AUTUMN MID SEMESTER EXAMINATION-2016

Design & Analysis of Algorithm

[CS-3001]

Full Marks: 25 **Time: 2 Hours**

Answer any five questions including question No.1 which is compulsory.
The figures in the margin indicate full marks.
Candidates are required to give their answers in their own words as far as practicable and all parts of a question should be answered at one place only.

DAA MID-SEM SOLUTION & EVALUATION SCHEME

Q1 Answer the following questions: **(1 x 5)**

a) Consider the following C function.

```
int fun(int n)
{
  if (n<=2)
    return 1;
  else
    return  fun(√n);
}
```

Represent this function by recurrence and find out the time complexity.

Answer:	**Scheme**
The recurrence is T(n) = T(√n) + 1 Time Complexity = Θ(log log n)	Each answer 0.5, No explanation required

Explanation:

T(n) = T(√n) + 1 ---------- (1)

Solving this recurrence by change in variable

Let m = log n => n = 2^m = $n^{1/2}$ => $2^{m/2}$

Substituting the value of n in recurrence (1), the recurrence becomes

$T(2^m) = T(2^{m/2}) + 1$ ----------(2)

let T(n) = S(m) => $T(2^m) = S(m)$ => $T(2^{m/2}) = S(m/2)$

Now substituting this value in equation (2), the recurrence becomes

S(m) = S(m/2) + 1 ----------(3)

Recurrence (3) is similar to master theorem, applying master theorem

a=1, b=2, f(m)=1

$m^{\log_b a}$ =1, As per master theorem. Comparing $m^{\log_b a}$ and f(m), we found both are same, so it comes under case-2. As per case-2 of master theorem, the solution of the recurrence is

$S(m) = \Theta(m^{\log_b a} \log m) = \Theta(m^{\log_b a} \log m) = \Theta(\log m)$ ----------(3)

Now substitute the value of n back to this eq. (3)

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

b) Consider the following pairs of functions f(n), g(n). Which pair the functions are such, that f(n) is O(g(n)) and g(n) is not O(f(n)) ?

A. $f(n)=n^3$, $g(n) = n^2\log(n^2)$ **B. $f(n)=\log(n)$ $g(n) = 10 \log n$**

C. $f(n)=1$, $g(n) = \log n$ **D. $f(n)=n^2$, $g(n) = 10n \log n$**

Answer:	**Scheme**
Option: C	Only answer 1 mark

Explanation:

As per the definition of Big-O,

In $f(n)=n^3$, $g(n) = n^2\log(n^2)$ => $n^3 \leq cn^2\log(n^2)$ is false and reverse is true

In $f(n)=\log(n)$ $g(n) = 10 \log n$ => $\log(n) \leq 10 \log n$ is true and the reverse is also true

In $f(n)=1$, $g(n) = \log n$ => $1 \leq c \log n$ is true and the reverse is false

In $f(n)=n^2$, $g(n) = 10n \log n$ => $n^2 \leq 10n \log n$ is false, but the reverse is true.

c) Which of the following sorting algorithms in its typical implementation gives best performance when applied on an array which is sorted, or almost sorted (at most two elements are misplaced).

A. Quick Sort **B. Heap Sort** **C. Merge Sort** **D. Insertion Sort**

Answer:	**Scheme**
Option: D If sorted is assumed in proper order Option: B and D If sorted is assumed in reverse order	Either answer 1 mark

Explanation:

sorted in proper order (almost)

- Quick sort performs worse case time complexity ($O(n^2)$), Merge and Heap performs its best case time complexity ($O(n \log n)$) and Insertion sort performs its best case time complexity that is $O(n)$. Hence Insertion sort will give the best solution.

sorted in reverse order (almost)

- Quick sort and insertion sort performs worse case time complexity ($O(n^2)$), Merge and Heap performs its best case time complexity ($O(n \log n)$) and Hence Merge and Heap sort will give the best solution.

d) A priority queue is implemented as a Max-Heap. Initially, it has 5 elements. The

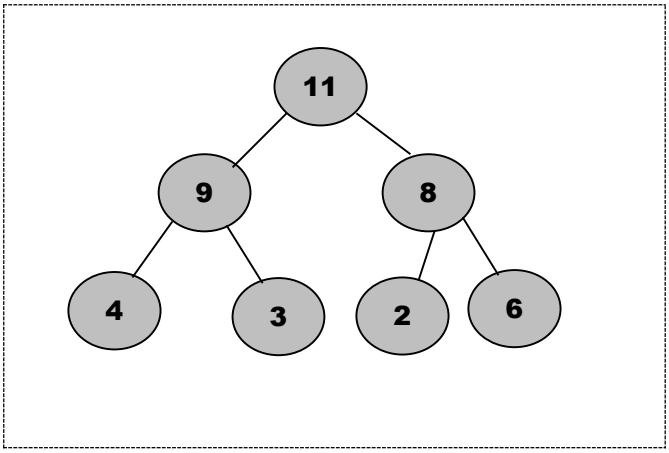
level order traversal of heap is as 11, 9, 6, 4, 3. Two new elements 2 and 8 and inserted in the heap in that order. Find out the level order traversal of the heap after the insertion of the elements?

