Priority Queue

Recall the definition of a queue. It is a collection where we remove the element that has been in the collection for the longest time. Alternatively stated, we remove the element that first entered the collection. A natural way to implement such a queue was using a linear data structure, such as a linked list or a (circular) array.

A *priority queue* is a different kind of queue, in which the next element to be removed is defined by (possibly) some other criterion. For example, in a hospital emergency room, patients are treated not in a first-come first-serve basis, but rather the order may also or instead be determined by the urgency of the case. To define the next element to be removed, it is necessary to have some way of comparing any two objects and deciding which has greater priority. Once a comparison method is chosen for determining priority, *the next element to be removed is the one with greatest priority.* Heads up: with priority queues, one typically assigns low numerical values to high priorities. Think "my number one priority", "my number 2 priority", etc.

Since we are talking about comparable elements, let's switch our terminology and refer the elements as *elements*. One way to implement a priority queue of elements is to maintain a sorted list. This could be done with a linked list or array list. Each time a element is added, it would need to be inserted into the sorted list. If the number of elements were huge, however, then this would be an inefficient representation since adds and removes would be $O(n)$. A second way to implement a priority queue would be to use a binary search tree. The element that is removed next is found by the `findMinimum()` operation. This would be a better way to implement a priority queue than the linear list method, since add and remove tend to take $\log n$ steps rather than n steps, if the tree is balanced. (I mentioned *balanced* binary search trees briefly in the last lecture. This topic will be covered in more depth – pun intended – in COMP 251.) One problem with using a balanced binary search tree for a priority queue is that it is overkill. We will look at a simpler data structure, called a *heap*.

Heaps

The most common way to implement a priority queue is to use a specific kind of binary tree, called a *heap*. To define a heap, we first need to define a complete binary tree.

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 nodes are comparable and satisfy the property that *each node is less than its children.* (To be precise, when I say that the nodes are comparable, I mean that the *elements* are comparable.) 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 a min heap in the next few lectures. Note that it follows from the definition that the smallest element in a heap is stored at the root.

As with stacks and queues, the two main operations we perform on heaps are `add` and `remove`.

add

To add an element to a heap, 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 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 element 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. This process of moving a node up the heap, is often called "upheap".

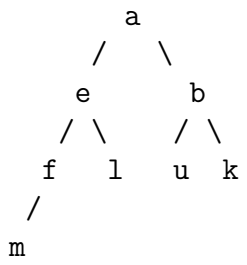
```
add(element){
   cur = new node at next leaf position
   cur.element = element
   while (cur != root) && (cur.element < cur.parent.element){
       swapElement(cur, parent)
       cur = cur.parent
   }
}
```

You might ask whether swapping the element at a node with its parent's element can cause a problem with the node's sibling (if it exists). It is easy to see that no problem exists though. Before the swap, the parent is less than the sibling. So if the current node is less than its parent, then the current node must be less than the sibling. So, swapping the node's element with its parent's element preserves the heap property with respect to the node's current sibling.

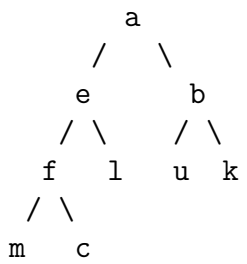
For example, suppose we have a heap with two element `e` and `g`. Then we add an element to the `*` position below and we find that `* < e`. So we swap them. But if `* < e` then `* < g`.

```
  e
/   \
g    *
```

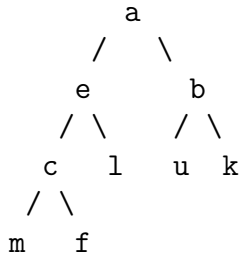
Here is a bigger example. Suppose we add element `c` to the following heap.



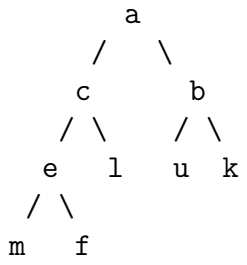
We add a node which is a sibling to `m` and assign `c` as the element of the new node.



