# Contents

Introduction 3

1 Lecture 1: Fibonacci Heaps 4
   1.1 MST problem review .............................................. 4
   1.2 Using d-heaps for MST ........................................... 5
   1.3 Introduction to Fibonacci Heaps ................................. 5

2 Lecture 2: Fib Heaps Cont, Persistent Data Structs 9
   2.1 Fibonacci Heaps Continued ....................................... 9
   2.2 Applications of Fibonacci Heaps ................................. 11
       2.2.1 Further Improvements ....................................... 12
   2.3 Introduction to Persistent Data Structures .................... 12

3 Lecture 3: Persistent Data Structs, Splay Trees 13
   3.1 Persistent Data Structures Cont. ................................. 13
   3.2 The Planar Point Location Problem ............................... 14
   3.3 Introduction to Splay Trees ..................................... 16

4 Lecture 4: Splay Trees 17
   4.1 Heuristics of Splay Trees ........................................ 17
   4.2 Solving the Balancing Problem ................................ 17
   4.3 Implementation and Analysis of Operations ..................... 18
   4.4 Results with the Access Lemma ................................. 20

5 Lecture 5: Splay Updates, Buckets 22
   5.1 Wrapping up Splay Trees ......................................... 22
   5.2 Buckets and Indirect Addressing ................................. 23
       5.2.1 Improvements ................................................... 24
       5.2.2 Formalization with Tries ................................... 24
       5.2.3 Laziness wins again ........................................... 25
6 Lecture 6: VEB queues, Hashing

6.1 Improvements to the Denardo and Fox queue .............................. 26
6.2 VEB queues ............................................................................. 26
6.3 Hashing .................................................................................... 28
Introduction

These notes were typeset live from the online 18.415 lectures, taught by David Karger.

**Remark 1.** Though the course was originally designed as ‘Advanced Algorithms,’ it is probably better to refer to it as ‘Classical Algorithms.’

The course doesn’t go into any cutting-edge algorithms; rather, it provides us a toolkit for algorithms that we are expected to know. We’ll mostly be looking at combinatorical problems, and introducing various models to be able to understand them as well as their efficiency.

Most course information can be found at the following link: courses.csail.mit.edu/6.854. The core of the course are long and challenging PSETs, and is complemented by a peer grading assignment and a final project. Collaboration is encouraged and essential, though they should be small groups of size 3. Academic integrity is critical to the course, though in recent years people have always been caught.

Due to current conditions, there are many experimental tactics in regards to lectures. Lectures will be recorded, though live lectures will allow for questions. Playing previous lectures may also happen, in where we can stop and ask questions. Self-study / giant office hours during lecture may also be possible. Problem set structure may also be changed, due to lack of facility of collaboration. Collaborators will be changed throughout the semester. There will be sufficient ‘slack points’ for late PSET submissions, but extensions will be granted as well if necessary.
1 Lecture 1: Fibonacci Heaps

1.1 MST problem review

Fibonacci Heaps were developed by Fredman Tarjon in 1985, and were done so specifically to solve the Shortest Path/Minimum Spanning Tree problems. Previously (in 18.410) two algorithms for solving MST problems were introduced: Prim’s algorithm and Kruskal’s algorithm. Both of these algorithms are greedy algorithms, with Prim’s relying on building up the tree one by one, adding the closest neighbor to the tree. On the other hand, Kruskal’s builds up the tree through connecting different components, adding in the shortest useful edge that connects two components.

There is actually another, ‘simpler’ algorithm that is taught much less often to solve the MST problem: Boruvka’s algorithm, where we contract a minimum edge off of each vertex. It is somewhat similar to Kruskal’s algorithm. It starts off by putting every vertex in its own component. At every iteration, the minimum weight edge out of every component, that connects to a different component, is added to the MST. Since each iteration reduces the number of components by a factor of $\geq 2$, this algorithm takes $O(E \log V)$ time.

When we implement Prim’s algorithm, we want to use a priority queue, where we are able to insert items, find/delete the minimum element, and decreasing the key (updating edge vertices). Prim’s algorithm is actually isomorphic to Dijkstra’s algorithm; the only difference is the cost function. In Dijkstra’s algorithm, the key is the distance to the source, while it is the distance to the root in Prim’s.

What is the cost of Prim’s algorithm, in terms of $n$ vertices and $m$ edges? Let’s make a quick table for everything when we use a heap as a priority queue:

<table>
<thead>
<tr>
<th>operation</th>
<th>number of operations</th>
<th>runtime per operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert items</td>
<td>$n$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>find/delete minimum</td>
<td>$n$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>decrease key</td>
<td>$m$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

The total runtime is $O((m + n) \log n)$. 
Heaps should have been covered in standard undergraduate algorithms courses. As a review, a heap is a tree that satisfies the heap property, which states that if \( P \) is a parent node of \( C \), then the key of \( P \) satisfies some relation compared to the key of \( C \). In a max heap, that relation is \( \geq \); in a min heap, that relation is \( \leq \). A standard heap uses \( O(\log n) \) amortized time for most operations.

### 1.2 Using d-heaps for MST

Since the decrease key operation is the most expensive operation of our algorithm, and so any modification to decrease this runtime would help us. This brings us to our key principle of balancing, which is necessary for our heap structure. So, instead of using a standard heap, we would instead use a d-heap, which is a heap with \( d \) children per node, with height \( \log_d(n) \). This brings the insert/decrease runtime down to \( O(\log_d n) \), while the delete min operation requires time \( O(d \log_d n) \), ultimately changing the runtime to \( O(m \log_d n + nd \log_d n) \).

**Question 3.** How do we minimize runtime?

**Claim 4**

If we set \( m \log_d n = nd \log_d n \), then we will get a factor of 2 within the minimum.

**Proof.** Exercise. 

Either way, setting them equal \( (m = nd) \), the runtime becomes \( O(m \log_{m/n} n) \). This doesn’t seem like a big improvement, but in a dense graph, this actually makes it a linear time algorithm!

### 1.3 Introduction to Fibonacci Heaps

Can we do better? The answer is yes, with Fibonacci heaps.

“Fibonacci Heaps are an excellent demonstration of laziness. You should never work unless you have to, since there is too much work to do. Similarly, you should always procrastinate. If you start early, then you end up doing work that potentially may be never needed.”

-D. Karger
The core idea of data structures is to put off work until later. When you’re forced to work, make it count, not only to answer the question, but also to simplify the structure itself. A good model is to use an adversary, which tries to make you work, and we try to optimize and amortize our work versus theirs, which we formalize with a potential function.

So, let’s analyze our ‘Fibonacci heap’ (the name will become clear at the end of the analysis).

How do we be lazy with regard to insertion? We can use a linked list, and we can push with $O(1)$ time in this way.

**Joke 5.** *A more time efficient way to do this, is simply to say "ok," to a request for insertion. This uses 0 time, but obviously poses a problem in that every subsequent non-insertion operation will return the wrong answer.*

How about delete min? We need to scan for the minimum, which takes $O(n)$ time for $n$ elements. But amortized, this is only $O(1)$ per inserted item! This is actually faster than the priority queue, which takes $O(n \log n)$ time, since the heap property has to be maintained. But obviously we have a problem if a bunch of delete min operations are asked in succession, in where it will take $O(n^2)$ time asymptotically.