Scheme:

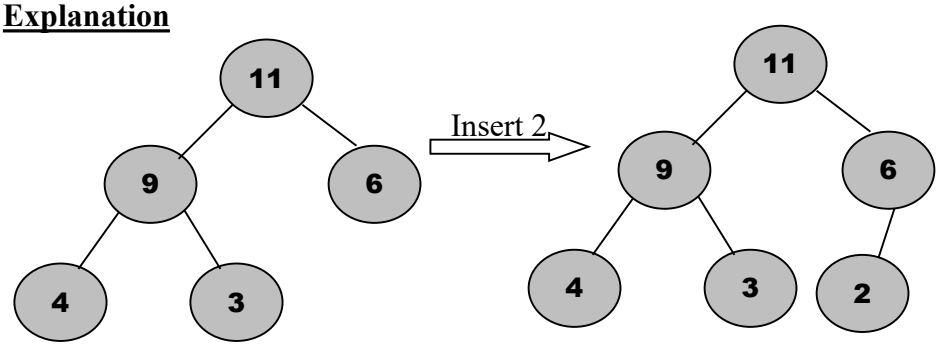
- Only answer : 1 Mark

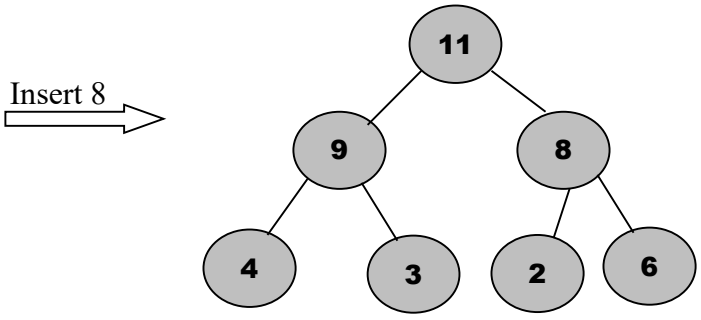
Answer:

The level order traversal of the heap after the insertion of the elements 2 and 8 is: 11, 9, 8, 4, 3, 2, 6



Explanation





e) What is the difference between Divide & Conquer and Greedy algorithm?

Scheme:

- Difference – 1 Mark

Answer:

Divide and Conquer (D-n-C) Algorithm

- In this approach, the problem is broken into several sub problem that are similar to the original problem but smaller in size, the sub problems are solved recursively, and then these solutions are combined to create a solutions to the original problem.
- The divide and conquer paradigm involves 3 steps at each level of the recursion.

  Divide the problem into a number of sub problems.

  Conquer the sub problems by solving them recursively. If the sub problem sizes are small enough, however, just solve the sub problems in a straight forward manner.

  Combine the solutions to the problems into the solution for the original problem.

Greedy Algorithm

- An optimization problem is one in which you want to find, not just *a* solution, but the *best* solution.
- A "**greedy algorithm**" sometimes works well for optimization problems.
- A greedy algorithm works in phases. At each phase:
  i. You take the best you can get right now, without regard for future consequences.
  ii. You hope that by choosing a *local* optimum at each step, you will end up at a *global* optimum.

Q2 a) What is purpose of algorithm analysis? Find out the complexity of the given function $\sum_{k=0}^{n-1} 2^k$. **(2.5)**

Answer:

Scheme:
Purpose of algorithm analysis – 1 mark
Time complexity – 1.5 mark

Purpose of Algorithm Analysis

- In computer science, the analysis of algorithms is the determination of the amount of resources (such as time and storage) necessary to execute them.
- The purpose of analyzing an algorithm is to discover its characteristics in order to evaluate its suitability for various applications or compare it with other algorithms for the same application. Moreover, the analysis of an algorithm can help us understand it better, and can suggest informed improvements. Algorithms tend to become shorter, simpler, and more elegant during the analysis process.

