4 Heaps

Heaps are a family of data structures for dynamic selection of a minimum. They maintain a set of items, each equipped with a *priority.* The priorities are usually numeric, but we will assume that they come from some abstract universe and they can be only compared. Beside the priority, the items can carry arbitrary data, usually called the *value* of the item.

Heaps typically support the following operations. Since a heap cannot efficiently find an item by neither its priority nor value, operations that modify items are given an opaque identifier of the item. The identifiers are assigned on insertion; internally, they are usually pointers.

INSERT(p, v)	Create a new item with priority p and value v and return its identifier.
MIN	Find an item with minimum priority. If there are multiple such items, pick any. If the heap is empty, return $\emptyset$.
EXTRACTMIN	Find an item with minimum priority and remove it from the heap. If the heap is empty, return $\emptyset$.
DECREASE(id, p)	Decrease priority of the item identified by id to p.
INCREASE(id, p)	Increase priority of the item identified by id to p.
DELETE(id)	Remove the given item from the heap. It is usually simulated by DECREASE$(id, -\infty)$ followed by EXTRACTMIN.
BUILD$((p_1, v_1), \ldots)$	Create a new heap containing the given items. It is equivalent to a sequence of INSERTS on an empty heap, but it can be often implemented more efficiently.

By reversing the order of priorities, we can obtain a *maximal heap,* which maintains the maximum instead of the minimum.

Observation: We can sort a sequence of n items by inserting them to a heap and then calling EXTRACTMIN n times. The standard lower bound on comparison-based sorting implies that at least one of INSERT and EXTRACTMIN must take $\Omega(\log n)$ time amortized.

We can implement the heap interface using a search tree, which yields $\mathcal{O}(\log n)$ time complexity of all operations except BUILD. Specialized constructions of heaps presented in this chapter will achieve $\mathcal{O}(\log n)$ amortized time for EXTRACTMIN, $\mathcal{O}(n)$ for BUILD, and $\mathcal{O}(1)$ for all other operations.

Dijkstra's algorithm

Let us see one example where a heap outperforms search trees: the famous Dijkstra's algorithm for finding the shortest path in a graph.

TODO

Lemma: Dijkstra's algorithm with a heap runs in time $\mathcal{O}(n\cdot T_I(n)+n\cdot T_X(n)+m\cdot T_D(n))$. Here n and m are the number of vertices and edges of the graph, respectively. $T_I(n)$ is the amortized time complexity of INSERT on a heap with at most n items, and similarly $T_X(n)$ for EXTRACTMIN and $T_D(n)$ for DECREASE.

4.1 Regular heaps

TODO

4.2 Binomial heaps

The binomial heap performs similarly to the regular heaps, but is has a more flexible structure, which will serve us well in later constructions. It supports all the usual heap operations. It is also able to MERGE two heaps into one efficiently.

The binomial heap will be defined as a collection of binomial trees, so let us introduce these first.

Definition: The *binomial tree of rank k* is a rooted tree B_k with ordered children of each node such that:

1. B_0 contains only a root node.
2. B_k for $k > 0$ contains a root with k children, which are the roots of subtrees B_0, B_1, ..., B_{k-1} in this order.⟨1⟩

If we wanted to be strictly formal, we would define B_k as any member of an isomorphism class of trees satisfying these properties.

Let us mention several important properties of binomial trees, which can be easily proved by induction on the rank. It might be useful to follow figures 4.1 and 4.2.

Observation:

⟨1⟩ In fact, this works even for $k = 0$.

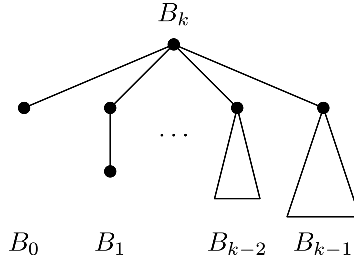

Figure 4.1: The binomial tree of rank k

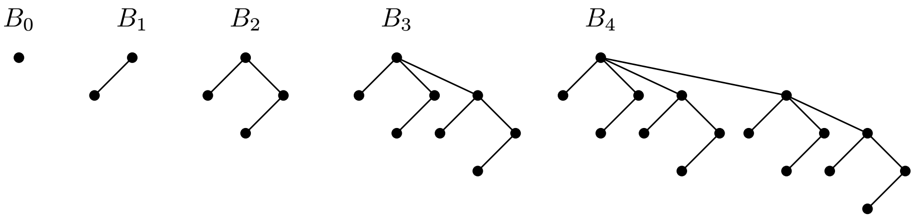

Figure 4.2: Binomial trees of small rank

- The binomial tree B_k has 2^k nodes on $k + 1$ levels.
- B_k can be obtained by linking the roots of two copies of B_{k-1} (see figure 4.3).

