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 a priority, which is a more general criterion. For example, in a hospital emergency room, patients are treated not in a first-come first-serve basis, but rather 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.

One way to implement a priority queue of elements (often called *keys*) 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 in the worst case the adds and removes would be $O(n)$.

Heaps

The usual way to implement a priority is to use a data structure 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, in the sense that they can all be ordered – recall the *Comparable* interface in Java, presented in lecture 16) 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 leaf. 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). No, it cannot. Before the swap, the parent is less than the sibling.[1] So if the current node is less than its parent, then the current node must be less than the sibling too. 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 elements `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.

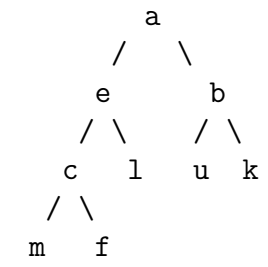
```
          a
        /   \
       e     b
      / \   / \
     f   l  u  k
    /
   m
```

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

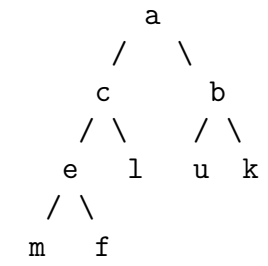
```
          a
        /   \
       e     b
      / \   / \
     f   l  u  k
    / \
   m   c
```

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

[1] Here I say that one node is less than another, but what I really mean is that the element at one node is less than the element at the other node.



Now we continue up the tree. We compare c with its new parent's element 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.

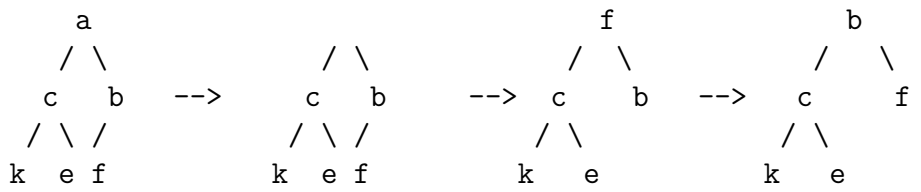
How do we fill the hole that is left by the element we removed ? We first copy the last element in the heap (the rightmost element in level h) into the root, and delete the node containing this last element. We then need to manipulate the elements in the tree to preserve the heap property that each parent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

We start at the root, which 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 larger child, since the smaller child is smaller than the larger child.

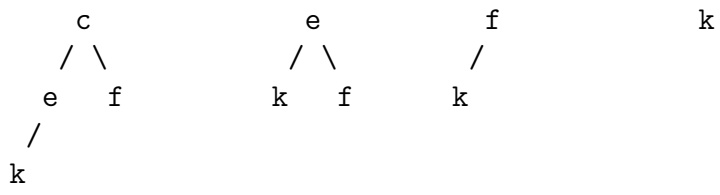
```
removeMin(){                // returns smallest element
   tmp = root.element
   remove last leaf node and put its element into the root
   cur = root
   while ((cur has a left child) and
          ((cur.element > cur.left.element) or
           (cur has a right child and cur.element > cur.right.element)))
       minChild = child with the smaller element
       swapElement(cur, minChild)
       cur = minChild
   }
   return tmp
}
```

The condition in the while loop is rather complicated, and you may have just skipped it. Don't. There are several possible events that can happen and you need to consider each of them. One is that the current node has no children. In that case, there is nothing to do. The second is that the current node has one child, in which case it is the left child. The issue here is that the left child might be smaller. The third is that the current node might have two children. In that case, one of these two children *has to be* smaller than the current node. (See Exercises.)

Here is an example:

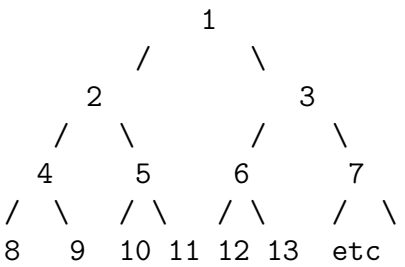


If we apply `removeMin()` again and again until all the elements are gone, we get the following sequence of heaps with elements removed in the following order: `b, c, e, f, k`.



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.

The idea is that an array representation defines a simple relationship between a tree 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 $i/2$.

