Chapter C4: Heaps and Fibonacci Heaps

C4.1 Heaps

A heap is the standard implementation of a priority queue. A min-heap stores values in nodes of a tree such that

**heap-order property:** for each node, its value is smaller than or equal to its children’s

So the minimum is on top. (A max-heap can be defined similarly.)

The standard heap uses a binary tree. Specifically, it uses a complete binary tree, which we define here to be a binary tree where each level except the last is full, and in the last level nodes are added left to right. Here is an example:

```
7
/   \\   \\
24 19  \\
/   /   \\
25 66 68 40
/   /   \\
29 31 58  \\
```

Now, any binary tree can be stored in an array using level numbering. The root is in cell 0, cells 1 and cells 2 are for its children, cells 3 through 6 are for its grandchildren, and so on. Note that this means a nice formula: if a node is in cell \(x\), then its children are in \(2x+1\) and \(2x+2\). Such storage can be (very) wasteful for a general binary tree; but the definition of complete binary tree means the standard heap is stored in consecutive cells of the array.

C4.2 Heap Operations

The idea for insertion is to: *Add as last leaf, then bubble up value until heap-order property re-established.*

```
Insert(v)
add v as next leaf
while v<parent(v) {
    swapElements(v,parent(v))
    v=parent(v)
}
```
One uses a “hole” to reduce data movement. Here is an example of inserting the value 12:

![Diagram showing insertion of value 12 into a binary heap]

The idea for `extractMin` is to: Replace with value from last leaf, delete last leaf, and bubble down value until heap-order property re-established.

```
ExtractMin()
    temp = value of root
    swap root value with last leaf
    delete last leaf
    v = root
    while v > any child(v) {
        swapElements(v, smaller child(v))
        v = smaller child(v)
    }
    return temp
```

Here is an example of `extractMin`:

![Diagram showing `extractMin` operation]

Variations of heaps include

- $d$-heaps; each node has $d$ children
- support of merge and other operations: leftist heaps, skew heaps, Fibonacci heaps

### C4.3 Fibonacci Heaps

Fibonacci heaps provide an interesting extension of heaps. They still retain the heap-property: each parent is smaller than its children (we’re assuming min-heap). But they
do away with the restriction to using a binary tree and also allow more than one root. Specifically, we will maintain:

\[ \text{a collection of heap-ordered trees.} \]

Again let us start by just supporting insert and extractMin. We get amortized constant time for insert and amortized \(O(\log n)\) time for delete-min (cannot do better).

Allowing a collection of trees, means that insertion can be made trivial:

<table>
<thead>
<tr>
<th>Insert</th>
</tr>
</thead>
<tbody>
<tr>
<td>create a new heap</td>
</tr>
</tbody>
</table>

Of course, this opting for lazy insertion adds work to extractMin. So, during that operation we will do extra computation to tidy up the heap.

In particular, the idea chosen by Fibonacci heaps ensures that each execution of extractMin runs in time proportional to \(d + \log n\), where \(d\) is the number of insert's since the previous extractMin. In essence, one can think of the lazy insertion operation putting aside a dollar for future computation.

Enough ideating. Here is the implementation. The tidy-up phase of the extractMin operation achieves the following situation: \textit{there is at most one tree of each given root-degree.}

<table>
<thead>
<tr>
<th>ExtractMin</th>
</tr>
</thead>
<tbody>
<tr>
<td>calculate minimum root and store its value</td>
</tr>
<tr>
<td>add children of it as trees to the collection</td>
</tr>
<tr>
<td>\textbf{while} there are two trees of same root-degree \textbf{do}</td>
</tr>
<tr>
<td>merge the two trees with larger-value root becoming child of smaller-value root</td>
</tr>
<tr>
<td>return stored value</td>
</tr>
</tbody>
</table>

A simple way to implement the loop is to use an array with slots for a tree of each root-degree. By iterating through the roots, the loop takes time proportional to the initial number of roots, which is at most \(d + x\), where as above \(d\) is the number of recent insert’s, and \(x\) is the maximum root-degree at the beginning.

Now, all trees with root-degree zero are just isolated nodes. Also, all trees with root-degree \(d\) are formed by combining two trees with root-degree \(d - 1\). It follows (by
induction) that every tree with root-degree $d$ has precisely $2^d$ nodes. In particular, the highest root-degree is at most $\log n$, and indeed all nodes have at most $\log n$ children.

At this point, the notions of Fibonacci heap and *binomial heap* essentially coincide: the Fibonacci numbers only show up when we introduce a fancy *decreaseKey*.

So the only thing that remains is some implementation details. It is customary to make use of doubly-linked circular linked lists (DCL). A DCL is used to store the roots. Further, each node has a children counter and a pointer to a DCL of its children. The DCL data structure is used because it enables constant-time removal of a node, addition of a node, and merging of two DCLs.

### C4.4 Delete and Decrease-Key

Now, any priority queue that supports *decreaseKey* can trivially be extended to handle *delete* as well: simply decrease the key to some value guaranteed to be the smallest, and then do *extractMin*. (This assumes the input to both operations is a pointer to the entry/node—if one needs to *find* the node as well, then the operation takes linear time anyway.)

So, it remains to support *decreaseKey*. In any heap-like structure, one implementation of this operation is: decrease the value, and then bubble the value up if smaller than parent, until it reaches the correct level. This can be achieved if one adds parent pointers to all the nodes. Since the trees all have depth at most $O(\log n)$, this means that we have *decreaseKey* in that time.

But the inventors of Fibonacci heaps provided a way to do *decreaseKey* in $O(1)$ amortized time, as we now show.

◊ *Potential Functions*

A common way to keep track of the arithmetic for amortized analysis is to define a *potential function*. This allowance is adjusted by every operation, both by contributions to future computation, and then by actual computation that reduces it.

Here it is in action in Fibonacci heaps. To get *decreaseKey* to run in amortized constant time, the first idea is that *decreaseKey* is implemented by detaching that node from its parent and making it into a new tree for the collection.

This sounds like we are done. But this upsets the earlier analysis of *extractMin*, because now trees of the same root-degree can have varying sizes. The solution to this is to take action if the same node loses too many children...

In the final version, a node is *marked* if it loses a child. The potential of a Fibonacci heap is defined to be

$$\Phi = t + 2m$$

12
where $t$ is the number of trees/roots in the collection and $m$ is the number of marked nodes. If a marked node loses a second child, then some rebalancing occurs.

The Fibonacci numbers, finally appear (boy we had to wait a long time), as now the analysis shows that a node with root degree-$d$ has size at least the $d$th Fibonacci number. This is approximately $\Phi^d$, where $\Phi = (1 + \sqrt{5})/2$ is the golden ratio.

One still has to go back and check that the claimed amortized running times of all previous operations remain valid, et cetera

**Exercises**

1. **Add Me**