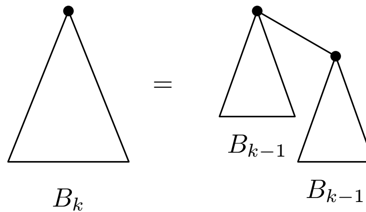

Figure 4.3: Joining binomial trees of the same rank

Definition: The *binomial heap* for a given set of items is a sequence $\mathcal{T} = T_1, \ldots, T_\ell$ of binomial trees such that:

1. The ranks of the trees are increasing: $rank(T_i) < rank(T_{i+1})$ for all i. In particular, this means that the ranks are distinct.
2. Each node of a tree contains a single item. We will use $p(v)$ for the priority of the item stored in a node v.
3. The items obey the *heap order* — if a node u is a parent of v, then $p(u) \le p(v)$.

Note: To ensure that all operations have the required time complexity, we need to be careful with the representation of the heap in memory. For each node, we will store:

- the rank (a rank of a non-root node is defined as the number of its children)
- the priority and other data of the item (we assume that the data can be copied in constant time; if they are large, we keep them separately and store only a pointer to the data in the node)
- a pointer to the first child
- a pointer to the next sibling (the children of a node form a single-linked list)
- a pointer to the parent (needed only for some operations, see exercise 4)

Since the next sibling pointer plays no role in tree roots, we can use it to chain trees in the heap. In fact, it is often helpful to encode tree roots as children of a "meta-root", which helps to unify all operations with lists.

As the trees in the heap have distinct ranks, their sizes are distinct powers of two, whose sum is the total number of items n. This is exactly the binary representation of the number n — the k-th digit (starting with 0) is 1 iff the tree B_k is present in the heap. Since each n has a unique binary representation, the shape of the heap is fully determined by its size. Still, we have a lot of freedom in location of items in the heap.

Corollary: A binomial heap with n items contains $\mathcal{O}(\log n)$ trees, whose ranks and heights are $\mathcal{O}(\log n)$. Each node has $\mathcal{O}(\log n)$ children.

Finding the minimum

Since the trees are heap-ordered, the minimum item of each tree is located in its root. To find the minimum of the whole heap, we have to examine the roots of all trees. This can be done in time linear in the number of trees, that is $\mathcal{O}(\log n)$.

If the MIN operation is called frequently, we can speed it up to $\mathcal{O}(1)$ by caching a pointer to the current minimum. The cache can be updated during all other operations at the cost of a constant slow-down. We are leaving this modification as an exercise.

Merging two heaps

All other operations are built on Merge. This operation takes two heaps H_1 and H_2 and it constructs a new heap H containing all items of H_1 and H_2. The original heaps will be destroyed.

In Merge, we scan the lists of trees in each heap in the order of increasing ranks, as when merging two sorted lists. If a given rank is present in only one of the lists, we move that tree to the output. If we have the same rank in both lists, we cannot use both trees, since it would violate the requirements that all ranks are distinct. In this case, we *join* the trees to form B_{k+1} as in figure 4.3: the tree whose root has the smaller priority will stay as the root of the new tree, the other root will become its child. Of course, it can happen that rank $k+1$ is already present in one or more lists, but this can be solved by further merging.

The whole process is similar to addition of binary numbers. We are processing the trees in order of increasing rank k. In each step, we have at most one tree of rank k in each heap, and at most one tree of rank k carried over from the previous step. If we have two or more trees of rank k, we join a pair of them and send it as a carry to the next step. This leaves us with at most one tree, which we can send to the output.

As a minor improvement, when we exhaust one of the input lists and we have no carry, we can link the rest of the other list to the output in constant time.

Let us analyze the time complexity. The maximum rank in both lists is $\mathcal{O}(\log n)$, where n is the total number of items in both heaps. For each rank, we spend $\mathcal{O}(1)$ time. If we end up with a carry after both lists are exhausted, we need one more step to process it. This means that Merge always finishes in $\mathcal{O}(\log n)$ time.

