Here's the outline for today:

- Introduction: describe the heap operations and desired time bounds, and why they're useful
- binomial heaps (algorithm and analysis)
- fibonacci heaps (algorithm and analysis)

1 Introduction

We want to represent a set of keys. The keys are from a totally ordered universe. (That is, you can compare any pair of keys and get an answer $>$, $=$ or $<$.) We further want to support these operations:

makeheap()	Return an empty heap
insert(k, S)	Insert a key k into a heap S. Return a pointer to the node in S containing the key k.
deletemin(S)	Delete the minimum element from S and return it.

A straightforward way to implement this is a balanced binary search tree. If we keep the tree balanced, we can achieve $O(\log n)$ insertion and deletemin. You've probably also seen in other classes a way to implement heaps efficiently using an array — that's used in heapsort. This is called an "implicit heap", because there's an implicit tree structure that is used by the algorithm (not explicitly represented using pointers.) The running time of insert and deletemin are $O(\log n)$, where n is the current size of the heap.

[One drawback of implicit heaps is that they do not grow gracefully – when a heap overflows, you have to grow the array to make more room. We earlier saw that by using a doubling strategy, you can ensure that in the amortized sense, this is not a problem. You'd also like it to shrink appropriately. This can be done as well.]

Another operation that we're going to be discussing is:

decreasekey(p, k', S)	This takes as input a pointer into the data structure (p) to a place where a certain key k is stored. And it updates that key with k'. $k' < k$.

Note that to use this, the main program that is using this ADT needs to have these handles (or pointers, like p) into the internal of the data structure.

It's not *a priori* obvious why this is useful. Just to give the most compelling example, consider Dijkstra's algorithm for finding all shortest paths from a source vertex in a directed graph with non-negative edge lengths. Let the number of nodes in the graph be n, and the number of edges be m. In general, m could be as large as $\Omega(n^2)$. The running time of the algorithm boils down to:

- n inserts
- n deletemins
- m decreasekeys

In all cases, the size of the heap is bounded by n. (For clarity, assume that the graph is connected and $m \geq n - 1$.) If the running time of all these operations are $\log(n)$, this gives a running time bound for the whole algorithm of:

$$O(m \log(n)).$$

However, if we can get decreasekey down to O(1) time (and we can) this running time becomes:

$$O(m + n \log(n))$$

This is considered a breakthrough – we're basically removing a $\log(n)$ factor from the running time.

Another operation that is often discussed is:

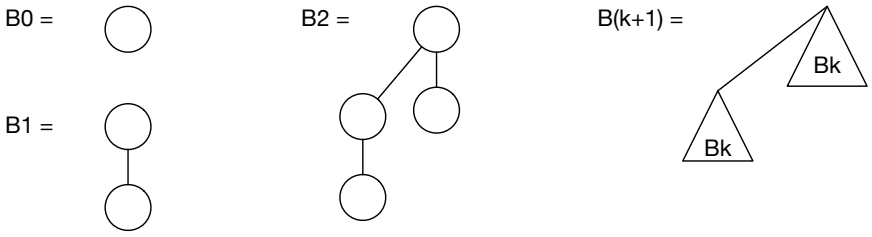
meld(A, B) Take two heaps A and B as input and return a new heap representing the union of the sets stored in A and B.

We'll begin our description of solutions to these problems with binomial heaps. This algorithm can handle makeheap, insert and meld in $O(1)$ time, and deletemin in $O(\log n)$ time (decreasekey is not supported). Then we show how to modify binomial heaps and create a data structure called fibonacci heaps that that also supports decreasekey in $O(1)$. (All of these time bounds are amortized.)

2 Binomial Heaps

First we describe what the data structure looks like, then we show how to implement all of the operations (except decreasekey) using it.

Basically a binomial heap is a collection of trees called binomial trees. The binomial tree of rank k, B_k, is defined recursively as follows:



Simple properties of binomial trees:

- B_k has 2^k nodes
- The root of Bk has k children
- These children are each a root of a binomial tree, and the ranks of these are $0, 1, 2, \ldots, k-1$.
- The number of nodes at distance d from the root in Bk is k choose d.

Each node of a binomial tree is going to be used to store a key, and the keys are in heap order in the tree, that is, the key in a node is always $\geq$ the key in its parent. Therefore the minimum key is at the root of a binomial tree.