Today we will revisit the `add` and `removeMin` methods and rewrite them in terms of this array. We will also show how to build a heap using the array representation, and we can use a heap to sort a set of comparable elements – called *heapsort*.

add(element)

Suppose we have a heap with `k` elements which is represented using an array, and now we want to add a `k+1`-th element. Last lecture we sketched an algorithm doing so. Here I'll re-write that algorithm using the simple array indexing scheme. Let `size` be the number of elements in the heap. These elements are stored in array slots 1 to `size`, i.e. recall that slot 0 is unused so that we can use the simple relationship between a child and parent index.

```
add(element ){
   size = size + 1          // number of elements in heap
   heap[ size ] = element   // assuming array has room for another element
   i = size

   //  the following is sometimes called "upHeap"  -- see below

   while (i > 1  and heap[i] < heap[ i/2 ]){
     swapElements( i, i/2 )
     i = i/2
   }
}
```

Here is an example. Let's add `c` to a heap with 8 elements.

```
1   2   3   4   5   6   7   8   9
-------------------------------------
a   e   b   f   l   u   k   m   c
a   e   b   c   l   u   k   m   f  <----  c swapped with f (slots 9 & 4)
a   c   b   e   l   u   k   m   f  <----  c swapped with e (slots 4 & 2)
```

Building a heap

We can use the `add` method to build a heap as follows. Suppose we have a list of `size` elements and we want to build a heap.

```
buildHeap(list){
   create new heap array         //  size == 0,  length > list.size
   for (k = 0; k < list.size; k++)
        add( list[k] )                    //  add to heap[ ]
}
```

We can write this in a slightly different way.

```
buildHeap(list){
   create new heap array      //  size == 0,  length > list.size
   for (k = 1; k <= list.size; k++){
       heap[k] = list[k-1]       //  say list index is 0, .. ,size-1
       upHeap(k)
   }
}
```

where `upHeap(k)` is defined as follows.

```
upHeap(k){    //  assumes we have an underlying array structure
    i = k
    while (i > 1  and heap[i] < heap[ i/2 ]){
       swapElements( i, i/2 )
       i = i/2
    }
}
```

Are we sure that the buildHeap() method indeed builds a heap? Yes, and the argument is basic mathematical induction. Adding the first element gives a heap with one element. If adding the first k elements results in a heap, then adding the $k+1$-th element also results in a heap since the upHeap method ensures this is so.

Time complexity

How long does it take to build a heap in the best and worst case? Before answering this, let's recall some notation. We have seen the "floor" operation a few times. Recall that it rounds down to the nearest integer. If the argument is already an integer then it does nothing. We also can define the ceiling, which rounds up. It is common to use the following notation:

- $\lfloor x \rfloor$ is the largest integer that is less than or equal to x. $\lfloor\ \rfloor$ is called the *floor* operator.
- $\lceil x \rceil$ is the smallest integer that is greater than or equal to x. $\lceil\ \rceil$ is called the *ceiling* operator.

Let i be the index in the array representation of elements/nodes in a heap, then i is found at level *level* in the corresponding binary tree representation. The level of the corresponding node i in the tree is such that

$$2^{level} \leq i < 2^{level+1}$$

or

$$level \leq \log_2 i < level + 1,$$

and so

$$level = \lfloor \log_2 i \rfloor.$$

We can use this to examine the best and worst cases for building a heap.

In the best case, the node i that we add to the heap satisfies the heap property immediately, and no swapping with parents is necessary. In this case, building a heap takes time proportional to the number of nodes n. So, best case is $\Theta(n)$.

What about the worst case? Since $level = \lfloor \log_2 i \rfloor$, when we add element i to the heap, in the *worst case* we need to do $\lfloor \log_2 i \rfloor$ swaps up the tree to bring element i to a position where it is less than its parent, namely we may need to swap it all the way up to the root. If we are adding n nodes in total, the worst case number of swaps is:

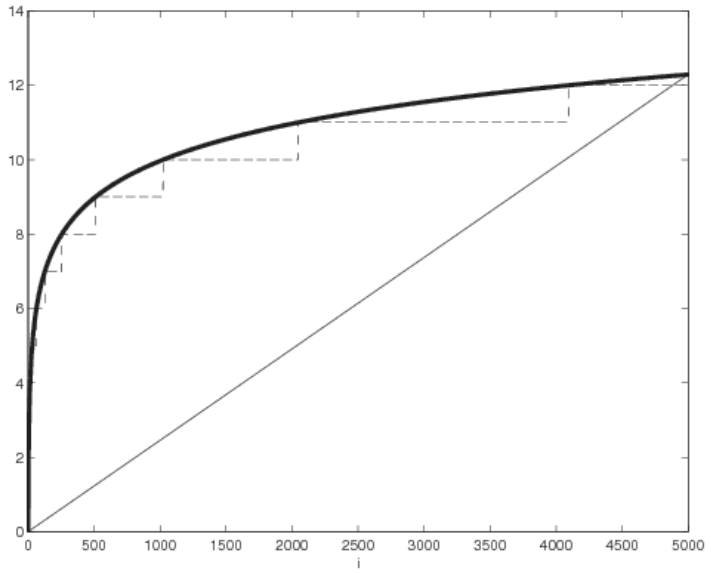
$$t(n) = \sum_{i=1}^{n} \lfloor \log_2 i \rfloor$$

To visualize this sum, consider the plot below which show the functions $\log_2 i$ (thick) and $\lfloor \log_2 i \rfloor$ (dashed) curves up to $i = 5000$. In this figure, $n = 5000$.

