Winter 2016 CMPS 101
Algorithms and Abstract Data Types
Written 3 Answer Key

Kivanc Yazan    Ryan Compton

Friday 12th February, 2016
1 Question 1 (1+1 pts)

Prove complete binary tree with height $h$ has $2^{h+1} - 1$ nodes. Then, prove $n$ element heap has at most $\left\lceil \frac{n}{2^{h+1} - 1} \right\rceil$ nodes of height $h$.

Part 1 (1 pt): Define $IH(h) = \text{all complete binary trees of height } h \text{ have } 2^{h+1} - 1 \text{ nodes}$.

Base case: Show $IH(0)$. By definition every complete binary tree of height 0 consists of a single root, and thus contains $1 = 2^{0+1} - 1$ nodes.

Inductive step: Assume $h > 0$ and that $IH(h-1)$ is true to show that $IH(h)$ is also true.

Let $T$ be an arbitrary complete binary tree of height $h$. Since $h > 0$, tree $T$ does not consist of a single root, so it must satisfy the second half of the definition. Therefore $T$ consists of a root, a left sub-tree $T_L$ and a right sub-tree $T_R$ where $T_L$ and $T_R$ are disjoint complete binary trees of height $h - 1$. The number of nodes in $T$ is 1 for the root, plus the number of nodes in $T_L$ plus the number of nodes in $T_R$. However, using $IH(h-1)$, the number of nodes in $T_L = 2^{h-1}$ and the number of nodes in $T_R = 2^{h-1}$. Therefore the number of nodes in $T$ is: $1 + 2^{h-1} + 2^{h-1} = 2^{h+1} - 1$, showing $IH(h)$.

Part 2 (1 pt):

Let $h$ and $n$ be arbitrary non-negative numbers and let $S$ be the set of nodes at height $h$ in an heap. No nodes in $S$ can be an ancestor or descendant of another node in $S$ (since otherwise they would have different heights). Because of the packed shape of the heap, all but one of the nodes in $S$ is the root of complete binary tree of height $h$ - only one node in $S$ can have leaves at two different depths in its sub-tree. Therefore the heap contains at least $|S| - 1$ disjoint complete binary height $h$ sub-trees plus at least one additional node (for the node at height $h$ that may not be the root of a complete binary sub-tree). Adding up these nodes we get:

$$n \geq (|S| - 1)(2^{h+1} - 1) + 1$$

$$\tfrac{n}{2^{h+1} - 1} \geq |S| - 1 + \frac{1}{2^{h+1} - 1}$$

$$\left\lceil \frac{n}{2^{h+1} - 1} \right\rceil \geq \left\lceil |S| - 1 + \frac{1}{2^{h+1} - 1} \right\rceil$$

$$\left\lceil \frac{n}{2^{h+1} - 1} \right\rceil \geq |S| \quad (1)$$
2 Question 2 (1+1 pts)

Show steps of heapsort and quicksort for [6, 2, 9, 5, 7, 10, 4].
3 Question 3 (1 pt)

Write $\text{INSERT}(S, x)$, $\text{MAXIMUM}(S)$, $\text{EXTRACT-MAX}(S)$, $\text{INCREASE-KEY}(S, x, k)$ with linked-list. Show their time complexities and compare with implementation of class.

**INSERT**

Algorithm 1 Insertion

```plaintext
1: procedure INSERT(S,x)  
2:     New_Node = new Node(x)  // A new node for the linked list is created. Parameter x is set as it’s value
3:     Current_Node = S.Head // Current Node points to the head of the list
4:     Parent = NULL  
5:     if Current_Node.value > New_Node.value then  // Check if the head’s priority is higher than the new node’s
6:         while Current_Node.Next.value > New_Node.value do  // Lines 8 through 11 can take N time as it can loop through the entire list
7:             Parent = Current_Node  
8:             Current_Node = Current_Node.Next  
9:         end while
10:     end if
11:     New_Node.Next = Current_Node  
12:     if Parent != Current_Node then  // Lines 13 through 17 take constant time
14:     end if
15: end procedure
```

Runtime Analysis:
Lines 2 through 6 take constant time
Lines 8 through 11 can take N time as it can loop through the entire list
Lines 13 through 17 take constant time
Therefore $F(n) = n$
Proof: $F(n) = O(n)$
$F(n) \leq c \cdot n$
Given a constant $c = 2$
$n \leq 2 \cdot n$ for an initial $n_0 = 1$

A Heap implementation of insert takes $O(\log(n))$ time due to keeping the structure of the heap, which takes less time than maintaining the order within a linked list ($O(n)$).

**MAXIMUM**

Runtime Analysis:
Line 1 takes constant time
Since returning the value of the head of the list takes only constant time, therefore $F(n) = 1$
Algorithm 2 Find the Maximum Priority Element

1: procedure MAXIMUM(S) 
2: RETURN S.Head.value \Comment{Return the first value within the list} 
3: end procedure

Proof: \( F(n) = O(1) \)
\( F(n) \leq c \cdot 1 \)
Given a constant \( c = 2 \)
\( 1 \leq 2 \cdot 1 \)

A Heap implementation will also take \( O(1) \) time to find the maximum, when the heap is a max heap, therefore the top of the heap will also contain the maximum just like the linked list.