Inserting items

Insert is easy. We create a new heap with a single binomial tree of rank 0, whose root contains the new item. Then we merge this heap to the current heap. This obviously works in time $\mathcal{O}(\log n)$.

Building a heap of n given items is done by repeating Insert. We are going to show that a single Insert takes constant amortized time, so the whole Build runs in $\mathcal{O}(n)$ time. We observe that the Merge inside the Insert behaves as a binary increment and it runs in time linear in the number of bits changed during that increment. (Here it is crucial that we stop merging when one of the lists is exhausted.) Thus we can apply the amortized analysis of binary counters we developed in section **??**.

Extracting the minimum

The ExtractMin operation will be again based on Merge. We start by locating the minimum as in Min. We find the minimum in the root of one of the trees. We unlink this

tree from the heap. Then we remove its root, so the tree falls apart to binomial trees of all lower ranks (remember figure 4.1). As the ranks of these trees are strictly increasing, the trees form a correct binomial heap, which can be merged back to the current heap.

Finding the minimum takes $\mathcal{O}(\log n)$. Disassembling the tree at the root and collecting the sub-trees in a new heap takes $\mathcal{O}(\log n)$; actually, it can be done in constant time if we are using the representation with the meta-root. Merging these trees back is again $\mathcal{O}(\log n)$. We conclude that the whole ExtractMin runs in time $\mathcal{O}(\log n)$.

Decreasing and increasing

A Decrease can be performed as in a regular heap. We modify the priority of the node, which can break heap order at the edge to the parent. If this happens, we swap the two items, which can break the order one level higher, and so on. In the worst case, the item bubbles up all the way to the root, which takes $\mathcal{O}(\log n)$ time.

An Increase can be done by bubbling items down, but this is slow. As each node can have logarithmically many children, we spend $\mathcal{O}(\log n)$ time per step, which is $\mathcal{O}(\log^2 n)$ total. It is faster to Delete the item by decreasing it to $-\infty$ and performing an ExtractMin, and then inserting it back with the new priority. This is $\mathcal{O}(\log n)$.

Summary of operations

operation	*worst-case complexity*
Insert	$\Theta(\log n)$
Min (cached)	$\Theta(1)$
ExtractMin	$\Theta(\log n)$
Merge	$\Theta(\log n)$
Build	$\Theta(n)$
Decrease	$\Theta(\log n)$
Increase	$\Theta(\log n)$
Delete	$\Theta(\log n)$

We proved only the asymptotic upper bounds, but it is trivial to verify that these are tight on some inputs.

Compared with the binary heap, the binomial heap works in the same time, but it additionally supports efficient merging.

Exercises

1. The name of binomial trees comes from the fact that the number of nodes on the i-th level (starting with 0) in a rank-k tree is equal to the binomial coefficient $\binom{k}{i}$. Prove it.

2. How many leaves does the tree B_k have?

3. The binomial tree B_k is a spanning tree of the k-dimensional hypercube. And indeed a special one: show that it is a shortest path tree of the hypercube.

4. Show that if we need only Insert and ExtractMin, we can omit the parent pointers. Which other operations need them?

5. Show that we can cache the minimum with only a constant slowdown of all operations.

6.* Define a *trinomial heap,* whose trees have sizes equal to the powers of 3. In a single heap, there will be at most two trees of any given rank. Show how to implement heap operations. Compare their complexity with the binomial heap.

4.3 Lazy binomial heaps

The previous construction is sometimes called the *strict binomial heap,* as opposed to the *lazy* version, which will be introduced in this section.

The lazy binomial heap is slow in the worst case, but fast in the amortized sense. It is obtained by dropping the requirement that the ranks of trees in a heap are sorted and distinct. This can lead to a very degenerate heap, which is just a linked list of single-node trees. Repeated ExtractMins on such heaps would be too slow, so we make each extraction *consolidate* the heap — bring it to the strict form with distinct ranks. The consolidation is expensive, but an amortization argument will show that all consolidations can be pre-paid by the other operations.

The representation of the heap in memory will be the same, except that we will need the list of children of each node to be doubly linked. In fact, it is easier to use a circular list of children, in which every node points to its predecessor and successor. This way, the pointer to the first child can be used to find the last child in constant time, too.

A Merge of two lazy heaps is trivial: we just concatenate their lists of trees. With circular lists, this can be done in constant time. Therefore both Merge and Insert run in $\mathcal{O}(1)$ time. A build makes n Inserts, so it runs in $\mathcal{O}(n)$ time.