So instead, we’ll go back to our original principle of trying to simplify the data structure. Each comparison that we make in finding the minimum element also reveals to us information about smaller/larger elements. We can use this information to simplify runtime - simply ignore the larger elements when looking for the minimum element. To implement this, we’ll make a tree in where we compare every element, and the larger element in the comparison is made the child of the smaller element. This ends up making a **heap-ordered tree (HoT)** (also known as a **caterpillar**)! Now, the minimum is at the root after creating this tree structure. Then, after the first delete-min operation, only roots of subtrees are possible candidates for the next delete-min operation! This is a big time improvement, changing the runtime to just be proportional to the maximum number of children of the root.

But this is exactly the problem - our data structure is too inefficient when the root has many children. If we start comparing the elements starting with the minimum, then we just get a tree of height 2, with everything connected to the root minimum. After deleting the minimum node, then this just turns into $n - 1$ isolated elements, and so we lose all the information that we obtained regarding comparisons. Our previous approach put too much information on the minimum, which is lost after deletion, and hence we want to limit the number of children/node. To solve this, we can instead arrange a competition, in bracket style (similar to a binary tree). In this scheme, we compare random pairs of elements, then compare the smaller ones of those pairs, then compare
the smaller ones of those, and so on and so forth. After rearranging everything then, we end up getting a tree known as a **binomial tree**. This is because there are \( \binom{\log_2(n)}{k} \) elements at every tree depth \( k \), where \( n \) is the number of nodes. This also implies that deleting the minimum will only produce \( \binom{\log_2(n)}{1} = \log_2(n) \) subtrees.

This binomial tree structure still does not solve all our problems, since insertions and delete-mins can be interleaved, and we can’t run this bracket system each time. To get around this, we’ll instead record each node’s degree, which is equal to the number of children that a node had, or in other words, which ‘tournament round’ it is in (for example, a node that lost the first ‘round’ would have a degree of zero, while a node that won two rounds would have degree 2).

After delete-min is run, we will end up with a number of new HoTs, equal to the degree of the previous minimum. These will all be added to our collection of HoTs. Afterwards, we’ll only face off HoTs whose roots have the same degrees (known as the **union by rank heuristic**, similar to the union-find data structure). To consolidate them through this heuristic, we will append HoTs of degree \( d \) with each other, by connecting the root of one tree to be a child of the other. This reduces two HoTs of degree \( d \) to one of degree \( d + 1 \). This process starts from the HoTs with root degree 0, in order to end up with only one HoT of each root degree. By a similar analysis of the union-by-rank heuristic as the one in union find, the maximum degree we have is \( O(\log n) \) and hence the total number of HoTs we have is also \( O(\log n) \). This collection of HoTs is what we know as the Fibonacci heap.

If we combine this with lazy insert, we’ll have insertion just add a degree 0 HoT to the collection instead. On a delete-min, we will first delete the minimum element, and then put its children into the collection of HoTs. We then consolidate to find the next minimum element, simultaneously shrinking down to only having \( O(\log n) \) HoTs.

**Question 6.** *What is the runtime of the delete-min operation?*

<table>
<thead>
<tr>
<th><strong>Claim 7</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>In the worst case, the runtime of the Fibonacci heap delete-min is just ( O(n) ), which happens when there are ( n ) inserts (making ( n ) degree 0 HoTs) followed by a delete-min. This operation is amortized ( O(1) ) due to the insertions, but also allows for restructuring of the entire data structure to be a Fibonacci heap. Afterwards, the runtime for delete-min is proportional to the number of HoTs we have, plus the number of new trees created by deleting the root, which is ( O(\log n) ).</td>
</tr>
</tbody>
</table>

**Proof.** At the beginning, when there are \( n \) HoTs of degree 0, each node needs to be compared to build up the original Fibonacci heap, taking \( O(n) \) time. Afterwards, due to the structure of
the Fibonacci heap, a deletion only requires consolidation of the remaining HoTs. There are at most $O(\log n)$ HoTs originally, plus the number of children we have (bounded by the binomial heap structure as $O(\log n)$), and so consolidation will take at most that much time.

To formalize this, we can go through an amortized analysis with a potential function.

**Note 8**
As a review, in the potential method for amortized analysis, we have that

$$a_i = r_i + \Phi(i) - \Phi(i - 1)$$

where $\Phi(i)$ is our potential function, $r_i$ is the real cost, and $a_i$ is the amortized cost.

We have that $\sum a_i = \sum r_i + \Phi(\text{final}) - \Phi(0)$ by telescoping. If $\Phi(\text{final}) \geq \Phi(0)$ then the sum of amortized costs $\sum a_i$ is greater than or equal to the sum of the real costs $\sum r_i$, and hence the amortized cost becomes an upper bound on the actual cost. If the potential is always positive and $\Phi(0) = 0$ then this condition is automatically satisfied.

For our case here, we can let $\Phi(i)$ simply be equal to the number of HoTs we have. Then, the amortized cost $a_i$ will equal the final number of HoTs we have, plus the number of children that we make $O(\log n)$. Since the number of HoTs after a delete-min operation is $\log n$, the final amortized cost is simply $O(\log n)$!!

Hence, such a data structure provides $O(1)$ insertion and $O(\log n)$ amortized delete-min. But this is not good enough for us, since we’re still missing the decrease-key operation. This will be covered in depth in the next lecture, but the algorithm is simple - just cut off that subtree and make it a new HoT. The problem is that we can make trees that are not binomial trees, which is a problem due to their size (the number of children).

**Remark 9.** Usually, the location of the minimum element is also marked in the delete-min operation, such that accessing the minimum will take $O(1)$ time rather than the $O(\log n)$ time required to look through the tree roots we have in Fibonacci heaps.
2 Lecture 2: Fib Heaps Cont, Persistent Data Structs

2.1 Fibonacci Heaps Continued

We start off today with a review of how we defined the Fibonacci tree, that we defined last time with our collection of HoTs. We had the potential function being equal to the number of trees we had, and since the number of trees is $O(\log n)$, our amortized cost for delete-min is $O(\log n)$. Finally, we closed off by mentioning the decrease-key operation, which was simply to cut off the tree that decreases-key. However, this has the problem that if a node loses children over and over again, we no longer have our $\log n$ bound.

Note 10
Prof. Karger mentions the movie “Saving Private Ryan,” in where four children went to fight in war. The family ultimately decided that it was best to rescue the last of the children, since it would be horrible if a family lost all their children.

And so we can do the same with the Fibonacci tree, to fix our decrease-key operation. “If a node loses more than 1 child, then we will take it out of the line of fire” (i.e. make it a root node and cut it off as well) To implement this, we will add a “mark bit” to each tree that represents if a child has already been cut from a node. Then, a parent of the key which decrease-key is run on will be cut from the tree if it was marked. This has the potential to cause a cascading cut, in where we keep cutting until we find a clear mark bit.

Claim 11
We will have a nice data structure with $O(1)$ insertion and decrease-key and $O(\log n)$ for delete-min, if we can prove the following:

1. Show that cascading cuts are “free” amortized.
2. Show that the tree size will be exponential in the degree of the root. The base of the exponent doesn’t matter, since a size of $n$ will correspond to a degree of $O(\log_b n) = O(\log n)$.

Proof. Claim 1 To show the cascading cuts are free, we will make a potential function for the marks, namely $\Phi = \text{number of marked nodes}$, in all our trees in the data structure. Now, what is the cost of the decrease-key operation? The cost is just simply equal to the number of cut nodes $c$, plus $O(1)$ for ‘housekeeping operations.’ The number of marked nodes decreases by $c - 1$, plus
the one new node we mark, for a total change in potential of $2 - c$. This means that the amortized cost is constant $O(1)$!