The primitive operation on binomial trees is the link:

link(A, B) Take two binomial trees of the same rank, and link them together to make a new binomial tree of one greater rank.

This is implemented simply by comparing the root nodes of the two trees, and the one with the smaller root gets to be the root of the new tree, and the other one is made a child of the root. This will take $O(1)$ time.

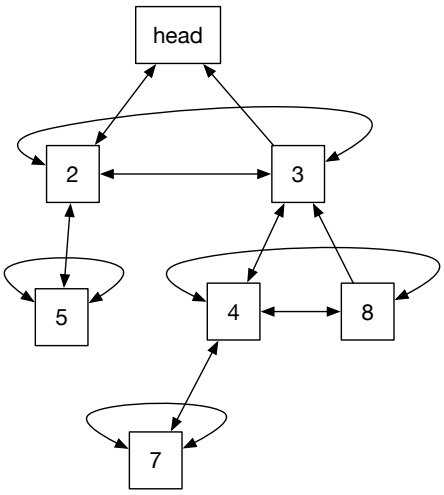
Here's how we'll represent binomial trees. (Note: Some of the choices made here are to make it easier to explain the jump to fibonacci heaps, to support decreasekey, and are not really needed for binomial heaps. One example is the parent pointers.)

Each node of the tree has the following fields:

- Parent pointer
- Left and right sibling pointers
- Child pointer
- My rank (the number of children I have)
- Key

So the children of a node are linked together in a doubly linked circular list. Each of them points to the parent, and the parent points to one of the children.

This allows us to, for example, walk up the tree, in $O(1)$ time per step. It also allows us to, given a pointer to the root of a subtree, in $O(1)$ time to delete that subtree. (This functionality is needed for fibonacci heaps, but not for binomial heaps.)



This is how a binomial tree is represented. The binomial heap is simply a doubly linked collection of binomial trees. There's an additional node that is the "head" of the whole heap. It has the same structure as any other node but it does not store any keys, and instead stores the number of keys in the whole heap.

Here's how we implement the operations:

makeheap() create a head node pointing to an empty list of binomial trees. Time: $O(1)$.

meld(A, B) We simply merge the doubly linked lists of binomial trees pointed to by the head nodes. Make a new head node, point it to the merged list, and return it. Time: $O(1)$.

insert(k, A) Create a binomial tree just for the key k, and add this to the list pointed to by the head of A. Time: $O(1)$.

deletemin(A) This is a bit more complex. There are several steps involved.

First we allocate an array of pointers, all initially NULL. The size of the array is $L = 1 + \lfloor \log(n) \rfloor$, where n is the number of nodes in the heap before the delete. (The array is indexed by integers in $[0, L-1]$.) Then we scan through all the trees in A, one at a time, and insert them into this array. We maintain the property that position k in the array will either be NULL or it will point to a binomial tree of rank k. Here's how we do this:

If we are processing a binomial tree of rank k, and the kth position of the array already has no binomial tree in it, then we just point that array element to the tree. If the kth position has a binomial tree in it, we link these two trees together and it becomes a binomial tree of rank $k+1$. We try to put that into position $k+1$ of the array, etc.

This array is sufficiently large, because if we tried to use position L (one after the last position) we would be trying to put a binomial tree of rank L into that position. A binomial tree of rank L has at least $2^{1+\lfloor \log(n) \rfloor}$ nodes in it. This is greater than n, which is impossible.

Call this the compression step.

Now, we have this collection of trees stored in the array. We scan the roots of these trees and find the minimum key. This is the one we have to delete. We simply delete it. Now all we do is take all the trees (from the array and the children of the deleted node) and put them all together into a single binomial heap.

How long does this take? First note that the time to allocate and initialize the array is $O(\log n)$. What about the cost of inserting the trees into the array? For this we do an amortized analysis. We'll maintain two tokens on the root of each binomial tree. And during the compression step, we'll maintain the invariant that each tree stored in the array has one token on it. When we insert a tree into the array, if the cell is empty, we use one of the tokens to do the work, and leave the other one on the tree. If we're inserting it into a cell in the array that already has a tree in it, we use one token to link the two trees together, and we have two left to put on the resulting tree. This tree now has two tokens on it, which is sufficient to pay for its insertion into the array. After we're done, we have to restore two tokens to the root of each tree. At most $1 + \lfloor \log(n) \rfloor$ tokens are required to do this. So the whole cost of the deletemin is $O(\log n)$ amortized time.