Decrease does not need to be modified. It is local to the tree containing the particular item. Even in a lazy heap, the rank of this tree and its height are still $\mathcal{O}(\log n)$, and so is the time complexity of Decrease. Both Delete and Increase will be again reduced to Decrease, ExtractMin, and Insert.

ExtractMin and consolidation

An ExtractMin is implemented similarly to the strict heap. We find the tree whose root has minimum priority. This requires $\mathcal{O}(t)$ time, where t is the number of trees in the heap. Then we remove the tree from the heap, disassemble it to subtrees and merge these trees back to the heap. This can be done in constant time with circular lists.

Then we consolidate the heap by bucket sorting. We create an array of $\lfloor \log n \rfloor + 1$ buckets, one for each possible rank. We process the trees one by one. If we have a tree of rank r, we look in the r-th bucket. If the bucket is empty, we put the tree there. Otherwise, we remove the tree from the bucket and join it with our tree. This increases its rank r by one. We repeat this until we find an empty bucket. Then we proceed to the next tree of the heap. At the end, we collect all trees from the buckets and put them back to the heap.

How much time does the consolidation take? We spend $\mathcal{O}(\log n)$ time on initializing the buckets and collecting the trees from them at the end. We add t trees and since each join decreases the number of trees by 1, we spend $\mathcal{O}(t)$ time on the joins. Therefore the whole consolidation runs in $\mathcal{O}(t + \log n)$ time.

We conclude that ExtractMin runs in $\mathcal{O}(t + \log n)$ time, which is $\mathcal{O}(n)$ in the worst case.

Amortized analysis

Let us show that while ExtractMin is very expensive in the worst case, it actually amortizes well.

We define a potential Φ equal to the total number of trees in all heaps. The potential is initially zero and always non-negative.

Merge has constant real cost and since it only redistributes trees between heaps, it does not change the potential. Hence its amortized cost is also $\mathcal{O}(1)$. An Insert creates a new node, which has constant real cost and it increases the potential by 1. Together with the merge, its amortized cost is $\mathcal{O}(1)$.

Decrease does not change the potential, so its amortized cost is equal to the real cost $\mathcal{O}(\log n)$.

The real cost of ExtractMin on a heap with t trees is $\mathcal{O}(t + \log n)$. At the end, all trees have distinct ranks, so the potential drops from t to at most $\log n$. The potential difference is therefore at most $\log n - t$, so after suitable scaling it offsets the $\mathcal{O}(t)$ term of the real cost. This makes the amortized cost $\mathcal{O}(\log n)$.

Build, Increase, and Delete are composed of the other operations, so we can just add the respective amortized costs.

Summary of operations

operation	*worst-case complexity*	*amortized complexity*
INSERT	$\Theta(1)$	$\Theta(1)$
MIN (cached)	$\Theta(1)$	$\Theta(1)$
EXTRACTMIN	$\Theta(n)$	$\Theta(\log n)$
MERGE	$\Theta(1)$	$\Theta(1)$
BUILD	$\Theta(n)$	$\Theta(n)$
DECREASE	$\Theta(\log n)$	$\Theta(\log n)$
INCREASE	$\Theta(n)$	$\Theta(\log n)$
DELETE	$\Theta(n)$	$\Theta(\log n)$

We proved only the asymptotic upper bounds, but it is trivial to verify that these are tight on some inputs.

Compared with its strict counterpart, the lazy binomial heap achives better amortized complexity of INSERT and MERGE. Still, DECREASE is too slow to improve Dijkstra's algorithm.

Exercises

1. *Consolidation by pairing:* Consider a simplified procedure for consolidating a heap. When it joins two trees of the same rank, it immediately sends them to the resulting heap instead of trying to put them in the next higher bucket. This consolidation no longer guarantees that the resulting trees have distinct ranks, but prove that the amortized cost relative to the potential Φ is not harmed.

4.4 Fibonacci heaps

To make DECREASE faster, we need to implement it in a radically different way. If decreasing a node's priority breaks heap order on the edge to the parent, we cut this edge and make the decreased node a root of its own tree. Repeated cuts can distort the shape of the trees too much, so we avoid cutting off more than one child of a non-root node. If we want to cut off a second child, we cascade the cut to make the parent a new root, too.

This idea leads to the Fibonacci heaps, discovered by Fredman and Tarjan in 1987. We will develop them from the lazy binomial heaps, but we drop all requirements on the shape of the trees.

