Priority Queue/Heap

This document contains summary of some of the items discussed as part of the COMS 3110 Lectures. This document **does not replace the lecture materials**. This document may contain some topics that are not covered as part of the lecture; you will not be tested on those parts, they are made available to you for gaining further knowledge on topics/concepts that are related to class-lecture.

Heap is a tree structure, where

1. in any subtree the root (i.e., the parent) of the subtree has a value that is greater than (less than) equal to the values of its children.
2. the tree is almost complete. The leaf nodes are fill up the tree from left to right.

If the parent-node value is greater than or equal to that of its child-nodes, then the heap is termed as max heap; otherwise it is referred to as min heap. If each node in the heap tree has at most 2 children then the heap is called a binary heap. The height of a heap tree is $O(\log n)$ as the tree is balanced by definition.

1 Array-based Implementation

As the heap is an almost complete tree, the elements can be stored in an array while capturing the parent-children relationship (using the array index).

Let the root index is denoted by r. Typically, there are two ways the root index is decided: either 0 or 1. The parent-children indices relationship is based on r. It is as follows: given that parent is at index p, and its left-child is in index l and right-child is in index r.

$$l = \begin{cases} 2 \times p + 1 & \text{if } r = 0 \\ 2 \times p & \text{if } r = 1 \end{cases} \qquad r = \begin{cases} 2 \times p + 2 & \text{if } r = 0 \\ 2 \times p + 1 & \text{if } r = 1 \end{cases}$$

Similarly, one can identify the parent index from the index of the children ((`child-index` $-1)/2$ when $r = 0$; otherwise it is (`child-index`$/2$). The index of the last element n-th element is $n - 1$ when $r = 0$; otherwise it is n.

0	1	2	3	4	5	if $r = 0$
10	30	15	100	50	80	
1	2	3	4	5	6	if $r = 1$

min heap

0	1	2	3	4	5
100	80	30	50	10	15
1	2	3	4	5	6

max heap

2 Heap operations

The operation heapifyUp takes as input the index of an element, which is not in its correct position as per the heap property, in particular, it's value is lower than (higher than) the value of its parent in a min heap (resp. max heap). The operation pushes the element up the heap tree by exploring along the child-parent relationship.

```
// i: index from where heapify up is performed
// root is at index 1
// heap array is A: minheap

heapifyUp(int i) {
   if (i==1)
```

```
      return;
   if (A[i] > A[i/2]) // nothing to do
       return;
   else {
       swap(A[i], A[i/2]);
       heapifyUp(i/2);
   }
}
```

The dual operation is heapifyDown. In this case, the parent is not in correct position. The element is pushed down along the tree-path that contains the child with the larger (smaller) value for max heap (resp. min heap).

```
// i: index from where heapify down is performed
// size: number of elements in the heap-tree
// root is at index 1
// heap array is A: minheap

heapifyDown(int i, int size) {
  if (2i > size)  // i is leaf-index
      return
  if (2i+1 > size) // only left child exist
      j=2i;
  else            // both children exist
     if (A[2i] < A[2i+1]) // left is smaller
        j=2i;
     else         // right is smaller
        j=2i+1;
     if (A[i] < A[j]) // nothing to do
        return
     else {
        swap(A[i], A[j]);
        heapifyDown(j, size);
    }
}
```

The above operations are used to implement find-and-extract the maximal element for max heap or minimal element for min heap, and for inserting new elements in the heap.

```
// size: number of elements in the heap
// heap array: A
// root of the heap at index 1
// Example code for extracting the root of a heap
extractMin() {
   root = A[i];
   A[1] = A[size];       // write the last element to the root
   size--;               // update the size of the heap-tree
   heapifyDown(1, size); // heapifyDown to rearrange the heap
   return root;
}

// size: number of elements in the heap
// heap array: A
```

```
// root of the heap at index 1
// Example code for inserting an element e in the heap
insert(e) {
   size = size + 1; // number of elements is increased by 1
   A[size] = e;     // the new element is added to the last index
   heapifyUp(size); // heapifyUp to rearrange the heap
}
```

The run-time for the find-and-extract, insert, heapifyUp and heapifyDown are $O(\log n)$ as the operations involve exploration of the (potentially the longest) path in the tree, which is th height of the tree.

Consider inserting the following elements to form a heap.

```
1963 1776 1941 1963 1492 1865 1783 1804 1918 1941 2001
```

MakeHeap. The operation makeHeap rearranges the elements in an array such that heap property is satisfied after the rearrangement. The strategy applied in this algorithm is (a) to consider the tree representation of the array elements and (b) to rearrange elements to satisfy heap property starting from the lower levels of the tree. Note that, the sub-trees rooted at the leaf-level already satisfy the heap property as such trees contain just one element (that element being one of the leaf-nodes of the original tree).

The algorithm is as follows:

```
// A is the array to be arranged to be a heap
makeHeap() {
  size = number of elements in A;
  for (i=last-parent; i>=r; i--) // r: root of heap
     heapifyDown(i, size);
}
```

The last-parent is the highest index of the element that has at least one child. The runtime of the algorithm is $O(n)$. This is because, the maximum number of constant time operations (compare and swap in this case) that occur for elements at level l is l; the level describes the distance of any node from a leaf. All leaf nodes are at level $l = 0$. The number of nodes at level l is of the order $\frac{n}{2^{l+1}}$. Therefore, the runtime is defined by the series

$$1.\frac{n}{2^2} + 2.\frac{n}{2^3} + 3.\frac{n}{2^4} + \ldots + L.\frac{n}{2^{L+1}}$$

where L is the level of the root. Hence, $\frac{n}{2^{L+1}} = 1$ as there is only one root element. Proceeding further,

$$\begin{aligned} & 1.\frac{n}{2^2} + 2.\frac{n}{2^3} + 3.\frac{n}{2^4} + \ldots + L.\frac{n}{2^{L+1}} \\ = \; & \frac{n}{2}\left(\frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3} + \ldots + \frac{L}{2^L}\right) \end{aligned} \tag{1}$$

Some basic algebra is needed to find the sum of the series in parenthesis above.

$$\begin{aligned}
S &= \frac{1}{2^1} + \frac{2}{2^2} + \frac{3}{2^3} + \ldots + \frac{L}{2^L} \\
\frac{S}{2} &= \frac{1}{2^2} + \frac{2}{2^3} + \ldots + \frac{L-1}{2^L} + \frac{L}{2^{L+1}}
\end{aligned}$$

Subtracting the second from the first

$$\begin{aligned}
\frac{S}{2} &= \frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^3} + \ldots + \frac{1}{2^L} - \frac{L}{2^{L+1}} \\
S &= \frac{1}{2^0} + \frac{1}{2^1} + \frac{1}{2^2} + \ldots + \frac{1}{2^{L-1}} - \frac{L}{2^L} \\
&= 2 \times \left(1 - \frac{1}{2^L}\right) - \frac{L}{2^L} \\
&= 2 \times \left(1 - \frac{2}{n}\right) - 2 \times \frac{\log(n) - 1}{n}
\end{aligned}$$

Therefore, from Equation 1

$$\frac{n}{2}\left(2 \times \left(1 - \frac{2}{n}\right) - 2 \times \frac{\log(n) - 1}{n}\right) \in O(n)$$