To be rigorous, we have to apply the amortized analysis to all the operations, not just deletemin. But it's easy to see that meld doesn't need any extra tokens, and insert only needs two. (Also note that the initial potential (number of tokens) is zero and the final number is non-negative.)

By the way, this can be called the "lazy" way to implement binomial heaps – it defers as long as possible doing comparisons, to the time when it has to do a deletemin. It can be done "eagerly" instead, in which case the set of binomial trees at top level are all different sizes. The running time bounds for all operations are the same (amortized). And you can get by without using an array in the deletemin.

3 Fibonacci Heaps

We want to incorporate the decreasekey operation into the above data structure. How can we do this?

One idea to try is simply this. To decrease the key of a node, we simply change its key value. If its new key is less than that of its parent, we swap the node with its parent. We repeat this till we get to the top of the binomial tree, and stop.

This is a correct algorithm. Its running time is $O(\log n)$. But we're striving for $O(1)$ running time.

Another idea: Simply take the node you want to decrease, and change its key, and disconnect it and its entire subtree from where it is, and attach it to the tree root list.

This also gives a correct algorithm. (It gives a valid structure and will give the right answers.) This is clearly $O(1)$ time, so what goes wrong? The problem is that we can no longer prove the bound on the time of deletemin. We're depending on the fact that (after the compression step) the number of trees is $O(\log(\text{number of nodes}))$. This is no longer the case. For example, if we deleted a whole bunch of nodes distance 2 from the root of a tree, we could make a tree of rank k that had only $k+1$ keys in it.

There's a clever way to fix this devised by Fredman and Tarjan. This is the fibonacci heap algorithm.

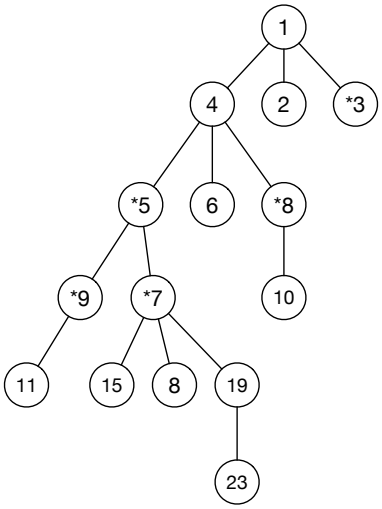
We add just one more bit of information to each node of the data structure. Call this the mark bit. The mark bits of all root nodes are always 0.

To implement decreasekey, we'll need what we call a cut. cut(i) takes a pointer to a node i in the tree. It works as follows:

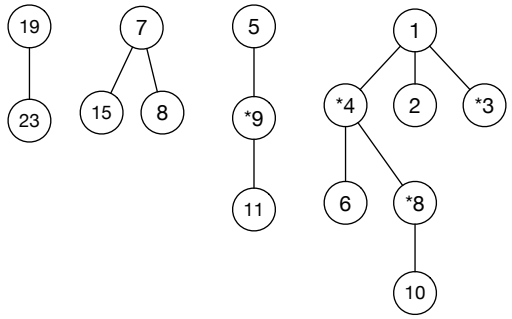
cut(i) If i is a tree root, do nothing. Let p be the parent of i. Remove the tree rooted at i from its parent p, and set it aside to later be merged in at top level. If p is not marked, then mark p (unless its a root) and return. If p is marked, then cut(p). This is recursive.

So the way we do decreasekey is simply to cut the node we went to decrease, change its key, then sew all the trees freed up by the cut together at top level.

Here's an example. Say we have one tree that looks like this. Marked nodes are indicated by *.