Then we observe that `c` is less than the element `f` of its parent, so we swap `c,f` to get:



Now we continue up the tree. We compare `c` with the element in its new parent `e`, see that the elements need to be swapped, and swap them to get:



Again we compare `c` to its parent. Since `c` is greater than `a`, we stop and we're done.

removeMin

Next, let's look at how we remove elements from a heap. Since the heap is used to represent a priority queue, we remove the minimum element, which is the root.

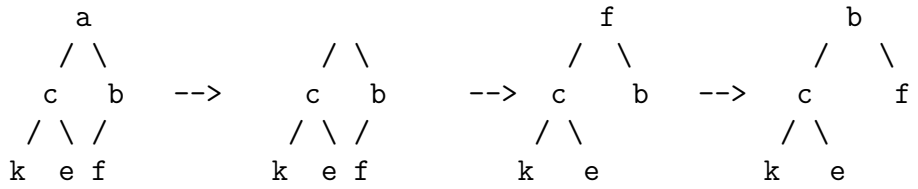
How do we fill the hole that is left by the element we removed ? We first copy the last element in the heap (the rightmost element in level h) into the root, and delete the node containing this last element. We then need to manipulate the elements in the tree to preserve the heap property that each parent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

We start at the root, which now contains an element that was previously the rightmost leaf in level h. We compare the root to its two children. If the root is greater than at least one of the children, we swap the root with the smaller child. Moving the smaller child to the root does not create a problem with the other child and with the heap property, since by definition the smaller child is greater than the larger child.

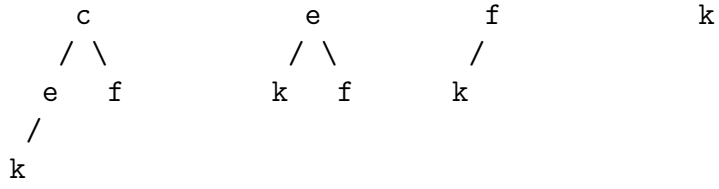
Here is a sketch of the algorithm:

```
removeMin(){
   remove last leaf node and put its element into the root
   node = root
   while ((node has at least one child) and
          ((node.element > leftchild.element) or
           (node has right child and node.element > rightchild.element)))
       minChild = child with the smaller element
       swapElement(node, minChild)
       node = minChild
   }
}
```

Here is an example:

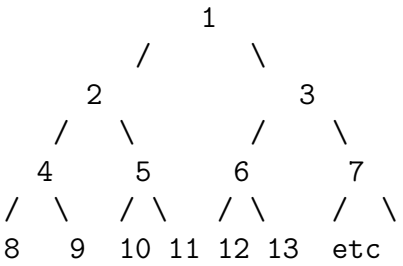


If we apply `removeMin()` again and again until all the elements are gone, we get the following sequence of heaps (with elements removed in the following order: `b, c, e, f, k`.



Implementing a heap using an array

A heap is defined to be a complete binary tree. If we number the nodes of a heap by a level order traversal and start with index 1, rather than 0, then we get an indexing scheme as shown below.

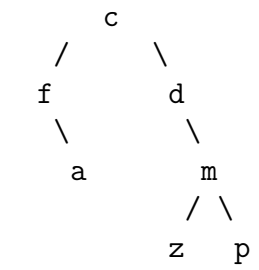


These numbers are NOT the elements stored at the node, rather we are just numbering the nodes so we can index them.

This indexing scheme gives a simple relationship between a node's index and its children's index. If the node index is i, then its children have indices $2i$ and $2i+1$. Similarly, if a non-root node has index i then its parent has index $i/2$.

It is very common to use an array to represent a heap, rather than a binary tree, and to use the parent/child indexing scheme described above rather than using nodes and parent, leftchild, and right child references. Note that the heap is still a complete binary tree. We will still be talking about "parent", "left child", and "right child". However, we will be using an array to represent the binary tree.

ASIDE: note that you can always use an array to represent a binary tree if you want, by using the above indexing scheme. However, there are costs to doing so when the binary tree is NOT a complete binary tree, namely there maybe large gaps in the array. We would just have null references at these gap point. For example, below is a binary tree (left) and its array representation (right) where - indicates null. Note there is a `null` in the 0-th element, but this is also the case for the heap. Its only there to make the parent-child indexing formula simpler to think about.




```
-cfd-a-m------zp
```