But this has a problem. The potential function differs from the previous potential function, and that’s cheating, since we’re making a bunch of new HoTs in our fibonacci heap as well. We need to reconcile these potentials, and so we define the final potential to be $\Phi = (\text{number of HoTs}) + 2\times(\text{mark bits})$. The reason we have a factor of two in front is so we have one unit of work for the cascading cut itself, and the other unit of work to account for the increase in the number of HoTs. Going through with this analysis, we will then see that decrease-key will have amortized $O(1)$ time.

**Claim 2** To show a bound in the tree size, we’ll consider a node $x$ and its current children $y_1, y_2, \ldots, y_k$, where each $y_i$ is ordered by order of addition.

**Lemma** $y_i$ has degree $\geq i - 2$

**Proof.** Consider the arbitrary child $y_i$. It was added after children $y_1, y_2, \ldots, y_{i-1}$. When $y_i$ is added, then $x$ itself had degree $\geq i - 1$ since the other $y_i$ are children. Since we only consolidate nodes of same degree, then the degree of $y_i \geq i - 1$ when it was added. Because $y_i$ is still in the tree, then due to our rule for cutting, at most one child could have been cut from $y_i$. This means that the degree is $\geq i - 2$.

Now that we have this bound on degree, we can figure out how big the tree itself is. If we let $s_k$ be the minimum number of descendants (including children) of any subtree with a degree $k$ root, then we aim to find a bound on $s_k$ itself. We can easily find $s_0 = 1$ and $s_1 = 2$. We further have that $s_k \geq \sum_{i=2}^{k} s_{i-2} = \sum_{i=0}^{k-2} s_k$.

Solving this at inequality at equality, we get that $s_k - s_{k-1} = s_{k-2}$, and we see that rearranging it shows that $s_k$ satisfies the fibonacci recurrence! So $s_k = O(\phi^k)$ and hence is exponential in the degree of the tree.

So, we that fib-heaps are pretty useful. We have $O(1)$ insertion and decrease-key, and $O(\log n)$ delete-min. We also have $O(1)$ merging. This is actually optimal, since otherwise we could get a comparison sort with time $\Omega(n \log n)$, which is impossible.

Is it practical? Well, the constants of the operations aren’t bad, and sometimes they outperform binary heaps. However, fib heaps use pointers to store everything, rather than arrays (in implicit heaps). Additionally, CPUs also use caching, and memory-accesses are free in arrays rather than all over the place pointers, and so we lose caching efficiency if we stick with fibonacci heaps.
2.2 Applications of Fibonacci Heaps

Recall that Fibonacci heaps were originally designed to solve the Prim/Dijkstra shortest path-finding/minimum-spanning tree problems, and a regular heap could get time efficiency $O(m \log n)$. With a fibonacci heap, the decrease-key operations are all $O(1)$, and so the runtime improves to $O(m + n \log n)$, and this is linear except when the graph is really sparse.

Can we do better, even for sparse graphs, trying to get rid of the logarithmic term? Fredman Let’s look at MSTs to start with - they’re still slow with fibonacci heaps, since $n$ delete-mins from a size $n$ heap will take $O(n \log n)$ time. To get around this, we should try to keep the heap size small, say less than a constant $k$.

We’ll start of with a standard Prim’s algorithm: start from a node, and insert neighbors, delete min from the heap, etc, until the heap size hits $k$. The heap itself contains the neighbors of the current tree, and we will keep running the algorithm until we have $\geq k$ potential neighbors. To actually find the MST then, we will go to another node that has not yet been added to the MST, and start running Prim’s on that one, ending with the same condition or when we connect to a previously established tree.

After one iteration of this procedure, we end up with many subtrees that have yet to be connected to each other. This takes $O(m + t \log k)$ time (through a Fib heap) where $t$ is the number of nodes we have by standard Prim’s algorithm. To combine all the trees, we can contract each created subtree to a node (in $O(m)$ time), and restart the MST algorithm with a new $k$ (this will be known as one phase). Then, if we make $t \log k = m \implies k = 2^{m/t}$, we get linear time in $m$ per phase.

Now, how does these constants change at the conclusion of every phase? Since each contracted node will have $\geq k$ incident edges, the number of vertices at the after contraction at the end of every phase will be $\leq \frac{m}{k}$. This means that the number of nodes in the next phase is going to be $\leq \frac{m}{k}$, and so to achieve linear time per phase, the new $k$ of the next phase can be $2^{m/t'} \geq 2^k$.

Initially, we start with $t = n$ by definition, and with $k = \max \left( \frac{m}{n}, 2 \right)$ (so we don’t start with a crazy small $k$). What is our number of phases? Well, since by definition we will have an MST if $k = n$, then we can simply look at the time it takes for $k$ to reach $n$. With every phase, the number $k$ gets exponentially larger. We call this number of phases $\beta(m, n) = \min \{ i \mid (\log_2)^{i}(n) \leq k_0 = m/n \}$, and it is easy to show that $\beta(m, n) \leq \log^{*}(n)$. $\log^{*}(N) \approx 8$, where $N$ is the number of atoms in the universe. This is essentially linear.
2.2.1 Further Improvements

But Theoreticians are never satisfied. This bound was improved to $m \cdot \log \beta(m, n)$ using edge bundles. This was further improved by Chazelle to $O(m\alpha(n) \log \alpha(n))$ where $\alpha(n)$ is the inverse Ackermann Function, where $\alpha(N) \approx 4$.

We can get even better with randomization, into time $O(m)$. It is unknown if this is able to be solved in linear time yet, but surprisingly (!!!) we can actually get an optimal deterministic algorithm for solving MSTs with unknown runtime. (proof by Pettie and Ramachandran: brute force all algorithms for tiny spanning trees, choose the minimum, and the exhaustive search for best algorithm takes linear time. Then generalize to larger trees.)

Note 12
Logistical Note: in general, PSETs will use material covered mostly in Wednesday lecture, with a little bit of spillover into Friday. This time, we didn’t get to Persistent Data Structures in time, and there is no lecture Monday, so the last problem in the PSET will be dropped for this week. The next lecture will be next Wednesday.

2.3 Introduction to Persistent Data Structures

Tarjan is basically the God of data structures. He is also one of the best writers of theory papers. He and Sarnak also introduced the idea of persistent data structures. The general idea is to augment the data structure to support operations in past versions as well. (for example, to answer questions like: “what was the min yesterday?”) The ability to query the requisite task is known as partial persistence, but they even provide an ability to “insert a new key in the heap yesterday?” and ask “what was the min 3 hours ago?” incorporating the insertion of the key yesterday as well. This is basically time travel, called full persistence, where we can query and modify the past.

