + O(1). Focusing on bits, not words. The ideal is to use O(n) bits. The O(1) is there so that we can round up if OPT is fractional. Most implicit data structures just store some permutation of the data: that is all we can really do. As some simple examples, we can refer to Heap which is a Implicit Dynamic Data Structure, and Sorted array which is static example of these data structures.
- *Succinct Data Structures* – Space = OPT + o(OPT). In other words, the leading constant is 1. This is the most common type of space-efficient Data Structures.
- *Compact Data Structures* – Space = O(OPT). Note that some "linear space" data structures are not actually compact because they use O(w·OPT) bits. This saves at least an O(w) from the normal data structures. For example, suffix tree has $O(n)$ words space, but its information theoretic lower bound is n bits. On the other hand, BST can be seen as a Compact Data Structure.

Succinct is the usual goal here. Implicit is very hard, and compact is generally to work towards a succinct data structure.

17.1.1 mini-survey

- *Implicit Dynamic Search Tree* – Static search trees can be stored in an array with $\ln n$ per search. However, in order to inserts and deletes makes the problem more tricky. There is an old results that was done in $lg^2 n$ using pointers and permutation of the bits In 2003, Franceschini and Grossi [89] developed an implicitly dynamic search tree which supports insert, delete, and predecessor in O(log(n)) time worst case, and it is cache-oblivious.

- *Succinct Dictionary* – This is static, which has no inserts and deletes. In a universe of size u, how many ways are there to have n items. Use $\lg \binom{u}{n} + O\left(\frac{\lg \binom{u}{n}}{\lg \lg \lg u}\right)$ bits [90] or $\lg \binom{u}{n} + O(\frac{n(\lg \lg n)^2}{\lg n})$ bits [94] , and support $O(1)$ membership queries, same amount of time but with less space than normal dictionaries; u is the size of the universe from which the n elements are drawn.

- *Succinct Binary Tries* – The number of possible binary tries with n nodes is the nth Catalan number, $C_n = \binom{u}{n}/(n+1) \approx 4^n$. Thus, $OPT \approx \log(4^n) = 2n$. We note that this can be derived from a recursion formula based on the sizes of the left and right subtrees of the root. In this lecture, we will show how to use $2n + o(n)$ bits of space. We will be able to find the left child, the right child, and the parent in O(1) time. We will also give some intuition for how to answer subtree-size queries in O(1) time. Subtree size is important because it allows us to keep track of the rank of the node we are at. Motivation behind the research was to fit the Oxford dictionary onto a CD, where space was limited.

- *Almost Succinct k-ary trie* – The number of such tries is $\mathrm{C}_n^k = \binom{kn+1}{n}/(kn+1) \approx 2^{(log(k)+log(e))n}$. Thus, OPT = (log(k) + log(e))n. The best known data structures was developed by Benoit *et al.* [91]. It uses $(\lceil log(k) \rceil + \lceil log(e) \rceil)$n + o(n) + O(loglog(k)) bits . This representation still supports the following queries in O(1) time: find child with label i, find parent, and find subtree size.

- *Succinct Rooted Ordered Trees* – These are different from tries because there can be no absent children. The number of possible trees is C_n, so OPT = 2n. A query can ask us to find the ith child of a node, the parent of a node, or the subtree size of a node. Clark and Munro [92] gave a succinct data structure which uses 2n + o(n) space, and answers queries in constant time.

- *Succinct Permutation* – In this data structure, we are given a permutation π of n items, and the queries are of the form $\pi^k(i)$. Munro et. al. present a data structure with constant query time and space $(1+\epsilon)n\log(n) + O(1)$ bits in [95]. They also obtain a succinct data structure with $\lceil \log n! \rceil + o(n)$ bits and query time $O(\log n / \log \log n)$.

- *Compact Abelian groups* – Represent abelian group on n items using $O(\lg n)$ bits, and represent an item in that list with $\lg n$ bits.

- *Graphs* – More complicated. We did not go over any details in class.

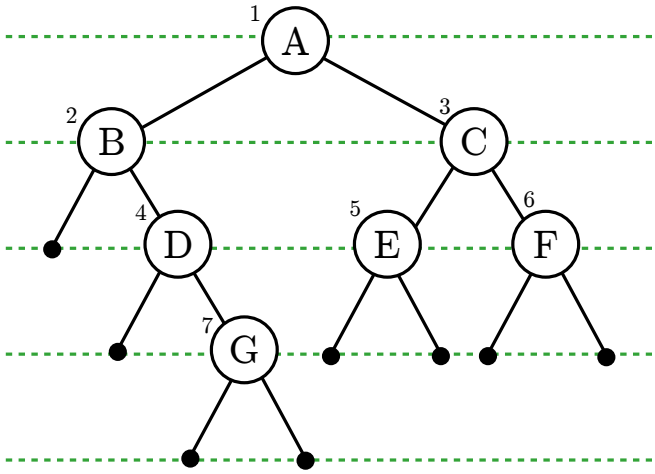
- *Integers* – Implicit n-bit number of integers can do increment or decrement in $O(\lg n)$ bits reads and $O(1)$ bit writes.

  OPEN: $O(1)$ word operations.

17.2 Level Order Representation of Binary Tries

As noted above, the information theoretic optimum size for a binary trie is $2n$ bits, two bits per node. To build a succinct binary trie, we must represent the structure of the trie in $2n + o(n)$ bits. We represent tries using the *level order representation*. Visit the nodes in level order (that is, level by level starting from the top, left-to-right within a level), writing out two bits for each node. Each bit describes a child of the node: if it has a left child, the first bit is 1, otherwise 0. Likewise, the second bit represents whether it has a right child.

As an example, consider the following trie:



We traverse the nodes in order A, B, C, D, E, F, G. This results in the bit string:

11	01	11	01	00	00	00
A	*B*	*C*	*D*	*E*	*F*	*G*

External node formulation Equivalently, we can associate each bit with the child, as opposed to the parent. For every missing leaf, we add an *external node*, represented by $\cdot$ in the above diagram. The original nodes are called *internal nodes*. We again visit each node in level order, adding 1 for internal nodes and 0 for external nodes. A tree with n nodes has $n + 1$ missing leaves, so this results in $2n + 1$ bits. For the above example, this results in the following bit string:

1	1	1	0	1	1	1	0	1	0	0	0	0	0	0
A	*B*	*C*	$\cdot$	*D*	*E*	*F*	$\cdot$	*G*	$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdot$	$\cdot$

Note this new bit string is identical to the previous but for an extra 1 prepended for the root, A.

17.2.1 Navigating

This representation allows us to compute left child, right child, and parent in constant. It does not, however, allow computing subtree size. We begin by proving the the following theorem:

Theorem 44. *In the external node formulation, the left and right children of the ith internal node are at positions $2i$ and $2i+1$.*

Proof. We prove this by induction on i. For $i = 1$, the root, this is clearly true. The first entry of the bit string is for the root. Then bits $2i = 2$ and $2i + 1 = 3$ are the children of the root.

Now, for $i > 1$, by the inductive hypothesis, the children of the $i-1$st internal node are at positions $2(i-1) = 2i-2$ and $2(i-1)+1 = 2i-1$. We show that the children of the ith internal immediately follow the $i-1$st internal node's children.

Visually, there are two cases: either $i-1$ is on the same level as i or i is on the next level. In the second case, $i-1$ and i are the last and first internal nodes in their levels, respectively.

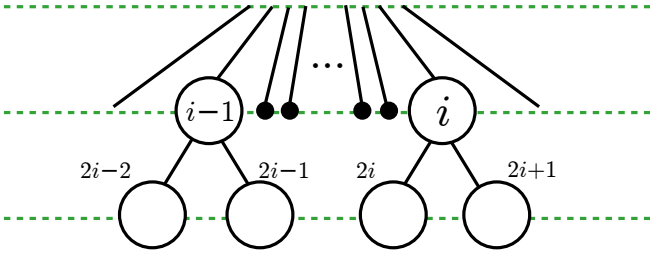


Figure 17.1: $i-1$ and i are on the same level.

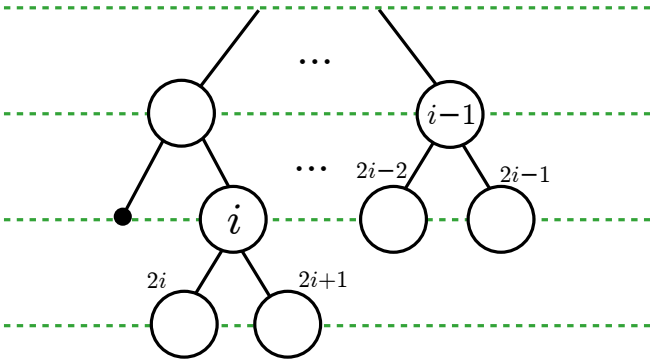


Figure 17.2: $i-1$ and i are different levels.

Level ordering is preserved in children, that is, A's children precede B's children in the level ordering if and only if A precedes B. All nodes between the $i-1$st internal node and the ith internal node are external nodes with no children, so the children of the ith internal node immediately follow the children of the $i-1$st internal node, at positions $2i$ and $2i+1$. □

As a corollary, the parent of *bit position* i is $\lfloor i/2 \rfloor$. Note that we have two methods of counting bits: we may either count bit positions or rank only by 1s. If we could translate between the two in $O(1)$ time, we could navigate the tree efficiently.

17.3 Rank and Select

Now we need to establish a correspondence between positions in the n-bit string and actual internal node number in the binary tree.

Say that we could support the following operations on an n-bit string in O(1) time, with o(n) extra space:

- rank(i) = number of 1's at or before position i
- select(j) = position of jth one.

This would give us the desired representation of the binary trie. The space requirement would be 2n for the level-order representation, and o(n) space for rank/select. Here is how we would support queries:

- left-child(i) = 2rank(i)
- right-child(i) = 2rank(i) + 1
- parent(i) = select($\lfloor i/2 \rfloor$)

Note that level-ordered trees do not support queries such as subtree-size. This can be done in the balanced parentheses representation, however.

17.3.1 Rank

This algorithm was developed by Jacobsen, in 1989 [93]. It uses many of the same ideas as RMQ. The basic idea is that we use a constant number of recursions until we get down to sub-problems of size $k = \lg(n)/2$. Note that there are only $2^k = \sqrt{n}$ possible strings of size k, so we will just store a lookup table for *all possible bit strings of size k.* For each such string we have k = O(log(n)) possible queries, and it takes log(k) bits to store the solution of each query (the rank of that element). Nonetheless, this is still only $O(2^k \cdot k \log k) = O(\sqrt{n} \lg(n) \lg \lg(n)) = o(n)$ bits.

Notice that a naive division of the entire n-bit string into chunks of size $\frac{1}{2} \lg n$ does not work. because we need to save $\frac{n}{\lg n}$ relative indices, each taking up to $\lg n$ bits, for a total of $\Theta(n)$ bits, which is too much. Thus, we need to use a technique that uses indirection twice.

Step 1: We build a lookup table for bit strings of length $\frac{1}{2} \lg n$. As we argued before, this takes $O(\sqrt{n} \lg n \lg \lg n) = o(n)$ bits of space.

Step 2: Split the the n-bit string into $(\lg^2 n)$-bit chunks.

At the end of each chunk, we store the cumulative rank so far. Each cumulative rank would take $\lg n$ bits. And there are at most $n/\lg^2 n$ chunks. Thus, the total space required is $O(\frac{n}{\lg^2 n} \lg n) = O(\frac{n}{\lg n})$ bits

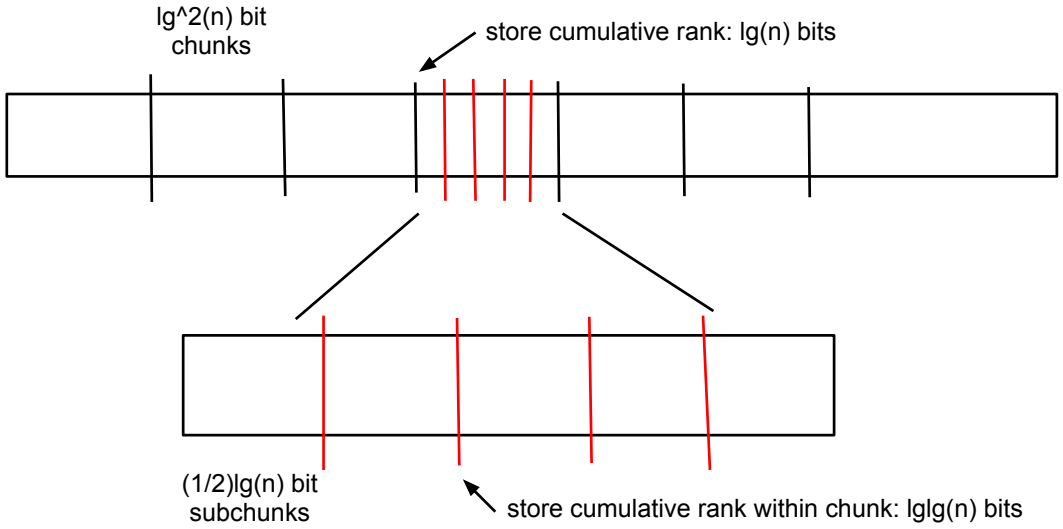


Figure 17.3: Division of bit string into chunks and sub-chunks, as in the rank algorithm

Step 3: Now we need to split each chunk into $(\frac{1}{2}\lg n)$-bit sub-chunks.

Then, at the end of each sub-chunk, we store the cumulative rank **within each chunk**. Since each chunk only has size $\lg^2 n$, we only need $\lg\lg n$ bits for each index. There are at most $O(n/\lg n)$ sub-chunks. Thus, the total space required is $O(\frac{n}{\lg n}\lg\lg n) = o(n)$ bits. Notice how we saved space because the size of the cumulative rank within each small chunk is less.

Step 4: To find the total rank, we just do:

Rank = rank of chunk + relative rank of sub-chunk within chunk + relative rank of element within sub-chunk (via lookup table).

This algorithm runs in $O(1)$ time, and it only takes $O(n\frac{\lg\lg n}{\lg n})$ bits.

It is in fact possible to improve the space to $O(\frac{n}{\lg^k n})$ bits for any constant k. But this is not covered in class.

17.3.2 Select

This algorithm was developed by Clark and Munro in 1996 [92]. Select is similar to rank, although more complicated. This time, since we are trying to find the position of the ith one, we will break our array up into chunks with equal amounts of ones, as opposed to chunks of equal size.

Step 1: First, we will pick every $(\lg n\lg\lg n)$th 1 to be a *special one*. We will store the index of every special one. Storing an index takes $\lg n$ bits, so this will take $O(n\lg n/(\lg n\lg\lg n)) = O(n/\lg\lg n) = o(n)$.

Then, given a query, we can find divide it by $\lg n\lg\lg n$ to teleport to the correct chunk containing our desired result.

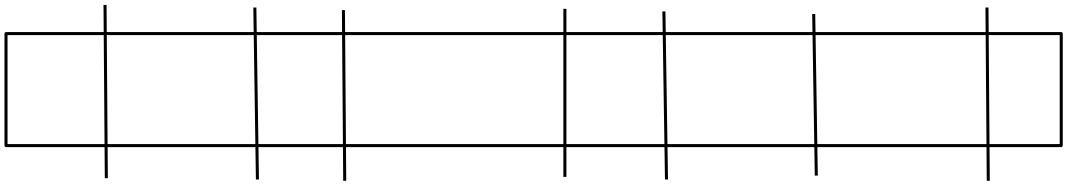


Figure 17.4: Division of bit string into uneven chunks each containing the same number of 1's, as in the select algorithm

Step 2: Now, we need to restrict our attention to a single chunk, which contains $\lg n \lg\lg n$ 1 bit. Let r be the *total* number of bits in a chunk.

