Priority Queue

Recall the definition of a queue. The fundamental property is that the next item to be removed is the one that was in the queue longest. 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 is also determined by the urgency of the case (perhaps in combination with other factors, including how long the patient has been waiting). 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. Then, *the next item to be removed is the one with greatest priority.* Careful though: with priority queues, one typically assign low numerical values to high priorities. Think: "my number one priority", "my number 2 priority", etc.)

One way to implement 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. If the number of items is huge, then this would be an inefficient representation since inserts and removals are $O(n)$. A second way to implement a priority queue would be to use a binary search tree. The item that is removed next is found by the `findMinimum()` operation. This would be a better way to implement a priority queue than the linear method, since an insertion and deletion tend to take $\log n$ steps rather than n steps. However, there is no guarentee on that. Also, a binary search tree can have long paths.

ASIDE: Java's `PriorityQueue` class

The Java API defines a class `PriorityQueue<T>` where type `T` implements `Comparable`. The `PriorityQueue<T>` class has methods `add` and `remove`. It also has methods called `offer` and `poll`, which basically do the same thing as `add` and `remove`, respectively. There are differences between `add` and `offer` and between `remove` and `poll` but these do not concern us here. Have a look at the API if you are interested.

Heaps

The most common way to implement a priority queue is to use an different kind of binary tree, called a *heap*. A heap avoids the long paths that can arise with binary search trees. For now, we define a heap abstractly. Later in the lecture I will say how we typically implement it.

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[1] and satisfy the property that *each node is less than its children.* 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.

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

[1] I am not distinguishing the nodes being comparable from the elements at the nodes being comparable.

add

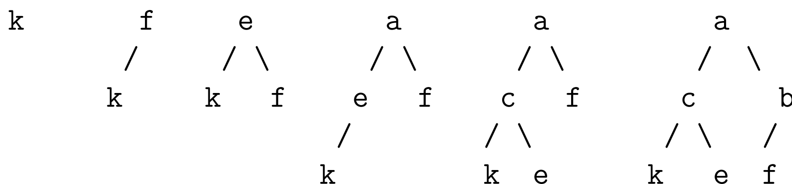
To add an element to a heap, we create a new node and insert it in the next available position of the (complete) tree. If level h is not full, then we insert it next to the rightmost element. If level h is full, then we start a new level at height $h + 1$.

Once we have inserted the new node, we need to be sure that the heap property is satisfied. The 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 (the element of) 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 this cannot happen though. The sibling must be greater than the parent (since we have a heap), and so if the node is less than its parent, then (by transitivity) the node must be less than the sibling. So, swapping the node with its parent preserves the heap property with respect to the node's current sibling.

Example

Let's suppose the heap is inititally empty and then we add items `k,f,e,a,c,b` in that order. Here is how the heap evolves. Later we will see the algorithm that was used.

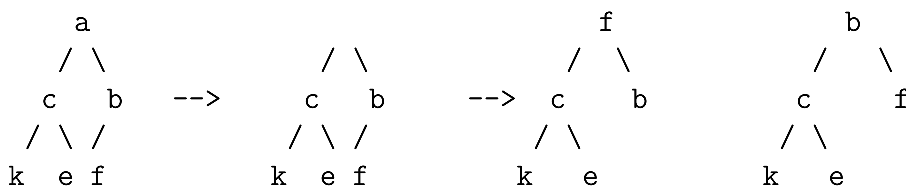


removeMin

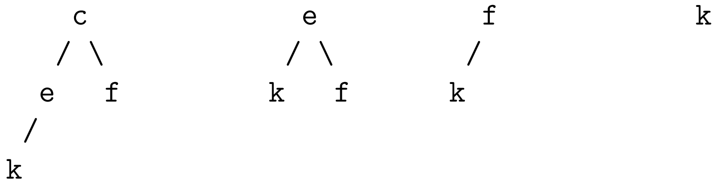
Next, let's look at how we remove elements from a heap. Since the heap is used to represent a priority queue, we remove the minimum element, which is the root.

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.

Starting at the root (which contains an element that was previously a leaf), we compare the root to its two children. If the root is less than its two children, then we are done. Otherwise, the root is greater than at least one of the children. In this case, we swap the root with the smaller child. 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. For example:

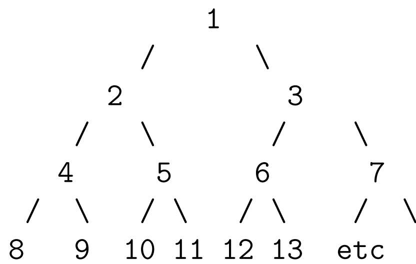


And we remove the remaining elements one by one as follows:



Implementing a heap using an array

We number the nodes of a heap by a level order traversal and start with index 1, rather than 0. This makes indexing slightly simpler (see below).



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

The above picture defines the relationship between a node's index and its children's index. If the node index is i, then its children have indices $2i$ and $2i + 1$ and its parent has index $i/2$ i.e. $\lfloor i/2 \rfloor$, since we are doing integer division.

Adding an element to a heap

Suppose we have a heap represented by an array, such that the first $i - 1$ elements define a heap (where $i > 1$). Now we wish to add element i to the heap. We do so by comparing the element at i to its parent's element. If its parent's element is greater, then we need to swap. We continue moving our element up the heap until it is less than or equal to its parent. When we are done, we can be sure that the first i elements define a value heap.

Let's write this slightly more generally. Suppose we have an array such that the first i-1 elements define a heap. We now want to add another element to the heap.

```
add( element ){
   i = a.size + 1
   a[i] = element
   upheap( i )
}
```

where we use the method `upHeap` defined as follows.

```
upHeap( i ){
   if (i > 1  and a[i] < a[i/2]){
      swap( i, i/2 )              //  swap elements a[i] and a[i/2]
      upHeap( i/2 )
   }
}
```

Building a heap

We can use the above `upHeap` operation to make a heap as follows. You begin with an array `a` whose elements are in positions 1 to $n =$ `a.size`. The elements are in no particular order. If we just take the element `a[1]`, then this single element defines a heap. Then, given a heap with i-1 elements, we upHeap the element i, given us a heap out of the first i elements. By induction, this gives us a heap of size n.

```
buildHeap(){

//  INPUT:   an array a of unsorted elements, indexed from 1 to size
//        (assume size > 1)
//  OUTPUT:  the elements of the array rearranged so that the array is a heap

  for (i = 2; i <= a.size; i++)
     upHeatp( i );
}
```