Definition: The *Fibonacci heap* for a given set of items is a sequence $\mathcal{T} = T_1, \ldots, T_\ell$ of rooted trees such that:

1. Each node contains a single item. We will use $p(v)$ for the priority of the item stored in a node v.
2. The items obey the *heap order* — if a node u is a parent of v, then $p(u) \leq p(v)$.
3. Each node can be *marked* if it lost a child. Tree roots are never marked.

Note: The representation of each node in memory will contain:

- the rank, defined as the number of children (the rank of a tree is defined as the rank of its root)
- the priority and other data of the item
- the boolean mark
- a pointer to the first child
- a pointer to the previous and next sibling in a circular list of children
- a pointer to the parent

Operations

MERGE works as in lazy binomial heaps: It just concatenates the lists of trees.

INSERT creates a heap consisting of one unmarked node and merges it to the current heap. BUILD is an iterated INSERT.

EXTRACTMIN is again similar to the lazy binomial heap. The only difference is that when it disassembles a tree, it has to unmark all new roots.

DECREASE changes the priority of the node. If it becomes lower than the priority of its parent, we CUT the node to make it a root.

CUT disconnects a node from the list of siblings, unmarks it, and connects it to the list of roots. If the original parent is not a root, it sets its mark. If the mark was already set, it calls CUT on the parent recursively.

As usual, INCREASE and DELETE are reduced to the other operations.

Worst-case analysis

Worst-case complexity of operations easily follows from their description. The only exception is the consolidation inside EXTRACTMIN, which needs to create an array of buckets indexed by ranks. In binomial heaps, the maximum rank was easily shown to be at most $\log n$. The trees in Fibonacci heaps can have less regular shapes, but is is also possible to prove that their ranks are at most logarithmic.

The analysis involves Fibonacci numbers, from which the heap got its name. We define the Fibonacci sequence by $F_0 = 0$, $F_1 = 1$, and $F_{n+2} = F_{n+1} + F_n$. It is well known that

$F_n = \Theta(\sigma^n)$, where $\sigma = (1+\sqrt{5})/2 \doteq 1.618$ is the golden ratio. The following lemma will be useful:

Lemma: $F_0 + F_1 + \ldots + F_k = F_{k+2} - 1$.

Proof: By induction on k. For $k = 0$, we have $F_0 = F_2 - 1$, that is $0 = 1 - 1$. In the induction step, we increase both sides of the equality by F_{k+1}. The right-hand side becomes $F_{k+2} - 1 + F_{k+1} = F_{k+3} - 1$ as needed. □

Now we use Fibonacci numbers to show that the sizes of trees grow exponentially with their rank.

Lemma: A node of rank k has at least F_{k+2} descendants (including itself).

Proof: We will proceed by induction on the height of the node. If the height is 0, the node is a leaf, so it has rank 0 and $F_2 = 1$ descendants.

Now, let u be a node of rank $k > 0$. Let $x_1, \ldots, x_k$ be its children in the order of their linking to u, and $r_1, \ldots, r_k$ their ranks. In a binomial heap, the ranks would be $0, \ldots, k-1$. Here, we are able to prove a weaker, but still useful claim:

> **Claim:** For every $i \geq 2$, we have $r_i \geq i - 2$.
>
> *Proof:* Consider the moment when x_i was linked to u. At that time, $x_1, \ldots, x_{i-1}$ were already the children of u, but there might have been more children, which were removed later. So the rank of u was at least $i - 1$. Nodes are linked only when we join two trees of the same rank, so the rank of x_i must have been also at least $i-1$. As x_i is not a root, it might have lost at most one child since that time, so its current rank is at least $i - 2$. □

The ranks $r_1, \ldots, r_k$ are therefore at least $0, 0, 1, 2, \ldots, k - 2$. As the height of children is smaller that the height of the parent, we can apply the induction hypothesis, so the sizes of subtrees are at least $F_2, F_2, F_3, F_4, \ldots, F_k$. As $F_0 = 0$ and $F_1 = 1 = F_2$, the total number of descendants of u, including u itself, can be written as $F_0 + F_1 + F_2 + \ldots + F_k + 1$. By the previous lemma, this is equal to F_{k+2}. This completes the induction step. □

Corollary R: The ranks of all nodes are $\mathcal{O}(\log n)$.