Say we do a cut(19). Here is the set of trees that results:

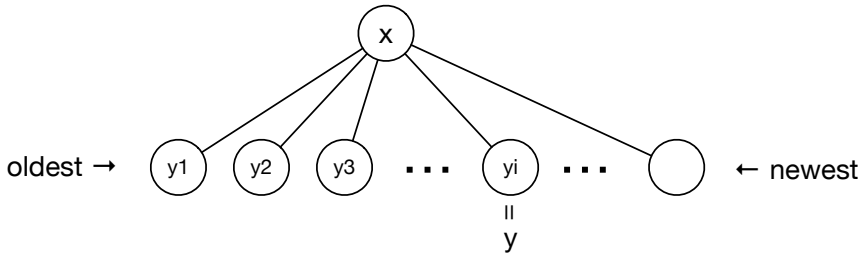


There are two things we need to do now. We need to prove that the amortized cost of cut(i) is $O(1)$, and we need to prove that deletemin is still $O(\log n)$. Let's start with the latter.

We're going to show that a tree of rank r has lots of nodes in it. Specifically, we'll show that $r \leq \log_\phi(n)$. That is, the rank of a tree with n nodes is at most the log base ϕ of n. Here ϕ is the golden ratio, or 1.61...

Lemma 1 *For any node in a tree, sort its children in the order in which they were linked to it, from oldest to newest. The ith child has rank at least $i - 2$.*

Proof:



Let the ith oldest child be called y. We want to show that the rank of y is at least $i - 2$.

When y and x were linked, y was given at least $i - 1$ siblings. Therefore the rank of x was at least $i - 1$. The ranks of x and y were equal when they were linked. So we know the rank of y was at least $i - 1$. Since that time, only one of y's children was removed. (If more than one, then y would have been cut from x.) Therefore the rank of y is at least $i - 2$. ■

What does this tell us about the relationship between the size of a tree and the rank? Let $s(r)$ be the minimum number of nodes in a tree of rank r. We can write a recurrence for $s()$:

$$s(0) = 1 \tag{1}$$

$$s(r) \geq 1 + 1 + \sum_{2 \leq i \leq r} s(i-2) \tag{2}$$

The first 1 is for the root, and the second 1 is for the first child. Here's a small table of these

bounds:

$$
\begin{aligned}
s(1) &\geq 2 + 0 = 2 \\
s(2) &\geq 2 + s(0) = 3 \\
s(3) &\geq 2 + s(0) + s(1) \geq 5 \\
s(4) &\geq 2 + s(1) + s(1) + s(2) \geq 8
\end{aligned}
$$

Lemma 2 $s(r) \geq \phi^r$

Proof: First define a recurrence $t()$, that is the same as the one for $s()$ but we replace all $\geq$ by $=$ signs. Then it's easy to see that $s(r) \geq t(r)$. We now can prove that

$$t(r) \geq \phi^r.$$

This will be by induction. Clearly it's true for $r = 0$. Assume it's true for all smaller values, and prove it for r.

$$t(r) = 2 + \sum_{2 \leq i \leq r} t(i-2)$$

The right hand side is:

$$
\begin{aligned}
&\geq 2 + \sum_{2 \leq i \leq r} \phi^{i-2} \\
&= 2 + \phi^0 + \phi^1 + \phi^2 + \cdots + \phi^{r-2} \\
&= 2 + \frac{\phi^{r-1} - \phi^0}{\phi - 1}
\end{aligned}
$$

But $\phi - 1 = 1/\phi$ and $\phi^0 = 1$ so we get

$$
\begin{aligned}
&= 2 + \phi^r - \phi \\
&= \phi^r + (2 - \phi) > \phi^r
\end{aligned}
$$

■

Now let's revisit the analysis for deletemin. Everything still goes through as above, except if there are n nodes, then the rank could be as large as $\log_\phi(n)$ instead of $\log_2(n)$. This does not change the big-oh bound.

It remains to analyze the cost of decreasekey.

Here again we need to do an amortized analysis. To do this we'll maintain three tokens on each marked node. (We continue to maintain two token on each root node.) We'll allocate 6 tokens to do the decreasekey. Consider the first cutting step. We use one token to do the work, and we use two to put on the cut node that is now at top level. If the parent is not marked, we put the last three there. If the parent is marked, we take the three from that (we now have 6) and we are back where we started: about to do a cut and having 6 tokens to work with. This guarantees that we won't run out of tokens.

Thus, we've proven this theorem:

Theorem 3 *Fibonacci heaps use $O(1)$ amortized time for makeheap, insert, meld, and decreasekey. They use $O(\log n)$ amortized time for deletemin.*