Time complexity of the function:

$\sum_{k=0}^{n-1} 2^k = (2^{n-1+1}-1)/(2-1) = 2^n-1 = O(2^n)$

b) Solve the following recurrence. **(2.5)**

$T(n) = \sqrt{n}\, T(\sqrt{n}) + n$

Assume that T(n) is constant for sufficiently small n.

Scheme:

- Correct answer with necessary steps : 2.5 Marks
- Incorrect answer with some valid steps – step marking (0.5 to 2)

Answer:

$$T(n) = \sqrt{n}\, T(\sqrt{n}) + n$$

Dividing by n, we get

$$\frac{T(n)}{n} = \frac{T(\sqrt{n})}{\sqrt{n}} + 1$$

Putting $n=2^m$, we can get

$$\frac{T(2^m)}{2^m} = \frac{T(2^{m/2})}{2^{m/2}} + 1$$

Let,

$$S(m) = \frac{T(2^m)}{2^m}$$

Now the original recurrence becomes

S(m)=S(m/2)+1

This looks like master theorem, soling we can get

S(m) = Θ(log m)

Now returning from S to T with $n=2^m$ => m=log n, we have

S(m) = Θ(log m) => $\frac{T(n)}{n} = \Theta(\log\log n)$ => **T(n)= Θ(n log log n)**

Q3 a) Solve the following recurrence. **(2.5)**

$T(n) = 3\, T(n/4) + n^2$

Assume that T(n) is constant for sufficiently small n.

Scheme:

- Correct answer with proper explanation – 2.5 marks

Answer:

$T(n) = 3\, T(n/4) + n^2$

This recurrence looks like master theorem form. So

a=3, b=4, $f(n)= n^2$

$n^{\log_b a} = n^{\log_4 3} = n^{0.75}$, Comparing $n^{\log_b a}$ and f(n) we found f(n) is asymptotically larger than $n^{\log_b a}$, So we guess the solution is case-3 of master theorem. Testing of case-3 is done as follows:

$f(n)=\Omega(n^{\log_b a+\epsilon})=\Omega(n^{0.75+\epsilon})$ => $n^2 \geq c\, n^{0.75+\epsilon}$ =>True for c=1, ∈=0.25

Now ,$af(n/b) \leq cf(n)$ => $3 \times n^2/16 \leq c\, n^2$ => True

So as per case-3, the solution of the recurrence is $\mathbf{T(n) = \Theta(f(n)) = \Theta(n^2)}$

b) **Write a function in C language to arrange the numbers stored in an array as follows: odd numbers followed by even numbers, in O(n) time. The prototype of the function is given as void ARRANGE-ARRAY(int [], int);** **(2.5)**

Scheme:

- Correct algorithm solved in O(n) time , uses the given prototype – 2.5 marks
- Correct algorithm solved in O(n) time , uses different prototype – 2 marks
- Correct algorithm solved in other than O(n) time – 1.5 mark

Answer:

```
void ARRANGE-ARRAY(int a[], int n)
{
   int lb=0, ub=n-1;
   while(lb<ub)
   {
      //If lb index value is found even and ub indexed value is found  odd, then swap
     if(a[lb]%2==0 && a[ub]%2!=0) //Left=EVEN, right=ODD
     {
         temp=a[lb]; a[lb]=a[ub];a[ub]=temp;
          lb=lb+1;
          ub=ub-1;
     }
     else if(a[lb]%2==0 && a[ub]%2==0) //Left=EVEN, right=EVEN
     {
          ub=ub-1;
     }
     else if(a[lb]%2!=0 && a[ub]%2!=0) //Left=ODD, right=ODD
     {
          lb=lb+1;
     }
     else if(a[lb]%2!=0 && a[ub]%2==0) //Left=ODD, right=EVEN
     {
          lb=lb+1;
          ub=ub-1;
     }
  }
}
```

Q4 **a)** **Write down the merge sort algorithm that uses the merge procedure. Describe the time complexity of each terms of the divide, conquer and combine strategy.** **(2.5)**

Scheme:

- Merge sort algorithm: 0.5 mark
- Merge procedure : 1 mark

- Time complexity: 1 Mark

Answer:

Merge Sort Algorithm

A is the name of the array having lower bound as p and upper bound as q

```
MERGE-SORT(A, p, r)
{
  if(p<r)
  {
    q←(p+r)/2
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q+1, r)
    MERGE(A, p, q, r)
  }
}
```

