Name : Seetharaman Krishnamoorthy

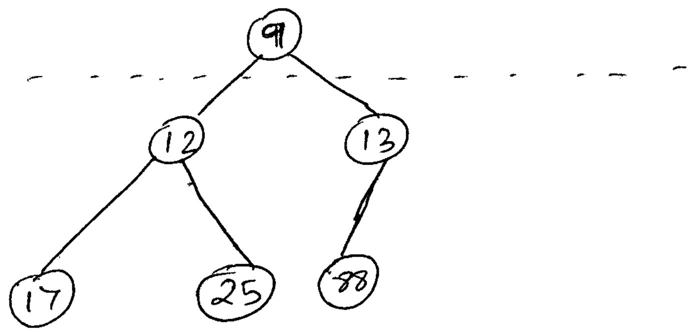
Id : N13274420

CS6033 Homework Assignment 2

1)

9	12	13	17	25	88

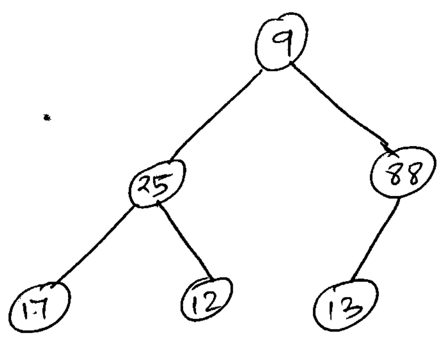
a)



BUILD-MAX-HEAP(A)
A. heap-size = A. length
for i = $\lfloor$A. length/2$\rfloor$ to 1
MAX-HEAPIFY(A, i)

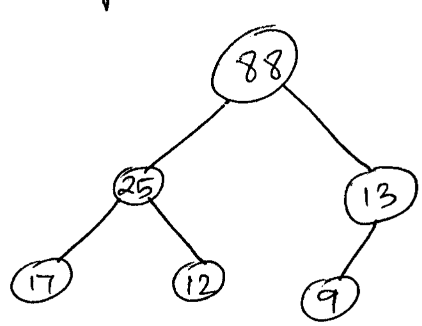
- First we take i = 3 and MAX-HEAPIFY it.
- Then we take i = 2 and repeat the same.

So,



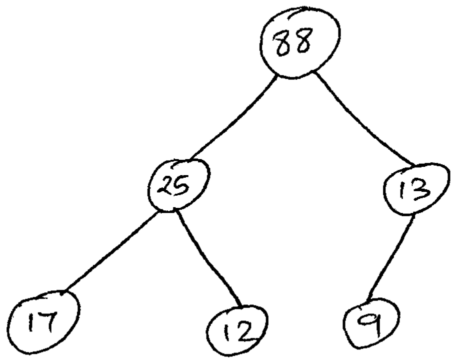
- And then we move top and take i = 1 and repeat the procedure.

Ans:



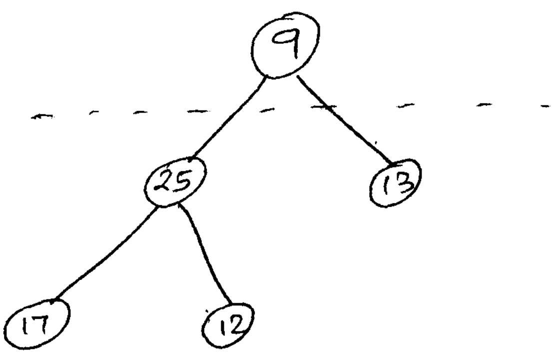
b)

```
HEAP-EXTRACT-MAX(A)
    if A.heapsize < 1
        error "heap underflow"
    max = A[1]
    A[1] = A[A.heapsize]
    A.heapsize = A.heap-size - 1
    MAX-HEAPIFY(A, 1)
    return max.
```

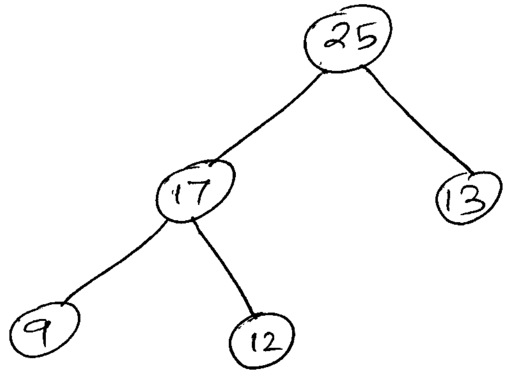


→ so extracting the root element which is the max of Array A, we get

max = 88



→ we do max-HEAPIFY on A[1] and so on to build the proper MAX-HEAP.



