Evan Louie
Austin Yen
CMPS 101
5/11/16

# Homework 5

ejlouie , Auyen

1. Delete (A, i)

```
Delete (A, i)
    n = A.length
    swap A[i], A[n]
    A[n] = NULL
    n = n - 1
    return Heapify (A, i)
```

```
Heapify (A, i)
    left = 2i , right = 2i+1
    n = len(A)
    if (left ≤ n) and A[left] < A[i]
        min = left
    else
        min = i
    if (right ≤ n) and (A[right] < A[min])
        min = right
    if (min != i)
        swap A[i], A[min]
        Heapify (A, min)
```

The time complexity of Delete is $\Theta(\log n)$. In the delete function, everything except for heapify runs in $\Theta(1)$ time. Heapify is what does the majority of the work. In heapify there is 1 recursive call that depends on the case, and the rest (swapping) runs in $\Theta(1)$ time. The children's subtrees have at most $2n/3$. The worst case occurs when the bottom level of the tree is half full.

* Adapted from CLRS

$T(n) \le T(2n/3) + c$

$a = 1 \quad b = 3/2 \quad f(n) = n^0 = 1$

$n^{\log_b a} = n^{\log_{3/2} 1} = n^0 = 1$

$f(n)$ equals 1, which is equal to $n^{\log_3 1}$.

So the second case applies therefore

$T(n) = \Theta(n^{\log_{3/2} 1} \log n) = \Theta(\log n)$

2.1. K-Sort (A, K)

```
K-Sort (A, K)
    B = Build-MaxHeap (A)
    return Modified-Heapsort (B, i, K)
```

```
Build-MaxHeap (A)
    size = A.length
    for i = ⌊n/2⌋ to 1
        Max-Heapify (A, i)

Max-Heapify (A, i)
    left = 2i , right = 2i + 1
    n = len(A)
    if (left ≤ n) and (A[left] > A[i])
        max = left
    else
        max = i
    if (right ≤ n) and (A[right] > A[max])
        max = right
    if (max != i)
        swap A[i] , A[max]
        Max-Heapify (A, max)
```

```
Modified-Heapsort (A, i, K)
    Asize = A.length
    initialize output array of size k
    Ksize = K.length
    for i = 1 to K
        max = A[1]
        M[Ksize] = max
        Ksize = Ksize - 1
        A[1] = A[Asize]
        Asize = Asize - 1
        Max-Heapify (A, 1)
    return M
```

The time complexity K-sort is the time it takes for Build-MaxHeap and Modified-Heapsort. In class we proved that to build a max heap, it takes $O(n)$. For Modified-Heapsort, we extract the max from the root, then place that value at the end of the array which take $O(1)$. After we called Heapify, to maintain the heap properties. We do this entire process K times since we only want to sort k elements. In the last problem we proved heapify takes $O(\log n)$ time. In this case it only takes $\log(k)$ times because we are only sorting the first k largest elements. The time complexity is $O(n + k\log k)$, since we call Build-MaxHeap and Modified-Heapsort runs heapify K times.

2.2 Sort-Arrays ($A_1$ ... $A_k$)

```
Initialize array B of size k
for i to k
    B[i] = A_i  // Add each array to B so that B is an array of arrays
Build-MinHeap(B) // Build min heap assuming it considers each inner array's first index
Initialize array C of size n
j = 0
for i to n
    min = B[0][j]  // After build heap, the min value is the first index of the first array in B
    Add min to array C  // extract the first index of the root array
    B[0].remove(B[0][j])
    Heapify(B, B[0])
return C
```

Time complexity of Sort-Arrays is $O(n\log k)$. This is because the for-loop runs for $n$ times, which is the total amount of elements in all arrays. Within this for loop we are calling heapify, which we defined in question 1 to run $O(\log n)$ times. In this case since the size of the heap is $k$, Heapify runs in $O(\log k)$. This for loop runs $O(n\log k)$. Building the min-heap, and adding each array to B both run in $k$ times. So the total time complexity would be $O(2k + n\log k) = O(n\log k)$.

