**CS512 LECTURE NOTES - LECTURE 11**

# 1 Priority Queues

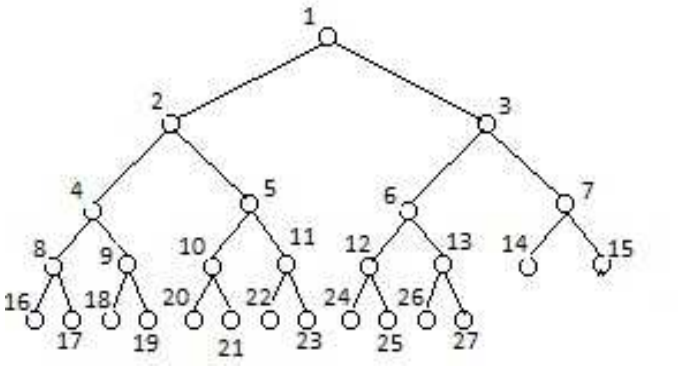
A priority queue is a data structure that has the following public methods:

1. extractMax() or extractMin(), depending on the type of queue.
2. insert(x)
3. biuldHeap()
4. decreaseKey(k)
5. increaseKey(k)

The best way to implement a priority queue is to use a **Heap**.

### 1.1 Heap

- We can have a MAX-heap where we are interested in extracting the maximum value of the entire queue, or a MIN-heap where we are interested in extracting the minimum value of the entire queue. We will assume that we are implementing a MAX-heap, a MIN-heap is similar, just change $\leq$ for $\geq$.
- A heap is a complete binary tree (each node has exactly two children), expect for (possibly) the last level which must be complete from left to right.
- The tree can be binary (each node can have at most two children) which is a binary heap, or it can be k-ary (each node can have several children) which is a k-ary heap.
- It must satisfy the **Heap Property:** Given any node $i$, the value of the element stored in position $i$ must be $\geq$ than the value of each one of its children.
- The elements of the heap can be placed in an array. The positions of the elements of the array are shown below for a heap of size $n = 27$.




### 1.2 Finding children and parent

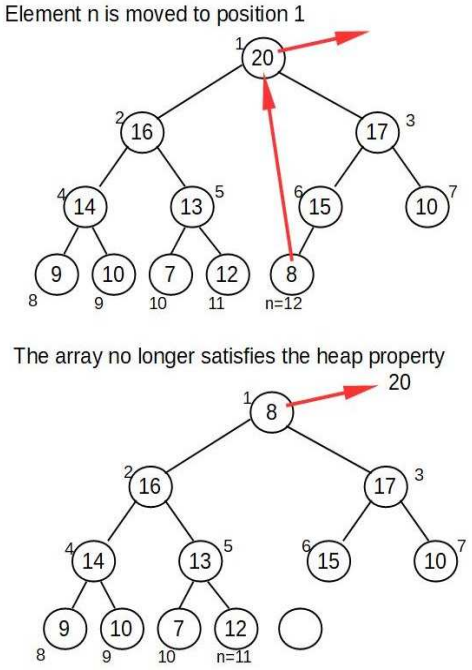
One advantage of using an array to store the heap is that it is easy to find the children and the parent of a given node $k$:

- The parent of $k$ is $\lfloor \frac{k}{2} \rfloor$ if $k > 1$, otherwise, $k$ has no parent.
- The left child of $k$ is $2k$
- The right child of $k$ is $2k + 1$

### 1.3 extractMax()

Notice that the heap property makes sure that the maximum is always kept in the root of the tree, so the extractMax method must return the value of the element stored in the root of the tree, and then delete the root.

But if we delete the root, the tree will no longer be a heap, so we will take element a[n] and put it in position 1 (root), so we can reduce the size of the tree to n-1. However, if we do this, the tree might no longer satisfy the heap property as can be seen in the following example:




Notice that the only element that is out of place is the element now at the root (call it $i$). The solution is to find the index ($j$) to the maximum of the three elements a[i], a[left(i)], a[right(i)], if $j \neq i$ we can swap a[i] with a[j], but then we have to repeat the same process with element $j$. This method is called *siftDown(i)*.

```
extractMax()
if (n>0)
    begin
    max=a[1]
    a[1]=a[n]
    n–
    siftDown(1)
    end
```

### 1.4 siftDown(i)

Since *siftDown(i)* is used internally by the public methods of the heap, it must be implemented as a *private* method.

