CMPS 101: Spring 2016: HW 5

Authors: Tarum Fraz - 1349796
Andres Segundo - 1408968

Q1 (1 point):

Psuedo Code:

```
1.) Delete(i):
2.) Swap node i with last element in deepest level of min-heap // takes $O(1)$
3.) Remove node i // takes $O(1)$
4.) heapify(A, i): // takes $O(\log(n))$, where i is the swapped element
5.)         find child of A[i] w/smaller value
6.)         swap A[i] w/that child
7.)         Heapify(A, child)
```

Time Complexity: $\Theta(\log(n))$

The time complexity of $Delete(i)$ is $\Theta(\log(n))$ because the swap and remove of the node happen at most once in the algorithm, taking $O(1)$ time. The recursion takes $O(\log(n))$ time, so if we add all of the times up, we find that the total time complexity of removing from a min-heap is $\Theta(\log(n))$.

Q2 (2 points):

Option 2:

In order to solve this problem, it would make the most sense to build a separate array, that is of length n*k. We can then simply build an a min-heap, that takes the first element from each of the k arrays, and stores it. If we keep track of which list and index we are pulling from, we can pop the root of the min-heap into the array we created, and then using the pointer information we stored, inserting the next value of the array we took from, into our Min-heap. If we keep using this method, we would merge k sorted arrays, into one sorted array, in a efficient way.

Pseudo Code:

```
1.) merge($k_1, k_2, ..., k_n$) // takes as input all k sorted arrays
2.) initialize an array called sort of size n*k
3.) buildMinHeap(k) // Build of size k
4.) for i = 1 to k
5.)     rewrite the first element of every array with a pointer (k[1], i)
6.) for i = 1 to k
7.)     insert(Heap, k[1]) // insert first val of every k into heap
8.)     point to the new first element in every k
9.) for i = 1 to n
10.)    sort[i] = extractMin(Heap)// extract root of heap, put in array
11.)    if sort[i] = (k[1], i) // find the array we took from
12.)        insert(Heap, k[2]) // insert next val in array into Heap via pointer
13.)        point the pointer to next element in that list
14.) return sort
```

```
1.) buildMinHeap(k) // Build a min-heap
2.) heapsize = k // initialize heapsize to amount of sorted arrays
3.) return empty heap called Heap // we will use this to fill up k vals later
```

```
1.) minHeapify(A, i)
2.) left = LEFT(i) // assign the left and right child of of i
3.) right = RIGHT(i)
4.) if left $\leq$ A.heapsize and A[left] < A[i]
5.)     smallest = left // Mark the child that is smaller than its parent
6.) else
7.) if right $\leq$ A.heapsize and A[right] < A[i]
8.)     smallest = right
8.) if smallest $\neq$ i // maintain min-heap properties
10.)     Swap A[i] and A[smallest] // swap child smaller than parent
11.)     minHeapify(A, smallest)
```

```
1.) insert(Heap, k[i])
2.) heapsize = heapsize + 1 // make room for new element
3.) x = heapsize // start at new element
// iterate through heap and don't stop till child is larger than parent
4.) while x > 1 and Heap[x//2] > k[i] // if we are not at root & parent is larger
5.)     do Heap[x] = Heap[x//2] // swap with parent
6.)         x = x//2
7.) Heap[x] = k[i]
```

```
1.) extractMin(Heap)
2.) if heapsize < 1 , return "error"// check to make sure heap is not too small
3.) minimum = A[1] // min equals root in heap, put in variable for swap
5.) A[1] = A[heapsize] // swap first value with last in heap
6.) A[heapsize] = A[heapsize] - 1 // decrement heap, acts as partition
7.) minHeapify(A,1) // rebuild heap to maintain minheap properties
8.) return minimum // extraction of minimum
```

Time Complexity: $O(n \log(k))$

It takes $O(k)$ time to build the min-heap. Every time we extract the min from the min-heap, it takes us $O(\log(k))$ time. When we use the pointers to and insert the next element from the respect array into the heap, it also takes us $O(\log(k))$ time. All of the other elementary operations in the pseudo code happen a constant amount of times, therefor we will not include them in the complexity analysis. If we add up all of the times, we get $O(k + n\log(k)) = O(n\log(k))$ time. *Therefore, the time complexity of our algorithm is* $O(n\log(k))$.

Q3 (2 points):

Option 1:

Pseudo Code:

```
1.) Heapsort(A):
2.) If A.length == 1, return A // Array is trivially sorted
3.) buildMinHeap(A) // Call to function in order to build min-heap
4.) Initialize a new array of size A.length called heapsorted
5.)     for i = 1 to A.length // loop to iterate through heap
6.)     heapsorted[i] = extractMin(A) // sort array
7.)     return heapsorted // return the sorted array

1.) buildMinHeap(A) // Build a min-heap from given A
2.) heapsize = A.length
3.) for i = A.length//2 downto 1 // iterate downwards
4.)     minHeapify(A, i) // recursively call heapify to fill heap

1.) minHeapify(A, i)
2.) left = LEFT(i) // assign the left and right child of of i
3.) right = RIGHT(i)
4.) if left ≤ A.heapsize and A[left] < A[i]
5.)     smallest = left // Mark the child that is smaller than its parent
6.) else
7.) if right ≤ A.heapsize and A[right] < A[i]
8.)     smallest = right
8.) if smallest ≠ i // maintain min-heap properties
10.)     Swap A[i] and A[smallest] // swap child smaller than parent
11.)     minHeapify(A, smallest)

1.) extractMin(A)
2.) if heapsize < 1 , return "error"// check to make sure heap is not too small
3.) minimum = A[1] // min equals root in heap, put in variable for swap
5.) A[1] = A[heapsize] // swap first value with last in heap
6.) A[heapsize] = A[heapsize] - 1 // decrement heap, acts as partition
7.) minHeapify(A,1) // rebuild heap to maintain minheap properties
8.) return minimum // extraction of minimum
```

