Priority Queue

Let's briefly go back to thinking about queues. Our definition of a queue was that we remove items based on the amount of time they have been the queue, so that the removed element is the one that has been there the longest. A natural way to implement such a queue was using a linear structure, such as a linked list or a (circular) array.

A priority queue is a different kind of queue, in which the element that is removed is decided based on some criterion other than the time of entry into the queue. For example, in a hospital emergency room, patients arrive and are treated not in a first-come first-serve order, but rather the order is determined by the relative severity of the case (perhaps in combination with other factors, including how long the patient has been waiting). The priority is assumed to be something that is computable. The objects in a priority queue can be compared and their relative priorities determined. Let's say that high priority is low value – think: "my number one priority" vs. "my number 2 priority" etc refers to the order in which these items should be removed from the priority queue.

One way you might imagine implementing a priority queue is to maintain a sorted list of the elements in the queue. This could be done with a linked list or array. Each time an item is added, it would need to be inserted into the sorted list. An alternative would be to use a binary search tree such as we discussed last lecture. The item that is removed next is found by the `findMinimum()` operation i.e. follow the left child references until reaching a node with no left child. This is perhaps a better way to implement a priority queue than the linear method, since an insertion and deletion tend to occur relatively quickly. It can still take long though, since we have not guarenteed that the binary tree will be flat. We can still have long paths in the tree.

One can avoid these long paths with an alternative binary tree called a *heap*, which is commonly used for representing priority queues. Heaps have the advantage that they are balanced, namely the minimum and maximum depth of all leaves differs by at most 1. This guarentees that the binary tree is as flat as possible.

To define a heap, though, we need a few more definitions about binary trees.

Binary trees - definitions and properties

How many nodes can a binary tree have at each level? The root has one node. Level 1 has two nodes (the two children of the root). Level 2 has four nodes, namely each child at level 1 can have two children. You can easily see that the maximum number of nodes doubles at each level, level l can have 2^l nodes. For a binary tree of height h, the maximum number of nodes is thus:

$$n_{max} = \sum_{l=0}^{h} 2^l = \frac{2^{h+1} - 1}{2 - 1} = 2^{h+1} - 1.$$

The minimum number of nodes in a binary tree of height h is of course $h + 1$, namely each node has at most one child (the sole leaf has no children). It follows that

$$h + 1 \leq n \leq 2^{h+1} - 1$$

We can rearrange this equation to give

$$\lg(n + 1) - 1 \leq h \leq n - 1.$$

Heaps

We say a binary tree of height h is *complete* if every level l less than h has the maximum number (2^l) of nodes, and in level h all nodes are as far to the left as possible.

A *heap* is a complete binary tree, whose elements can be ordered (in Java, they are `Comparable`), such that the element stored at each node is less than the element stored at its children's node(s). This is the default definition of a heap, and is sometimes called a *min heap*. A *max heap* is defined similarly, except that the element stored at each node is greater than the elements stored at the children of that node. Unless otherwise specified, we will assume the definition of a min heap.

It follows from the definition that the smallest element in a heap is stored at the root.

Notice that we have not specified how to implement a heap. (There are multiple ways to implement a binary tree.) For now, we want to treat a heap as abstractly as we can.

As with stacks and queues, the two main operations we perform on heaps are `add` and `remove`. Let's have a look at how to perform these.

add

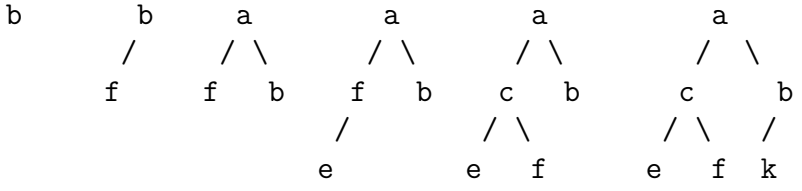
To add an item, we create a new node and insert it in the next available position of the (complete) tree. If level h is not full, then we insert it next to the rightmost element. If level h is full, then we start a new level at height $h + 1$.

Once we have inserted the new node, we need to be sure that the heap property is satisfied. The only problem could be that the parent of the node is greater than the node. This problem is easy to solve. We can just swap the elements of the node and its parent. We then need to repeat the same test on the new parent node, etc, until we reach either the root, or until the parent node is less than the current node.

You might ask whether swapping with the parent can cause a problem in the case that the new node is a right child, since there is already a left child, and swapping the new node with the parent might disturb the heap property with respect to the left child. Fortunately, this cannot happen. Before inserting the right child node, the parent was less than the left child (since we had a heap). So if the new right child node is less than the parent this implies that the new right child node must already be less than the left child node. So swapping the right child with the parent would preserve the heap property.

Example

Let's suppose the heap is inititally empty and then we add items `b,f,a,e,c,k` in that order.



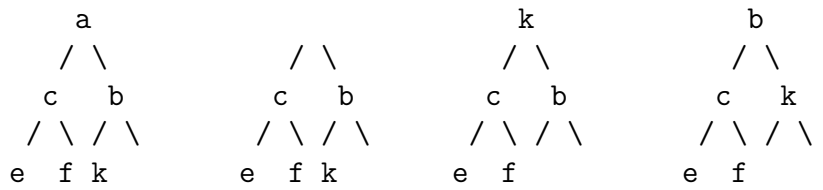
removeMin

Now let's remove elements from the priority queue. We always remove the minimum element, which is at the root.

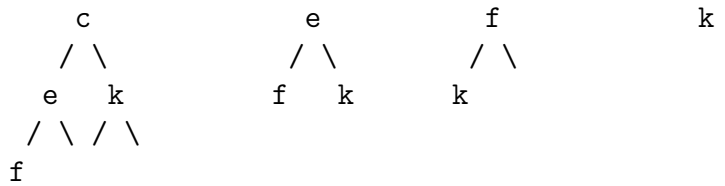
How do we fill the hole ? We first copy the last element in the heap (the rightmost element in level h) to the root, and delete this last element. We then need to manipulate the elements in the tree to preserve the heap property that each parent is less than its children.

Starting at the root (which contains the new element), we compare the root to its two children. If the element at the root is less than the elements at its two children, then we are done. Otherwise, the root element is greater than the elements of at least one of the children. In this case, we swap the element at the root with the child having the smaller element. Moving the smaller child to the root does not create a problem, since by definition the smaller child will be greater than the larger child.

In the example above, we now remove the minimum element (`k`).



And remove the remaining elements in turn:



Next lecture, we will look at how to implement heaps.