Q3.1

- Heapsort (A)
  Initialize new array B
  Build Heap(A)
  for i = 0 to A.length − 1
    min = Extract Min(A)
    B.append (min)
    Build Heap. (A)
  return B

Extract Min (A)
  min = A[0]
  A[0] = A[A.length − 1]
  A.remove (A[A.length − 1])
  return min

The stability of heap sort depends on the heapify function. If using a min-heap, one has to consider the child comparisons for a parent. If heapify decides to swap from the right side first when the left and right are equal, then heap sort is not stable. This can happen when you use " if (A[left] < A[right]) " to swap the parent with a child because it then swaps the right side in an else statement. To ensure that the left child is swapped first, " if (A[left] ≤ A[right]) " could be used.

Q3.2

- To extract something and maintain the min-heap, heapify must be called. Although the extraction by swapping the root with a leaf takes O(1) time, we must call heapify on the resulting heap so that the leaf doesn't stay at the root of the min-heap where it likely doesn't belong. Since the time complexity of heapify is O(log n), O(log n) is the best we can do for Extract Min.

Q4.

```
Insert (Root, A)
  if (Root = null)
    initialize new node with key A
  else if (A < Root.key)
    Insert (Root.left, A)
  else if (A > Root.key)
    Insert (Root.right, A)
```

Insert recursively moves down the tree until it finds an empty spot (a leaf). The recurrence to do this would be

$T(n) \le T(2n/3) + 1$

Master Theorem: $a=1$, $b=\frac{3}{2}$, $c=1$

$n^{\log_{1.5} 1} = n^0 = 1$, $c = \Theta(1)$

$\Rightarrow T(n) = \Theta(\log n) = \Theta(D)$

```
Delete (node, A)
  if (node.key = A)
    if node only has one child
      node = its child
      its child = null
    else if node has 2 children
      if the key of node.left ≥ key of node.right
        node = node.left
        Delete (node.left, A)
      else
        node = node.right
        Delete (node.right, A)
  else if node.key < A
    Delete (node.left, A)
  else
    Delete (node.right, A)
```

Delete has to search for any node with a key equal to A. The actual deleting process takes $O(1)$. Since Delete recursively searches, its worst case would involve traversing the depth of the tree. Like Insert, the reccurence is $T(n) \le T(2n/3)+1$. Therefore, like Insert, its time complexity is $\Theta(\log n) = \Theta(D)$

• Range(a, b)

```
Range(a, b)
    return FindRange(root, a, b)

FindRange(node, a, b)
  if (node.key <= b and node.key >= a)
      return 1 + FindRange(node.left, a, b) + FindRange(node.right, a, b)
  else if (node.key < a)
      return FindRange(node.right, a, b)
  else
      return FindRange(node.left, a, b)
```

Range calls Find Range. Find Range begins at the root. It recursively searches for any node with a key within the range given. When it finds such a node, it adds 1 and continues searching. In essence, it moves down the depth of the tree. Like Insert and Delete, its recurrence is $T(n) \leq T(^{2n}/_3) + 1$. Its time complexity would therefore be $\theta(\log n) = O(D)$

4.2 · BlockDelete(a)
    Block(root, a)

Block(node, a)
  if (node.key < a)
    Delete(node, node.key)
    Block(node.left, a)
    Block(node.right, a)
  if (node.key = a)
    Delete(node, node.key)
    Block(node.left, a)
  if (node > a)
    Block(node.left, a)

BlockDelete searches and deletes any node with a key less than or equal to a. Like Range, it can traverse the depth of the tree, starting from the root. Like Range, the movement down the tree is $\theta(\log n)$. However, Block calls Delete several times, or the number of nodes less than or equal to a. The worst case would be if a is equal to or greater than the largest key in the tree. Therefore, the time complexity is $\theta(n \log n + \log n)$, or $\theta(nD + D)$.