Here is a recursive version of the *siftDown(i)* algorithm, the exit condition is when element $i$ has no children, i.e. $2i > n$

```
siftDown(i)
   begin
   if (2i≤ n)
         begin
         Let j  be the index of max(a[i],a[left(i)],a[right(i)])
         if (i≠j)
               begin
               Swap a[i] with a[j]
               siftDown(j)
               end
         end
   end
```

### 1.5 Analysis of *extractMax* and *siftDown*

In the *extractMax* algorithm, *siftDown* starts from the root and goes one level at a time until it finds a leaf (node with no children). Since it is a "complete" binary tree, the total depth is at most $\lg n$, since at each level only $\Theta(1)$ operations are performed by *siftDown*, because finding the max of at most three elements can be done in constant time, the total running time of *extractMax* is $T(n) \in \Theta(\lg n)$

### 1.6 insert(x)

The insert method takes an element $x$ and adds it at position a[n+1] of the array. The problem is that after inserting $x$ the heap might no longer satisfy the heap property. In order to satisfy the heap property, we will make the element $x$ *bubbleUp* until it finds its proper position.

```
insert(x)
   n++
   a[n]=x
   bubbleUp(n)
```

### 1.7 bubbleUp(i)

The *bubbleUp(i)* method takes an index $i$ and if that index is not the root it will compare the value store in position $i$ with the value of its parent. If the value in position $i$ is larger than its parent, then the method will swap a[i] with a[parent(i)], and recursively call *bubbleUp* on the parent of $i$.

```
bubbleUp(i)
if (i>1)
    begin
    Let j be the index of max(a[i],a[parent(i)])
    if (j=i)
       begin
       swap a[i] with a[parent(i)]
       bubbleUp(parent(i))
       end
    end
```

### 1.8 Analysis of *insert* and *bubbleUp*

Notice that *insert* calls *bubbleUp* on element $n$, which is a leaf. *bubbleUp* will keep swapping one level at a time until it reaches the root of the tree (in the worst case). Therefore the worst case time complexity of *insert* is $T(n) \in O(\lg n)$

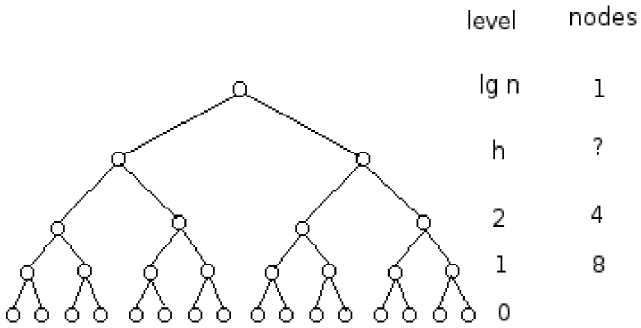
### 1.9 buildQueue(a,n)

To build a heap given an array $a$ and its size $n$, we initially consider the array as a representation of a heap (tree), but it might not satisfy the heap property.

In order to make it satisfy the heap property we will start from the node ($i$) with the largest index such that it has children, and work our way up to the root one node at a time sifting down each node. Notice that the largest node index with children is $\lfloor \frac{n}{2} \rfloor$.

```
buildHeap(a,n)
    if n>1
        begin
        for i=⌊n/2⌋ down to 1
            begin
            siftDown(i)
            end
        end
```

### 1.10 Analysis of *buildQueue*




From the figure we can see that the **maximum** number of nodes in level $h$ is

$$\frac{n}{2^h}$$

The total number of operations of the *buildQueue* algorithm can be computed by counting the number of calls to *siftDown* in each level. We know that **siftDown** will take $i$ operations for a node in level $i$.

$$T(n) \leq \sum_{i=1}^{\lg n} i\frac{n}{2^i} = n\sum_{i=1}^{\lg n} \frac{i}{2^i}$$

Now all we have to do to find $T(n)$ is solve the summation $\sum_{i=1}^{\lg n} \frac{i}{2^i}$

Notice that

$$\sum_{i=1}^{\lg n} \frac{i}{2^i} \leq \sum_{i=1}^{\infty} \frac{i}{2^i}$$

Let

$$S = \sum_{i=1}^{\infty} \frac{i}{2^i} \quad = \quad \frac{1}{2} + \frac{2}{2^2} + \frac{3}{2^3} + \frac{4}{2^4} + \ldots$$

Multiplying by $1/2$

$$\frac{1}{2}S \quad = \quad \frac{1}{2^2} + \frac{2}{2^3} + \frac{3}{2^4} + \frac{4}{2^5} + \ldots$$

Subtract from $S$

$$S - \frac{1}{2}S \quad = \quad \frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \ldots = 1$$

So we have that $S = 2$, and

$$T(n) \in O(n)$$

### 1.11 Heapsort

A simple application of priority queues is to sorting. It is easy to write a sorting algorithm that takes the input array, transforms it into a priority queue and repeatedly extracts the maximum of the array placing it as the last position.

**Algorithm Heapsort(a)**
Q.buildQueue(a)
while not Q.isEmpty()
  a[n]=Q.extractMax()
  (we assume that extractMax decrements n)

The worst case complexity of the Heapsort algorithm is $O(n \lg n)$ since `buildQueue` is $O(n)$ and `extractMax` is in $O(\lg n)$ (repeated $n$ times).