CSC263: Problem Set 1

Angus Fung,[1] Syed Kamran,[1] Michelle Leung[1]

[1]Division of Engineering Science, University of Toronto,

Question 1

(a) Suppose an item with priority chosen uniformly at random between 1 and 50 is inserted into the heap tree. Since the heap must follow the property of *completeness*, the node is inserted at the end of the array (the right child of 30). From the heap property, it can be ascertained that no swaps will occur under two conditions: (i) the inserted value is less than 30 and (ii) the inserted value is equal to 30, which is the boundary condition stipulated by the assignment.

The number of input values that satisfy (i) and (ii) are 30. Therefore, the probability that no swaps will occur is $\frac{30}{50}$ or $\frac{3}{5}$.

(b) By inspection, the input value range that would lead to one, two, and three *swaps* are 31 to 35, 36 to 45, and 46-50, respectively. Note that the maximum number of possible swaps is no more than $\lceil \log_2 n \rceil - 1$, where n is the number of nodes.

Let $T(n)$ be the random variable representing the number of swaps that occur, then the expected number of swaps is given by the following:

$$\begin{aligned}\mathbb{E}[T] &= \sum_{t=1}^{3} t \cdot P[T = t] \\ &= 1 \cdot \frac{5}{50} + 2 \cdot \frac{10}{50} + 3 \cdot \frac{5}{50} \\ &= \frac{1}{10} + \frac{2}{5} + \frac{3}{10} \\ &= \frac{8}{10} \\ &= \frac{4}{5}\end{aligned}$$

Question 2

(a) Suppose we have a heap of size n, $n \geq 3$ and that the heap is full. Since the *heap* property does not place any restrictions on the left or right child, then any node at the bottom level of the tree (or height k, where k is numerated from top to bottom) could contain the value with the minimum priority. The number of nodes at the bottom level can be calculated by totaling the number of nodes from height 1 to k, and subtracting that with the number of number of nodes from height 1 to $k-1$. Let n_m be the exact number of nodes that could contain the value of minimum priority, then:

$$\begin{aligned}
n_m &= \sum_{n=0}^{k-1} 2^n - \sum_{n=0}^{k-2} 2^n \\
&= \frac{1-2^k}{1-2} - \frac{1-2^{k-1}}{1-2} \\
&= 2^k + 1 - 2^{k-1} - 1 \\
&= 2^k - 2^{k-1} \\
&= 2^{k-1}
\end{aligned}$$

(b) By inspection of the function definition of ExtractMin, we determine that the most computationally extensive calculation is determining the min_index value. In order to determine the min_index the function needs to perform at least 2^{k-1} heap accesses to check all the leaves at the bottom of the heap.We can use our definition of n to express 2^{k-1} in terms of n.

$$\begin{aligned}
n &= 2^k - 1 \\
n + 1 &= 2^k \\
k &= \log_2(n+1)
\end{aligned}$$

Substituting k in our expression for the minimum number heap accesses required we obtain:

$$\begin{aligned}
2^{k-1} &= 2^{\log_2(n-1)-1} \\
&= \frac{n-1}{2} \\
&= \Omega(n)
\end{aligned}$$

(c) Sample code:

```
def ExtractMin(heap):

    # assume index starts at 1
    min_index = 1
    smallest_key = heap[1].smallest

    while heap[min_index].priority != smallest_key:
        left_node = min_index
        right_node = min_index*2

        #check if the child nodes are smallest
        if heap[left_node].priority == smallest_key:
            min_index = left_node
            break
        elseif heap[right_node].priority == smallest_key:
            min_index = right_node
            break

        #if the child nodes of the parent aren't the smallest
        #then oneof the child's children must be the smallest
        #check which child has the smallest child
        if heap[left_node].smallest == smallest_key:
            min_index = left_node
        else:
            min_index = right_node

        #Now that we have the min_index we store it in temp and
        #remove it from the heap
        temp = heap[min_index]
        heap[min_index] = heap[heap.size]
        heap.size = heap.size - 1
    #Ensure heap is still a heap
    i = min_index
    while i > 1:
        curr_p = heap[i].priority
        parent_p = heap[i // 2].priority

        if curr_p <= parent_p:
            #ensure smallest attribute is correctly set
            if heap[i].smallest < curr_p:
                heap[i // 2].smallest = heap[i].smallest
            else:
                heap[i // 2].smallest = curr_p

            i = i // 2

        else:
            #swap priorities
            heap[i], heap[i // 2] = heap[i // 2], heap[i]

            #ensure smallest attribute is correctly set
            if heap[i].smallest < heap[i].priority:
                heap[i // 2].smallest = heap[i].smallest
            else:
                heap[i // 2].smallest = heap[i].priority

            i = i // 2

    return temp
```

Note: The above code assumes that the smallest attribute of all bottom leaves is inf.

Justification: The code above traverses through the heap taking advantage of the smallest attribute to find the index of the smallest priority node in $\log_2(n)$ time. Once the index of the smallest priority node is found, the node is removed from the heap in unit time. Lastly a "bubble up" operation is used to restore the heap property and to ensure the smallest attribute of each node is correctly set in $\log_2(n)$ time. The overall algorithm should run in $\mathcal{O}(\log_2(n))$ time.

Question 3

(a) If we perform 2^{k-1} $ExtractMax$ operations, and each operation moves the last leaf to replace the existing root before bubbling the notes down to satisfy the heap property, then the remaining tree is still complete but the highest priority 2^{k-1} will have been removed. Since the remaining tree has $(2^{k-1} - 1$ nodes, which is equal to the number of non-leaf nodes in the original tree, we know that the remaining nodes must all be of priority P since the original tree had the property where all of its leaves have priority equal to P,.

(b) We know that for the first $ExtractMax$ operation where the last leaf takes the place of the root, $k - 2$ swaps is needed to bubble the top most node to the second last level, after which no more swaps occur because we know all the nodes of the last level are of priority P, which is the same as the current node being bubbled. We know that exactly half of the nodes in the last level is the same number of nodes int the previous level, and thus can be expressed as 2^{k-2} number of nodes, hence performing 2^{k-1} consecutive $ExtractMax$ operations requires at least half the nodes on the bottom most level bubbled down fill to the second last level, which is $(2^{k-2})(k-2)$. Therefore, at least $(2^{k-2})(k-2)$ node swaps are needed.

(c) When the size of the heap n is one less than a power of two, we know the tree is a full. Given the property that all its leaves have priority P where P is less than all other node priorities in the tree, and that the $HeapSort$ algorithm is similar to that of the $ExtractMax$ algorithm, we know that at least $(2^{k-2})(k-2)$ node swaps are needed to move $\frac{(n+1)}{4}$ nodes, which is equivalent to 2^{k-2} nodes, hence we have

$$\begin{aligned}
2^{k-2} &= \frac{(n+1)}{4} \\
k - 2 &= log_2(\frac{n+1}{4}) \\
&= \frac{log(n+1)}{log(2)} - 2
\end{aligned}$$

which is the number of array swaps per node. Then, the total number of array swaps, F(n), in the worst case would be

$$F(n) = (\frac{n+1}{4})(\frac{log(n+1)}{log(2)} - 2) + (\frac{n+1}{4^2})(log_2(\frac{n+1}{4^2})) + ... + (\frac{n+1}{4^{k-1}})(log_2\frac{n+1}{4^{k-1}})$$

Then, taking the first term and ignoring the constants, we see that $F(n) = \Omega(nlogn)$