If $r > (\lg n \lg\lg n)^2$ bits:

This is when the 1 bits are sparse. Thus, we can afford to store an array of indices of every 1 bit in this chunk. There are $(\lg n \lg\lg n)$ 1 bits. Storing each would take up at most $\lg n$ space (The size of the chunk is already polylog(n), so $\lg n$ bits is more than enough). And there are at most $\frac{n}{(\lg n \lg\lg n)^2}$ such sparse chunks. So, in total, we use space:

$O(\frac{n}{(\lg n \lg\lg n)^2}(\lg n \lg\lg n)\lg n) = o(\frac{n}{\lg\lg n})$ bits.

If $r < (\lg n \lg\lg n)^2$ bits:

We have reduced to a bit string of length $r \leq (\lg n \lg\lg n)^2$. This is good, because analogously to Rank, these chunks are small enough to be divided into sub-chunks without requiring too much space for storing the sub-chunk's relative indices.

Step 3: We basically repeat steps 1 and 2 on all reduced bit strings, and further reduce them into bit strings of length $(\lg\lg n)^{O(1)}$.

Step 1':

With the reduced chunks of length $O(\lg n \lg\lg n)^2$, we again pick out special 1's, and divide it into every $(\lg\lg n)^2$th 1 bit. Since within each chunk, the relative index only takes $O(\lg\lg n)$ bits, the total space required is:

$O(\frac{n}{(\lg\lg n)^2}\lg\lg n) = O(\frac{n}{\lg\lg n})$ bits.

Step 2':

Within groups of $(\lg\lg n)^2$ 1 bits, say it has r bits total:

if $r \geq (\lg\lg n)^4$, then store relative indices of 1 bits. There are $\frac{n}{(\lg\lg n)^4}$ such groups, within each there are $(\lg\lg n)^2$ 1 bits, and each relative index takes $\lg\lg n$ bits to store. For a total of:

$O(\frac{n}{(\lg\lg n)^4}(\lg\lg n)^2 \lg\lg n) = O(\frac{n}{\lg\lg n})$ bits.

if $r < (\lg\lg n)^4$, then $r < \frac{1}{2}\lg n$. Then it is small enough for us to use step 4:

Step 4: use lookup table for bit string of length $\leq \frac{1}{2}\lg n$. Like in rank, the total space for the lookup table is at most:

$O(\sqrt{n}\lg n \lg\lg n)$.

Thus, we again have $O(1)$ query time and $O(\frac{n}{\lg\lg n})$ bits space.

Again, this result can be improved to $O(n/\lg^k n)$ bits for any constant k, but we will not show it here.

17.4 Subtree Sizes

We have shown a Succinct binary trie which allows us to find left children, right children, and parents. But we would still like to find sub-tree size in O(1) time. Level order representation does not work for this, because level order gives no information about depth. Thus, we will instead try to encode our nodes in depth first order.

In order to do this, notice that there are C_n (catalan number) binary tries on n nodes. But there are also C_n rooted ordered trees on n nodes, and there are C_n balanced parentheses strings with n parentheses. Moreover, we will describe a bijection: binary tries $\Leftrightarrow$ rooted ordered trees $\Leftrightarrow$ balanced parentheses. This makes the problem much easier because we can work with balanced parentheses, which have a natural bit encoding: 1 for an open parentheses, 0 for a closed one.

17.4.1 The Bijections

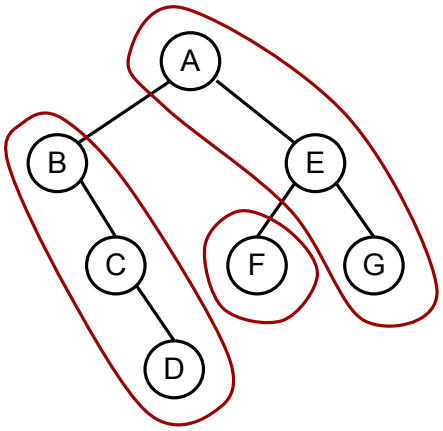


Figure 17.5: An example binary trie with circled right spine.

We will use the binary trie in Figure 17.5. Finding the right spine of the trie, then recurse until every node lives in a spine. To make this into a rooted ordered tree, we can think of rotating the trie 45 degrees counter-clockwise. Thus, the top three nodes of the tree will be the right spine of the trie (A,C,F). To make the tree rooted, we will add an extra root *. Now, we recurse into the left subtrees of A,C, and F. For A, the right spine is just B,D,G. For C, the right spine is just E: C's

only left child. Figure 17.6 shows the resulting rooted ordered tree. There is a bijection between the two representations.

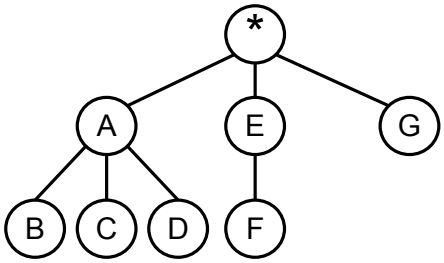


Figure 17.6: A Rooted Ordered Tree That Represents the Trie in Figure 17.5.

To go from rooted ordered trees to balanced parentheses strings, we do a DFS (or Euler tour) of the ordered tree. We will then put an open parentheses when we first touch a node, and then a closed parentheses the second time we touch it. Figure 17.7 contains a parentheses representation of the ordered tree in Figure 2.

```
( ( ( ) ( ) ( ) ) ( ( ) ) ( ) )
* A B B C C D D A E F F G G E *
```

Figure 17.7: A Balanced Parentheses String That Represents the Ordered Tree in Figure 17.6

Now, we will show how the queries are transformed by this bijection. For example, if we want to find the parent in our binary trie, what does this correspond to in the parentheses string? The bold-face is what we have in the binary trie, and under that, we will describe the corresponding queries from the 2 bijections.

Binary Trie	Rooted Ordered Tree	Balanced Parentheses
Node	Node	Left Paren[and matching right]
Left Child	First Child	Next char [if '(', else none]
Right Child	Next Sibling	Char after matching ')' [if '(']
Parent	Previous Sibling or Parent	if prev char ')', its mathcing '('; if '(', that '('
Subtree size	size(node)+size(right siblings)	1/2 distance to enclosing ')'

Could use rank and select to find the matching parenthesis for the balanced parenthesis representation.

Lecture 18

Succinct 2

Scribes: Sanja Popovic(2012), Jean-Baptiste Nivoit(2012), Jon Schneider(2012), Sarah Eisenstat (2010), Martí Bolívar (2007)

18.1 Overview

In the last lecture we introduced the concept of implicit, succinct, and compact data structures, and gave examples for succinct binary tries, as well as showing the equivalence of binary tries, rooted ordered trees, and balanced parentheses expressions. Succinct data structures were introduced which solve the *rank* and *select* problems.

In this lecture, we introduce compact data structures for suffix arrays and suffix trees. Recall the problem that we are trying to solve. Given a text T over the alphabet Σ, we wish to preprocess T to create a data structure. We then want to be able to use this data structure to search for a pattern P, also over Σ.

A suffix array is an array containing all of the suffixes of T in lexicographic order. In the interest of space, each entry in the suffix array stores an index in T, the start of the suffix in question. To find a pattern P in the suffix array, we perform binary search on all suffixes, which gives us all of the positions of P in T.

18.2 Survey

In this section, we give a brief survey of results for compact suffix arrays. Recall that a compact data structure uses $O(OPT)$ bits, where OPT is the information-theoretic optimum. For a suffix array, we need $|T| \lg |\Sigma|$ bits just to store the text T.

18.2.1 Compact suffix arrays and trees

Grossi and Vitter 2000 [98] Suffix array in

$$\left(\frac{1}{\varepsilon} + O(1)\right) |T| \lg |\Sigma|$$

bits, with query time

$$O\left(\frac{|P|}{\log^{\varepsilon}_{|\Sigma|} |T|} + |\text{output}| \cdot \log^{\varepsilon}_{|\Sigma|} |T|\right)$$

We will follow this paper fairly closely in our discussion today. Note how one can trade better query time for higher space usage.

Ferragina and Manzini 2000 [96] The space required is

$$5H_k(T) \cdot |T| + O\left(|T| \cdot \frac{|\Sigma| + \lg |T|}{\lg \lg |T|} + |T|^{\varepsilon} \cdot |\Sigma|^{|\Sigma|+1}\right)$$

bits, for all fixed values of k. $H_k(T)$ is the k^{th}-order empirical entropy, or the regular entropy conditioned on knowing the previous k characters. More formally:

$$H_k(T) = \sum_{|w|=k} \Pr\{w \text{ occurs}\} \cdot H_0(\text{characters following an occurrence of } w \text{ in } T).$$

Note that because we are calculating this in the empirical case,

$$\Pr\{w \text{ occurs}\} = \frac{\#\text{ of occurrences of } w}{|T|}.$$

For this data structure, query time is

$$O(|P| + |\text{output}| \cdot \lg^{\varepsilon} |T|).$$

Sadakane 2003 [100] Space in bits is

$$\frac{1+\varepsilon'}{\varepsilon} H_0(T)|T| + O(|T| \lg \lg |\Sigma| + |\Sigma| \lg |\Sigma|),$$

and query time is

$$O(|P| \lg |T| + |\text{output}| \lg^{\varepsilon} |T|).$$

where $0 < \varepsilon < 1$ and $\varepsilon' > 0$ are arbitrary constants. Note that this bound is more like a suffix array, due to the multiplicative log factor. This bound is good for large alphabets.

18.2.2 Succint suffix arrays and trees

Grossi, Gupta, Vitter 2003 [97] Space in bits is

$$H_k(T) \cdot |T| + O\left(|T| \lg |\Sigma| \cdot \frac{\lg \lg |T|}{\lg |T|}\right),$$

and query time is

$$O(|P| \lg |\Sigma| + \frac{\lg^2 |T|}{\lg \lg |T|}).$$

Ferragina, Manzini, Mäkinen, Navarro [101] Space in bits is

$$H_k(T) \cdot |T| + O\left(\frac{|T|}{\lg^{\varepsilon} n}\right),$$

and query time is

$$O(|P| + |\text{output}| \cdot \lg^{1+\varepsilon} |T|).$$

Also exhibits $O(|P|)$ running time for counting queries.

18.2.3 Extras

Hon, Sadakane, Sung 2003 [103] and Hon, Lam, Sadakane, Sung, Yiu 2007 [102] details low-space construction with $O(|T| \cdot \lg |\Sigma|)$ working space.

Hon, Sadakane 2002 [104] and Sadakane 2007 [105] detail suffix-tree operations like maximal common substrings.

Sadakane 2007 [106] solves the document retrieval problem on those structures.

Chan, Hon, Lam, Sadakane 2007 [107] describes the dynamic version of the problem.

18.3 Compressed suffix arrays

For the rest of these notes, we will assume that the alphabet is binary (in other words, that $|\Sigma| = 2$). In this section, we will cover a simplified (and less space-efficient) data structure, which we will adapt in the next section for the compact data structure. Overall, we will cover a simplified version of Grossi's and Vitter's paper [98]. We will achieve the same space bound, but slightly worse query time.

The problem we are solving with suffix arrays is to find $SA[k]$, i.e. where does the k-th suffix start, assuming suffixes are sorted in lexicographic order. The data structure we will be using is similar to the one seen in Lecture 16. We are still using the same divide-and-conquer approach, except that it is 2-way instead of 3-way.

18.3.1 Top-Down

Let us introduce some notation we will use throughout the notes.

start: The initial text is $T_0 = T$, the initial size $n_0 = n$, and the initial suffix array $SA_0 = SA$ where SA is the suffix array of T.

step: In every successive iteration, we are combining two adjacent letters:

$$T_{k+1} = \left\langle (T_k[2i], T_k[2i+1]) \text{ for } i = 0, 1, \ldots, \tfrac{n}{2} \right\rangle.$$

This means the size is halved in each step:

$$n_{k+1} = \tfrac{n_k}{2} = \tfrac{n}{2^k}.$$

We define a recursive suffix tree as

$$SA_{k+1} = \tfrac{1}{2} \cdot (\text{extract even entries of } SA_k)$$

where "even entries" are defined to be suffixes whose index in T_k are even.

Clearly, it is fairly easy to calculate SA_{k+1} from SA_k, and since SA_0 is known, this means that we can go top-down without much difficulty. However, in order to make this data structure work, we need to go bottom-up.

18.3.2 Bottom-Up

We need a way to represent SA_k using SA_{k+1}. To do so, we define the following functions:

is-even-suffix$_k(i)$ This tells us whether $SA_k[i]$ is an even suffix. More formally:

$$\textit{is-even-suffix}_k(i) = \begin{cases} 1 & \text{if } SA_k[i] \text{ is even} \\ 0 & \text{otherwise} \end{cases}$$

even-succ$_k(i)$ Returns j-th suffix such that $j \geq i$ and $SA_k[j]$ is even. More compact:

$$\textit{even-succ}_k(i) = \begin{cases} i & \text{if } SA_k[i] \text{ is even} \\ j & \text{if } SA_k[i] = SA_k[j] - 1 \text{ is odd} \end{cases}$$

even-rank$_k(i)$ The "even rank" of i is the number of even suffixes preceding the i^{th} suffix.

Using this functions, we can write:

$$SA_k[i] = 2 \cdot SA_{k+1}[\textit{even-rank}_k(\textit{even-succ}_k(i))] - (1 - \textit{is-even-suffix}_k(i))$$

Here, the factor of 2 is decompressing the size of the array. Ff the predicate $\textit{is-even-suffix}_k(i)$ is true, $\textit{even-succ}_k(i) = i$, so this is equivalent to saying

$$SA_k[i] = 2 \cdot SA_{k+1}[\textit{even-rank}_k(i)].$$

This basically means that we are looking up the correct value in the array SA_{k+1}. If $\textit{is-even-suffix}_k(i)$ is false, then this is equivalent to performing the same action on i's "even successor" — which is the index into SA_k of the suffix starting one position after $SA_k[i]$ — and then subtracting 1 to get the correct starting position in the text T_k. We use $(1 - \textit{is-even-suffix}_k(i))$ instead of $\textit{is-odd-suffix}_k(i)$ because we can use it to calculate $\textit{even-rank}_k(i)$ as $\textit{even-rank}_k(i) = rank_1(\textit{is-even-suffix}_k(i))$.

If we can perform the above operations in constant time and a small amount of space, then we can reduce a query on SA_k to a query on SA_{k+1} in constant time. Hence, a query on SA_0 will take $O(\ell)$ time if our maximum recursion depth is ℓ. If we set $\ell = \lg \lg n$, we will reduce the size of the text to $n_\ell = n / \lg n$. We can then use a normal suffix array, which will use $O(n_\ell \lg n_\ell) = O(n)$ bits of space, and thus be compressed. If we want to, we can further improve by setting $l = 2 \lg \lg n$ and get $n_l = n / \lg^2 n$.

18.3.3 Construction

We can store $\textit{is-even-suffix}_k(i)$ as a bit vector of size n_k. Because n_k decreases by a factor of two with each level (geometric series), this takes a total of $O(n)$ space. Then we can implement $\textit{even-rank}_k(i)$ with the rank from last lecture on our bit vector, requiring $o(n_k)$ space per level, for a total of $O(n)$ space.

Notice that $\textit{even-rank}_k(i)$ is $rank_1(i)$ on the array where $a[i] = \textit{is-even-suffix}_k(i)$. We saw in lecture 17 how to do that in $O(n_k \frac{\lg\lg n_k}{\lg n_k})$. Again, this decreases geometrically, so overall it takes $o(n)$ space.

Doing $\textit{even-succ}_k(i)$ is trivial in the case that $SA_k[i]$ is even because it is an identity function. This leaves $n_k/$ odd values, but we cannot store them explicitly because each takes $\lg n_k$ bits.