Ans: The array is $A = $

25	17	13	9	12

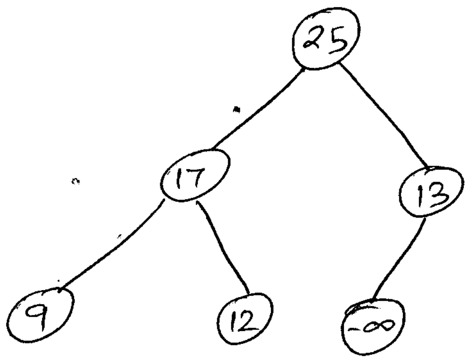
c) MAX-HEAP-INSERT (A, key)

A.heapsize = A.heapsize + 1

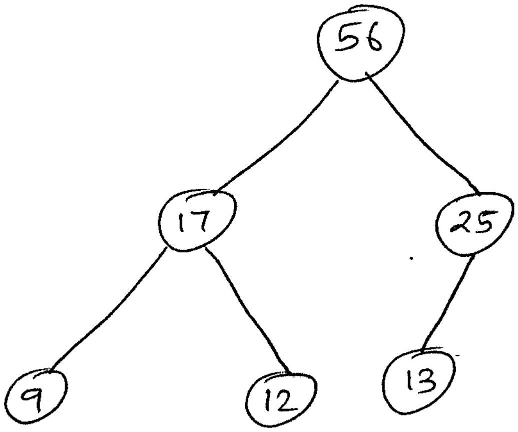
A[A.heapsize] = $-\infty$

HEAP-INCREASE-KEY (A, A.heapsize, key)

- Here our key is 56



- Now our key is placed in last point and its compared and exchanged with parent according to HEAP-INCREASE-KEY.
- This process is repeated until our key is inserted and we get a proper max-heap.



Ans: The Array is A = | 56 | 17 | 25 | 9 | 12 | 13 |

2) - we are given 'n' movies and we should watch best 'k' movies out of it with Higher ranks.

- This is clearly a priority queue problem and we solve it with MAX-HEAP.

Algorithm:

Array 'A' contains list of movies with size 'n'.

```
A.heapsize = A.length
for i = ⌊A.length/2⌋ downto 1
    MAX-HEAPIFY(A, i)

for i = 1 to k
    max = A[1]
```

$A[1] = A[A.heapsize]$

$A.heapsize = A.heapsize - 1$

MAX-HEAPIFY $(A, 1)$

return max

→ Here 'max' is the movie we watch.

→ So everytime we watch the movie with high rank.

→ <u>Running time</u>:

Building the max-heap	–	$\Theta(n)$
Extracting the highest rank movie from heap	–	$\Theta(k \log n)$

Total time $= \Theta(k \log n + n)$

3) a) $\text{Parent}(i) = \lfloor (i-1)/4 \rfloor$

$\text{child1}(i) = 4i+1$

$\text{child2}(i) = 4i+2$

$\text{child3}(i) = 4i+3$

$\text{child4}(i) = 4i+4$

- Assuming the index starts at '0'.

$\text{Parent}(i) = (4+i-2)/4$

$\text{child1}(i) = 4i-2$

$\text{child2}(i) = (4i-2)+1$

$\text{child3}(i) = (4i-2)+2$

$\text{child4}(i) = (4i-2)+3$

- Assuming the index starts at '1'.

b) - Each level has elements $4^0, 4^1, 4^2 ..$

- So total elements

$$n = 1 + 4 + 4^2 + \dots + 4^h$$

$$4n = 4 + 4^2 + 4^3 + \dots + 4^{h+1}$$

which gives

$$n = \frac{4^{h+1} - 1}{4 - 1}$$

So,

$$h = \ln(4n - n + 1) / \ln(4) - 1$$

$$\Rightarrow \quad h = \text{ceil}\left(\ln(4n-n+1) / \ln(4) - 1 \right)$$

c) <u>HEAP-EXTRACT-MAX</u>

```
if A.heapsize < 1
    error "heap underflow"
max = A[1]
A[1] = A[A.heapsize]
A.heapsize = A.heapsize - 1
MAX-HEAPIFY (A, 1)
return max
```

<u>Note:</u>
The implementation is similar to algorithm written.
<u>Ex:</u>
A.heapsize = n
(length of Array)