They came up with a way to do this, without regard to the type of data structure that is used. Any structure that is pointer based, which has fixed-size nodes whose fields hold values and pointers to other nodes, can be augmented. (Glaring Exception: no arrays)

Now, the idea is to wrap the data structure and the primitive operations on it. When modifying, we change the value or pointer in a node itself. Thus, if we can figure out how to make a node persistent, then we can extend this to the entire data structure. Changing each field to an array of time/value pairs (the fat-node method) allows us to do this. We pay a $O(1)$ space cost per insertion. The runtime of insertion is still $O(1)$, however queries become longer, to $O(\log t)$ time with binary search. This is a multiplicative slowdown, unacceptable for theoreticians, which we will resolve in the next lecture.
3 Lecture 3: Persistent Data Structs, Splay Trees

3.1 Persistent Data Structures Cont.

Recall from the last lecture that data structures are mostly pointer based structures, which are collections of \( O(1) \) size nodes that contain \( O(1) \) scalar values, as well as \( O(1) \) pointers to other nodes. We want to make what we call atomic operations, query and modify operations, persistent; doing this for every node makes the whole data structure persistent. We previously also talked about the fat-node method, which was unacceptable due to the multiplicative \( O(\log t) \) slowdown for lookup (need to do this for every node).

To start off in looking at how we can implement this, let’s consider a binary search tree, when we try to add a node to it. If we just overwrite the entire binary tree, then we lose persistence, even partial persistence. A brute force approach would result in storing a copy of the binary tree at every time step, the copy method, which is unfortunately quite slow. The lookup time is \( O(\log t) \), where this \( t \) is now the total number of modifications we ever made. But the good thing is that this is an additive \( \log t \) term, since we only need to find the root (rather than for every node), and then we have a data structure to work with. The problem with this approach is that a modify operation is \( O(n) \) and also takes \( O(n) \) space, which is highly unacceptable.

How do we get around this? The key idea is that almost the whole tree is the same after each operation. This means that if we just keep track of changes, rather than the whole tree at each time \( t \), then we can improve our modify operation. We copy the new node, as well as everything that changed because of this addition. The parent of the new node changes (it now has a pointer to the new root), and the parent of that also changes (since its child now has an additional pointer), and so on. So, if we make new nodes corresponding to the ones that need to be modified, with pointers to the old unmodified nodes, then we maintain persistency with less time and space (still unbounded: \( O(\text{number of ancestors}) \)). This method is known as path copying.

Tarjan found a way to combine the path-copying \( O(\log t) \) additive slowdown with the \( O(1) \) modification in fat-nodes - to consider “plump nodes” instead. The idea is that fat nodes are bad, and so if it’s too fat, then we do path-copy instead.

We’ll only go over this idea for tree structures, but it is not too hard to generalize to all pointer-based data structures (one can see this in the original Tarjan paper). In each node, we will have one extra timestamped field. We’ll use it to store the timestamp and action of the first modification (even changes in pointers) of the node, and on the second modification we make a copy of this node (with a new empty extra field). Once the second modification is made, then we will change
the time-stamped field of the parent to store the fact that its child changed to the new copied node at that time \( t \). Thus, we will have two options when we traverse to this node, depending on the time \( t \) - we will either point to the original child, or to the new node (this information is completely stored in the extra field, so no information is lost). Then, at a later time, when we want to modify that node, we will simply modify the new copy instead.

How do we implement the lookup operation? We can start at the root of the tree, and check if the field has a timestamp of modification before or after time \( t \). If it was, then we go to the new node as specified by the extra field. Otherwise, we go to the previous node. This is the operation of looking up a particular node, and since we only add a constant amount of operations, this is a \( O(1) \) slowdown. But there is a slight problem, because we still need an array of roots ordered by time. Every time the root is modified, then there is no parent that can store that change, and so we need to make a copy of the node itself. As a result, there is an additive \( O(\log t) \) slowdown since we need to find the appropriate root to start with.

What about the new space that this uses? Let’s do an amortized analysis, with the potential being equal to the number of live nodes that have a non-empty extra field. This is only a problem for nodes that we may still modify, and we aren’t going to modify nodes back in time. Therefore, we define a live node to simply be nodes reachable from the current root.

Let’s calculate the amortized space cost for an atomic modification. If we modify a node and the extra field is empty, then we can simply do this with a constant space increase, and the potential increases by 1. If we modify a node with a non-empty extra field, then we copy it, point the parent to the new copy, and then the old node becomes ‘dead,’ resulting in a decrease of 1 in the potential. Then, since the copy operation increases by \( O(1) \) space but the potential decreases by 1, then this is a free operation! These costs are going to cascade because it ends once we get to a node with an empty field, so we just get an overall space cost of \( O(1) \) for modification.

### 3.2 The Planar Point Location Problem

This is a problem in computational geometry, which is a field we will return to later in the course, but it can be solved with a persistent data structure approach!

**Definition 13**

*Computational geometry* involves problems dealing with points and lines in the plane or in space, and ignores the algebra for computation. Instead, we focus on geometrical concepts, and have operations such as checking for line intersections, checking if points are on lines or not, and the length of the segment itself, as \( O(1) \) primitive operations.
**Definition 14**

The **Planar Point Location** problem is where we have a planar subdivision, which is a splitting of the plane into a bunch of polygons. The question is to determine which polygon is a certain query point in. Line segments can emanate from any vertex, and can also end at any vertex, and the input size is in $n$, the number of segments.

The 1D version is simple - it is simply to ask which segment a query point itself lies on. We can easily solve this with binary search, and a binary search with sorted list of the endpoints can easily answer each query in $O(\log n)$ time, with $O(n \log n)$ time and $O(n)$ space to build.

A trick that’s often applied in computational geometry is **dimensionality reduction**, where we try to change a 2D problem into a 1D problem. We’ll use it here. If we project all the vertices of polygonal intersection onto a line, then we know which "slab" (x-coordinate region) the query point is in. We can then binary-search the segments, asking if the query point is above or below a specific line segment. The reason that the slabs are important is that it removes the ambiguity of above/below, and restricting it to just one slab results in no ambiguity and a total ordering.

What is the cost of this? We have two costs - a build cost and a query cost. For the query cost, we can find out which slab we are in in $O(\log n)$. After that, we can find the y region that we are in also in $O(\log n)$. Therefore, we can find the query cost in $O(\log n)$ with two binary searches in 2D. This generalizes to $n$ dimensions easily, becoming $O(d \log n)$.

For the build cost, let’s first consider the space cost. The number of slabs is proportional to the number of the vertices, and for every slab we need to keep track of the ordering of the segments, resulting in a total space cost proportional to $O(\text{vertices} \times \text{segments})$. And this can easily become $O(n^2)$, which is bad - this also means that our runtime is at least $O(n^2)$, unacceptable.

To solve this, we’ll use persistent data structures instead. Instead of considering the standard $xy$ plane, we’ll instead consider the x axis as a measure of time $t$, and keep track of a vertical line that moves from left to right (this is known as the **sweep line** method). Our question is to find out what the sweep line sees as time progresses. It sees intersections with some line segments, and these intersections do change coordinates as the sweep line moves to the right. But we don’t care about numbers, so we don’t need to keep track of them. The only thing that actually needs to be taken care of are changes in the ordering of the segments, as well as the appearing/disappearing of segments themselves. Segments can only appear/disappear/cross at vertices themselves, and so the only topological changes we need to keep track of are precisely at the $x$ coordinates of the vertices! Each change is relatively small, and so we can keep a persistent binary search tree on the segments in the sweep line.
These modifications of addition/deletion/crossing are all cheap with persistent data structures. We can use a balanced red-black tree, or more generally, any balanced binary search tree (BST). Each update to the persistent BST will only take $O(\log n)$ time (as is standard for a red/black tree), and a $O(\log n)$ space change (due to data structure persistency). As we sweep the line across the data structure, we will at most be doing $O(\text{segments}) = O(n)$ changes, and so all our insertions and deletions will just be $O(n \log n)$ time and space!