Whatever data structure we use, let's order the values of j by i; that is, if the answers are stored in array called $\textit{odd}_a\textit{nswers}$, then we would have $\textit{even-succ}_k(i) = \textit{odd}_a\textit{nswers}[i - \textit{even-rank}_k(i)]$, because $i - \textit{even-rank}_k(i)$ is the index of i among odd suffixes. This ordering is equivalent to ordering by the suffix in the suffix array, or $T_k[SA_k[i]:]$. Furthermore, this is equivalent to ordering by $(T_k[SA_k[i]], T_k[SA_k[i]+1:]) = (T_k[SA_k[i]], T_k[SA_k[\textit{even-succ}_k(i)]:])$. Finally, this is equivalent to ordering by $(T_k[SA_k[i]], \textit{even-succ}_k(i))$.

So to store $\textit{even-succ}_k(i)$, we store items of the form $(T_k[SA_k[i]], \textit{even-succ}_k(i))$. Each such item requires $(2^k + \lg n_k)$ bits, because the characters in T_k are of length 2^k and $\textit{even-succ}_k(i)$ takes $\lg n_k$ bits. This means that the first $\lg n_k$ bits won't change very often, so we can store the leading $\lg n_k$ bits of each value v_i using unary differential encoding:

$$0^{\text{lead}(v_1)}10^{\text{lead}(v_2)-\text{lead}(v_1)}10^{\text{lead}(v_3)-\text{lead}(v_2)}1\ldots$$

Where $\text{lead}(v_i)$ is the value of the leading $\lg n_k$ bits of v_i. There will then be $n_k/2$ ones (one 1 per value) and at most $2^{\lg n_k} = n_k$ zeros (the maximal value we can store with $\lg n_k$ bits is n_k which is the maximal number of incrementations), and hence at most $(3/2)n_k$ bits total used for this encoding. Again by the geometric nature of successive values of n_k, this will require $O(n)$ bits total, so the overall data structure is still compressed. We can store the remaining 2^k bits explicitly. This will take

$$2^k \cdot \frac{n_k}{2} = 2^k \frac{n}{2^{k+1}} = \frac{n}{2} \text{ bits.}$$

Note that if we maintain rank and select data structures, we can efficiently compute

$$\text{lead}(v_i) = rank_0(\text{select}_1(i)).$$

Therefore, the requirement for $even - succ_k$ is

$$\frac{1}{2}n + \frac{3}{2}n_k + O(\frac{n_k}{\log\log n_k}).$$

The total requirement for the entire structure is summing this expression plus n_k (for $is - even - suffix_k$) over all ks:

$$\sum_{k=0}^{\log\log n} \left(\frac{1}{2}n + \frac{3}{2}n_k + O\left(\frac{n_k}{\log\log n_k}\right) + n_k\right) = \frac{1}{2}n\log\log n + 5n + O\left(\frac{n_k}{\log\log n_k}\right).$$

Unfortunately, the factor $n\log\log n$ makes this not compact, so we need to improve further.

18.4 Compact suffix arrays

To reduce the space requirements of the data structure, we want to store fewer levels of recursion. We choose to store $(1 + 1/\varepsilon)$ levels of recursion, one for the following values of k:

$$0, \varepsilon\ell, 2\varepsilon\ell, ..., \ell = \lg\lg n.$$

In other words, instead of pairing two letters together with each recursive step, we are now clustering $2^{\varepsilon\ell}$ letters in a single step. We now need to be able to jump $\varepsilon\ell$ levels at once.

18.4.1 Level jumping

In order to find a formula for $SA_{k\varepsilon\ell}$ in terms of $SA_{(k+1)\varepsilon\ell}$, we proceed similarly as when our construction for compressed suffix arrays, but we redefine the word "even" as follows. A suffix $SA_{k\varepsilon\ell}[i]$ is now "even" if its index in $T_{k\varepsilon\ell}$ is divisible by $2^{\varepsilon\ell}$. This changes the definition of $\textit{even-rank}_{k\varepsilon\ell}(i)$ in the obvious way. However, we will not change the definition of $\textit{even-succ}_{k\varepsilon\ell}(i)$: it should still return the value j such that $SA_{k\varepsilon\ell}[i] = SA_{k\varepsilon\ell}[j] - 1$. It should do this for all "even" values of $SA_{k\varepsilon\ell}[i]$.

With these modified definitions, we can compute $SA_{k\varepsilon\ell}[i]$ as follows:

- Calculate the even successor repeatedly until index j is at the next level down — in other words, so that $SA_{k\varepsilon\ell}[j]$ is divisible by $2^{\varepsilon\ell}$.
- Recursively compute $SA_{(k+1)\varepsilon\ell}[\textit{even-rank}_{k\varepsilon\ell}(j)]$.
- Multiply by $2^{\varepsilon\ell}$ (the number of letters we clustered together) and then subtract the number of calls to successor in the first step.

This process works for much the same reason that it does in the compressed suffix array. We first calculate the j such that $SA_{k\varepsilon\ell}[j]$ is divisible by $2^{\varepsilon\ell}$ and $SA_{k\varepsilon\ell}[j] - m = SA_{k\varepsilon\ell}[i]$, where $0 \le m < 2^{\varepsilon\ell}$. The recursive computation gives us the index in $T_{(k+1)\varepsilon\ell}$ of the suffix corresponding to $SA_{k\varepsilon\ell}[j]$. We can compute the true value of $SA_{k\varepsilon\ell}[j]$ if we multiply the result of the recursive computation by $2^{\varepsilon\ell}$. We then subtract the value m to get $SA_{k\varepsilon\ell}[i]$.

18.4.2 Analysis

We may have to look up the even successor of an index $2^{\varepsilon\ell}$ times before getting the value we can recur on. Therefore, the total search time is $O(2^{\varepsilon\ell} \lg\lg n) = O(\lg^{\varepsilon} n \lg\lg n) = O(\lg^{\varepsilon'} n)$ for any $\varepsilon' > \varepsilon$.

We use the same unary differential encoding for successor as in the compressed construction, for a total of $2n_{k\varepsilon\ell} + n + o(n)$ bits per level in total. We also must store the $\textit{is-even-suffix}_{k\varepsilon\ell}(\cdot)$ vectors and the rank data structure, for a total of $n_{k\varepsilon\ell} + o(n)$ per level. There are $1 + 1/\varepsilon$ levels in total. Hence, the total space is something like $(6 + 1/\varepsilon)n + o(n)$ bits, which is compact.

There are some optimizations we can perform to improve the space. We don't have to store the data for $\textit{even-succ}_0(\cdot)$, because it's the top level, which means that the total space required storing

even successor information is:

$$O\left(\frac{n}{2^{\varepsilon\ell}}\right) = O\left(\frac{n}{\lg^{\varepsilon} n}\right) = o(n).$$

If we store the *is-even-suffix*$_{k\varepsilon\ell}(\cdot)$ vectors as succinct dictionaries (because they are, after all, fairly sparse), then the space required is:

$$\lg \binom{n_{k\varepsilon\ell}}{n_{(k+1)\varepsilon\ell}} \approx n_{(k+1)\varepsilon\ell} \lg \frac{n_{k\varepsilon\ell}}{n_{(k+1)\varepsilon\ell}} = n_{(k+1)\varepsilon\ell} \lg 2^{\varepsilon\ell} = \frac{n_{k\varepsilon\ell} \cdot \varepsilon\ell}{2^{\varepsilon\ell}} = \frac{n_{k\varepsilon\ell} \cdot \varepsilon \lg\lg n}{\lg^{\varepsilon} n} = o(n_{k\varepsilon\ell})$$

Hence, the total space is $o(n)$. This gives us a total space of $(1 + 1/\varepsilon)n + o(n)$ bits.

OPEN: Is it possible to achieve $o(\lg^{\varepsilon} n)$ in $O(n)$ space?

18.5 Suffix trees [99]

18.5.1 Construction

In the previous lecture, we saw how to store a binary trie with $2n+1$ nodes on $4n+o(n)$ bits using balanced parens. We can use this to store the structure of the compressed suffix tree. Unfortunately, we don't have enough space to store the edge lengths or the letter depth, which would allow us to traverse the tree with no further effort.

To search for a pattern P in the tree, we must calculate the letter depth as we go along. Say that we know the letter depth of the current node x. To descend to its child y, we need to compute the difference in letter depths, or the length in letters of the edge between them.

The letter depth of y is equivalent to the length of the substring shared by the leftmost descendant of y and the rightmost descendant of y. Let ℓ be the leftmost descendant, and let r be the rightmost descendant. If we know the index in the suffix array of both ℓ and r, then we can use the suffix array to find their indices in the text. Because ℓ and r are both descendants of x, we know that they both match for *letter-depth*(x) characters. So we can skip the first *letter-depth*(x) characters of both, and start comparing the characters of ℓ, r, and P. If P differs from ℓ and r before they differ from each other, we know that there are no suffixes matching P in the tree, and we can stop the whole process. Otherwise, ℓ and r will differ from each other at some point, which will give us the letter depth of y. Note that the total number of letter comparisons we perform is $O(|P|)$, for the entire process of searching the tree.

18.5.2 Necessary binary trie operations

To find ℓ, r, and their indices into the suffix array, note that in the balanced parentheses representation of the trie, each leaf is the string "())".

leaf-rank(here**)** The number of leaves to the left of the node which is at the given position in the string of balanced parentheses. Can be computed by getting $rank_{())}((n)$

leaf-select(i) The position in the balanced parentheses string of the i^{th} leaf. Can be computed by calculating $select_{()}(i)$.

leaf-count(here**)** The number of leaves in the subtree of the node at the given position in the string of balanced parens. Can be computed using the formula:

$$rank_{()}(\text{matching) of parent}) - rank_{()}(\text{here}).$$

leftmost-leaf(here**)** The position in the string of the leftmost leaf of the node at the given position. Given by the formula:

$$\textit{leaf-select}(\textit{leaf-rank}(\text{here}) + 1).$$

rightmost-leaf(here**)** The position in the string of the rightmost leaf of the node at the given position. Given by the formula:

$$\textit{leaf-select}(\textit{leaf-rank}(\text{matching) of parent} - 1)).$$

Hence, we can use a rank and select data structure on the string of balanced parentheses to find the first and last leaves in the part of the string representing the subtree rooted at y. We can then calculate the rank of those leaves to determine the index into the suffix array. Hence, we can perform the search described in the previous paragraph at a cost of $O(1)$ operations per node.

18.5.3 Analysis

The total time required to search for the pattern P is

$$O(|P| + |\text{output}|) \cdot O(\text{cost of suffix array lookup}).$$

Grossi and Vitter [98] improved upon this to achieve a search time of

$$O\left(\frac{P}{\log_\Sigma T} + |\text{output}| \cdot \log_\Sigma^\epsilon T\right)$$

18.5.4 Succinct Suffix Trees

It is also possible to improve this, creating a succinct suffix tree given a suffix array. In the above algorithm, storing the suffix tree takes too much space to achieve succinctness. Instead, we store the above compact suffix tree on every b^{th} entry in the suffix array, which gives us an extra storage cost of $O(n/b)$.

First, modify the tree to return something reasonable if P doesn't match any of the items in the tree, such as returning the predecessor of P in the set of leaves. If we consider the suffixes to be divided into blocks of size b, then when we query on P, the suffix tree will give us an interval of block dividers such that any suffix matching P must lie in a block touching one of those dividers. This gives us a range of blocks in which to look for the true answer.

The rest of the algorithm was not covered in lecture, but is explained in [99] and in the handwritten lecture notes. The main idea is to use a look-up table which stores the following information: given

a sorted array A of b length (at most) b bit-strings and a pattern string P of length at most b^2, our table stores the first and last positions of P in A. Note that this takes space $O(2^{b^2+b} \lg b) = O(\sqrt{n})$ bits of space if $b \leq \frac{1}{2}\sqrt{\lg n}$. We can then use this to efficiently find the first and last match in a block.

Lecture 19

Dynamic graphs 1

Scribes: Justin Holmgren (2012), Jing Jian (2012), Maksim Stepanenko (2012) Mashhood Ishaque (2007)

19.1 Overview

In this lecture we discuss dynamic trees that have many applications such as Network Flow and Dynamic Connectivity problems in addition to them being interesting theoretically. This is the beginning of our discussion of dynamic graphs, where in general we have an undirected graph and we want to support deletions and insertions. We will discuss one data structure called Link-Cut trees that achieves logarithmic amortized time for all operations.

19.2 Link-cut Trees

Using link-cut trees we want to maintain a forest of rooted trees whose each node has an arbitrary number of unordered child nodes. The data structure has to support the following operations in $O(\lg n)$ amortized time:

- *make_tree()* – Returns a new vertex in a singleton tree. This operation allows us to add elements and later manipulate them.
- *link(v,w)* – Makes vertex v a new child of vertex w, i.e. adds an edge (v, w). In order for the representation to remain valid this operation assumes that v is the root of its tree and that v and w are nodes of distinct trees.
- *cut(v)* – Deletes the edge between vertex v and its parent, $parent(v)$ where v is not the root.
- *find_root(v)* – Returns the root of the tree that vertex v is a node of. This operation is interesting because path to root can be very long. The operation can be used to determine if two nodes u and v are connected.

- *path_aggregate(v)* – Returns an aggregate, such as max/min/sum, of the weights of the edges on the path from the root of the tree to node v. It is also possible to augment the data structure to return many kinds of statistics about the path.

Link-Cut Trees were developed by Sleator and Tarjan [108] [109]. They achieve logarithmic amortized cost per operation for all operations. Link-Cut Trees are similar to Tango trees in that they use the notions of preferred child and preferred path. They also use splay trees for the internal representation.

19.3 Preferred-path decomposition

Link-cut trees are all about paths in the tree, so we want to split a tree into paths. We need to define some terms first.

19.3.1 Definition of Link-Cut Trees

We say a vertex has been *accessed* if was passed to any of the operations from above as an argument. We call the abstract trees that the data structure represents **represented trees**. We are not allowed to change the represented tree and it can be unbalanced. The represented tree is split into paths (in the data structure representation).

The **preferred child** of node v is equal to its i-th child if the last access within v's subtree was in the i-th subtree and it is equal to null if the last access within v's subtree was to v itself or if there were no accesses to v's subtree at all. A **preferred edge** is an edge between a preferred child and its parent. A **preferred path** is a maximal continuous path of preferred edges in a tree, or a single node if there is no preferred edges incident on it. Thus preferred paths partition the nodes of the represented tree.

Link-Cut Trees represent each tree T in the forest as a tree of **auxiliary trees**, one auxiliary tree for each preferred path in T. Auxiliary trees are splay trees with each node keyed by its depth in its represented tree. Thus for each node v in its auxiliary tree all the elements in its left subtree are higher(closer to the root) than v in v's represented tree and all the elements in its right subtree are lower. Auxiliary trees are joined together using **path-parent pointers**. There is one path-parent pointer per auxiliary tree and it is stored in the root of the auxiliary tree. It points to the node that is the parent(in the represented tree) of the topmost node in the preferred path associated with the auxiliary tree. We cannot store path-to-child pointers because there can be many children. Including auxiliary trees with the path-parent pointers as edges, we have a representation of a represented tree as a tree of auxiliary trees which potentially can have a very high degree.