EXTRACT-MAX

Algorithm 3 Extract the Max Priority Element

1: procedure EXTRACT-MAX(S) 
2: value = MAXIMUM(S) \Comment{Get the MAX value} 
3: S.Head = S.Head.Next \Comment{Make Head Node the next node in the list} 
4: RETURN value \Comment{Return the max value} 
5: end procedure

Runtime Analysis:
Lines 2 through 4 run in Constant Time Since it was shown that MAXIMUM runs in constant time, as well as setting the head of the list and returning value running in constant time
Therefore \( F(n) = 1 \)
Proof: \( F(n) = O(1) \)
\( F(n) \leq c \cdot 1 \)
Given a constant \( c = 2 \)
\( 1 \leq 2 \cdot 1 \)

A Heap implementation will take \( O(\log(n)) \) time as it will need to remove the Max element and then reorder the heap to maintain the Max-Heap property. While the linked list only needs to set a new head.

INCREASE-KEY

Runtime Analysis:
Lines 2 through 6 take constant time
Lines 7 through 10 take linear time since it can run until the end of the list if \( x \) is at the end of the list
Line 11 takes constant time
Line 12 takes linear time as proven before
Therefore \( F(n) = n + n = 2n \)
Algorithm 4 Increase element X to value K

1: procedure INCREASE-KEY(S,x,k)  
2:      \triangleright This Function assumes that x is within S and k is greater than x  
3:   if S.Head.value != x then  \triangleright Only need to increase x if it is not the head, otherwise it cannot be increased  
4:         Parent == NULL  
5:         Current_Node = S.Head  
6:   while Current_Node.Next.value != x do  
7:       Parent = Current_Node  
8:       Current_Node = Current_Node.Next  
9:   end while  
10:   Current_Node.Next = Current_Node.Next.Next  \triangleright Delete the Node with x  
11:   INSERT(S,k)  \triangleright Insert K into S  
12:   end if  
13: end procedure  

Proof: \( F(n) = O(n) \)
\( F(n) \leq c*n \)
Given a constant \( c = 3 \)
\( 2n \leq 3*n \) for an initial \( n_0 = 3 \)

A Heap implementation will only take \( O(log(n)) \) time since it can find a element within \( log(n) \) time, while also inserting an element in \( log(n) \), which is much faster than the linked list having to reinsert into the list.
4 Question 4 (1 pt)

Write pseudocode for deleting an element (given index) from a max binary heap. Give the time complexity.

Runtime Analysis:
Lines 2 through 5 take constant time Lines 6 through 20 will be called at most logn time since it is iterating through the heap by children of the element k which at most can be logn deep since a heap is always a complete binary tree.

\[ F(n) = 2 \times \log n + C \]

Proof: \( F(n) = O(\log n) \)

\[ 2 \times \log n + C \leq C_1 \times \log n \]

Given \( C_1 \) is 3 then:

\[ F(n) \leq C_1 \times \log n \] with an initial \( n_0 \) of 2
Algorithm 5 Delete element $k$

1: procedure DELETE(k) $\triangleright$ This function is assuming a 0 based index 

This function assumes that a global variable named "Heap" is the array holding the heap

2: if $k ==$ Heap.length-1 then $\triangleright$ If $k$ is the last element of the Heap, then just remove it

3: Heap[k] == NULL

4: else if $k \geq 0$ and $k <$ Heap.length then $\triangleright$

This function only needs to execute if $k$ is within the range of indices within Heap

5: End = Heap[Heap.length-1] $\triangleright$ Get the last element within the array

6: Heap[k] = End $\triangleright$ Replace the value at $k$ with the value at the end of the array

7: Heap[End] = NULL $\triangleright$ Set element at End to NULL

8: $k =$ MoveElementUp(Heap,k) $\triangleright$

Element must be moved up in heap if it is less than its parent

9: while (Heap[2k+1] != NULL or Heap[2k+2] != NULL) and (Heap[k] < Heap[2k+1] or Heap[k] < Heap[2k+2]) do $\triangleright$

10: Check that element $k$ has at least one child and If one of the children of the new element at $k$ is less than the new value at $k$, then the max of the children need to be swapped with the element at $k$

11: if Heap[k] < Heap[2k+2] and Heap[k] < Heap[2k+1] then $\triangleright$

Check if both children are greater than the element at $k$, if so then swap the max between the two children

12: if Heap[2k+1] > Heap[2k+2] then
13: Swap elements at $k$ and $2k+1$
14: $k = 2k+2$
15: else
16: Swap elements at $k$ and $2k+2$
17: $k = 2k+2$
18: end if
19: else if Heap[k] < Heap[k2+2] then $\triangleright$

20: It is known both children aren’t greater than the element at $k$ but due to the while loop logic we know one of the children is greater than element $k$. Find which one and swap it with $k$

21: Swap elements at $k$ and $2k+2$
22: $k = 2k+2$
23: else
24: Swap elements at $k$ and $2k + 1$
25: $k = 2k + 1$
26: end if
27: end while
28: end if
29: end procedure
Algorithm 6 Move k up the Heap

1: procedure MoveElementUp(S,k) 
2:   if k mod 2 == 0 then 
3:     if S[k] < S[(k-2)/2] then 
4:       Swap elements at k and (k-2)/2 
5:       RETURN MoveElementUp(S,(k-2)/2) 
6:     end if 
7:   else 
8:     RETURN k 
9:   end if 
10: else 
11:   if S[k] < S[(k-1)/2] then 
12:     Swap elements at k and (k-1)/2 
13:     RETURN MoveElementUp(S,(k-1)/2) 
14:   end if 
15: else 
16:   RETURN k 
17: end if 
18: end procedure 

5 Versions

Verison 2. Updated Algorithm in problem 4 to better hold to constraints of the problem Verison 3. Added helper function to algorithm in problem 4