Design and Analysis of Algorithms
CS575, Spring 2023

Theory Assignment 2.2

Due on 3/7/23 (Tuesday)

1. (15 points) We want to find the largest item in a list of n items.
   a) Use the divide-and-conquer approach to write an algorithm (pseudo code is OK). Your algorithm will return the largest item (you do not need to return the index for it). The function that you need to design is *int maximum (int low, int high)*, which *low* and *high* stands for low and high index in the array. (8 points)
   b) Analyze your algorithm and show its time complexity in order notation (using θ). (7 points)

int maximum (int low, int ~~maximum~~ high):
    if low = high:
        return array [low]
    else:
        mid = (low + high)/2
        left_max = maximum (low, mid)
        right_max = maximum (mid +1, high)

        if left_max > ~~right_max~~ right_max:
            return left_max
        else:
            return ~~right~~ right_max

— The recurrence relation can be expressed as, for the running time of the algorithm.

$$T(n) = 2T(n/2) + 1 . O(1).$$

This is because we make two recursive calls on arrays of size n/2 & each recursive call takes constant time to perform the ~~compar~~ comparisons and return the max value

$$T(n) = 2T(n/2) + \theta(1).$$

$a = 2 \quad b = 2 \quad \& \quad k = 0$

$b^k = 2^0 = 1$

$a > b^k$

Case 3 of Master's method.

$$T(n) = \theta(n^{\log_b a})$$

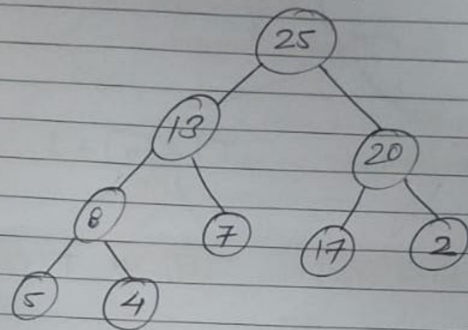
$$= \theta(n^{\log_2 2})$$

$$\boxed{T(n) = \theta(n)}$$

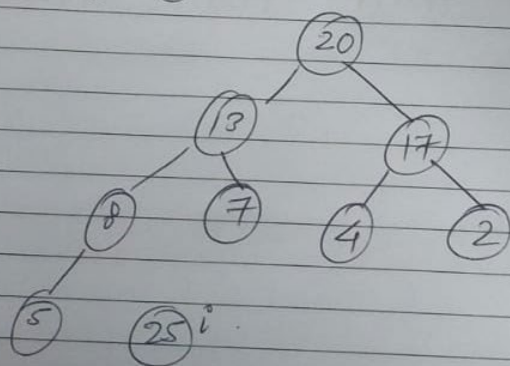
2. (15 points) Illustrate the operation of Heapsort on the input array A = <5, 13, 2, 25, 7, 17, 20, 8, 4>. Draw the heap just after Build-Max-Heap was executed. Then draw a new heap after another (the next) element has been sorted; the last heap you draw has a single element (see example figure in slide 24 of Ch6-sorting-heap-linear lecture notes).

- input array $A = \langle 5, 13, 2, 25, 7, 17, 20, 8, 4 \rangle$

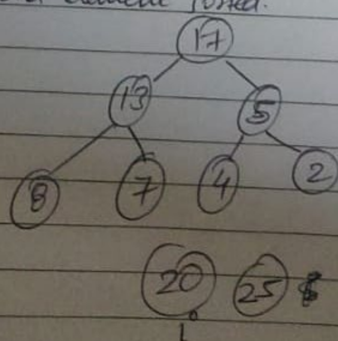
Heap after Build-Max-Heap



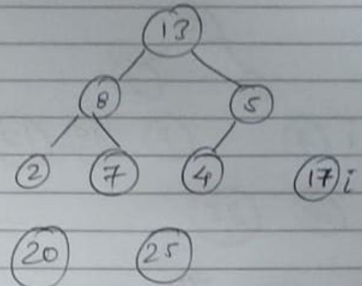
Heap after first ~~iteration~~ element sorted.



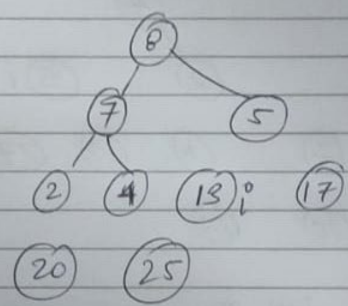
Heap after 2nd element sorted.



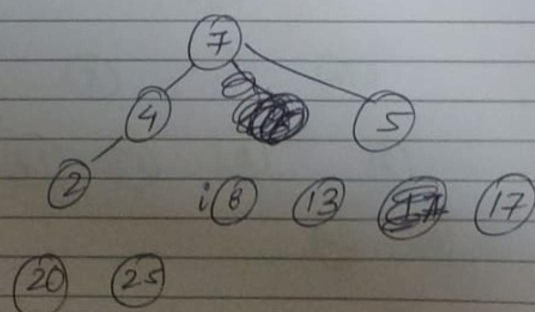
Heap after 3rd element sorted.