19.3.2 Operations on Link-Cut Trees

Access

All operations above are implemented using an *access(v)* subroutine. It restructures the tree T of auxiliary trees that contains vertex v so that it looks like v was just accessed in its represented tree

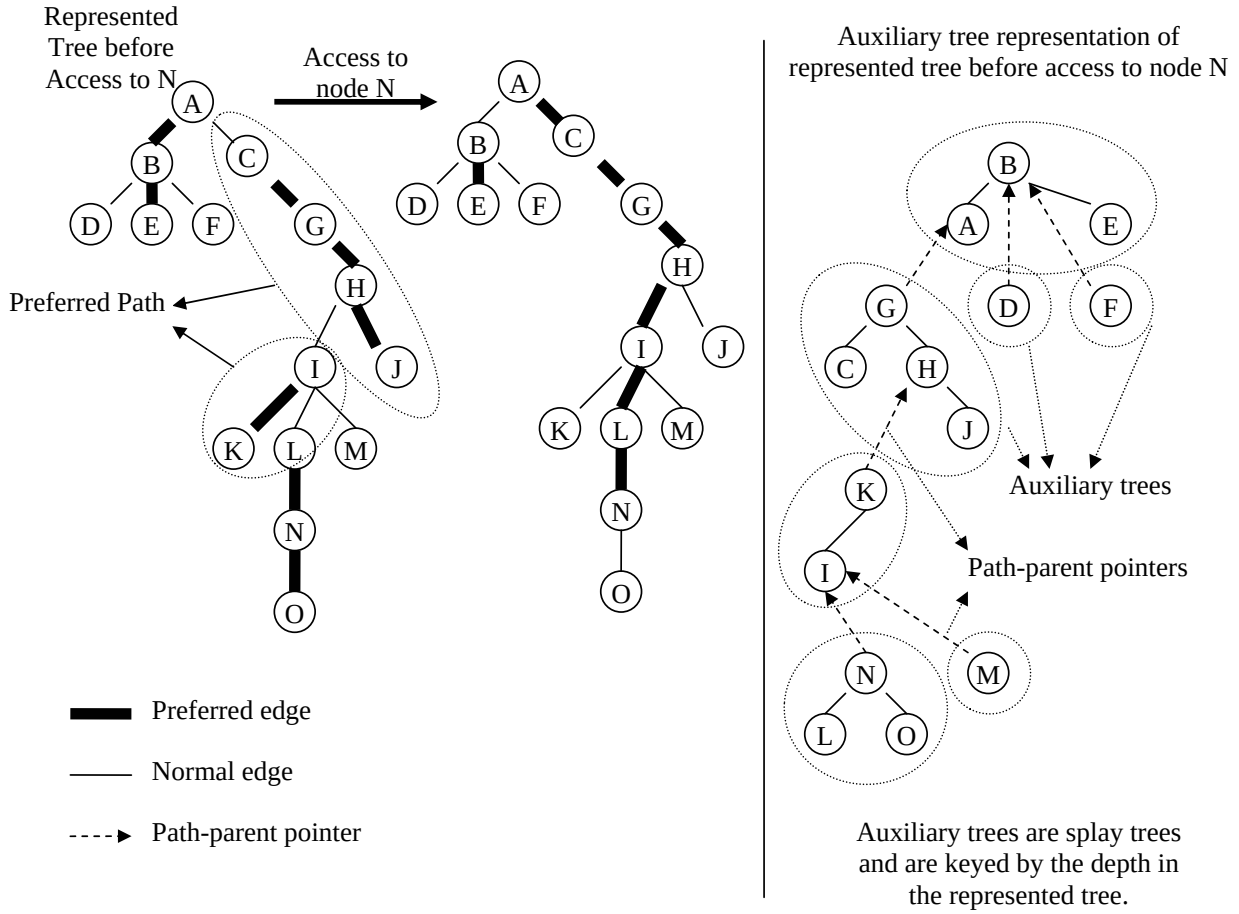


Figure 1: Structure of Link-Cut Trees and Illustration of Access Operation

R. When we access a vertex v some of the preferred paths change. A preferred path from the root of R down to v is formed. When this preferred path is formed every edge on the path becomes preferred and all the old preferred edges in R that had an endpoint on this path are destroyed, and replaced by path-parent pointers.

Remember that the nodes in auxiliary trees are keyed by their depth in R. Thus nodes to the left of v are higher than v and nodes to the right are lower. Since we access v, its preferred child becomes *null*. Thus, if before the access v was in the middle of a preferred path, after the access the lower part of this path becomes a separate path. What does it mean for v's auxiliary tree? This means that we have to separate all the nodes less than v in a separate auxiliary tree. The easiest way to do this is to splay on v, i.e. bring it to the root and then disconnect it right subtree, making it a separate auxiliary tree.

After dealing with v's descendants, we have to make a preferred path from v up to the root of R. This is where path-parent pointer will be useful in guiding us up from one auxiliary tree to another. After splaying, v is the root and hence has a path-parent pointer (unless it is the root of T) to its parent in R, call it w. We need to connect v's preferred path with the w's preferred path, making

ACCESS(v)

- Splay v within its auxiliary tree, i.e. bring it to the root. The left subtree will contain all the elements higher than v and right subtree will contain all the elements lower than v
- Remove v's preferred child.
  - path-parent(right(v)) $\leftarrow v$
  - right(v) $\leftarrow null$ (+ symmetric setting of parent pointer)
- loop until we reach the root
  - $w \leftarrow$ path-parent(v)
  - splay w
  - switch w's preferred child
    - path-parent(right(w)) $\leftarrow w$
    - right(w) $\leftarrow v$ (+ symmetric setting of parent pointer)
    - path-parent(v) $\leftarrow null$
  - $v \leftarrow w$
- splay v just for convenience

w the real parent of v, not just path-parent. In other words, we need to set w's preferred child to v. This is a two stage process in the auxiliary tree world. First, we have to disconnect the lower part of w's preferred path the same way we did for v (splay on w and disconnect its right subtree). Second we have to connect v's auxiliary tree to w's. Since all nodes in v's auxiliary tree are lower than any node in w's, all we have to do is to make v auxiliary tree the right subtree of w. To do this, we simply disconnect w with it's right child, and change the pointer to point to v. Finally, we have to do minor housekeeping to finish one iteration: we do a second splay of v. Since v is a child of the root w, splaying simply means rotating v to the root. Another thing to note is that we keep the path-parent pointer at all stages of concatenation.

We continue building up the preferred path in the same way, until we reach the root of R. v will have no right child in the tree of auxiliary trees. The number of times we repeat is equal to the number of preferred-child changes.

Find Root

FIND_ROOT operation is very simple to implement after we know how to handle accesses. First, to find the root of v's represented tree, we access v thus make it on the same auxiliary tree as the root of the represented tree. Since the root of the represented tree is the highest node, its key in the auxiliary tree is the lowest. Therefore, we go left from v as much as we can. When we stop, we have found the root. We splay on it and return it. The reason we need to splay is the root might be linearly deep in the tree; by splaying it to the top, we ensure that it is fast to find root next time.

FIND_ROOT(v)

- access(v)
- Set v to the smallest element in the auxiliary tree, i.e. to the root of the represented tree
  - $v \leftarrow$ left(v) until left(v) is $null$
- access r
- return r

19.3.3 Path Aggregate

Like all of our operations, the first thing we do is access(v). This is an aux. tree representing the path down to v. We augment the aux. tree with the values you care about, such as "sum" or "min" or "max." We will see that it is easy to maintain such augmentations when we look at cut and link.

PATH-AGGREGATE(v)

- access(v)
- return v.subtree-sum (augmentation within each aux. tree)

Cut

To cut $(v, parent(v)$ edge in the represented tree means that we have to separate nodes in v's subtree (in represented tree) from the tree of auxiliary trees into a separate tree of auxiliary trees. To do this we access v first, since it gathers all the nodes higher than v in v's left subtree. Then, all we need to do is to disconnect v's left subtree (in auxiliary tree) from v. Note that v becomes in an auxiliary tree all by itself, but path-parent pointer from v's children (in represented tree) still point to v and hence we have the tree of auxiliary trees with the elements we wanted. Therefore there are two trees of aux trees after the cut.

CUT(v)

- access(v)
- left(v) $\leftarrow null$ (+ symmetric setting of parent pointer)

Link

Linking two represented trees is also easy. All we need to do is to access both v and w so that they are at the roots of their trees of auxiliary trees, and make latter left child of the former.

LINK(v, w)

- access(v)
- access(w)
- left(v) $\leftarrow w$ (+ symmetric setting of parent pointer)

19.4 Analysis

As one can see from the pseudo code, all operations are doing at most logarithmic work (amortized, because of the splay call in find_root) plus an access. Thus it is enough to bound the run time of access. First we show an $O(\lg^2 n)$ bound.

19.4.1 An $O(\lg^2 n)$ bound.

From access's pseudo code we see that its cost is the number of iterations of the loop times the cost of splaying. We already know from previous lectures that the cost of splaying is $O(\lg n)$ amortized (splaying works even with splits and concatenations). Recall that the loop in access has $O(\#$ preferred child changes) iterations. Thus to prove the $O(\lg^2 n)$ bound we need to show that the number of preferred child changes is $O(\lg n)$ amortized. In other words the total number of preferred child changes is $O(m \lg n)$ for a sequence of m operations. We show this by using the Heavy-Light Decomposition of the represented tree.

The Heavy-Light Decomposition

The Heavy-Light decomposition is a general technique that works for any tree (not necessarily binary). It calls each edge either heavy or light depending on the relative number of nodes in its subtree.

Let $size(v)$ be the number of nodes in v's subtree (in the represented tree).

Definition 45. *An edge from vertex $parent(v)$ to v is called **heavy** if $size(v) > \frac{1}{2} size(parent(v))$, and otherwise it is called **light**.*

Furthermore, let light-depth(v) denote the number of light edges on the root-to-vertex path to v. Note that light-depth(v) $\leq \lg n$ because as we go down one light edge we decrease the number of nodes in our current subtree at least a factor of 2. In addition, note that each node has at most one heavy edge to a child, because at most one child subtree contains more than half of the nodes of its parent's subtree. There are four possibilities for edges in the represented tree: they can be preferred or unpreferred and heavy or light.

Proof of the $O(\lg^2 n)$ bound

To bound the number of preferred child changes, we do Heavy-Light decomposition on represented trees. Note that access does not change the represented tree, so it does not change the heavy or light classification of edges. For every change of preferred edge (possibly except for one change to the preferred edge that comes out of the accessed node) there exists a newly created preferred edge. So, we count the number of edges which change status to being preferred. Per operation, there are at most $\lg n$ edges which are light and become preferred (because all edges that become preferred are on a path starting from the root, and there can be at most $\lg n$ light edges on a path by the observation above). Now, it remains to ask how many heavy edges become preferred. For any one operation, this number can be arbitrarily large, but we can bound it to $O(\lg n)$ amortized. How come? Well, during the entire execution the number of events "heavy edge becomes preferred" is

bounded by the number of events "heavy edge become unpreferred" plus $n-1$ (because at the end, there can be $n-1$ heavy preferred edges and at the beginning the might have been none). But when a heavy edge becomes unpreferred, a light edge becomes preferred. We've already seen that there at most $\lg n$ such events per operation in the worst-case. So there are $\leq \lg n$ events "heavy edge becomes unpreferred" per operation. So in an amortized sense, there are $\leq \lg n$ events "heavy edge becomes preferred" per operation (provided $(n-1)/m$ is small, i.e. there is a sufficiently large sequence of operations).

19.4.2 An $O(\lg n)$ bound.

We prove the $O(\lg n)$ bound by showing that the cost of preferred child switch is actually $O(1)$ amortized. From access's pseudo code one can easily see that its cost is

$$O(\lg n) + (\textit{cost of pref child switch}) * (\#\ \textit{pref child switches})$$

From the above analysis we already know that the number of preferred child switches is $O(\lg n)$, thus it is enough to show that the cost of preferred child switch is $O(1)$. We do it using the potential method.

Let $s(v)$ be the number of nodes under v in the tree of auxiliary trees. Then we define the potential function $\Phi = \sum_v \lg s(v)$. From our previous study of splay trees, we have the Access theorem, which states that:

$$\textit{cost}(\textit{splay}(v)) \leq 3(\lg s(u) - \lg s(v)) + 1,$$

where u is the root of v's auxiliary tree. Now note that splaying v affects only values of s for nodes in v's auxiliary tree and changing v's preferred child changes the structure of the auxiliary tree but the tree of auxiliary trees remains unchanged. Therefore, on access(v), values of s change only for nodes inside v's auxiliary tree. Also note that if w is the parent of the root of auxiliary tree containing v, then we have that $s(v) \leq s(u) \leq s(w)$. Now we can use this inequality and the above amortized cost for each iteration of the loop in access. The summation telescopes and is less than

$$3(\lg s(\text{root of represented tree}) - \lg s(v)) + O(\#\ \textit{preferred child changes})$$

which in turn is $O(\lg n)$ since $s(\textit{root}) = n$. Thus the cost of access is thus $O(\lg n)$ amortized as desired.

To complete the analysis we resolve the worry that the potential might increase more than $O(\lg n)$ after cutting or joining. Cutting breaks up the tree into two trees thus values of s only decrease and thus Φ also decreases. When joining v and w, only the value of s at v increases as it becomes the root of the tree of auxiliary trees. However, since $s(v) \leq n$, the potential increases by at most $\lg s(v) = \lg n$. Thus increase of potential is small and cost of cutting and joining is $O(\lg n)$ amortized.

Lecture 20

Dynamic graphs 2

Scribes: Josh Alman (2012), Vlad Firoiu (2012), Di Liu (2012) TB Schardl (2010)

20.1 Overview

Last lecture we covered dynamic trees, also known as link-cut trees. Link-cut trees are able to represent a dynamic forest of rooted trees in $O(\log n)$ amortized time per operation.

In this lecture we will see how to maintain connectivity information for general graphs. We will start by examining a simpler, although not strictly better, alternative to link-cut trees known as Euler-tour trees. Then, for the special case of *Decremental Connectivity*, in which edges may only be deleted, we will achieve constant runtime. We will then use Euler-tour trees to achieve dynamic connectivity in general graphs in $O(\log^2 n)$ time. Finally we will survey some of what is and is not known for dynamic graphs. The next class will be about lower bounds.

20.2 Dynamic Connectivity

Our goal is to maintain an *undirected* graph subject to:

- `insert`/`delete` of edges or vertices (with no edges).
- `connected`(u, v): whether u and v are connected, on general undirected graphs that are subject to vertex and edge insertion and deletion.
- Another possible query is to determine whether the entire graph is connected; though this may seem easier, the current lower and upper bounds are the same as for pairwise connectivity.

We will focus on the former type of query.

The different types of updates that we will consider are:

- **Fully Dynamic**. Supports insertion and deletion of vertices and edges.
- **Incremental**. Supports only insertions of vertices and edges.
- **Decremental**. Supports only deletions of vertices and edges.

20.2.1 Results in Dynamic Connectivity

There are several results for dynamic connectivity in specific types of graphs.

- Trees. For trees, we can support $O(\log n)$ time for queries and updates using Euler-Tour trees or Link-Cut trees. If the trees are only decremental, then we can support queries in constant time, as we will see in this lecture.
- Planar graphs. Eppstein et al. have proven that we can attain $O(\log n)$ queries and updates [111].

Here are some results for general dynamic graphs.

- In 2000 Thorup showed how to obtain $O(\log n(\log\log n)^3)$ updates and $O(\log n/\log\log\log n)$ queries [112].
- Holm, de Lichtenberg, and Thorup obtained $O(\log^2 n)$ updates and $O(\log n/\log\log n)$ queries [113].
- For the incremental data structure, the best result we have is $\Theta(\alpha(m,n))$ using union-find [123].
- For the decremental data structure, we have a $O(m\log n + \text{polylog}n + \ \#\text{queries})$ solution, which essentially amounts to an $O(\log n)$ solution for dense graphs [124].
- **OPEN**: It remains an open problem whether $O(\log n)$ queries and updates are attainable for general graphs.

All of the aforementioned runtimes are amortized. Less is known about worst-case bounds:

- Eppstein et al. showed that an $O(\sqrt{n})$ worst-case update and $O(1)$ query time is achievable [115].
- **OPEN**: It remains an open problem whether we can achieve $O(\text{poly}\log n)$ worst-case updates and queries.

Patrascu and Demaine proved two lower bounds on dynamic connectivity, namely that an $O(x\log n)$ update requires an $\Omega(\log n/\log x)$ query time, and an $O(x\log n)$ query requires an $\Omega(\log n\log x)$ update time for $x > 1$ [114]. We will be focusing more on lower bounds next lecture. Note that [112, 113, 115] are all optimal in a sense, by the trade-off bounds shown in [114].