Merge procedure

```
MERGE(A, p, q, r)
{
  //Create a temporary array TA having (r-p+1) elements
  //Array A divided into two parts (i.e. Left array A is sorted in ascending from index p to q and right array A is sorted from index q+1 to r
   i←p, j←q+1
   k←0
   //Compare one element of left part of array A with one element of right part of array A one by one and do the following till each part has at least one element
   while (i≤q and j≤r)
   {
      if (A[i]<A[j])
      {
        TA[k]=A[i]
        i←i+1
        k←k+1
      }
      else
      {
        TA[k]=A[j]
        j←j+1
        k←k+1
      }
   }
  //Copy rest elements if any from left array to temporary array TA
   while(i≤q)
  {
```

```
        TA[k]=A[i]
        i←i+1
        k←k+1
    }
    //Copy rest elements if any from right array to temporary array TA
    while(j≤r)
    {
        TA[k]=A[j]
        j←j+1
        k←k+1
    }
    //Copy all elements from temporary array TA back to array A
    k←0
    for i← p to r
    {
      A[i]←TA[k]
       k←k+1
    }
}
```

Time Complexity of Merge Sort Algorithm

- Merge sort on just one element takes constant time.
- When we have n>1 elements, we break down the running time as follows:

Divide: The divide step just computes the middle of the sub array, whichn takes constant time. Thus D(n) = Θ(1)

Conquer: When two sub problems are recursively solved, each of size n/2, it contributes 2T(n/2) to the running time

Combine: The MERGE procedure on an n-element subarray takes time Θ(n), and so C(n)= Θ(n)

Now the recurrence for the worst case running time T(n) of merge sort

$$\mathbf{T(n)}=\begin{cases}\Theta(n) & \text{if } n=1\\ \mathbf{2T(n/2)} + \Theta(n) & \text{if } n>1\end{cases}$$

Solving this recurrence by master theorem, we can get the solution as

T(n)=Θ(n log n)

b) Given 12 activities, A=< a_1, a_2,….,a_{10},a_{11},a_{12} > along with their start time (s_i) and finish time (f_i) are given as follows: (2.5)

i	1	2	3	4	5	6	7	8	9	10	11	12
s_i	44	7	37	83	27	49	16	44	44	58	27	26
f_i	86	25	96	89	84	62	17	70	84	94	79	57

Use an efficient method that computes a schedule with largest number of activities on that stage.

Scheme:

- Description of efficient method such as GREEDY-ACTIVITY-SELECTOR in

terms of language or algorithm or c-function – 1 mark

- Getting the correct solution schedule with proper steps -1.5 marks

Answer:

- It uses the following GREEDY-ACTIVITY-SELECTOR method that computes a schedule with largest number of activities.
- It assumes that the input activities are ordered by monotonically increasing finishing time.
- It collects selected activities into a set A and returns this set when it is done.

```
GREEDY-ACTIVITY-SELECTOR(s, f)
{
   n←length[s]
   A←{a1}
   k←1
   for m←2 to n
   {
        if(s(m)≥f(k)
        {
           A←AU{am}
           k←m
        }
   }
   Return A
}
```

a_i	s_i	f_i	Selection (Yes/No)
a7	16	17	√
a2	7	25	x
a12	26	57	√
a6	49	62	x
a8	44	70	x
a11	27	79	x
a5	27	84	x
a9	44	84	x
a1	44	86	x
a4	83	89	√
a10	58	94	x
a3	37	96	x

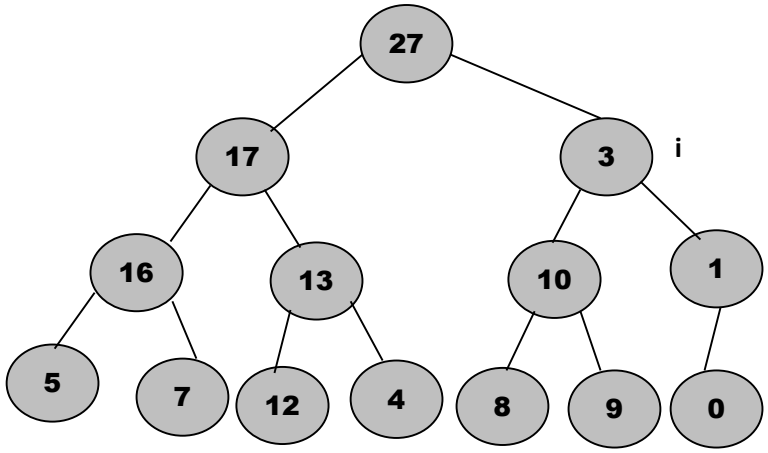
The solution schedule is <a7,a12,a4>

Q5 a) Write the algorithm for MAX-HEAPIFY(A,i). Illustrate the operation of MAX-HEAPIFY(A,3) on the array A={27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0} **(2.5)**

