ESO211 (Data Structures and Algorithms) Lecture Notes Set 4

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 a valid 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 how to rearange the elements of the sub-tree rooted at i such that finally the subtree rooted at i also becomes a heap. It may be noted that the sub-tree at $parent(i)$ may not be valid heap. Here A is the array in which the heap is implemented and $HeapSize$ denotes the number of elements in the heap. Observe that the process ripples down to a leaf node.

Verify that the resulting sub-tree rooted at vertex i is a valid heap.

The time complexity of the procedue is $O(depth - depth(i))$ where $depth(x)$ denotes the depth of the node x, the number of edges on the path from the root

```
large := i;
if leftChild(i) ≤ HeapSize − 1 AND A[leftChild(i)] > A[i] then
 |  large := leftChild(i);
end
if rightChild(i) ≤ HeapSize − 1 AND A[rightChild(i)] > A[large] then
 |  large := rightChild(i);
end
if large ≠ i then
 |  Swap(A[large], A[i]);
 |  FixDown(A, large, HeapSize);
end
```

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

to i in the tree. *depth* is the depth of the tree, i.e., depth of the deepest node in the tree.

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 root (top) key, copy the value of $A[HeapSize]$ to the root, and then perform FixDown at the root. See algorithm 2.

```
Output A[0];
A[0] := A[HeapSize − 1];
HeapSize := HeapSize − 1;
FixDown(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. So we donot use $FixDown$ here. See Algorithm 3.

```
A[HeapSize] := x;
HeapSize := HeapSize + 1;
i := HeapSize − 1;
while  i > 0 AND key(i) > key(parent(i)) do
 |  swap(A[i], A[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 of 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 $i := parent(m-1)$ *Down to* 0 **do**
  | $FixDown(A, i, m)$;
end

Algorithm 4: BuildHeap(A, m)

Suppose the depth of the tree is d. Also suppose there are r nodes at depth d. So $n = (2^d - 1) + r$ where $1 \le r \le 2^d$. The cost of FixDown from a node at depth i is at most $(d-i)$. So the cost of FixDown for the nodes upto depth $d-2$ is $\sum_{i=0}^{d-2}(d-i).2^i$. The cost of $\lceil r/2 \rceil$ nodes at level $d-1$ (which have children) is $\lceil r/2 \rceil$. So the total cost is

$$
\begin{aligned}
Cost \le & c(\sum_{i=0}^{d-2}(d-i).2^i + \lceil r/2 \rceil) \\
= & c(d.\sum_{i=0}^{d-2} 2^i - \sum_{i=1}^{d-2} i.2^i + \lceil r/2 \rceil) \\
= & c(d(2^{d-1} - 1) + \lceil r/2 \rceil - 2.d(\sum_{i=1}^{d-2} x^i)/dx|_{x=2}) \\
= & c(d(2^{d-1} - 1) + \lceil r/2 \rceil - 2.d((x^{d-1} - x)/(x-1))/dx|_{x=2}) \\
= & c(d(2^{d-1} - 1) + \lceil r/2 \rceil - 2((d-1)2^{d-2} - 1) + 2(2^{d-1} - 2)) \\
= & c(3.2^{d-1} - 2 - d + \lceil r/2 \rceil) \\
\le & c.3n/2
\end{aligned}
$$

So the time complexity of $BuildHeap$ is $O(n)$.

Exercise:

1. Given a max-Heap in which the key of each node its priority. Does it act a priority queue, i.e., if there are more than one nodes with the same priority, then does $DeleteMax$ otuput the oldest of these highest priority node value? Justify your answer.

Explain a suitable modification so the max-Heap acts like priority queue.