We end with an **OPEN** question: are $o(\log n)$ update and $\text{poly}\log n$ query achievable?

20.3 Euler-Tour Trees

Euler-Tour trees are due to Henzinger and King [110] and are an alternative to link-cut trees for representing dynamic trees. Euler-tour trees are simpler and easier to analyze than link-cut trees, but do not naturally store aggregate information about paths in the tree. Euler-tour trees are well suited for storing aggregate information on subtrees, which is a feature we will use in the next section.

The idea behind Euler-Tour trees is to store the Euler tour of the tree. In an arbitrary graph, an Euler tour is a path that traverses each edge exactly once. For trees we say that each edge is bidirectional, so the Euler tour of a tree is the path through the tree that begins at the root and ends at the root, traversing each edge exactly twice — once to enter the subtree, and once to exit it. The Euler tour of a tree is essentially the depth-first traversal of a tree that returns to the root at the end. The correspondence between a tree and its Euler tour is shown in Figure 20.1.

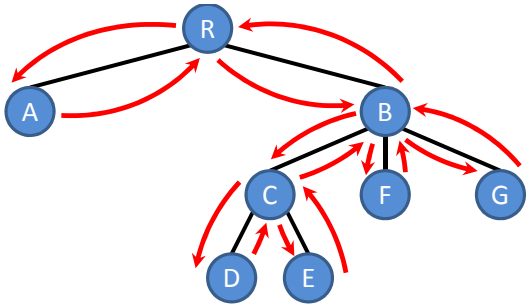


Figure 20.1: An example tree and its Euler tour. The order in which nodes in the tree are visited is indicated by the curved red arrows. The Euler tour of the tree shown here corresponds to the sequence: $R\ A\ R\ B\ C\ D\ C\ E\ C\ B\ F\ B\ G\ B\ R$.

In an Euler-Tour tree, we store the Euler tour of a represented tree in a balanced binary search tree (BST). For some represented tree, each visit to a node in that tree's Euler tour corresponds to a node in the BST. Each node in the represented tree holds pointers to the nodes in the BST representing the first and last times it was visited. As an example, the pointers between the root of the tree in Figure 20.1 and its Euler-tour node visitations is shown in Figure 20.2.

An Euler-Tour tree supports the following operations:

`FindRoot`(v) Find the root of the tree containing node v. In the Euler tour of a tree, the root is visited first and last. Therefore we simply return the minimum or maximum element in the BST.

`Cut`(v) Cut the subtree rooted at v from the rest of the tree. Note that the Euler tour of v's subtree is a contiguous subsequence of visits that starts and ends with v, contained in the sequence of visits for the whole tree. To cut the subtree rooted at v, we may simply split the BST before its first and after its last visit to v. This splitting gives us the Eulertour of the tree before reaching v, the tour of v's subtree, and the tour of the tree after leaving v. Concatenating the first and last pieces together, and possibly deleting one redundant visitation between the end of the first piece and beginning of the last, gives us our answer.

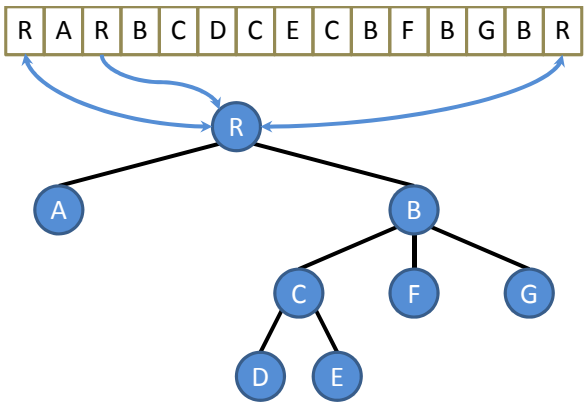


Figure 20.2: We represent a tree's Euler tour as a sequence of visitations starting and ending at the root. Pointers from that sequence point to the node, and pointers from the node point to the first and last visitations. In this figure only the pointers relating to the root are shown.

`Link`(u, v) Insert u's subtree as a child of v. In the resulting Euler tour, we need to traverse u's subtree immediately after and immediately before visits to v. Therefore we will split v's traversal before the last visit to v, and then concatenate onto the left piece a singleton visit to v, followed by the Euler tour of u's subtree, followed by the right piece.

`Connectivity`(v, w) Two nodes are connected if they are in the same rooted tree, so it suffices to check that `FindRoot`$(v) =$ `FindRoot`(w).

`Subtree Aggregate`(v) Compute the min, max, sum, etc. of the nodes in the subtree rooted at v. This is accomplished through a range query in the BST between the first and last occurrence of v.

Each of these operations performs a constant number of search, split, and merge operations on the Euler-tour tree. Each of these operations takes $O(\log n)$ per operation on a balanced BST data structure. Consequently, the total running time for Euler-Tour trees is $O(\log n)$ per operation. If we use a B-tree with fanout of $\Theta(\log n)$ instead of a balanced BST in our Euler-tour trees, we can achieve $O(\log n / \log \log n)$ searches (from the depth of the tree) and $O(\log^2 n / \log \log n)$ updates (from the depth times the branching factor).

20.3.1 Decremental Connectivity in Trees

We are going to achieve constant amortized time per operation if we are only working with decremental trees. We assume that all edges eventually get deleted for our amortized analysis to work. We make a few observations:

1. We can easily get $O(\log n)$ per operation by using link-cut or Euler-tour trees.
2. To reduce this, we use leaf trimming: we cut below the maximally-deep nodes that have at least $\log n$ descendants. The top tree will have $O(n / \log n)$ branching nodes; compressing

paths results in a tree of size $O(n/\log n)$. We can use a link-cut or Euler-tour tree on this structure, giving a total time of $O(n)$ for queries on the top tree.

3. The bottom trees have only $O(\log n)$ edges, so we can store the edge set as a bit vector (only one machine word). We also preprocess a lookup table storing the path to the root from an any vertex as a bit vector on edges. This allows us to determine whether there is still a path to the root in constant time (compute the bitwise AND between the path to the root and the current edge set).

4. Finally, we need to answer queries about getting between nodes in paths that we compressed in 2. We can split each path into $O(n/\log n)$ chunks of size $\log n$. Each chunk can be stored as a bit vector. Then, we can answer queries within a chunk by masking the bit vector. When we answer queries across chunks, we can treat each chunk in-between as existing if the chunk is the all 1 vector, and not existing otherwise, then use a structure from 1. Thus, we can answer all path queries in $O(\log n)$ time, and like before this gives us $O(n)$ time for all the queries.

20.3.2 Dynamic Connectivity in $O(\log^2 n)$

We will examine the dynamic connectivity algorithm described in [113], which achieves an $O(\log^2 n)$ amortized bound.

The high-level idea for this data structure is to store a spanning forest (one spanning tree for each connected component) for the input graph G using an Euler-Tour tree. With this idea alone, we see that we can answer connectivity queries by checking if both input nodes are in the same tree; this can be done by performing find root on each and seeing whether they are the same, since the root is a canonical name for a tree. An `Insert`(u, v) can be done by first checking if u and v are in the same tree, and if not adding (u, v) to the spanning forest. The problem with this approach is `Delete`(u, v), and in particular we cannot tell whether deleting (u, v) will disconnect u and v. To deal with deletes, we will modify our initial idea by hierarchically dividing G into $\log n$ subgraphs and then storing a minimum spanning forest for each subgraph.

To implement this idea, we start by assigning a *level* to each edge. The level of an edge is an integer between 0 and $\log n$ inclusive that can only decrease over time. (We will use the level of each edge as a charging mechanism.) We define G_i to be the subgraph of G composed of edges at level i or less. Note that $G_{\log n} = G$. Let F_i be the spanning forest of G_i. We will implement our F_i's using Euler-Tour trees built on B-trees.

During the execution of this algorithm we will maintain the following two invariants:

We will also maintain the adjacency matrix for each G_i.

We implement the three operations on this data structure as follows:

`Connected`(u, v) Check if vertices u and v are connected. To do this, we first query $F_{\log n}$ to see if u and v are in the same tree. This can be done by checking $F_{\log n}$ if `Findroot`$(u) =$ `Findroot`(v). This costs $O(\log n/\log\log n)$ using B-tree based Euler-Tour trees.

`Insert`$(e = (u, v))$ Insert edge $e = (u, v)$ into the graph. To do this, we first set the level of edge e to $\log n$ and update the adjacency lists of u and v. If u and v are in separate trees in $F_{\log n}$, add e to $F_{\log n}$. This costs $O(\log n)$.

`Delete`$(e = (u, v))$ Remove edge $e = (u, v)$ from the graph. To do this, we first remove e from the adjacency lists of u and v. If e is not in $F_{\log n}$, we're done. Otherwise:

1. Delete e from F_i for all $i \geq \text{level}(e)$.

   Now we want to look for a replacement edge to reconnect u and v.

   - Note that the replacement edge cannot be at a level less than level(e) by Invariant **??** (recall that each F_i is a minimum spanning forest).
   - We will start searching for a replacement edge at level(e) to preserve the Invariant refinv:subset.

   We will look for this replacement edge by doing the following:

2. For $i = \text{level}(e)$ to $\log n$:

   (a) Let T_u be the tree containing u, and let T_v be the tree containing v. WLOG, assume $|T_v| \leq |T_u|$.

   (b) By Invariant **??**, we know that $|T_u| + |T_v| \leq 2^i$, so $|T_v| \leq 2^{i-1}$. This means that we can afford to push all edges of T_v down to level $i - 1$.

   (c) For each edge (x, y) at level i with x in T_v:

      i. If y is in T_u, add (x, y) to $F_i, F_{i+1}, \ldots, F_{\log n}$, and stop.

      ii. Otherwise set level$(x, y) \leftarrow i - 1$.

In order for `Delete` to work correctly, we must augment our Euler-Tour trees. First, each Euler-Tour tree must keep track of its subtree sizes in order to find which of $|T_v|$ and $|T_u|$ is smaller in $O(1)$. (This augmentation is standard and easy.) We also need to augment our trees to know for each node v in the minimum spanning forest F_i whether or not v's subtree contains any nodes incident to level-i edges. We can augment the tree to keep track of whether a subtree rooted at some internal node contains any such nodes. With this augmentation we can find the next level-i edge incident to $x \in T_v$ in $O(\log n)$ time using successor and jumping over empty subtrees.

Lines 1 and 2(c)i cost $O(\log^2 n)$ total, while 2(c)ii in the loop costs $O(\log n \cdot \#$ of edges decremented) total. Note that an edge's level cannot be lower than 0, so an edge is decremented at most $O(\log n)$ times. Therefore, the amortized cost of `Delete` is $O(\log^2 n)$ as desired.

20.4 Other Dynamic Graph Problems

We also have results for other kinds of dynamic graph problems.

k-Connectivity

A pair of vertices (v, w) is *k-connected* if there are k vertex-disjoint (or edge-disjoint) paths from v to w. A graph is k-connected if each vertex pair in the graph is k-connected. To determine whether the whole graph is k-connected can be solved as a max-flow problem. An $O(\sqrt{n}\text{poly}(\lg n))$ algorithm for $O(\text{poly}(\lg n))$-edge-connectivity was shown in [117]. There have been many results for k-connectivity between a single pair of vertices:

- $O(\text{poly}(\lg n))$ for $k = 2$ [113]
- $O(\lg^2 n)$ for planar, decremental graphs [118]

The above are amortized bounds. There have also been worst case bounds shown in [115]:

- $O(\sqrt{n})$ for 2-edge-connectivity
- $O(n)$ for 2-vertex-connectivity and 3-vertex-connectivity
- $O(n^{2/3})$ for 3-edge-connectivity
- $O(n\alpha(n))$ for $k = 4$
- $O(n \lg n)$ for $O(1)$-edge-connectivity
- **OPEN:** Is $O(\text{poly}(\lg n))$ achievable for $k = O(1)$? Or perhaps $k = \text{poly}(\lg n)$? This problem is open for whole graph k-connectivity as well.

Minimum spanning forest — $O(\log^4 n)$ updates can be obtained [113], as well as an $O(\sqrt{n})$ worst-case update [115] for general graphs and a $O(\log n)$ update on planar graphs [111]. Bipartiteness, also known as 2-colorability, is reducible to minimum spanning forest.

Planarity testing — in which we ask if inserting some edge (u, v) into our graph violates planarity — can be done in general in $O(n^{2/3})$ [119]. For a fixed embedding, planarity testing can be done in $O(\log^2 n)$ [115]. La Poutré showed an $O(\alpha(m, n) \cdot m + n)$ algorithm for a total of m operations with an incremental graph in [121].

Directed graphs — we now want to know if vertex v is connected to vertex w in a directed graph. Notice that the problem is now no longer transitive. We assume that insertions and deletions are done in bulk, meaning we insert or delete a vertex and all its incident edges at the same time. The current results are:

- $O(n^2)$ amortized updates, O(1) worst-case query [130] [134]
- Sankowski et al. showed the same bound in worst case. This is optimal if we are explicitly storing the transitive closure matrix [133]
- **OPEN**: $o(n^2)$ worst-case for all updates?
- $O(m\sqrt{n}t)$ amortized bulk update, $O(\sqrt{n}/t)$ worst case query for any $t = O(\sqrt{n})$ [132]
- $O(m + n \lg n)$ amortized bulk update, $O(n)$ worst case query [131]
- **OPEN**: Full trade-off: number of updates times number of querries $O(mn)$ or $O(n^2)$?

- On an acyclic graph: $O(n^{1.575}t)$ update, $O(n^{0.575}/t)$ query, with once again $t = O(\sqrt{n})$ [130].

All-pairs shortest paths — weight of shortest $v \to w$ path for all pairs of vertices, still with undirected graphs. Results include:

- $O(n^2(\lg n + \lg^2(1 + m/n)))$ amortized bulk update, $O(1)$ worst case query [129] (improvised on [128])
- **OPEN**: $O(n^2)$ or $o(n^2)$ update, even with undirected graphs?
- $O(n^{2.75})$ worst case update, $O(1)$ query [127]
- For unweighted graphs, $O(m\sqrt{n}$ polylg $n)$ amortizes updates, $O(n^{3/4})$ worst case query [126]
- undirected, unweighted and $(1+\epsilon)$-approx: $O(\sqrt{mn}t)$ amortized update, $O(\sqrt{m}/t)$ worst case query, $t = O(\sqrt{n})$ [125]

Lecture 21

Dynamic graphs 3

Scribes: R. Cohen(2012), C. Sandon(2012), T. Schultz(2012), M. Hofmann(2007)

21.1 Overview

In the last lecture we introduced Euler tour trees [137], dynamic data structures that can perform link-cut tree operations in $O(\lg n)$ time. We then showed how to implement an efficient dynamic connectivity algorithm using a spanning forest of Euler tour trees, as demonstrated in [138]. This yielded an amortized time bound of $O(\lg^2 n)$ for update operations (such as edge insertion and deletion), and $O(\lg n / \lg \lg n)$ for querying the connectivity of two vertices. In this lecture, we switch to examining the lower bound of dynamic connectivity algorithms. Until recently, the best lower bound for dynamic connectivity operations was $\Omega(\lg n / \lg \lg n)$, as described by Fredman and Henzinger in [135] and independently by Miltersen in [136]. However, we will show below that it is possible to prove $\Omega(\lg n)$, using the method given by Pătraşcu and Demaine in [140].

21.2 Cell Probe Complexity Model

The $\Omega(\lg n)$ bound relies on a model of computation called the *cell probe complexity model*, originally described in the context of proving dynamic lower bounds by Fredman and Saks in [139]. The cell probe model views a data structure as a sequence of *cells*, or words, each containing a w-bit field. The model calculates the complexity of an algorithm by counting the number of reads and writes to the cells; any additional computation is free. This makes the model comparable to a RAM model with constant time access. Because computation is free, the model is not useful for determining upper bounds, only lower bounds.