With this implementation, when we get a query, we can just query a slab by looking at the by 'time,' the x coordinate. Lookup is going to be $O(\log n)$ as discussed earlier with persistent data structures, and so we have the same query cost and the same building time cost as the 1D case. Space is still $O(n \log n)$, but we can refine this by looking at how red-black trees work. They modify $O(\log n)$ bits at each operation, but only do one rotation. We don’t care about the historical red-black bits, since they don’t matter for anything except for rebalancing. We’ll only persist the rotations, and since there’s only one per insert/delete operation, this improves the space usage to $O(1)$ per operation. Therefore, the space used by our red-black tree without persistently storing the color bits is simply $O(n)$, exactly equal to the 1D case. These time and space bounds are optimal!

### 3.3 Introduction to Splay Trees

Splay Trees, invented by Sleator and of course Tarjan, are another type of balanced binary search tree, which maintain $O(\log n)$ operations. There have been many balanced BSTs developed: red-black, AVL, scapegoat, 2-3 trees, etc. But all of these previous trees are kind of annoying, since you have the hassle of tracking extra information. Sleator and Tarjan developed self-adjusting trees, which don’t store any balance info, and are “absolutely astounding”.

These self adjusting BSTs don’t have any balance info, and are sometimes not even balanced, but nevertheless take $O(\log n)$ amortized time. They even outperform regular BSTs in formalizable ways, and contain more operations than a regular BST. They can be merged and split in $O(\log n)$, and other more complicated operations can even be free.

The drawback of splay trees? A much more sophisticated analysis, involving “cleverness and black magic.” The idea is to use a potential function to measure imbalance, and you need many insertions to cause imbalance. However, searches will decrease the potential, and so are paid for even if there is an imbalance. We’ll elaborate on this for the next lecture.
4 Lecture 4: Splay Trees

We know how binary trees work - they will have all operations proportional to the depth, which is usually $O(\log n)$ but can be as much as $O(n)$, unless we use some sort of balancing. But splay trees are special. They don’t store any auxiliary information, and yet are still able to be provably balanced. We’ll have a potential function $\Phi$ which measures how unbalanced the tree is, in order to make any operation amortized $O(\log n)$. Splay trees, like the data structures we’ve seen so far, are efficient because they are lazy.

4.1 Heuristics of Splay Trees

One of the primary heuristics in making the splay tree is by shortening long paths. Long paths take a long time to traverse and have a large potential, and shortening them reduces the potential and pays for the traversal itself. When we insert, the path increases, and the potential also increases such that we have enough to pay for a search later.

Another heuristic is to rebalance with rotations. We’ll say that a tree is balanced if the left and right subtrees of the root roughly have the same size. During a search, if the tree is balanced, then searches are fast and searches can be $O(\log n)$ time. In general, if the subtree is less than some constant factor of its parents’ size, then the tree is balanced.

4.2 Solving the Balancing Problem

The problem with trees appears when we have fat children. Then, when we descend, a search through a fat child takes significant time, so we’ll say that fat children have a large potential. Thus, to reduce the time of operations, we should eliminate fat children. One proposal to do this is to rebalance through rotations, but this doesn’t work, since rotations may still leave behind a long path.

What does work, however, is a double rotation. These double rotations will take different cases, based on the relative positions of the wanted node, their parent, and their grandparent. We’ll define a zig-zag operation of a node $x$ to take place if $x$ is the right child of its parent $y$, which is the left child of the grandparent $z$. The zig-zag operation will involve first rotating $x$ with $y$, and then rotating $x$ with $z$. The transformation is shown below:
We'll also define a **zig-zig** operation to take place if \( x \) is the left child of its parent \( y \), which is the left child of the grandparent \( z \). Here, we'll first perform a rotation between the \( y \) and \( z \), and then \( x \) and the now-parent \( z \). Finally, a **zag-zag** operation is the symmetric variant of the zig-zig operation, except with everything reversed, and takes place when \( x \) is the right child of \( y \), which is the right child of \( z \). The zig-zig operation is shown below:

Where did the intuition for thinking double rotations may work come from? Prof. Karger doesn't really know himself:

“This is what separates the truly great. A few lightning strikes of insight are enough to get you into the textbooks.”

### 4.3 Implementation and Analysis of Operations

We now define the **splay** operation of a node \( x \) to effect a double rotation of the node up the tree, until it becomes the root node. If \( x \) eventually becomes the root’s child, then we just do a single rotation at the end.

With the splay operation, we define the **search** operation to simply find the node \( x \) with standard techniques. Afterwards, we splay \( x \), bringing it to the root. This is \( O(\log n) \).
For our analysis, we will assume that each item $x$ has a weight $w_x$, which is not intrinsic to the data structure, but will help with the analysis. For now, we’ll set all the weights $w_x$ to be equal to 1, and we’ll additionally define the **size** function $s(x)$ of a node $x$ to give us the total weight of the subtrees of $x$. For example, when all the weights $w_x$ are equal to 1, the size function simply counts the number of nodes of the subtrees rooted at $x$. We will define the **rank** function $r(x)$ to be $\log_2 s(x)$, which intuitively gives us the ‘height’ of the splay tree. Finally, we define the potential $\Phi$ to be the sum of all ranks of all the nodes in the tree. Intuitively, $\Phi$ highly penalizes large deep subtrees, and rotations/splays to raise the subtree itself will help us raise the potential.

We will now introduce a key lemma in our analysis:

**Lemma 15 (Access Lemma)**
The amortized time to splay a node $x$ given the root $t$ is equal to

$$3(r(t) - r(x)) + 1 = O(\log(s(t)/s(x)))$$

where the functions $r$ and $s$ refer to the initial positions of $x$ and $t$. We will also define a function $r'(x)$ to refer to the rank of $x$ after splaying, and so the Access Lemma can also be written as follows:

$$3(r'(x) - r(x)) + 1 = O(r'(x) - r(x))$$

**Proof.** If we can prove that the amortized cost is $3(r'(x) - r(x))$ for a double rotation, then we can simply telescope our sum to prove the key lemma. So, we will focus on a single double rotation. The +1 in the lemma itself comes from the potential single rotation we may need to do at the end.

When we do a zig-zig rotation, the real cost is 2 rotations. The potential of all the nodes not being rotated doesn’t change. Thus, the only potential changes we have to consider are that of $x, y, z$. The change in potential is $\Delta \Phi = (r'(x) - r(x)) + (r'(y) - r(y)) + (r'(z) - r(z))$. The rank of $x$ increases while the rank of $z$ decreases, and the potential of $y$ can either increase or decrease.

Intuitively, if $r'(x) \gg r(x)$, then the cost of rotation is easily paid. If $r'(x) \approx r(x)$, then the cost of rotation is not paid with $x$’s rotation, but we do have the additional information that subtree $A$ is pretty fat, which means that the decrease in $y$ and $z$ potentials can be used to pay for the rotation.

