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 \(\text{findMinimum}()\) 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, if the tree is balanced. However, there is no guarantee on that. A binary search tree can have long paths.

**ASIDE: Java’s PriorityQueue class**

The Java API defines a class \(\text{PriorityQueue<T>}\) where type \(T\) implements \(\text{Comparable}\). The \(\text{PriorityQueue<T>}\) class has several methods, including an \(\text{add}\) and \(\text{remove}\) method. You can check out the Java API for the details. If you do, you’ll note that there are some subtle differences between these methods. I won’t go into them here, since there are more fundamental things to learn now, namely.....

Heaps

The most common way to implement a priority queue is to use a special 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\(^1\) and satisfy the property that \(\text{each node is less than its children}\). 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 in the next few lectures.) 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 \(\text{add}\) and \(\text{remove}\).

\(^1\)I am not distinguishing the nodes being comparable from the keys/elements at the nodes being comparable.
How many heaps are there?

Suppose we have letters a, b, c, d, e, f, g. How many heaps can we make with these letters?

The root has to be a since it is the minimum element. The letter b has to be in level 1 all elements other than a and b are greater than b and hence none of them can be b’s parent. Letter c does not have to be the sibling of b, though. Letter c could be a child of b, and in that case the sibling of b could be either d or e. See the slides for several examples.

Note that if you swap any two sibling nodes, you automatically preserve the heap property. Note that swapping sibling nodes is different from swapping sibling keys. If you swap nodes, you take the children with. For example, here I swap nodes b and c in the heap.

```
     a
    /  \
   b   c
  /  \
 d   e
```

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.

```java
add(key){
    create a new node at next available leaf position
    cur = new node
    cur.key = key
    while (cur != root) && (cur.key < cur.parent.key){
        swapKey(cur, parent)
        cur = cur.parent
    }
}
```

You might ask whether swapping the key at a node with its parent’s key can cause a problem with the node’s sibling (if it exists). It is easy to see that no problem exists though. The sibling must be greater than or equal to its arent (since we have a heap), and so if the current node is strictly less than its parent, then (by transitivity) the node must be less than the sibling. So, swapping the node’s key with its parent’s key preserves the heap property with respect to the node’s current sibling.
Here is an example. Suppose we add key \( c \) to the following heap.

```
   a
  /   \\
 e   b
 / \   / \   /
 f   l   u   k
   /   \\
 m
```

We add a node which is a sibling to \( m \) and assign \( c \) as the key of the new node.

```
   a
  /   \\
 e   b
 / \   / \   /
 f   l   u   k
   /   \\
 m   c
```

Then we observe that \( c \) is less than the key \( f \) of its parent, so we swap keys to get:

```
   a
  /   \\
 e   b
 / \   / \   /
 c   l   u   k
   /   \\
 m   f
```

Could this move have violated the heap property, with respect to its former sibling \( m \)? No. The sibling \( m \) was a child of \( f \) beforehand, and so we know that the sibling \( m \) was greater than or equal to \( f \). But we are swapping \( c \) and \( f \) because \( c \) is strictly less than \( f \). Thus, \( c \) is less than its former sibling \( m \).

Now we continue up the tree. We compare \( c \) with the key in its new parent \( e \), see that the keys need to be swapped, and swap them to get:

```
   a
  /   \\
 c   b
 / \   / \   /
 e   l   u   k
   /   \\
 m   f
```

Again we compare \( c \) to its parent’s key – but since 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.

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.

Here is a sketch of the algorithm:

```cpp
removeMin()
{
    remove last leaf node and put its key into the root
    node = root
    while ((node has at least one child) &&
            (node.key > leftchild.key) &&
            (node.key > rightchild.key))
        c = child with the smaller key
        swapKey(node, c)
        node = c
}
```

Here is an example:

```
a       f     b
/ \     / \    / \    / \\
c b --c b --c b --c f
/ \ / / / / / \ / \\
k e f k e f k e k e
```

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

```
c       e     f     k
/ \     / \     / \\
e f     k f     k
/ \\
k
```