We empirically assume that the size of each cell, w, is at least $\lg n$. This is because we would like the cells to store pointers to each of our n vertices, and information theory tells us that we need $\lg n$ bits to address n items. For the the following proof, we will further assume that $w = \Theta(\lg n)$. In this sense, the cell probe model is a *transdichotomous model*, as it provides a bridge between the problem size, n, and the cell or machine size, w.

21.3 Dynamic Connectivity Lower Bound for Paths

The lower bound we are trying to determine is the best achievable worst-cast time for a sequence of updates and queries to a path. We will prove the following:

Theorem 46. *Under the cell probe model, the lower bound worst-case cost is* $\Omega(\lg n)$ *per operation.*

It is possible to show that the lower bound is also $\Omega(\lg n)$ in the amortized case, but we will only examine the worst case cost.

21.3.1 Path Grid

We start by arranging the n vertices in a $\sqrt{n}$ by $\sqrt{n}$ grid, as shown in Figure 1. The grid has perfect matching between consecutive columns C_i and C_{i+1}. This results in a graph with $\sqrt{n}$ disjoint paths across the columns. The paths can be coded by the permutations $\pi_1, \pi_2, \ldots \pi_{\sqrt{n}-1}$, where π_i is the permutation of edges that joins column C_i and C_{i+1}.

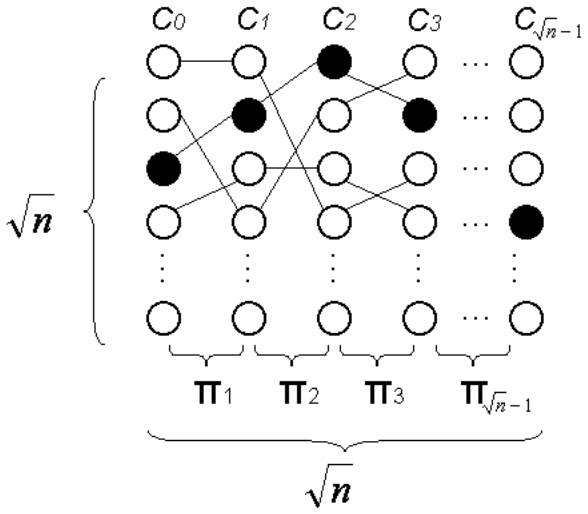


Figure 21.1: A $\sqrt{n}$ by $\sqrt{n}$ grid of vertices, with one of the disjoint paths darkened.

21.3.2 Block Operations

There are two operations that we define on the grid:

- *UPDATE*(i, π) – Replaces the i^{th} permutation π_i in the grid with permutation π. This is equivalent to $O(\sqrt{n})$ edge deletions and insertions.
- *VERIFY_SUM*(i, π) – Check if $\sum_{j=1}^{i}(\pi_j) = \pi$. In other words, check if the first i permutations $\pi_1, \pi_2, \ldots \pi i$ is equivelent to a single permutation π. This is equivalent to $O(\sqrt{n})$ connectivity queries.

We can now described the dynamic connectivity problem in terms of these operations and cell probe operations. Thus, we make the following claim:

Claim 47. *Performing $\sqrt{n}$ UPDATE(i, π) operations and $\sqrt{n}$ VERIFY_SUM(i, π) operations requires $\Omega(\sqrt{n}\sqrt{n}\lg n) = \Omega(n \lg n)$ cell probes.*

21.3.3 Construction of Bad Access Sequences

To prove the above claim, we need to use a family of random, "bad" sequences so as to ensure that we achieve the true lower bound. Otherwise, it would be possible for the data structure to tune to the input sequences, allowing it to take advantage of patterns in the sequences and run faster than it would in the worst case. We want to come up with the most difficult sequence of operations and sequences possible to ensure a worst case lower bound.

First, we will alternate the update and query operations, UPDATE(i, π) and VERIFY_SUM(i, π). Then, we will carefully select the arguments to these operations as follows:

- π *for UPDATE* – The permutation π for each UPDATE operation will be uniformly random, which will randomly change the result of each SUM operation (where SUM computes the sum of i permutations, which must also be performed by VERIFY_SUM).
- π *for VERIFY_SUM* – Using uniformly random permutations is not appropriate, because it is easy for the operation to check that a permutation *doesn't* match the sum; it only needs to find one path in the sum that doesn't satisfy π. The worst case permutation is actually the "correct" permutation, $\pi = \sum_{j=1}^{i}(\pi_j)$, because this forces the operation to check *every* path before returning TRUE.
- i – For both operations, the i's follow the *bit reversal sequence*. To generate this sequence, take all n-bit numbers n_i, write them in binary, then reverse (or "mirror") the bits in each number n_i to create a new number m_i. This results in a new sequence of numbers with some special properties. For example, the sequence maximizes the WILBER_1 function, as it chooses a path with the maximal number of non-preferred children at each step.

Example Using the argument selection rules above, we will alternate between update and query operations using worst case arguments, resulting in a "bad" access sequence. For example, with 3-bit cells, the bit reversal sequence is

$$(000_2, 100_2, 010_2, 011_2, 001_2, 101_2, 011_2, 111_2) = (0, 4, 2, 6, 1, 5, 3, 7)$$

The access sequence we would give would then be

$$QUERY(0, \pi_{correct}), UPDATE(0, \pi_{random}), QUERY(4, \pi_{correct}), UPDATE(4, \pi_{random}), \ldots$$

Notice that the sequences of queries defined above interleaves betwen adjacent blocks perfectly.

21.3.4 Tree of Time

To keep track of the order of the order of the sequence of acesses, we can create a binary tree in which each leaf represents one $QUERY/UPDATE$ pair of operations, and the leaves are in chronological order. We will refer to this as the tree of time. Note that for all h and m, the mth node at height m has all leaves with i such that i's last h digits are the bit reversal of m as descendents. As a result, the leaves in the left and right subtrees of any non-leaf node in this tree are interleaved. This implies that one can determine exactly what updates were performed in the left subtree of any node by making the right queries on values of i represented by leaves in the right subtree. The need to store and then retrieve this information is what makes this access sequence hard.

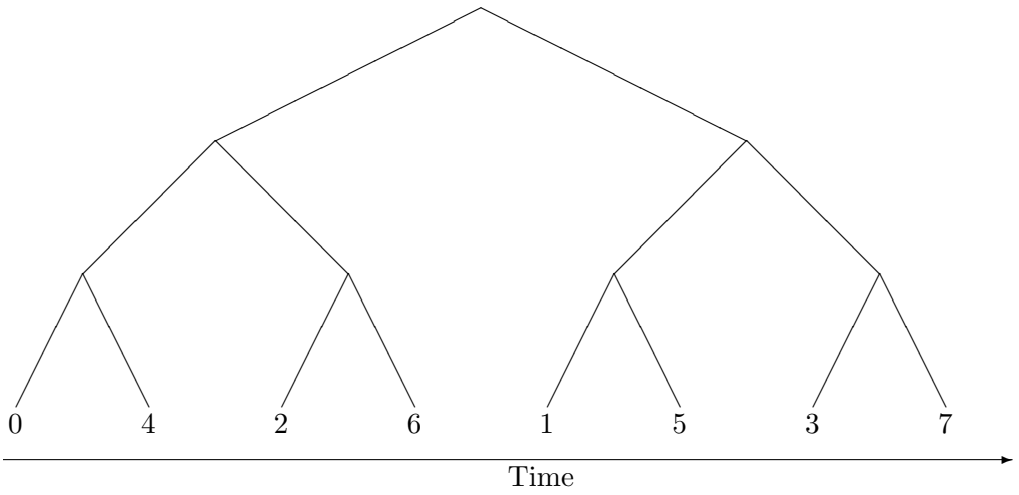


Figure 21.2: The tree of time for $w = 3$

Claim 48. *For any non-leaf node of the tree of time, v, let ℓ be the number of leaves in each of its subtrees. Then the series of operations represented in the right subtree of v must execute $\Omega(\ell\sqrt{n})$ probes of cells that were last modified during operations represented in the left subtree of v.*

For any given cell probe, there is at most one leaf that represents the step when the cell was last modified, and one leaf representing the step in which this probe occurs. So, the only node that includes this probe in its $\Omega(\ell\sqrt{n})$ probes is the least common ancestor of these two leaves.

There are $\sqrt{n}$ leaves total, so the sum of the values of ℓ corresponding to all of the nodes on a given level is $\sqrt{n}$. There are $\Omega(\log n)$ levels; therefore, this claim implies that the total number of cell probes used by this sequence of operations is $\Omega(\log n) \cdot \Omega(\sqrt{n} \cdot \sqrt{n}) = \Omega(n \log n)$. Thus, this claim implies claim 2.

21.3.5 Plan for proving claim 3

In order to prove Claim 3, we will show that for any node, v, in order to answer the queries represented in v's right subtree the program will need to use $\Omega(\ell\sqrt{n}\log n)$ bits of information that were set during operations represented in the left subtree of v.

Assuming that we knew the exact configuration of the paths prior to the operations represented in v's subtrees, it is possible to determine exactly what permutations were used by each $UPDATE$ operation represented in v's left subtree by executing the right series of $VERIFY\_SUM$ operations using i with leaves in v's right subtree. There are $\sqrt{n} = 2^{\theta(\sqrt{n}\log n)}$ possible permutations of $\sqrt{n}$ elements, so it takes $\Omega(\ell\sqrt{n}\log n)$ bits of information to track the set of modifications performed in the left subtree of v.

21.3.6 Simplified proof using SUM

Rather than doing the relevant proof immediately, we will start with a warmup. So, imagine that instead of using $VERIFY\_SU <$ as our queries, we are using

$$SUM(i) = \sum_{j=0}^{i} \pi_j$$

In other words, $SUM(i)$ returns the composition of the first i permutations.

Let R be the set of all cells accessed during the right subtree of v and W be the set of all cells written during the left subtree of v. Assuming that the size of each cell is $w = \theta(\log n)$, $R \cap W$ can be encoded in $O(|R \cap W|w)$ bits, and I claim that the full series of operations performed in the right subtree of v can be simulated using only a recording of the states of the cells immediately before the left subtree of v began, and an encoding of $R \cap S$.

Each time our simulation needs to know the information in a cell, there are three possible cases, based on where in the tree of time that cell was last modified.

1. v's right subtree: Its current value was set by another operation in the simulation, so we can easily determine it.
2. v's left subtree: This cell must be in $R \cap W$, so we can get its value from the encoding of $R \cap W$.
3. past subtrees: We can simply look its value up in the recording of the cells' states before v's subtree started executing.

In every case, we can give the simulation the data it needs to continue correctly, so it can give the correct outputs in response to all queries. Having the values of $SUM(i)$ for each leaf in v's right subtree provides enough information to determine exactly what permutations were performed in v's left subtree, so the output of the simulation must be different for each possible set of permutations that could be performed there. The only part of the simulation's input that depends on which permutations were performed in v's left subtree is $R \cap W$, so there must be at least as many possible values of $R \cap W$ as there are possible sets of permutations. Thus, $|R \cap W| = \Omega(\ell\sqrt{n}\log n)/\log n = \Omega(\ell\sqrt{n})$.

21.3.7 Proof using VERIFY_SUM

We have shown that Claim 3's bound of $\Omega(l\sqrt{n})$ cell probes is necessary to transfer the information from the UPDATE operations on the left subtree to the SUM operations on the right. Now we will try to prove a similar result using VERIFY_SUM.

Query: VERIFY_SUM(i, π): $\sum_{j=1}^{i}(\pi_j) \stackrel{?}{=} \pi$. Returns TRUE if the composition of the first i permutations is equivalent to permutation π, and returns FALSE otherwise.

Setup: Again, we assume we know the entire past. However, this time, we will not assume that we know updates in the left subtree or queries in the right subtree. We do know though, that for any i, only one π will result in TRUE. Furthermore, recall that we are using the worst case π inputs for VERIFY_SUM operations. Because the worst case π is the "correct" permutation, we know that VERIFY_SUM will always return true!

Encoding: As before, define R, W, and P. Instead of just having to encode the sums, however, we also have to encode the input π's. This is avoided by noting that the input π must always be the permutation that causes VERIFY_SUM to return true, as described in the setup. Instead of encoding π, we can just recover during decoding by trying all possible π_i permutations and selecting the one which matches the encoded P permutation.

Decoding: We start by simulating all possible input permutations to VERIFY_SUM so as to recover π, as described above. As before, the decoding algorithm relies on the knowledge of where the cell was last written. Unfortunately, this is no longer easy to discern. Because we simulated all possible input permutations, we queried cells in set R' as well as in R. Let R' be the set of cells read by these incorrect queries. If we read cell $r \in R' \setminus R$, then the permutation π must be incorrect. However, there is a chance that r will intersect with W in cells not in P, so that the state data needed to evaluate these read operations will get the value from the past, instead of getting data written during the left subtree. Since the algorithm is reading incorrect bits on its cell probes when the query has an answer that should be no, it might return yes instead, resulting in a false positive.

Result: We could get around this problem by encoding all of R or W, allowing us to check whether $r \in W \setminus R$ or $r \in \text{past} \setminus P$. However, this requires a large number of bits; W could be as large as $\ell\sqrt{n}\log n$ cells, or $\ell\sqrt{n}\log^2 n$ bits, prevent our encoding from fitting in the desired $\ell\sqrt{n}\log n$ space. Instead, we will try to find a way to cheaply encode whether we are in $R \setminus W$ or $W \setminus R$.

21.3.8 Separators

To solve the decoding problem described for VERIFY_SUM queries, we encode a *separator*.

Definition 49. *A* separator family *for size m sets is a family $S \subset 2^U$ with the property that for any $A, B \subset U$, with $(|A|, |B| \leq m)$ and $(A \cap B = \varnothing)$, there exists a $C \in S$, such that $(A \subset C)$ and $(B \subset U \setminus C)$.*

Theorem 50. *There exist separator families S such that $|S| \leq 2^{O(m+\log\log U)}$.*

Theorem 50 results from the fact that we can have a perfect hash family H with $|H| \leq 2^{O(m+\lg\lg|U|)}$, which maps $A \cup B$ to an $O(m)$-size table with no collisions. Each of these table entries stores two

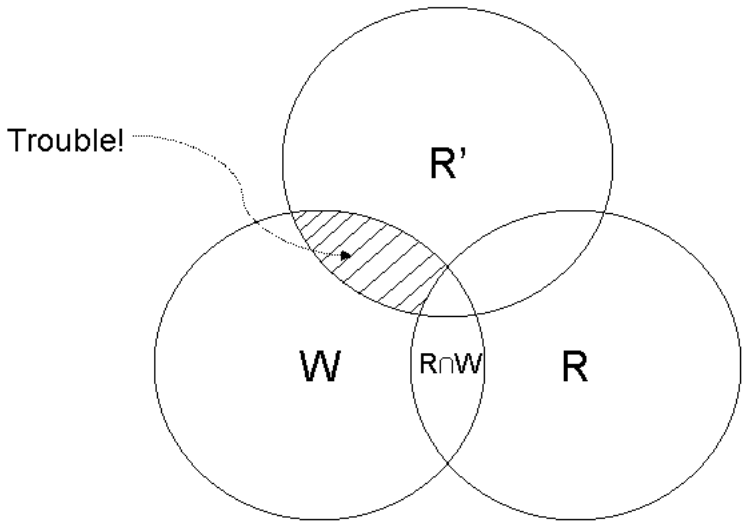


Figure 21.3: Venn diagram showing the intersections of R, W, and R'.

bits indicating if that element is in A or B. Storing the perfect hash function takes $\lg |H|$ bits, and the table entries take $2O(m)$ bits, so overall $\lg |H| + 2O(m) = O(m + \lg\lg |U|) + O(m) = O(m + \lg\lg |U|) = \lg |S|$.

Therefore, we can encode a separator S in $O(|R| + |W| + \lg\lg n)$ bits. The separator will have the property that $R \setminus W \subseteq S$ and $W \setminus R \subseteq \overline{S}$. By encoding the separator S along with P, we can successfully decode each write in the following manner.