Let us return back to the complexity of operations. Merge and Insert run in $\mathcal{O}(1)$ time. Build does n Inserts, so it runs in $\mathcal{O}(n)$ time.

ExtractMin on a heap with t trees uses $\mathcal{O}(\log n)$ buckets (because ranks are logarithmic by corrolary **R**), so it takes $\mathcal{O}(t + \log n)$ time.

Decrease does a Cut, which runs in time linear in the number of marked nodes it encounters. This can be up to $\mathcal{O}(n)$.

Amortized analysis

There are two procedures which can take $\mathcal{O}(n)$ time in the worst case: extraction of the minimum (which is linear in the number of trees) and cascaded cut (linear in the number of marked nodes it encounters). To amortize them, we define a potential, which combines trees with marked nodes:

$$\Phi := \textit{number of trees in all heaps} + 2 \cdot \textit{number of marked nodes}.$$

The potential is initially zero and always non-negative.

Merge has constant real cost and it does not change the potential. Hence the amortized cost is also constant.

Insert has constant real cost and it increases Φ by 1, so the amortized cost is constant.

ExtractMin on a heap with t trees runs in real time $\mathcal{O}(t + \log n)$. The final number of trees is $\mathcal{O}(\log n)$ and marks are only removed, so the potential difference is $\mathcal{O}(\log n) - t$. With suitable scaling of units, this makes the amortized cost $\mathcal{O}(\log n)$.

Cut of a node whose parent is either unmarked or a root takes $\mathcal{O}(1)$ real time and changes the potential by $\mathcal{O}(1)$. If the parent is marked, we cascade the cut. This involves unmarking the parent, which decreases Φ by 2 units. Then we make the parent a root of a stand-alone tree, which increases Φ by 1. The total change of Φ is -1, which pays for the constant real cost of the cut. The amortized cost of the cascading is therefore zero, so the total amortized cost of Cut is $\mathcal{O}(1)$.

Summary of operations

operation	*worst-case complexity*	*amortized complexity*
Insert	$\Theta(1)$	$\Theta(1)$
Min (cached)	$\Theta(1)$	$\Theta(1)$
ExtractMin	$\Theta(n)$	$\Theta(\log n)$
Merge	$\Theta(1)$	$\Theta(1)$
Build	$\Theta(n)$	$\Theta(n)$
Decrease	$\Theta(n)$	$\Theta(1)$
Increase	$\Theta(n)$	$\Theta(\log n)$
Delete	$\Theta(n)$	$\Theta(\log n)$

As usual, we proved only the asymptotic upper bounds, but it is easy to verify that that are tight on some inputs (see exercise 2).

Dijkstra's algorithm with Fibonacci heap therefore runs in $\mathcal{O}(m + n \log n)$ time. For dense graphs ($m \approx n^2$), this is $\mathcal{O}(n^2)$; for sparse graphs ($m \approx n$), we get $\mathcal{O}(n \log n)$. We note

that $\Omega(n \log n)$ is not easy to surpass, since Dijkstra's algorithm sorts vertices by their distance. There exist more efficient algorithms, but they always make further assumptions on the edge lengths — e.g., they assume that the lengths are small integers.

Exercises

1. Caching of minimum works in the Fibonacci heaps, too. Show how to do it and prove that the amortized cost of operations does not change.

2. Show that there is a sequence of $\mathcal{O}(n)$ operations on a Fibonacci heaps, which produces a heap, whose only tree is a path with all n nodes marked except for the root. This gives a lower bound for worst-case complexity of Decrease. It also implies that the trees of Fibonacci heap are not necessarily subtrees of the binomial trees of the same rank.

3. Is it possible to implement an Increase operation running in better than logarithmic amortized time?

4. For each k, construct a sequence operations on the Fibonacci heap, which produces a tree with exactly F_{k+2} nodes.

5.* Consider a Fibonacci-like heap which does not mark nodes. Show that there exists a sequence of operations, which produces a tree with n nodes and rank $r \approx \sqrt{n}$, where the children of the root have ranks $0, \ldots, r-1$. Then we can force this tree to disassemble and re-assemble again, which costs $\Theta(\sqrt{n})$ repeatedly.

6. Show that using a different potential, the amortized cost of ExtractMin can be lowered to $\mathcal{O}(1)$ at the expense of raising the amortized cost of Insert to $\mathcal{O}(\log n)$. Beware that the n here is the current number of items as opposed to the maximum number of items over the whole sequence of operations.