<u>MAX-HEAPIFY (A, i)</u>

```
child1 = CHILD1(i)
child2 = CHILD2(i)
child3 = CHILD3(i)
child4 = CHILD4(i)
if child1 ≤ A.heapsize and A[child1] > A[i]
    largest = child1
else largest = i
if child2 ≤ A.heapsize and A[child2] > A[largest]
    largest = child2
else largest = i
```

<u>Note:</u>
In implementation the CHILD(i) is found with formula on question (a)

if child3 $\leq$ A.heapsize and A[child3] > A[largest]

largest = child3.

else largest = i

if child4 $\leq$ A.heapsize and A[child4] > A[largest]

largest = child4

if largest $\neq$ i

exchange A[i] with A[largest]

MAX-HEAPIFY (A, largest)

- Here clearly the running time depends on MAX-HEAPIFY as other work done takes constant time.

- In MAX-HEAPIFY, we compare the node with all four children and we move down to the depth of tree.

$\Rightarrow$ Running time is $O(4 \log_4 n) \Leftrightarrow O(\log_4 n)$

Note:

$A.heapsize = n$

d) MAX-HEAP-INSERT (A, Key)

$A.heapsize = A.heapsize + 1$

$A[A.heapsize] = -\infty$

HEAP-INCREASE KEY (A, A.heapsize, Key)

- Here we know that HEAP-INCREASE Key runs in $O(\log_4 n)$ time.
- Since insert just adds element in constant time

Running time is $O(\log_4 n)$

e) HEAP-INCREASE-KEY (A, i, Key)

$A[i] = \max(A[i], Key)$

if $Key = A[i]$

while $i > 1$ and $A[\lfloor i/4 \rfloor] < A[i]$

do

Exchange $A[i]$ and $A[\lfloor i/4 \rfloor]$

$i = \lfloor i/4 \rfloor$

- Running time is $O(\log_4 n)$ since it is proportional to the depth of tree.

4) HEAP-DECREASE-KEY (A, i, key)

```
if key > A[i]
    error "new key is larger"
A[i] = key
while i > 1 and A[Parent(i)] > A[i]
    exchange A[i] with A[Parent(i)]
    i = Parent(i)
```

Loop Invariant : At the start of each iteration, each node i+1, i+2 ... n is the root of min-heap or is a leaf node.

Initialization : i would ~~either~~ be leaf node, so it satisfies the condition.

maintenance : At start of each iteration the subtree will be a min-heap.

Termination : At the end, i would be 1 and the entire tree would be a min-heap.

5) a) Here we divide the Array A into 2 different parts continuosly and then compare to find the maximum.