Decoding (with Separator): To decode, we first simulate all input permutations to VERIFY_SUM so as to recover π. Then, we determine when the cell being read was last written:

- *Right subtree* – As before, we have knowledge of the permutations performed on the right subtree, so decoding here is trivial.
- P – We again use the information encoded in the simulated P set to determine what permutations were performed.
- S – If the cell is in S but not $R \setminus W$, then it must have last been written in the past, and we assume we know the effects of past permutations
- $\overline{S}$ – If the cell is not in S, then it must not be in R, meaning that this can't be the correct π. So we abort this decoding and look elsewhere for the appropriate π.

21.3.9 Conclusion

We have shown that we can decode the permutations caused by the left subtree UPDATE operations by using an encoded representation of P and a separator S of size $O(|R| + |W| + \lg\lg n)$, giving us

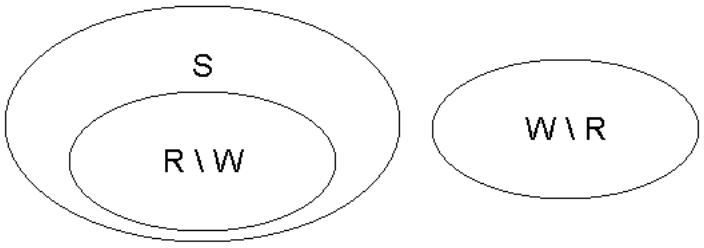


Figure 21.4: The separator S distinguishing between $R \setminus W$ and $W \setminus R$.

a lower bound of

$$|P|O(\lg n) + O(|R| + |W| + \lg \lg n) = \Omega(\ell\sqrt{n} \lg n).$$

There are two possibilities to consider.

- $|R| + |W| = \Omega(\ell\sqrt{n} \lg n)$ – If $|R| + |W|$ is large, then we have not proven anything, because our estimate is then trivially true. However, in this case, there were a total of $O(\ell)$ operations handled by each of the left and right subtrees, each of which does $\sqrt{n}$ dynamic connectivity operations. Then at least one of those dynamic connectivity operations used $\Omega(\lg n)$ time, and the main theorem holds.
- $|R| + |W| = o(\ell\sqrt{n} \lg n)$ – This implies that there were at least that many cell probes were forced on the right subtree by the left subtree, which is exactly what we were trying to show.

Bibliography

[1] Gerth Stlting Brodal: *Partially Persistent Data Structures of Bounded Degree with Constant Update Time*. Nord. J. Comput. 3(3): 238-255 (1996)

[2] Erik D. Demaine, John Iacono, Stefan Langerman: *Retroactive data structures*. SODA 2004: 281-290

[3] Erik D. Demaine, Stefan Langerman, and Eric Price: *Confluently Persistent Tries for Efficient Version Control*. Algorithmica (2008).

[4] Paul F. Dietz, Daniel Dominic Sleator: *Two Algorithms for Maintaining Order in a List* STOC 1987: 365-372

[5] Paul F. Dietz: *Fully Persistent Arrays (Extended Array)*. WADS 1989: 67-74

[6] James R. Driscoll, Neil Sarnak, Daniel Dominic Sleator, Robert Endre Tarjan: *Making Data Structures Persistent*. J. Comput. Syst. Sci. 38(1): 86-124 (1989)

[7] Yoav Giora, Haim Kaplan: *Optimal dynamic vertical ray shooting in rectilinear planar subdivisions* ACM Transactions on Algorithms 5(3) (2009)

[8] Haim Kaplan, Chris Okasaki, Robert Endre Tarjan: *Simple Confluently Persistent Catenable Lists*. SIAM J. Comput. 30(3): 965-977 (2000)

[9] Amos Fiat, Haim Kaplan: *Making data structures confluently persistent*. J. Algorithms 48(1): 16-58 (2003)

[10] Chris Okasaki: *Purely Functional Data Structures*. New York: Cambridge University Press, 2003.

[11] Sebastien Collette, John Iacono, and Stefan Langerman. *Confluent Persistence Revisited.* In Symposium on Discrete Algorithms (SODA), pages 593-601, 2012.

[12] T. H. Cormen and C. E. Leiserson and R. L. Rivest and C. Stein, *Introduction to Algorithms*. 3rd. Edition. The MIT Press, 2009.

[13] Nicholas Pippenger. *Pure Versus Impure Lisp*. ACM Transactions on Programming Languages and Systems, Vol. 19, No. 2. (March 1997), pp. 223-238.

[14] Gerth Stlting Brodal, Christos Makris, Kostas Tsichlas: *Purely Functional Worst Case Constant Time Catenable Sorted Lists*. ESA 2006: 172-183

[15] Erik D. Demaine, John Iacono, Stefan Langerman: *Retroactive data structures*. ACM Transactions on Algorithms 3(2): (2007)

[16] Yoav Giora, Haim Kaplan: *Optimal dynamic vertical ray shooting in rectilinear planar subdivisions* ACM Transactions on Algorithms 5(3) (2009)

[17] Umut A. Acar, Guy E. Blelloch, Kanat Tangwongsan: *Kinetic 3D convex hulls via self-adjusting computation*. Symposium on Computational Geometry 2007: 129-130

[18] Gudmund Skovbjerg Frandsen, Gudmund Skovbjerg Frandsen, Peter Bro Miltersen: *Lower Bounds for Dynamic Algebraic Problems*. Information and Computation 171(2): 333-349 (2001)

[19] Jon Louis Bentley, James B. Saxe: *Decomposable Searching Problems I: Static-to-Dynamic Transformation*. J. Algorithms 1(4): 301-358 (1980)

[20] S. Alstrup, G. Stolting Brodal, T. Rauhe, *New data structures for orthogonal range searching*, Foundations of Computer Science, 2000. Proceedings, 41st annual symposium, 198-207.

[21] J.L. Bentley, *Multidimensional Binary Search Trees in Database Applications*, IEEE Transactions on Software Engineering, 4:333-340, 1979.

[22] M. de Berg, O. Cheong, M. van Kreveld, M. Overmars, *Computational Geometry: Algorithms and Applications*, Springer, 3rd edition, 2008.

[23] G. Blelloch, *Space-efficient dynamic orthogonal point location, segment intersection, and range reporting*, SODA 2008:894-903.

[24] B. Chazelle, *Reportin and counting segment intersections*, Journal of Computer and System Sciences 32(2):156-182, 1986.

[25] B. Chazelle, L. Guibas, *Fractional Cascading: I. A Data Structuring Technique*, Algorithmica, 1(2):133-162, 1986.

[26] B. Chazelle, L. Guibas, *Fractional Cascading: II. Applications*, Algorithmica, 1(2):163-191, 1986.

[27] D. Dobkin, R. Lipton, *Multidimensional searching problems*, SIAM Journal on Computing, 5(2):181-186, 1976.

[28] Gabow H., Bentley J., Tarjan R., *Scaling and related techniques for geometry problems*, Symposium on Theory of Computing, 1984. Proceedings, 16th annual

[29] Y. Giyora, H. Kaplan, *Optimal dynamic vertical ray shooting in rectilinear planar subdivisions*, ACM Transactions on Algorithms, 5(3), 2009.

[30] D.T. Lee, C.K. Wong, *Quintary trees: a file structure for multidimensional database systems*, ACM Transactions on Database Systems, 5(3), 1980.

[31] G. Lueker, *A data structure for orthogonal range queries*, Foundations of Computer Science, 1978. Proceedings, 19th annual symposium, 28-34.

[32] K. Mehlhorn, S. Näher, *Dynamic Fractional Cascading*, Algorithmica, 5(2): 215-241, 1990.

[33] J. Nievergelt, E. M. Reingold, *Binary search trees of bounded balance*, Symposium on Theory of Computing, 1972. Proceedings, 4^{th} annual symposium.

[34] D.E. Willard, *New Data Structures for Orthogonal Range Queries*, SIAM Journal on Computing, 14(1):232-253. 1985.

[35] D.E. Willard, *New Data Structures for Orthogonal Queries*, Techincal Report, January 1979.

[36] D.E. Willard, Ph.D. dissertation, Harvard University, 1978.

[ABdB+99] Pankaj K. Agarwal, Julien Basch, Mark de Berg, Leonidas J. Guibas, and John Hershberger. Lower bounds for kinetic planar subdivisions. In *SCG '99: Proceedings of the fifteenth annual symposium on Computational geometry*, pages 247–254, New York, NY, USA, 1999. ACM.

[ABG+02] Pankaj K. Agarwal, Julien Basch, Leonidas J. Guibas, John Hershberger, and Li Zhang. Deformable free-space tilings for kinetic collision detection. *I. J. Robotic Res.*, 21(3):179–198, 2002.

[AEGH98] Pankaj K. Agarwal, David Eppstein, Leonidas J. Guibas, and Monika Rauch Henzinger. Parametric and kinetic minimum spanning trees. In *FOCS*, pages 596–605, 1998.

[AGMR98] Gerhard Albers, Leonidas J. Guibas, Joseph S. B. Mitchell, and Thomas Roos. Voronoi diagrams of moving points. *Int. J. Comput. Geometry Appl.*, 8(3):365–380, 1998.

[AHP01] Pankaj K. Agarwal and Sariel Hal-Peled. Maintaining approximate extent measures of moving points. In *SODA '01: Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 148–157, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

[BGH99] Julien Basch, Leonidas J. Guibas, and John Hershberger. Data structures for mobile data. *J. Algorithms*, 31(1):1–28, 1999.

[C86] Bernard Chazelle. Filtering Search: A new approach to query-answering. In *Siam J. Comput., Volume 15*, 1986.

[CG86] Bernard Chazelle, Leonidas J. Guibas, Fractional cascading: I. A data structuring technique. In *Algorithmica, Volume 1*, 1986.

[FF03] Guilherme D. da Fonseca, Celina M.H. de Figueiredo, Kinetic heap-ordered trees: Tight analysis and improved algorithms. In *Information Processing Letters, Volume 85, Issue 3*, 14 February 2003.

[Gui04] Leonidas J. Guibas. Kinetic data structures. 2004.

[GXZ01] Leonidas J. Guibas, Feng Xie, and Li Zhang. Kinetic collision detection: Algorithms and experiments. In *ICRA*, pages 2903–2910, 2001.

[KSS00] David Kirkpatrick, Jack Snoeyink, and Bettina Speckmann. Kinetic collision detection for simple polygons. In *SCG '00: Proceedings of the sixteenth annual symposium on Computational geometry*, pages 322–330, New York, NY, USA, 2000. ACM.

[37] Richard Cole, Bud Mishra, Jeanette P. Schmidt, Alan Siegel: *On the Dynamic Finger Conjecture for Splay Trees. Part I: Splay Sorting log n-Block Sequences.* SIAM J. Comput. 30(1): 1-43 (2000)

[38] Richard Cole: *On the Dynamic Finger Conjecture for Splay Trees. Part II: The Proof.* SIAM J. Comput. 30(1): 44-85 (2000)

[39] Erik D. Demaine, Dion Harmon, John Iacono, Daniel M. Kane, Mihai Patrascu: *The geometry of binary search trees.* SODA 2009: 496-505

[40] John Iacono: *Alternatives to splay trees with O(log n) worst-case access times.* SODA 2001: 516-522

[41] Daniel Dominic Sleator, Robert Endre Tarjan: *Self-Adjusting Binary Search Trees* J. ACM 32(3): 652-686 (1985)

[42] Robert Wilber, *Lower bounds for accessing binary search trees with rotations*, SIAM Journal on Computing, 18(1):56-67, 1989

[43] John Iacono, *Key independent optimality*, Algorithmica, 42(1):3-10, 2005 *Key Independent Optimality*

[DHIP04] Erik D. Demaine, Dion Harmon, John Iacono, and Mihai Patrascu. Dynamic optimality — almost. In *FOCS '04: Proceedings of the 45th Annual IEEE Symposium on Foundations of Computer Science (FOCS'04)*, pages 484–490. IEEE Computer Society, 2004.

[44] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[45] L. Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, June 2003.

[46] L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proc. STOC '02*, pages 268–276, May 2002.

[47] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *Proc. FOCS '00*, pages 399–409, Nov. 2000.

[48] M. A. Bender, Z. Duan, J. Iacono, and J. Wu. A locality-preserving cache-oblivious dynamic dictionary. In *Proc. SODA '02*, pages 29–38, 2002.

[49] G. S. Brodal and R. Fagerberg. Funnel heap — a cache oblivious priority queue. In *Proc. ISAAC '02*, pages 219–228, 2002.

[50] G. S. Brodal and R. Fagerberg. Cache oblivious distribution sweeping. In *Proc. ICALP '03*, page 426, 2003.

[51] G. S. Brodal and R. Fagerberg. On the limits of cache-obliviousness. In *Proc. STOC '03*, pages 307–315, 2003.

[52] G. S. Brodal, R. Fagerberg, and R. Jacob. Cache oblivious search trees via binary trees of small height. In *Proc. SODA '02*, pages 39–48, 2002.

[53] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. FOCS '99*, pages 285–298, 1999.

[54] H. Prokop. Cache-oblivious algorithms. Master's thesis, Massachusetts Institute of Technology, June 1999.

[55] Alon Itai, Alan G. Konheim, and Michael Rodeh. A Sparse Table Implementation of Priority Queues. *International Colloquium on Automata, Languages, and Programming (ICALP)*, p417-432, 1981.

[56] M. A. Bender, R. Cole, E. Demaine, M. Farach-Colton, and J. Zito. Two Simplified Algorithms for Maintaining Order in a List. *Proceedings of the 10th European Symposium on Algorithms (ESA)*, p152-164, 2002.

[57] D.E. Willard. A density control algorithm for doing insertions and deletions in a sequentially ordered file in good worst-case time. In *Information and Computation*, 97(2), p150-204, April 1992.

[58] P. Dietz, and D. Sleator. Two algorithms for maintaining order in a list. In *Annual ACM Symposium on Theory of Computing (STOC)*, p365-372, 1987.

[59] L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proc. STOC '02*, pages 268–276, May 2002.

[60] Gerth Stølting Brodal and Rolf Fagerberg. Cache oblivious distribution sweeping. *In Proceedings of the 29th International Colloquium on Au- tomata, Languages, and Programming*, volume 2380 of Lecture Notes in Computer Science, pages 426-438, Malaga, Spain, July 2002.

[61] Lars Arge and Norbert Zeh. 2006. *Simple and semi-dynamic structures for cache-oblivious planar orthogonal range searching.* In Proceedings of the twenty-second annual symposium on Computational geometry (SCG '06). ACM, New York, NY, USA, 158-166.

[62] Lars Arge, Gerth Stlting Brodal, Rolf Fagerberg, and Morten Laustsen. 2005. *Cache-oblivious planar orthogonal range searching and counting.* In Proceedings of the twenty-first annual symposium on Computational geometry (SCG '05). ACM, New York, NY, USA, 160-169.

[63] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* 2009.

[64] P. van Emde Boas, *Preserving Order in a Forest in less than Logarithmic Time*, FOCS, 75-84, 1975.

[65] Dan E. Willard, *Log-Logarithmic Worst-Case Range Queries are Possible in Space* $\Theta(n)$, Inf. Process. Lett. 17(2): 81-84 (1983)

[Ajt88] M. Ajtai: *A lower bound for finding predecessors in Yao's cell probe model*, Combinatorica 8(3): 235–247, 1988.

[BF99] P. Beame, F. Fich: *Optimal Bounds for the Predecessor Problem*, Symposium on the Theory of Computing 1999: 295–304.

[BF02] P. Beame, F. Fich: *Optimal Bounds for the predecessor problem and related problems*, Journal of Computer and System Sciences 65(1): 38–72, 2002.

[Mil94] P. Miltersen: *Lower bounds for union-split-find related problems on random access machines*, Symposium on the Theory of Computing 1994: 625–634.

