CSC263, Assignment #1

Brendan Neal[1], Filip Tomin[2]

[1]Section L0201 Student number: 1001160236

[2]Section L0201 Student number: 1001329984

1. (a) $T(n)$ is $O(n^2)$. Analysis:
Lines 1, 2 and 7 will only be executed once when nothing(A) is called.
The for-loop in line 3 runs from 3 to n, which means that it has $n-2$ iterations.
Within the for-loop, there are two constant-time lines (4 and 6 are assignment and comparisons, respectively), and another for-loop.
This inner for-loop on line 5 runs from 1 until $i-1$, and it runs for every iteration of the for-loop on line 3.
The for-loop on line 3 will exit early if $A[i] \neq s$, so the worst case is if this condition is never true.
Therefore, the total number of steps taken by this algorithm is going to be:
$(\sum_{i=3}^{n}(i-1+2))+3 = (\sum_{i=3}^{n}(i+1))+3 = (\sum_{i=1}^{n}(i+1) - \sum_{i=1}^{n}(2))+3 =$
$(\sum_{i=1}^{n}(i+1)-3-2)+3 = (\sum_{i=1}^{n}(i+1)-5)+3 = (\sum_{i=1}^{n}(i)+\sum_{i=1}^{n}(1)-5)+3 =$
$(\frac{(n(n+1)}{2}+n-5)+3 = \frac{1}{2}(n^2+n+2n-10)+3 = \frac{1}{2}(n^2+3n-4)$

Therefore, $T(n) = \frac{1}{2}(n^2+3n-4)$, which is $O(n^2)$ (pick $c=1$, $n_0=1$ for upper bound).

(b) Using the formal definition of Big-Omega, we can multiply n^2 by some $c \in \mathbb{R}^+$. In this case, we will use $c = \frac{1}{2}$ to cancel out the $\frac{1}{2}n^2$ at the front of $T(n) = \frac{1}{2}(n^2+3n-4)$. This means that we only need an input that shows that $n^2+3n-4 \geq n^2$. From the equation, any input with size $n \geq 2$ would work, but it must not terminate early from the if statement at line 6.
Lines 1 and 2 set the first two elements of any input to 0 and 1 respectively. From the effects of the for loop on line 5 and the conditions required for line 6 to not return early, it is necessary that the value at any index of the array A (where the index is ≥ 3) must equal the sum

of all previous values in the array. An array where each value at index i is 2^{i-3} satisfies this requirement. Using this input, the procedure never has an early return, and takes a full $T(n)$ steps for an array of size n. If $n \geq 2$, then $3n > 4$, thus $n^2 + 3n - 4 \geq n^2$.
Therefore, $T(n)$ is $\Omega(n^2)$.

2. (a) **Input that Causes Output Discrepancy:**
Pick $A = [1, 2, 3]$. Then:
Build-Max-Heap(A) produces $[3, 2, 1]$. Build-By-Inserts(A) produces $[3, 1, 2]$.
Walkthrough:

Build-By-Inserts(A):
First call of Build-By-Inserts:
A = $[1, 2, 3]$
HeapSize = 1
Heap = (1)

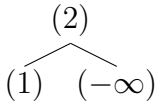
Build-By-Inserts starts it's for loop with $i = 2$ which calls Max-Heap-Insert$(A, A[i] = A[2] = 2)$:
A = $[1, -\infty, 3]$ | Max-Heap-Insert replaces A[2] with $-\infty$
HeapSize = 2 | Max-Heap-Insert increments HeapSize by 1
Heap =
(1)
|
$(-\infty)$

Max-Heap-Insert$(A, A[i])$ calls Heap-Increase-Key(A, A.HeapSize, key = 2):
A = $[1, 2, 3]$ | A[2] is changed from $-\infty$ to key = 2
HeapSize = 2
Heap =
(1)
|
(2)

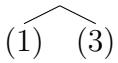
Max-Heap-Insert executes it's while-loop, it swaps the newly added child node with it's parent if the parent is less than the child:
A = $[2, 1, 3]$ | A[1] and A[2] are swapped by the loop
HeapSize = 2
Heap =
(2)
|
(1)

Build-By-Inserts increments i from 2 to 3 and calls Max-Heap-Insert$(A, A[i] = A[3] = 3)$:
A = $[2, 1, -\infty]$ | Max-Heap-Insert replaces A[3] with $-\infty$
HeapSize = 3 | Max-Heap-Insert increments HeapSize by 1

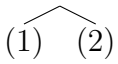
Heap =



Max-Heap-Insert($A, A[i]$) calls Heap-Increase-Key(A, A.HeapSize, key = 3):
A = [2, 1, 3] | A[3] is changed from $-\infty$ to key = 3
HeapSize = 3
Heap =