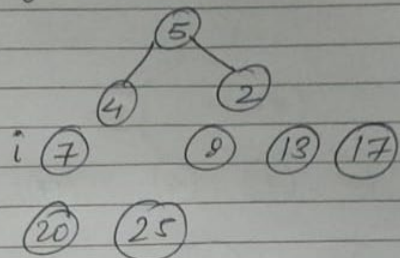
Heap after 4th element is sorted.



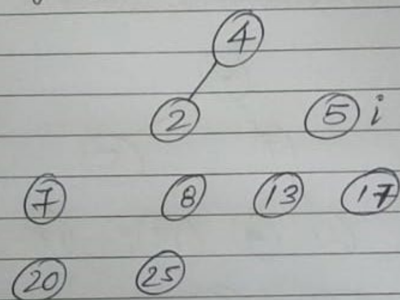
Heap after 5th element is sorted.



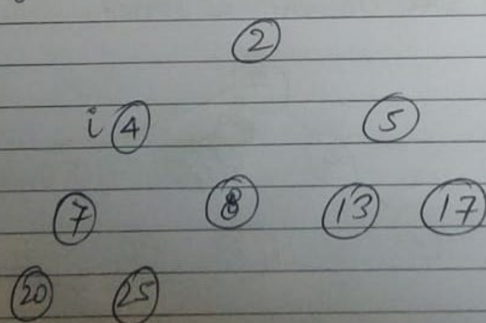
Heap ~~Element~~ after 6th element is sorted.



~~Elem~~ Heap after 7th element is sorted.



Heap after last element is sorted



The array after sorting {2, 4, 5, 7, 8, 13, 17, 20, 25}

3. (15 points) Argue for the correctness of Heapsort (the slide 25 of Ch6-sorting-heap-linear lecture notes) using the following loop invariant: At the start of the iteration with an i of the for loop, (a) the subarray $A[1 .. i]$ is a max-heap containing the i smallest elements of $A[1 .. n]$, and (b) the subarray $A[i+1 .. n]$ contains the $n - i$ largest elements of $A[1 .. n]$ in correctly sorted order (i.e., in ascending order). Divide your proof into the three required parts: Initialization, Maintenance, and Termination.

- Loop Invariant

At the start of the iteration with an ~~i~~ i of the ~~for~~ for loop, the subarray ~~$A[i-1]$~~ $A[1...i]$ is a max-heap containing the i smallest elements of $A[1....n]$, and the subarray $A[i+1....n]$ contains the $n-i$ largest elements of $A[1....n]$ in correctly sorted order (ie ascending order).

Initialization

Before the loop starts, i is initialized to n. ~~At~~ At this point, $A[1....n]$ is a ~~heap~~ heap (since we have called BUILD-MAX-HEAP), and $A[n+1....n]$ is ~~a~~ an empty ~~sub~~ subarray. Therefore, the loop invariant ~~hat~~ holds true.

Maintenance:

During the ith iteration, we start with a max heap containing the i smallest ~~element~~ elements of $A[1...n]$, where the largest element is at the root $A[1]$. After swapping $A[1]$ with $A[i]$, the ~~ith~~ ith largest element is now in its correct position at the end of the subarray $A[i...n]$. The subarray $A[1....i-1]$ still contains the $i-1$ smallest elements of $A[1....n]$, but $A[i...n]$ now contains $n-i+1$ largest elements of $A[1...n]$, which are not in sorted order necessarily.
To restore the max-heap property of the subarray $A[1..i-1]$ we ~~we~~ call MAX-HEAPIFY on $A[1]$, which moves the largest element of $A[1....i-1]$ to the ~~rot~~ root $A[1]$. This ensures that $A[1...i-1]$ is a max heap containing the

__/__/__

$i-1$ smallest elements of $A[1...n]$. Moreover, the ~~subover~~ subarray $A[i...n]$ still contains the ~~n+i-1~~ $n-i+1$ largest elements of $A[1...n]$, but now sorted order ~~beca~~ because we swapped the ith largest element to the end of the subarray. Therefore, after the ~~iteration~~ ith iteration, the loop invariant still holds.

Termination:

The loop ends when $i=1$, at which point ~~A[1]~~ $A[1]$ is the smallest element of $A[1...n]$. The subarray $A[2...n]$ contains the $n-1$ largest elements of $A[1...n]$ which are already sorted due to previous iterations of the loop. This ~~ma~~ means that the entire array $A[1...n]$ is now sorted permutation of the input array A. Therefore, the termination of the loop ~~satis~~ satisfies the desired outcome of the algorithm.