Heapsort is unstable simply because our output is based on removing elements from our created heap. Any original order of the elements was destroyed during the original creation of the heap. For example, let us consider the array **[2,3,4,4,8]**. When we call **buildMinHeap**, the two children of **2**, which is the root, are **3** and **4**. The two children of **3** are the second **4** and **8**. Here, the order of the two **4's** are reversed. Using this example, we see that lines 4-8 in **minHeapify(A)** are not accounting for the order of the numbers in the original array, when determining the smallest value. Since the order is not being preserved here, we can say that *Heapsort is unstable.*

Q3 (2 points):

Option 2:

It is impossible to design a new kind of min-heap that can handle Extract-Min operations in $O(1)$ time. This is because, although the actual extraction of the root, which is also the minimum, is $O(1)$, we have to move around some nodes (depending on the values in the heap), in order to maintain the Min-heap properties that actually make it a Min-Heap. When we extract the root, we swap the last element with the root, and then we compare the swapped element to both its left and right children. Since we know that the left child is of size $2i$ and the right child is of size $2i + 1$, we would have to check $2i, 2 * 2i, 2 * 2 * 2i, ..., n - 1 = 2^1 i, 2^2 i, ..., 2^{\log_2 n} i = O(\log_2 n)$. To find a way to do this in $O(1)$ time is not possible, because although we can extract in $O(1)$, we would always have to shift the heap in some sort of way to maintain the Min-Heap conditions. Since we cannot swap an unknown amount of elements with their children in $O(1)$ time, and still call it a Min-Heap, it is impossible.

Q4 (3 points): // discussed problem with Julian Salazar and his partner

Option 1:

Pseudo Code:

```
1.) Insert(A,n):
2.) y = NIL // empty node that will keep track of parent of x
3.) x = A.root // start from the top of subtree
4.) while x $\neq$ NIL:
5.)       y = x // set y as parent of x
6.)       x.size = x.size + 1 // update subtree size at each node on path
7.)       if n.key < x.key: // does node belong on left or right
8.)             x = x.left
9.)       else x = x.right
10.) n.parent = y // y becomes parent of x, determined in while loop
11.) if y == NIL: // n has no parent so it must be the root
12.)       A.root = n
13.) elseif n.key < y.key // determines which side of y n is on
14.)       y.left = n
15.) else y.right = n
```

```
1.) Search(n): // used in Delete(A,n)
2.) while x.key $\neq$ NIL:
3.)       if x.key < n.key:
4.)             x = x.left
5.)       elif x.key > n.key:
6.)             x = x.right
7.)       else
8.)             return x
9.)       return NIL
```

```
1.) Delete(A,n):
2.) x = Search(n) // x points to node n
3.) y = Predecessor(y) // stores predecessor of y, TAKEN FROM CLASS
4.) if x points to only node in tree then make tree empty
5.) elif x has no children:
6.)       if x is a right child:
7.)             x.parent.right = NIL // remove x
8.)       elif x is left child:
9.)             x.parent.left = NIL // remove x
10.)       while (x $\neq$ A.root): // while loop will traverse up path to maintain count
11.)             x = x.parent
12.)             x.count = x.count - 1
13.) elif x has 1 left child:
14.)       x.parent = x.left // removes pointer to x
15.)       while(x $\neq$ A.root) // while loop will traverse up path to maintain count
16.)             x = x.parent
17.)             x = x.size - 1
18.) else:
```

```
19.)     swap x, y values
20.)     if y has a left child:
21.)         y.parent points to y.left
22.)         splices y node
23.)     else:
24.)         if y is left child:
25.)             y.parent.left = NIL
26.)         elif y is right child:
27.)             y.parent.right = NIL
28.)     while (y $\neq$ A.root): // while loop will traverse up path to maintain count
29.)         y = y.parent
30.)         y.count = y.count - 1
```

```
1.) range(a,b):
2.) nodeA = node with biggest value < a
3.) nodeB = node with smallest value > b
4.) parentA = leftmost parent from nodeA
5.) parentB = rightmost parent from nodeB
6.) elements = root.size - parentA.size - parentB.size + nodeA.right.size + nodeB.left.size
```

Time Complexity for Insert and Search: $O(D)$
The time complexity of $Insert(A, n)$ and $Search(n)$ is $O(D)$ where D is the maximum depth of A. The worst case assumes that the tree is not perfectly balanced with out at least 2^i nodes at each depth. The while loops will reach at worst depth D giving $O(D)$.

Time Complexity for Delete(A, n): $O(D)$
Search(), Predecessor(), and the while loop in each case take $O(D)$ time. Everything else takes constant time $O(1)$. Total is $O(D) + O(D) + O(D) = 3[O(D)]$. Limit as D approaches infinity of (cD)/D = C which is $\leq$ const. Therefore Delete() is $O(D)$.

Time Complexity for Delete(A, n): $O(D)$
Searching for a particular node in a BST takes $O(D)$. This is because the code could go to depth D. parentA and parentB could take $O(D)$ time if the tree were built in a way where most of the elements are left nodes or right nodes. Therefore, we have 4 different $O(D)$ and T(n) = 4$O(D)$. Limit as D approaches infinity of 4[(cD)/D] = 4c which is $\leq$ const. Therefore Range(a, b) is $O(D)$.