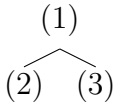


Max-Heap-Insert executes it's while-loop, it swaps the newly added child node with it's parent if the parent is less than the child:
A = [3, 1, 2] | A[2] and A[3] are swapped by the loop
HeapSize = 3
Heap =

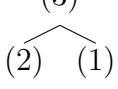


Build-Max-Heap(A):

First call of Build-Max-Heap: for-loop goes from 1 to $\lfloor A.length/2 \rfloor = \lfloor 3/2 \rfloor = 1$. Thus, only calls Max-Heapify(A, 1) once
A = [1, 2, 3]
HeapSize = 3
Heap =



Max-Heapify(A, 1): the left and right nodes of 1 are 2 and 3, respectively. After running through the if-statements, the largest node is determined to be the right node (3).
A = [3, 2, 1] | Max-Heapify swaps the largest node (3) with it's parent (1)
HeapSize = 3
Heap =



(b) **Runtime Analysis of Build-By-Inserts:**
Need to show two things to prove that $T(n)$ is $\Theta(n \log n)$:
$T(n)$ is $\Omega(n \log n) \wedge T(n)$ is $O(n \log n)$

$T(n)$ is $O(n \log n)$:

According to CLRS Ch 6-5's overview of Max-Heap-Insert (pg.164), Max-Heap-Insert has a worst-case time complexity of $O(\log n)$

According to the code for Build-By-Inserts, the for-loop runs from $i = 2$ to $i = n$ over a possible range of $\{1, 2, \ldots, n\}$, which means that the loop executes $|\{1, 2, \ldots, n\} \setminus \{1\}| = |\{1, 2, \ldots, n\}| - |\{1\}| = n - 1$ times.

Since each execution of the loop calls Max-Heap-Insert, Build-By-Inserts' execution time must be on the order of $(n-1) * \log n = n \log n - \log n$, which is $O(n \log n)$. Thus, the time complexity of Build-By-Inserts is $O(n \log n)$.

$T(n)$ is $\Omega(n \log n)$:

The code for Build-By-Inserts explicitly states that the loop will be executed from $i = 2$ to $i = n$, regardless of anything. Therefore, the minimum number of iterations that this loop will run through is $n - 1$ (using similar reasoning to the above)

The code for Max-Heap-Insert executes two constant-time lines (incrementation, assignment) and then calls Heap-Increase-Key(A, A.HeapSize, key).

The worst-case scenario for Heap-Increase-Key is if the parent of the newly added node is less than the newly added node. Therefore, the worst-case scenario for repeated calls of Heap-Increase-Key is if it is fed an increasing sequence of numbers (e.g. $[1, 2, 3, \ldots n]$).

Since each newly added node in this case would be the new maximum of the heap, and each newly added node would be at the "bottom" of the heap, this means that Heap-Increase-Key would have to switch the newly added node with it's parent repeatedly until the newest node is at the head of the heap to ensure that the heap satisfies the max-heap property.

Since the height of a heap with m nodes is $\lfloor \log_2 m \rfloor$, this means that in this scenario, each call to Heap-Increase-Key will result in the execu-

tion of $\lfloor \log_2 k \rfloor$ swaps, where k is the current number of nodes in the tree.

The total number of swaps needed to convert an n element array into a max-heap will be equal to:

$\sum_{k=1}^{n} \lfloor \log_2 k \rfloor \geq \sum_{k=\lceil n/2 \rceil}^{n} \lfloor \log_2 k \rfloor$ | Because the top-half of the sum of a positive sequence will be less than the entire sum.

$\sum_{k=\lceil n/2 \rceil}^{n} \lfloor \log_2 k \rfloor \geq \sum_{k=\lceil n/2 \rceil}^{n} \lfloor \log_2 (n/2) \rfloor$ | Because n/2 is the lowest possible value in the top half of the sequence.

$\sum_{k=\lceil n/2 \rceil}^{n} \lfloor \log_2 (n/2) \rfloor = \sum_{k=\lceil n/2 \rceil}^{n} \lfloor (\log_2 n) - 1 \rfloor$ | Simplifying the logarithm

$\sum_{k=\lceil n/2 \rceil}^{n} \lfloor (\log_2 n) - 1 \rfloor = \lfloor n/2 \rfloor * (log_2 n - 1)$ | Because the top half of the sequence has $\lfloor n/2 \rfloor$ terms

Therefore, a lower-bound for the worst-case time is $\lfloor n/2 \rfloor * (log_2 n - 1)$. Thus, the worst case time complexity of Build-By-Inserts is $\Omega(n \log n)$

Since it has been shown that Build-By-Inserts is $\Omega(n \log n)$ and $O(n \log n)$, it must be true that Build-By-Inserts is $\Theta(n \log n)$

3.