To formalize this, we know that $r'(x) = r(z)$ by definition, and likewise $r'(y) < r'(x)$ and $r(y) < r(x)$. The actual cost is as follows:

$$\text{cost} = 2 + \Delta \Phi \leq 2(r'(z) - r(x)) + (r'(x) - r(x)).$$

19
We will prove our lemma then if
\[ 2 + (r'(z) - r(x)) \leq 2(r'(x) - r(x)) \iff \log_2 \left( \frac{s'(z)}{s'(x)} \right) + \log_2 \left( \frac{s(x)}{s'(x)} \right) \leq -2 \]

Letting \(|S|\) denote \(s(S)\), looking at the effect of rotation, we have that
\[ s'(z) + s(x) = |A| + |B| + |C| + |D| < s'(x) = |A| + |B| + |C| + |D| + 2. \]

Since the maximum value of the function \(\log_2(x) + \log_2(1 - x)\) is \(-2\), after substitution, we have that
\[ \log_2 \left( \frac{s'(z)}{s'(x)} \right) + \log_2 \left( \frac{s(x)}{s'(x)} \right) \leq \log_2 \left( \frac{|C| + |D|}{|A| + |B| + |C| + |D|} \right) + \log_2 \left( \frac{|A| + |B|}{|A| + |B| + |C| + |D|} \right) \leq -2 \]
which proves the Access Lemma for a zig-zig rotation.

The zig-zag case is left as an exercise. \(\square\)

With this lemma, we can prove our desired result:

**Corollary 16** *(Balance Theorem)*

A splay operation takes amortized time \(O(\log n)\)

*Proof.* If all the weights are equal to 1, we know that \(r'(x) = \log n\). By the Access Lemma, the splay time is then \(O(\log n)\). Since the splay time is doing much more work than a find operation, then the entire search operation is \(O(\log n)\). \(\square\)

### 4.4 Results with the Access Lemma

When the weights are not all equal to 1, we can try to bound the potential in order to derive more results about the splay tree. The actual cost is the amortized cost minus the change in potential, which supposing we have \(m\) operations, is \(O(m \log n) - \Delta \Phi\). How large is this change in potential?

We can derive a few simple bounds. Letting the weights be \(w_x\) and having total sum \(W\), we have that \(\Phi(0) \leq n \log W\), which corresponds to the case where everything is a root. Likewise, we can also get a nice lower bound of \(\Phi(m) \geq \sum \log w_x\), where everything is a leaf. Thus, \(\Phi(0) - \Phi(m)\), the extra amortized cost, is \(\sum \log \frac{W}{w_x}\). Notice that the terms in the summation \(\log \frac{W}{w_x}\) are upper bounded by the time it takes to splay the node \(x\)! This tells us that the change in potential for the amortized cost is negligible and the real cost of splaying is approximately equal to the amortized cost.
With the access lemma, we can prove many results, for example, the static optimality theorem:

**Example 17**
Suppose we perform $m$ operations, such that item $x$ is accessed with probability $p_x$. To optimize the time of performing the operations, the best thing to do is to put the most-accessed items near the root. Since each level $k$ has $2^k$ spots, any item with $p_x \geq 2^{-k}$ can go to level $k$. Thus, the overall search cost is $m \sum -p_x \log_2 p_x = mS$, where $S = -\sum p_x \log_2 p_x$ is the entropy.

**Theorem 18** (Static Optimality Theorem)
Splay trees achieve this search cost as well, without knowing the individual $p_x$.

**Proof.** Let the weights $w_x = p_x$. By the access lemma, the cost of each element access is $\log_2 \frac{w}{w_x} = \log \frac{1}{p_x}$. The conclusion follows.

We will define a static finger, which comes from the idea of locality of reference. The idea is that nearby data will most likely be accessed, and so we may start a search from a ‘finger’ rather than a root. For certain types of accesses, starting from a ‘finger’ rather than the root is much faster, and splay trees do just as well. Additionally, even if the finger is dynamic and can move around, the splay tree also matches the time complexity.

What can’t we do with splay trees? Well, we think that they’re actually always optimal - there is a long standing unresolved conjecture about them:

**Conjecture 19** (Dynamic Optimality Conjecture)
Define a binary search tree strategy to be one in which we can do whatever operations we want in a binary tree, with complete knowledge of node requests. No matter what binary search tree strategy we have, splay trees will match the performance. No counterexamples have been found so far.

**Joke 20.** This conjecture will be a homework problem; anyone who solves it gets an A.

**Note 21**
The current state of the art towards the conjecture is from Demaine (another MIT prof), who introduced Tango Trees, which were provably shown to perform within $O(\log \log n)$ of the optimal binary search tree strategy.
5 Lecture 5: Splay Updates, Buckets

5.1 Wrapping up Splay Trees

Recall that previously we showed that the find operation in splay trees was amortized $O(\log n)$ time and we also got other bounds by changing the weights in our potential function. While we could just use a standard insert/delete operation, this is unacceptable, since without a splay a linked-list-like splay tree will have $O(n)$ insertion. So, we can just extend this easily - for an insertion, we first insert the element, then splay, for an amortized runtime of $O(\log n)$. However, since we inserted a new item, this also leads to an increase in potential and our analysis has to be made much more complicated. The analysis for delete is even worse, since we need to choose what exactly to splay.

Let’s try alternatives. Specifically, we’ll define the **split** and **join** operations as follows:

**Definition 22**

A **split** operation takes in a node as an argument, and returns two trees - one of which contains the items with key $\leq x$, and the other containing the items with key $> x$.

A converse operation, **join**, will take in two trees $T, S$ such that $T$ has all items with key less than any element in $S$, and return their union.

To implement these two operations, it is much simpler than our previous idea. For a split, we simply splay $x$ (or its predecessor if $x$ doesn’t exist), and then remove the right subtree, creating two subtrees that satisfy the requisite conditions. The amortized time is $O(\log n)$, and the potential obviously. For a join, we simply splay the minimum element of tree $S$ to the root, and then joins tree $T$ as a child of the minimum element of $S$. The potential increases by $O(\log n)$ which is fine, so the amortized cost is still $O(\log n)$.

Now that we have the split and join operations defined, we will implement insertion by simply calling split($T, x$), giving us subtrees $A$ and $B$ with $A < B$, then calling join($A, \text{join}(x, B)$). For deletion, we can first split($T, x$), creating subtrees $A \rightarrow x$ and $B$. Removing the element $x$ and then joining $A$ and $B$ effects deletion. Both take time $O(\log n)$.

When we implement splay trees in practice, we can use top-down splaying and to splay only on a long path, to improve runtime by a constant factor. The decrease in time is actually due to memory structures itself, since reading memory is much faster than writing memory. Another heuristic is to stop splaying after a while, which works since splay trees keep frequently accessed elements on the top. This heuristic is dangerous, however, if the probability distribution of inputs change.
Note 23
Another usage of splay trees is actually as a compression algorithm. Suppose that you want to compress a sequence of characters. We store them as a splay tree, but instead of sending a character over the communication link, we send the path in the tree, and then splay the element afterwards. The number of bits sent over the communication is equal to the search time, and so this is a way to show an information-theoretic result with data structures.

5.2 Buckets and Indirect Addressing

Previously, the things we have done were all with pointer-based structures, and we could essentially replace any arrays that we had with a pointer-based structure. But there are actually applications where arrays are optimal, for example, shortest paths when the edge weights are integers in \([1, C]\). Recall that the shortest path problem can be solved in time \(O(m + n \log n)\) with Fibonacci heaps, or \(O(m \log n)\) with standard methods. If \(C = 1\), then a BFS works and just takes \(O(m)\) time. To generalize, we can split up a weight \(w\) into just a chain of \(w\) edges, and this takes time \(O(mC)\), which beats the previous algorithm for small \(C\).