so

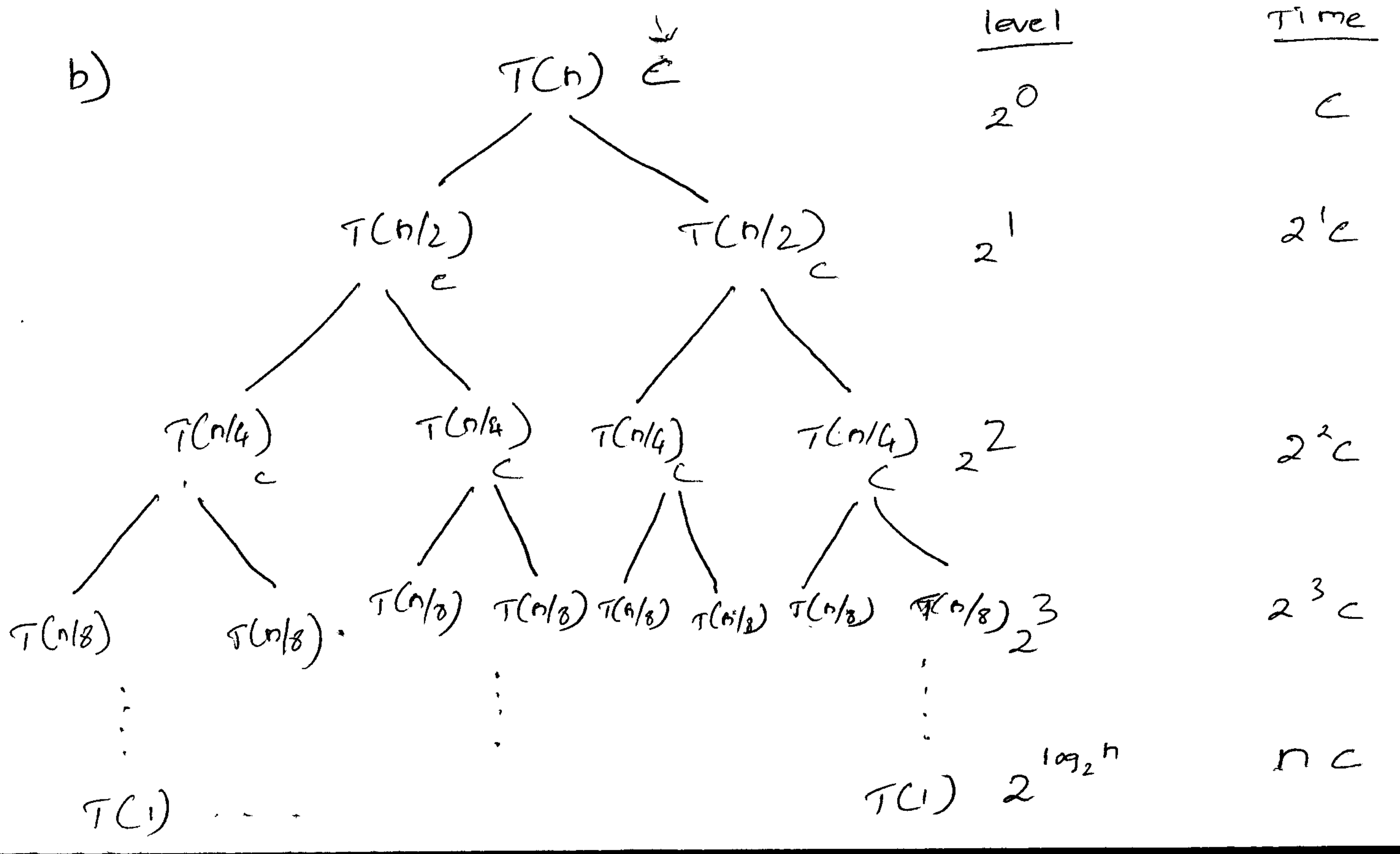
$$T(n) = 2(T(n/2)) + O(1)$$

since divide & combine takes constant time.

<u>Ans:</u>

$$T(n) = \begin{cases} O(1) & \text{if } n = 1 \\ 2T(n/2) + O(1) & \text{otherwise.} \end{cases}$$

b)



∴ So the Running time is

$$T(n) = \Theta(n)$$

c) CALCULATE-MEDIAN( )

```
MAX-HEAP = BUILD-MAX-HEAP(A)      // Constructing 2
MIN-HEAP = BUILD-MIN-HEAP (A)     // Heaps

if A.length == 1
    return median is A[1]
else if A.length == 2             // only for first
    item1 = A[1]                  // 2 items
    item2 = A[2]
    if item1 < item2
        MAX-HEAP-INSERT (MAX-HEAP, item1)
        MIN-HEAP-INSERT (MIN-HEAP, item2)
    else
        ~~MAX-INSE~~
        MAX-HEAP-INSERT (MAX-HEAP, item2)
        MIN-HEAP-INSERT (MIN-HEAP, item1)
else
    item = GET-ITEM (A)           // From 3rd item
    MAXROOT = HEAP-~~EXTRACT~~-MAXIMUM (MAX-HEAP)
    if item < MAXROOT
        MAX-HEAP-INSERT (MAX-HEAP, item)
```

```
else
        MIN-HEAP-INSERT (MIN-HEAP, item)

if   MAX-HEAP.size  -  MIN-HEAP.SIZE  > 1
        BALANCE-HEAPS (MAX-HEAP, MIN-HEAP)

MAXSIZE  =  MAX-HEAP.size
MINSIZE  =  MIN-HEAP.size
if   MAXSIZE == MINSIZE
        return   median  =  [ HEAP-MAXIMUM (MAXHEAP)
                              + HEAP-MINIMUM (MINHEAP) ] / 2
else
        if  MAXSIZE > MINSIZE
                return  median  =  HEAP-MAXIMUM(MAXHEAP)
        else
                return  median  =  HEAP-MINIMUM (MINHEAP)
```

BALANCE-HEAPS (A, B)

```
while (A.size - B.size > 1)
    i = HEAP-EXTRACT-MAX (A)
    MINHEAP-INSERT (B, i)
// remove root element from Heap with more
// elements and add it to other
```

d) Compare Binary trees with 4-ary trees and explain the scenario in which Binary Heap takes less time and prove it with example.