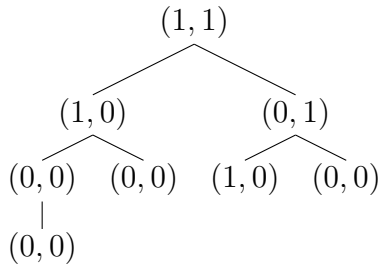
(a) **Description of Data Structure:**
We will be using a n-sized array of two bit tuples that is indexed like a heap, e.g. the parent of each node is at index $\lfloor \text{Index(node)}/2 \rfloor$, the left child of the node at index i is at $2i$, etc.

We will refer to the tuple in this array with index $i \in [1, n]$ as $A[i]$. The first and second elements of $A[i]$ will be referred to as $A[i][1]$ and $A[i][2]$, respectively.

$A[i][1]$ specifies whether or not $i \in S$ (1 if yes, 0 if no). $A[i][2]$ specifies whether or not $\exists D_i \in S$, where D_i is the first element of some descendent tuple of $A[i]$ (1 if yes, 0 if no).

For $n = 8$ and $S = 1, 2, 6$, the data structure D will look like:
$D = [(1,1), (1,0), (0,1), (0,0), (0,0), (1,0), (0,0), (0,0)]$

This is analagous to a binary tree that looks like:



Notice that the indices of nodes correlate directly to how far down/right a node is (i.e. the nodes associated with the largest values of I_n will be at the bottom of the tree in the rightmost portion of the tree).

Since this array is of n-size and is composed of tuples of 2 bits, it must take up $O(2n) \in O(n)$ bits of storage. Thus, the data structure uses only $O(n)$ bits of storage.

(b) **Execution of Insert and Maximum:**

Insert(j):

Insert(j) works as follows:
Assigns $A[j][1]$ to 1 (since now $j \in S$)

Re-assign the second element of A's ancestors' tuples to 1 (all located at $A[\lfloor \frac{i}{2^k} \rfloor]$), where k is the depth of the node in the tree.

Insert(j) takes $O(log_2 n)$ time: In the worst case, $A[j]$ will have a total of $(log_2 n)+1$ ancestors (if it is at the bottom of the tree), which means that the number of reassignments needed to keep the tree's tuple's consistent is $O(log_2 n)$.

Since reassigning the value at some array index takes $O(1)$ time, Insert(j)'s time complexity must be $O(((log_2 n) + 1) * 1) = O(log_2 n + 1) \in O(log_2 n) \in O(\log n)$. Thus, Insert(j)'s worst case time performance be in $O(\log n)$ time.

Delete(j) works similarily to Insert(j), but instead of assigning the appropriate bits (A[j] and it's ancestors' tuples) to 1, it assigns them to 0. Since the principle of it's operation is the same, Delete(j) must also be $O(\log n)$.

Maximum:
Maximum works as follows:

Check to see if $A[n][1]$ is 1. If it is, return n. Otherwise, recurse through the tree as follows:

Check if $A[1][2]$ is 1. If it is, check to see if either of $A[1]$'s children has a secondary value of 1. Go to the rightmost child of A[1] with a secondary value of 1 and repeat this process for all subsequent child nodes until the secondary value of both children is 0. This means that neither of the children have any more children, which means that we are at the bottom of the tree. Return the index of the rightmost node with value $(1, 0)$. This is the maximum.

$A[1][2]$ is not 1, it must be 0, which means that there are no elements in S, so return null, as there is no maximum in the empty set.

Maximum takes $O(log_2 n)$ time:

In the worst case, Maximum will fail the first step and will have to traverse down the tree. Since we are only checking two values with each execution of the process (left and right children's secondary values, which can be accessed in constant time), each execution will take constant time.

Since the height of an n-node binary tree is $O(log_2 n)$, the process will have to be executed $O(\log_2 n)$ times in the worst-case. Therefore, the total number of steps needed will be $O((\log_2 n)+1) \in O(\log_2 n) \in O(\log n)$.

Thus, Maximum's worst case time performance be in $O(\log n)$ time.

(c) **Execution of Member:**

Member(j) works as follows:
Return $A[j][1] == 1$

This works because if $A[j][1]$ is 1, it means that $j \in S$ (by the definition of the data structure)

Member(j) takes $O(1)$ time because accessing an array index takes $O(1)$ time and checking it's value takes $O(1)$ time. Thus, checking the value at some array index takes $O(1)$ time. Since Member(j) is checking if $A[j][1] == 1$, Member must take $O(1)$ time to execute.