Let’s go back to our ideas about priority queues, which we used in Dijkstra’s algorithm. One thing to note is that in the priority queue, the minimum distance is nondecreasing. If we can make a specialized priority queue for this case, then we may be able to simplify.

Dial said to simply make an array of buckets to do so. In bucket \(d\), we put all the items of distance estimate \(d\). For example, if we start with source \(s\), then we just insert the neighbors in their requisite buckets, and then remove the source from the array. Afterwards, as in Dijkstra’s, we need to find the new min. This we can do simply by going to the next bucket, via a forward scan!

Note 24
We need to do a forward scan here, since there are too many overlapping min neighbors from deleting the other elements, and it is hard to keep track of the min auxilliarily.

Our algorithm will work as follows: When we have a node in the priority queue, then we will do a forward scan to the minimum, remove the minimum, and then insert the minimum’s neighbors in our bucket. We repeat this, with our forward scan starting from the previous minimum.

The time it takes for neighbor updates is \(O(m)\), and the time for scans should just require \(O(\text{max distance}) = O(D)\). \(D\) itself is bounded by \(nC\), and so the overall runtime is \(O(m + nC)\) which is better than before!
The space we require is $O(n + D) = O(nC)$. But we can do better. The range of relevant values within the array is from $d$ to $d + C$. So, if we store our array in a circle mod $C + 1$, by using the same index to hold different values, none of the values will be occupied at the same time. Thus, with a circular array, we only need $O(n + C)$ space, which is optimal.

**Joke 25.** This algorithm, Dial’s Algorithm, was invented in ’73, when people could get things named after themselves with only arrays. Now, we need fancy data structures to get something named after ourselves.

### 5.2.1 Improvements

This is an algorithm that cannot be beaten with a fancy data structure, since the constants are good. However, if $C$ is large, then there is possibility for improvement. The largest time cost that we have is with the forward scan, and we can optimize this. We’ll do so with a 2-level bucket scheme, where we auxillarily make blocks of size $b$ that records how many buckets in each block which are nonempty, that are updated when elements are deleted/added. Now, on a scan, we can simply traverse blocks until we get to a nonempty bucket, and then traverse the nonempty bucket.

What are the improvements of runtime? The neighbor updates are still $O(m)$. The bucket scan is $O(b)$ per delete-min, for a total time of $O(nb)$. The block scan takes time $O(nC/b)$, since we go to at most bucket $nC$ and can skip $b$ buckets every time. Choosing $b = \sqrt{C}$ causes the total time to be reduced to $O(m + n\sqrt{C})$.

But if we can do it once, we can do it multiple times. Let’s try a 3-level bucket scheme, where we have superblocks that store blocks, which store buckets. Then, we scan for a non-empty superblock, then scan it for a non-empty block, then scan that for a nonempty bucket. The optimal time is from setting the block size to be $C^{1/3}$ and the superblocks size $C^{2/3}$, for a total runtime of $O(m + nC^{1/3})$.

When we have a k-level bucket scheme, then we can make the exponent of $C$ very small. But this is a problem, since when we have more and more layers, we need $O(k)$ time to update from all our insertions, and also need to scan $k$ layers. The overall runtime becomes $O(k(m + n \cdot C^{1/k}))$.

### 5.2.2 Formalization with Tries

This idea can be formalized to a trie, which is a depth $k$ tree over sibling arrays of size $\Delta$. Essentially, every element within the trie is an array of size $\Delta$, which itself also contains a subtrie of depth $k - 1$. We have a range of $C$ possible values (once again, thinking in a circle). Ideally, we choose $C + 1 = \Delta^k$ to store all the elements under consideration, and hopefully $\Delta$ is a power of two (we’ll see why later).
Let’s consider finding minima in this trie. Insertion and Deletion is $O(k)$ per operation, since that’s just the time needed to traverse down the trie. For the delete-min operation, we need to first find the minimum nonempty child of root, which takes time $\Delta$. Then, we can go down to look at the array pointed to by that element, and so on, until we reach the bottom layer of the trie. This takes time $O(k\Delta)$. The overall runtime is then $O(mk + nk\Delta)$, which when minimized for $k$, becomes $O(m \log_{m/n} C)$. This is linear for a dense graph and equal to $O(m \log C)$ for a sparse graph.

### 5.2.3 Laziness wins again

How do we make this even better? Be lazy! Denardo and Fox (’79) proposed to modify the insert operation to procrastinate more and be more lazy: “Don’t push items down until we must.” We’ll only make a child in the trie if we have more than one element in the node itself, essentially only expanding as necessary. This is a good thing to do, but if we’re only doing inserts and deletes, we don’t even need to compare things for inserts and deletes! Therefore, we will only invest time into expanding a node into its child array if we need to scan that bucket when we do a delete-min. We will then only have one “active” trie node per level, which are on the path to the current min. For the other elements, we just let them stay in the bucket in the lowest active node they belong to. To quickly check emptiness, we will keep track of the number of items in each layer (not descendants).

In implementation, for insert, we start at the top of the trie. We walk down until we land in a non-minimum bucket, and increment that level’s item count, and then stop. For the decrease-key operation, we may remove it from the current bucket, and find its new bucket, which is either behind or below its current bucket. For implementation, we simply check to see if our element can fit into the nearest previous bucket, and if they do, we may need to go down another layer. Since the min is monotonic, then we have no risk of going beyond our expanded bucket structure. The delete-min operation can be implemented by actually removing the min, and then checking the number of items in each layer. If the layer is non-empty, then we just do a forward scan. If the layer is empty, then we go up a layer, and do a forward scan to find the bucket in the layer above, and then expanding the bucket of the new min. Overall, we keep going up until we get to a non-empty layer, forward scan for the first non-empty bucket, and then expand it to the bottom.

The real cost for insert or decrease-key is $O(k)$, since we simply traverse down the trie. If we amortize the decrease-key cost into the insertion, then decrease-keys become $O(1)$. For a delete-min, we need $O(\Delta)$ to scan, then $O(k)$ to go up and down the trie, for a total time of $O(k + \Delta)$.

Thus, with laziness, the runtime is $O(m + n(k + \Delta))$, which when balanced (here a power of 2 for $\Delta$ is nice), becomes $O\left(m + \frac{n \log C}{\log \log C}\right)$. It’s not a big improvement, but it is from simple data structure, and $\log \log$ factors are relatively important in data structures. This can be improved even further with a priority queue on the scan, which we’ll implement in the next lecture.
6 Lecture 6: VEB queues, Hashing

6.1 Improvements to the Denardo and Fox queue

When we do a delete-min in our Denardo and Fox queue from previously, we have to do a forward scan to find the smallest non-empty bucket. But this may take a decent time, while it is just a problem in finding the next smaller number, a perfect application for a priority queue! So, Cherkassky, Goldberg, and Silverstein capitalized on this insight by making a Denardo and Fox queue, plus adding a standard heap to find the min entry at each level. (This is known as a HoT queue, heap on top queue.) Essentially, the top layer of the trie is made much bigger and into a heap. If we make the size of the heap $2^\Delta$, the cost of heap traversal is negligible compared to the rest of the structure, but the advantage is that we can actually reduce the number of layers needed to store values. This ultimately changes the runtime to $O(m + n(\log C)^{1/3})$.

Note 26
Goldberg, one of the inventors of this improvement, is a big practitioner of Experimental Algorithms. Experimentalists actually implement the algorithm and then see how it can be improved - what heuristics can improve access time, etc? These can also lead to new theorems, and Goldberg’s papers are a great demonstration of these techniques.