The area under the dashed curve is the above summation. It should be visually obvious from the figures that

$$\frac{1}{2} n \log_2 n < t(n) < n \log_2 n$$

where the left side of the inequality is the area under the diagonal line from (0,0) to $(n, \log_2 n)$ and the right side $(n \log_2 n)$ is the area under the rectangle of height $\log_2 n$. From the above inequalities, we conclude that in the worst of building a heap is $O(n \log_2 n)$.



removeMin

Next, recall the `removeMin()` algorithm from last lecture. We can write this algorithm using the array representation of heaps as follows.

```
removeMin(){
    element = heap[1]            //   heap[0] not used.
    heap[1] = heap[size]
    heap[size] = null
```

```
    size = size - 1
    downHeap(1, size)       //   see next page
    return element
}
```

This algorithm saves the root element to be returned later, and then moves the element at position `size` to the root. The situation now is that the two children of the root (node 2 and node 3) and their respective descendents each define a heap. But the tree itself typically won't satisfy the heap property: the new root will be greater than one of its children. In this typical case, the root needs to move down in the heap.

The `downHeap` helper method moves an element from a starting position in the array down to some maximum position in the heap. I will use this helper method in a few ways in this lecture.

```
downHeap(start,maxIndex){     //  move element from starting position
                              //  down to at most position maxIndex
   i = start
   while (2*i <= maxIndex){              // if there is a left child
      child = 2*i
      if (child < maxIndex) {           // if there is a right sibling
         if (heap[child + 1] < heap[child])
                                        //  if rightchild < leftchild ?
         child = child + 1
      }
      if (heap[child] < heap[ i ]){   // swap with child?
         swapElements(i , child)
         i = child
      }
      else  break    // exit while loop
   }
}
```

This is essentially the same algorithm we saw last lecture. What is new here is that (1) I am expressing it in terms of the array indices, and (2) there are parameters that allow the downHeap to start and stop at particular indices.

Heapsort

A heap can be used to sort a set of elements. The idea is simple. Just repeatedly remove the minimum element by calling `removeMin()`. This naturally gives the elements in their proper order.

Here I give an algorithm for sorting "in place". We repeatedly remove the minimum element the heap, reducing the size of the heap by one each time. This frees up a slot in the array, and so we insert the removed element into that freed up slot.

The pseudocode below does exactly what I just described, although it doesn't say "`removeMin()`. Instead, it says to swap the root element i.e. `heap[1]` with the last element in the heap i.e. `heap[size+1-i]`. After `i` times through the loop, the remaining heap has `size - i` elements,

and the last `i` elements in the array hold the smallest `i` elements in the original list. So, we only downheap to index `size - i` .

```
heapsort(list){
buildheap(list)
for i = 1 to size-1{
swapElements( heap[1], heap[size + 1 - i])
downHeap( 1,  size - i)
}
return reverse(heap)
}
```

The end result is an array that is sorted from largest to smallest. Since we want to order from smallest to largest, we must `reverse` the order of the elements. This can be done in $\Theta(n)$ time, by swapping i and $n+1-i$ for $i = 1$ to $\frac{n}{2}$.

Example

The example below shows the state of the array after each pass through the for loop. The vertical line marks the boundary between the remaining heap (on the left) and the sorted elements (on the right).

```
1   2   3   4   5   6   7   8   9
-------------------------------------
a   d   b   e   l   u   k   f   w |
b   d   k   e   l   u   w   f | a   (removed a, put w at root, ...)
d   e   k   f   l   u   w | b   a   (removed b, put f at root, ...)
e   f   k   w   l   u | d   b   a   (removed d, put w at root, ...)
f   l   k   w   u | e   d   b   a   (removed e, put u at root, ...)
k   l   u   w | f   e   d   b   a   (removed f, put u at root, ...)
l   w   u | k   f   e   d   b   a   (removed k, put w at root, ...)
u   w | l   k   f   e   d   b   a   (removed l, put u at root, ...)
w | u   l   k   f   e   d   b   a   (removed u, put w at root, ...)
w   u   l   k   f   e   d   b   a   (removed w, and done)
```

Note that the last pass through the loop doesn't do anything since the heap has only one element left (`w` in this example), which is the largest element. We could have made the loop go from `i = 1 to size - 1`.