[MNSW95] P. Miltersen, N. Nisan, S. Safra, A. Wigderson: *On data structures and asymmetric communication complexity*, Symposium on the Theory of Computing 1995: 103–111.

[MNSW98] P. Miltersen, N. Nisan, S. Safra, A. Wigderson: *On data structures and asymmetric communication complexity*, Journal of Computer and System Sciences, 57(1): 37–49, 1998.

[PT06] M. Patrascu, M. Thorup: *Time-space trade-offs for predecessor research*, Symposium on the Theory of Computing 2006: 232–240.

[PT07] M. Patrascu, M. Thorup: *Randomization does not help searching predecessors*, Symposium on Discrete Algorithms 2007: 555–564.

[Sen03] P. Sen: *Lower bounds for predecessor searching in the cell probe model*, IEEE Conference on Computational Complexity 2003: 73–83.

[SV08] P. Sen, S. Venkatesh: *Lower bounds for predecessor searching in the cell probe model*, Journal of Computer and System Sciences 74(3): 364–385, 2008.

[Xia92] B. Xiao: *New bounds in cell probe model*, PhD thesis, University of California, San Diego, 1992.

[66] Susanne Albers, Torben Hagerup: *Improved Parallel Integer Sorting without Concurrent Writing*, Inf. Comput. 136(1): 25-51, 1997.

[67] A. Andersson, T. Hagerup, S. Nilsson, R. Raman, *Sorting in Linear Time?*, J. Comput. Syst. Sci. 57(1): 74-93, 1998.

[68] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein: *Introduction to Algorithms*, Second Edition, The MIT Press and McGraw-Hill Book Company 2001.

[69] Y. Han: *Deterministic Sorting in $O(n \log\log n)$ Time and Linear Space*, J. Algorithms 50(1): 96-105, 2004.

[70] Y. Han, M. Thorup: *Integer Sorting in $O(n\sqrt{\log\log n})$ Expected Time and Linear Space*, FOCS 2002: 135-144.

[71] D.G. Kirkpatrick, S. Reisch: *Upper Bounds for Sorting Integers on Random Access Machines*, Theoretical Computer Science 28: 263-276 (1984).

[72] M. Thorup: *Equivalence between Priority Queues and Sorting*. FOCS 2002: 125-134 (2002).

[73] H. Gabow, J. Bentley, R. Tarjan. Scaling and Related Techniques for Geometry Problems. In *STOC '84: Proc. 16th ACM Symp. Theory of Computing*, pages 135-143, 1984.

[74] Dov Harel, Robert Endre Tarjan, *Fast Algorithms for Finding Nearest Common Ancestors*, SIAM J. Comput. 13(2): 338-355 (1984)

[75] Michael A. Bender, Martin Farach-Colton, *The LCA Problem Revisited*, LATIN 2000: 88-94

[76] M. Bender, M. Farach-Colton. The Level Ancestor Problem simplified. Lecture Notes in Computer Science. 321: 5-12. 2004.

[77] P. Dietz. Finding level-ancestors in dynamic trees. Algorithms and Data Structures, 2nd Workshop WADS '91 Proceedings. Lecture Notes in Computer Science 519: 32-40. 1991.

[78] R. Boyer and J. Moore. *A fast string searching algorithm.* Communications of the ACM, 20(10):762772, 1977.

[79] M. Farach. *Optimal suffix tree construction with large alphabets.* In Proc. 38th Annual Symposium on Foundations of Computer Science, pages 137-143. IEEE, 1997.

[80] Juha Karkkainen, Peter Sanders, *Simple linear work suffix array construction*, In Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP 03). LNCS 2719, Springer, 2003, pp. 943-955

[81] R. M. Karp and M. O. Rabin. *Efficient randomized pattern-matching algorithms.* IBM Journal of Research and Development, 31:249260, March 1987.

[82] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. *Linear-time longest-common-prefix computation in suffix arrays and its applications.* In Proc.12th Symposium on Combinatorial Pattern Matching (CPM 01), pages 181192. Springer-Verlag LNCS n. 2089, 2001.

[83] D. E. Knuth, J. H. Morris, and V. R. Pratt. *Fast pattern matching in strings.* SIAM Journal of Computing, 6(2):323350, 1977.

[84] E. M. McCreight. *A space-economic suffix tree construction algorithm.* J. ACM,23(2):262-272, 1976.

[85] S. Muthukrishnan. *Efficient algorithms for document retrieval problems.* SODA 2002:657-666

[86] Richard Cole, Tsvi Kopelowitz, Moshe Lewenstein. *Suffix Trays and Suffix Trists: Structures for Faster Text Indexing.* ICALP 2006: 358-369

[87] E. Ukkonen. *On-line construction of suffix trees.* Algorithmica, 14(3):249-260, 1995.

[88] P. Weiner. *Linear pattern matching algorithm.* In Proc. 14th Symposium on Switching and Automata Theory, pages 1-11. IEEE, 1973.

[89] G. Franseschini, R.Grossi *Optimal Worst-case Operations for Implicit Cache-Oblivious Search Trees*, Prooceding of the 8th International Workshop on Algorithms and Data Structures (WADS), 114-126, 2003

[90] A.Brodnik, I.Munro *Membership in Constant Time and Almost Minimum Space*, Siam J. Computing, 28(5): 1627-1640, 1999

[91] D.Benoit, E.Demaine, I.Munro, R.Raman, V.Raman, S.Rao *Representing Trees of Higher Degree*, Algorithmica 43(4): 275-292, 2005

[92] D.Clark, I.Munro *Eifficent Suffix Trees on Secondary Storage*, SODA, 383-391, 1996.

[93] G.Jacobson *Succinct Static Data Structures*, PHD.Thesis, Carnegie Mellon University, 1989.

[94] R. Pagh: *Low Redundancy in Static Dictionaries with Constant Query Time*, SIAM Journal of Computing 31(2): 353-363 (2001).

[95] J. Ian Munro, Rajeev Raman, Venkatesh Raman, and Satti Srinivasa Rao: *Succinct Representations of Permutations*, ICALP (2003), LNCS 2719, pp. 345-356.

[96] P. Ferragina and G. Manzini, *Indexing Compressed Text*, Journal of the ACM, Vol. 52 (2005), 552-581.

[97] R. Grossi, A. Gupta, J. S. Vitter, *High-order entropy-compressed text indexes*, SODA 2003: 841-850.

[98] R. Grossi and J. S. Vitter, *Compressed suffix arrays and suffix trees with applications to text indexing and string matching*, Thirty-Second Annual ACM Symposium on Theory of Computing, pp. 397-406 (2000).

[99] J. I. Munro, V. Raman, and S. S. Rao, *Space Efficient Suffix Trees*, Journal of Algorithms, 39(2):205-222.

[100] K. Sadakane, *New text indexing functionalities of the compressed suffix arrays*, Journal of Algorithms, 48(2): 294-313 (2003).

[101] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro, *Compressed Representations of Sequences and Full-Text Indexes*, ACM Transactions on Algorithms, 3(2) article 20 (2007).

[102] W-K. Hon, T-W. Lam, K. Sadakane, W-K. Sung, and S-M. Yiu *A Space and Time Efficient Algorithm for Constructing Compressed Suffix Arrays*, Algorithmica, 48(1) : 23-36 (2007).

[103] W-K. Hon, K. Sadakane, and W-K. Sung *Breaking a time-and-space barrier in constructing full-text indices*, 44th Annual IEEE Symposium on Foundations of Computer Science, pp. 251-260 (2003).

[104] W-K. Hon and K. Sadakane *Space-Economical Algorithms for Finding Maximal Unique Matches*, Combinatorial Pattern Matching, Lecture Notes in Computer Science, 2373 : 17-29 (2002).

[105] N. J. Larsson, and K. Sadakane *Faster suffix sorting*, Theoretical Computer Science, 387(3) : 258-272 (2007).

[106] K. Sadakane *Succinct data structures for flexible text retrieval systems*, Journal of Discrete Algorithms, 5(1) : 12-22 (2007).

[107] H-L. Chan, W-K. Hon, T-W. Lam, and K. Sadakane *Compressed indexes for dynamic text collections*, ACM Transactions on Algorithms, 3(2) article 21 (2007).

[108] D. D. Sleator, R. E. Tarjan, *A Data Structure for Dynamic Trees*, Journal. Comput. Syst. Sci., 28(3):362-391, 1983.

[109] R. E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, 1984.

[110] Monika Rauch Henzinger, Valerie King: Randomized dynamic graph algorithms with polylogarithmic time per operation. STOC 1995: 519-527

[111] David Eppstein, Giuseppe F. Italiano, Roberto Tamassia, Robert Endre Tarjan, Jeffery Westbrook, Moti Yung: Maintenance of a Minimum Spanning Forest in a Dynamic Plane Graph. J. Algorithms 13(1): 33-54 (1992)

[112] Mikkel Thorup: Near-optimal fully-dynamic graph connectivity. STOC 2000: 343-350

[113] Jacob Holm, Kristian de Lichtenberg, Mikkel Thorup: Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. J. ACM 48(4): 723-760 (2001)

[114] Mihai Patrascu, Erik D. Demaine: Lower bounds for dynamic connectivity. STOC 2005: 546-553

[115] David Eppstein, Zvi Galil, Giuseppe F. Italiano, Amnon Nissenzweig: Sparsification - a technique for speeding up dynamic graph algorithms. J. ACM 44(5): 669-696 (1997)

[116] Mikkel Thorup: Decremental Dynamic Connectivity. J. Algorithms 33(2): 229-243 (1999)

[117] Mikkel Thorup: Fully-dynamic min-cut. STOC 2001: 224-230

[118] Dora Giammarresi, Giuseppe F. Italiano: Decremental 2- and 3-Connectivity on Planar Graphs. Algorithmica 16(3): 263-287 (1996)

[119] Zvi Galil, Giuseppe F. Italiano, Neil Sarnak: Fully Dynamic Planarity Testing with Applications. J. ACM 46(1): 28-91 (1999)

[120] Giuseppe F. Italiano, Johannes A. La Poutré, Monika Rauch: Fully Dynamic Planarity Testing in Planar Embedded Graphs (Extended Abstract). ESA 1993: 212-223

[121] Johannes A. La Poutré: Alpha-algorithms for incremental planarity testing (preliminary version). STOC 1994: 706-715

[122] Neil Robertson, Paul D. Seymour: Graph Minors. XX. Wagner's conjecture. J. Comb. Theory, Ser. B 92(2): 325-357 (2004)

[123] Robert Tarjan: Efficiency of a Good But Not Linear Set Union Algorithm. J. ACM 22(2): 215-225 (1975)

[124] Mikkel Thorup: Decremental Dynamic Connectivity, J. Algorithms 33(2): 229-243 (1999)

[125] Liam Roditty, Uri Zwick: Dynamic Approximate All-Pairs Shortest Paths in Undirected Graphs. FOCS 2004: 499-508

[126] Liam Roditty, Uri Zwick: On Dynamic Shortest Paths Problems. ESA 2004: 580-591

[127] Mikkel Thorup: Worst-case update times for fully-dynamic all-pairs shortest paths. STOC 2005:112-119

[128] Camil Demetrescu, Giuseppe F. Italiano: A new approach to dynamic all pairs shortest paths. STOC 2003:159-166

[129] Mikkel Thorup: Fully-Dynamic All-Pairs Shortest Paths: Faster and Allowing Negative Cycles. SWAT 2004:384-396

[130] Camil Demetrescu, Giuseppe F. Italiano: Fully Dynamic Transitive Closure: Breaking Through the O(n2) Barrier. FOCS 2000:381-389

[131] Liam Roditty, Uri Zwick: A fully dynamic reachability algorithm for directed graphs with an almost linear update time. STOC 2004:184-191

[132] Liam Roditty, Uri Zwick: Improved Dynamic Reachability Algorithms for Directed Graphs. FOCS 2002:679

[133] Piotr Sankowski: Dynamic Transitive Closure via Dynamic Matrix Inverse (Extended Abstract). FOCS 2004:509-517

[134] Liam Roditty: A faster and simpler fully dynamic transitive closure. SODA 2003:404-412

[135] M. L. Fredman and M. R. Henzinger. *Lower bounds for fully dynamic connectivity problems in graphs*. Algorithmica, 22(3):351362, 1998.

[136] P. B. Miltersen, S. Subramanian, J. S. Vitter, and R. Tamassia. *Complexity models for incremental computation*. Theoretical Computer Science, 130(1):203236, 1994.

[137] M. R. Henzinger, V. King, *Randomized dynamic graph algorithms with polylogarithmic time per operation*. STOC 1995: 519-527

[138] J. Holm, K. Lichtenberg, M. Thorup, *Poly-logarithmic deterministic fullydynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity*. J. ACM 2001: 48(4): 723-760

[139] M. Fredman, M. Saks, *The cell probe complexity of dynamic data structures*. STOC 1989: 345-354

[140] M. Pătraşcu, E. D. Demaine, *Lower bounds for dynamic connectivity*. STOC 2004: 546-553

[141] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. 2009.

[142] Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms*, 25(1):19-51, 1997.

[143] Mark N.Wegman and Larry Carter. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences*, 22(3):265-279, 1981.

[144] Mikkel Thorup and Yin Zhang. Tabulation based 4-universal hashing with applications to second moment estimation. *SODA*, pages 615-624, 2004.

[145] Alan Siegel. On universal classes of extremely random constant-time high functions. *SIAM-COMP*, 33(3):505-543, 2004.

[146] Don Knuth. Notes on "open" addressing, 1963.

[147] Jeanette P. Schmidt and Alan Siegel. The Spatial Complexity of Oblivious k-Probe Hash Functions. *SIAM Journal on Computing*, 19(5):775-786, 1990.

[148] Anna Pagh, Rasmus Pagh, and Milan Ružić. Linear probing with constant independence. In *In STOC '07: Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 318-327. ACM Press, 2007.

[149] Mihai Pătraşcu and Mikkel Thorup. On the k-independence required by linear probing and minwise independence. In *Proc. 37th International Colloquium on Automata Languages and Programming (ICALP)*, pages 715-726, 2010.

[150] Mihai Pătraşcu and Mikkel Thorup. The power of simple tabulation hashing. In *Proc. 43rd ACM Symposium on Theory of Computing (STOC)*, pages 1-10, 2011. See also arXiv:1011.5200.

[151] Jeanette P. Schmidt, Alan Siegel, and Aravind Srinivasan. Chernoff-Hoeffding bounds for applications with limited independence. *SIAM J. Discrete Math.*, 8(2):223-250,1995.

[152] Mihai Pătraşcu. Blog, 2011.

[153] Martin Dietzfelbinger, Anna R. Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert Endre Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, 23(4):738-761, 1994.

[154] Martin Dietzfelbinger and Friedhelm Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *ICALP*, pages 6-19, 1990.

[155] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122-144, 2004.

[156] Jeffrey S. Cohen and Daniel M. Kane. Bounds on the independence required for cuckoo hashing. *ACM Transactions on Algorithms*, 2009.

[157] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0(1) worst case access time. *J. ACM*, 31(3):538-544, 1984.

[158] M. L. Fredman and D. E. Willard. BLASTING through the information theoretic barrier with FUSION TREES. *Proceedings of the twenty-second annual ACM symposium on Theory of Computing*, 1-7, 1990.

[159] M. L. Fredman and D. E. Willard. Surpassing the information theoretic barrier with fusion trees. *Journal of Computer and System Sciences*, 47:424-436, 1993.

[160] A. Andersson, P. B. Miltersen, and M. Thorup. Fusion trees can be implemented with AC^0 instructions only. *Theoretical Computer Science*, 215:337-344, 1999.

[161] A. Andersson and M. Thorup. Dynamic ordered sets with exponential search trees. *Journal of the ACM*, 54:3:13, 2007.

[162] R. Raman. Priority queues: Small, monotone, and trans-dichotomous. *Algorithms - ESA '96*, 121-137, 1996.