Note 27
Silverstein was Prof. Karger’s brother’s roommate and also Prof. Karger’s classmate in graduate school. Silverstein dropped out of graduate school to work at Google, and now is doing great work at Khan Academy, showing how a deep study of algorithms can lead to many places.

6.2 VEB queues

This insight in the HoT queue actually comes from a previously introduced data structure, van Emde Boas trees (1973). The general idea of VEB trees is to implement a priority queue as a two-layer trie, with each layer having $\sqrt{C}$ elements. We will work with $b$-bit words ($C = 2^b$).

The structure of the queue $Q$ is as follows:

- $Q.min$: This is stored in $Q$, but not in the tree itself (this is critical!)
- $Q.low$: This is an array of size $\sqrt{C}$. The intuition is that each will have $b/2$ bits. Each of the values in the array point to another VEB queue on $\sqrt{C}$ values in range.
• **Q.high**: This is another VEB queue, which stores the indexes of nonempty level-2 blocks. The maximum size of $Q.high$ is $\sqrt{C}$, each with $b/2$ bits.

This gives us a priority queue in $Q.high$, which allows us to easily access the minimum elements, as wanted. Also, note that the number of layers we have is $\log b = \log \log C$, a significant improvement.

Let’s now implement our operations as follows:

- For our insertion, if $x < Q.min$, we will just swap $x$ and $Q.min$, and the problem reduces to just inserting an element greater than the min. Now, we will say that $x = 2^{b/2}x_h + x_l$. We check if the queue $Q.low[x_h]$ is nonempty. If it is nonempty, we simply insert $x_l$ into $Q.low[x_h]$. Otherwise, we just make a VEB queue there, which we can insert $x_l$ into, making sure to also insert $x_h$ into $Q.high$ to keep track of the new queue.
- For delete-min, we can simply query $Q.min$. We remove it, and then we need to replace it with the minimum of the recursive structure. To do this, we look at $Q.high.min$, the index of the first nonempty level-2 block (or equivalently, the high bits of the new min). A special case occurs if $Q.high.min = null$, this means that the recursive structure is empty, and so there are no more elements in the queue. Otherwise, $Q.high.min$ tells us $x_h$, and deleting the min from $Q.low[x_h]$ gives us $x_l$. Finally, if deleting the min from the second layer results in an empty queue, we need to delete it from $Q.high$ in order to keep our queue consistent. The new minimum is $2^{b/2}x_h + x_l$.

Let’s do a quick runtime analysis. Insertion of a $b$ bit integer is $T(b) = 1 + 2T(b/2)$, which occurs when $Q.low[x_h]$ is not made yet, needing to insert two integers on size $b/2$ queues. Using Master’s Theorem tells us that insertion is $O(b)$. However, we can improve on this recurrence formula, since insertion $x_l$ of into an empty queue (happening if $Q.low[x_h]$ is previously empty) only takes time $O(1)$! Our recurrence then becomes $T(b) = 1 + T(b/2) = O(\log b) = O(\log \log C)$.

Likewise, for a deletion, the recurrence also looks like $T(b) = 1 + 2T(b/2)$, but the deletion of the level-2 min is just $O(1)$, and so the actual recurrence is $T(b) = 1 + T(b/2) = O(\log \log C)$. Note that here, if $Q.min$ was not stored separately, we could not do the simplification of the recurrence.

VEBs are exponentially better than every other heap structure we’ve seen. But it has a drawback in the space required. For a $b$ bit queue, we need approximately $2^{b/2}$ space for the top layer, and then another $2^{b/2}$ space for every item we insert.

Can we do better? Well, the arrays themselves are only used for guaranteeing $O(1)$ lookup. If we can find another data structure to do $O(1)$ lookup with significantly better lookup efficiency (spoiler alert: hash tables), then the space complexity becomes much better.
6.3 Hashing

Note 28
Hashing is an important topic to understand; one reason is since it involves randomization, which was a stated prerequisite of the course. We'll use hashing to create dictionaries, which are data structures that store key/value pairs, with insertion, deletion, and lookup key operations. While hashing should have been covered in standard undergraduate algorithms courses, we will now prove important results without the assumptions made in those previous courses.

For our model, we will make our keys integers in $[m] = \{1, 2, \ldots, m\}$, which can be stored in an array of size $m$ with $O(1)$ lookup. But we can do better. Supposing that we only have $n$ keys to insert, the question is to ask if we can use $s > n$ space to store everything in $O(1)$.

To do so, we'll use a hash function $h : [m] \to [s]$, and to store each key $k$ in an array at position $h(k)$. The problem with a standard hash function is with collision of values ($h(k_1) = h(k_2)$). We can get around it by storing all the values that hash to one particular value in a linked list, but then this makes the access time potentially not $O(1)$.

We need a good hash function, one that causes few collisions. However, this is impossible:

Theorem 29
There are no good hash functions.

Proof. This is impossible due to the pigeonhole principle, which says that at least one bucket will have at least $\frac{m}{s}$ elements. If we choose the input keys devilishly, the lookup time for that bucket is $O(m/s) = o(1)$, which means that that hash function is not good.

To get around this, we'll instead use randomization, with a hash family, which is simply a set of hash functions such that we can pick a good one for any set of items.

What if we just pick a random function from our hash family?

Theorem 30
If $n$ keys randomly distributed in $[s]$ are hashed, then the expected access time in our hashing function is $O(1 + \frac{n}{s})$.

Proof. We will define the indicator random variable $C_{ij}$ as follows: $C_{ij} = \begin{cases} 1 & \text{if items } i, j \text{ collide} \\ 0 & \text{otherwise} \end{cases}$.
The time to find element \( i \) is simply \( 1 + \sum_j C_{ij} \). The expected time to find an element is then
\[
\mathbb{E}[1 + \sum_j C_{ij}] = 1 + \sum \mathbb{E}[C_{ij}] = 1 + \sum \mathbb{P}[\text{collision between } i \text{ and } j]
\]
by linearity of expectation.

The probability that two randomly assigned elements \( i, j \in [s] \) collide is simply \( \frac{1}{s} \). Then, the expectation value simply becomes \( 1 + \frac{n}{s} \), as desired.

Even though we've proven that nice theorem, this doesn't help us, since remembering all these random functions take a huge amount of space! Specifically, there are \( m^s \) functions from \([m] \rightarrow [s]\), taking up \( m \log s \) space (this bound comes from information theory).

Getting around this, Carter and Wegman introduced \textbf{2-universal hash families}. This comes from the observation that we don't need a completely random hash function, but rather just a function that is \textbf{pairwise independent}, such that any two elements in \([s]\) will have a probability \( \frac{1}{s} \) of collision. While pairwise independence sounds like it would trivially lead to \textbf{mutual independence} (independence of all elements), the following example demonstrates that this assumption is false:

\begin{example}
Let's consider flipping three coins \( x, y, z \). The probability that any pair of two coin flips are the same is just \( \frac{1}{2} \).

What if we're lazy? Let's just flip two coins \( x, y \), and set \( z = x \oplus y \) (exclusive or). The probability that any pair of two coin flips are the same is still \( \frac{1}{2} \)!
But these values are obviously not independent.
\end{example}

Thus, if we can find a way to generate pairwise independent hash functions with less space, then we get around the bottleneck that our previous attempt at random hash functions had.