ESO211 (Data Structures and Algorithms)
Lectures 17 to 21

Shashank K Mehta

1 Heap Data Stucture

A max-heap is a combination of two structures: (i) It is an almost-complete binary tree and (ii) also a sequence/list such that any node preceeds its childr-nodes in the sequence. It stores a completely ordered set such that for any node j, $key(parent(j)) \geq key(j)$ for all j. One can similarly define a *min-heap* where the last relation is reversed, i.e., $key(parent(j)) \leq key(j)$ for all j. In this discussion we will only discuss max-heap but the entire discussion also applies to min-heaps.

Observation 1 *The root of a max-heap is a highest key node.*

Usually it is very convenient to implement a heap on an array because the second property is easily achieved. To define an almost complete binary tree structure on an array we use the following relations on the indices: $parent(i) = \lfloor (i-1)/2 \rfloor$, $leftChild(i) = 2i+1$, and $rightChild(i) = 2i+2$. The index of the root is 0.

A few points about the binary tree, especially almost complete binary trees, are in order. The depth of a node is the distance of the node from the root (number of edges on the path). At most 2^i nodes can exist in a binary tree at depth i. An almost complete binary tree of depth d contains 2^i nodes at depth i for all $i < d$ and contains at least 1 vertex in the depth d. Hence such a tree contains at least 2^d vertices and at most $2^{d+1} - 1$ vertices.

One very useful operation on a heap is when the sub-tree rooted at a node i is not avalid heap but those rooted at $leftChild(i)$ and $rightChild(i)$ are valid heaps. So we have, $key(i) < key(leftChild(i))$ or $key(i) < key(rightChild(i))$. Algorithm 1 shows that while not changing the elements of the heap we can rearange the elements of the sub-tree rooted at i such that finally this subtree becomes a heap. Here A is the array in which the heap is implemented and $HeapSize$ denotes the number of elements in the heap.

Verify that the resulting structure is a valid heap.

The time complexity of the procedue is $O(dep(i) + constant)$ where $dep(i)$ denotes the depth of the node i.

To extract the largest element from a max-heap, you need to output the key of the root and then re-fix the heap. This is done by removing the top key, copy

$large := i$;
if $leftChild(i) \leq HeapSize - 1$ AND $A[leftChild] > A[i]$ **then**
| $large := leftChild$;
end
if $rightChild(i) \leq HeapSize - 1$ AND $A[rightChild] > A[large]$ **then**
| $large := rightChild$;
end
if $large \neq i$ **then**
| $Swap(A[large], A[i])$;
| $FixHeap(A, parent(i), HeapSize)$;
end

Algorithm 1: FixHeap($A, i, HeapSize$)

the value of $A[HeapSize]$ to the root, and then perform FixHeap at the root. See algorithm 2.

Output $A[0]$;
$A[0] := A[HeapSize - 1]$;
$HeapSize := HeapSize - 1$;
$FixHeap(A, 0, HeapSize)$;

Algorithm 2: DeleteMax($A, HeapSize$)

To insert a new element in a heap we store the new element at $A[HeapSize]$ and then allow the new value to bubble up to its valid position. See Algorithm 3.

$A[HeapSize] := x$;
$HeapSize := HeapSize + 1$;
$i := HeapSize - 1$;
while $i > 0$ AND $key(i) > key(parent(i))$ **do**
| $swap(i, parent(i))$;
| $i := parent(i)$;
end

Algorithm 3: Insert($A, HeapSize, x$)

Prove that Algorithm 3 results in a valid heap.

Finally let us make a heap from a set m elements given in an array A stored in range $0 : m - 1$. We will perform this task iteratively in bottom-up order. Note that the elements in the range $m : parent(m) + 1$ are leaf nodes hence the sub-trees rooted at these vertices are single nodes and hence these are valid heaps. Starting at $parent(m)$ down to 0 we will fix the heap using $FixHeap$. See Algorithm 4.

```
for i := parent(m) Down to 0 do
|   FixHeap(A, i, m);
end
```

Algorithm 4: BuildHeap(A, m)

Let the depth of the tree be d, i.e., $d = \lfloor \log_2(m-1) \rfloor$. The cost of executing FixHeap at a node at depth i if $O(d-i)$. Hence the cost of $BuildHeap$ will be at most $\sum_{i=d-1}^{0} O((d-i) \cdot 2^i$. This bound is equal to $\sum_{j=1}^{d-j} j \cdot 2^{d-j} = 2^d \cdot \sum_{j=1}^{d-1} j/2^j$.

Note that the sum in the last expression is equal to $E = 2 \sum_{j=1}^{d-1} jx^{j-1}$ when we plug 1/2 for x. We can rewrite E as $d/dx(\sum_{j=1}^{d-1} x^j) \leq d/dx((x - x^d)/(1 - x)) = (1 - dx^{d-1})/(1 - x) + (x - x^d)/(1 - x)^2$. At $x = 1/2$ the right hand side is at most $2 + 2 = 4$. So that the total cost us at most $2^d.E \leq 4m$. Hence $BuildHeap$ has time complexity $O(m)$.