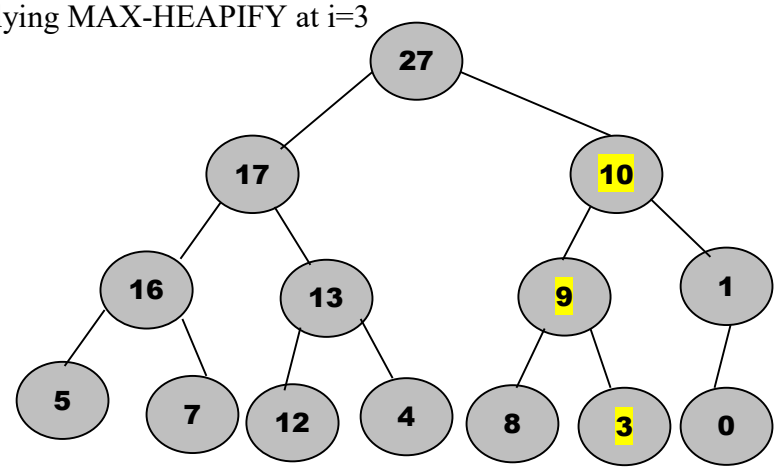
Scheme:

- Proper illustration through diagrams – 2.5 marks

Answer:



Applying MAX-HEAPIFY at i=3



b) Write the Insertion Sort algorithm. Analyze the best case and worst case time complexity of this algorithm. **(2.5)**

Answer:

Line No.	Insertion Sort Algorithm	Cost	Times
1	INSERTION-SORT(A)	0	
2	{	0	
3	for j←2 to length[A]	c1	n
4	{	0	
5	key←A[j]	c2	n-1
6	//Insert A[j] into the sorted sequence A[1..j-1]	0	
7	i←j-1	c3	n-1
8	while(i>0 and A[i]>key)	c4	$\sum_{j=2}^{n} tj$
9	{	0	
10	A[i+1]←A[i]	c5	$\sum_{j=2}^{n} (tj - 1)$

11	i←i-1	c6	$\sum_{j=2}^{n}(tj-1)$
12	}	0	
13	A[i+1]←key	c7	n-1
14	}	0	
15	}	0	

To compute T(n), the running time of INSERTION-SORT on an input of n values, we sum the products of the cost and times column, obtaining

$T(n)=c1n + c2(n-1) + c3(n-1) + c4\sum_{j=2}^{n} tj + c5\sum_{j=2}^{n} (tj - 1) + c6\sum_{j=}^{n} (tj - 1) + c7(n-1)$ ..... (1)

Best case Analysis:

- Best case occurs if the array is already sorted.
- For each j=2 to n, we find that A[i]≤key in line number 8 when I has its initial value of j-1. Thus tj=1 for j=2 to n and the best case running time is
  $T(n)=c1n + c2(n-1) + c3(n-1) + c4(n-1) + c7(n-1) =O(n)$

Worst case Analysis:

- Worst case occurs if the array is already sorted in reverse order
- In this case, we compare each element A[j] with each element in the entire sorted subarray A[1..j-1], so tj=j for j=2 to n.
- The worst case running time is
  $T(n)=c1n + c2(n-1) + c3(n-1) + c4(n(n+1)/2-1) + c5(n(n-1)/2) + c6(n(n-1)/2) +c7(n-1)=O(n^2)$

Q6 a) Find an optimal solution to the following knapsack instance: (2.5)
Number of items=n=8, Knapsack Capacity=M=17. Profit=P={10, 15, 8, 7, 3, 15, 8, 27} and weight=W= {5, 4, 3, 7, 2, 3, 2, 6}. Write a suitable algorithm to solve this problem.

Scheme:

- Fractional Knapsack Algorithm: 1 Mark
- Optimal Solution with proper steps: 1.5 Marks

Answer:

```
GREEDY-KNAPSACK(M, n)
/*w[1:n] and v[1:n] contains the weights and values/profits respectively. n items/objects ordered such that v[i]/w[i] ≥ v[i+1]/w[i+1]. M is the knapsack size and x[1:n] is the solution vector.*/
{
   for i ← 1 to n do
       x[i] ← 0  // Initialize x
   U ← M
   for i ← 1 to n do
   {
      if (w[i] > U) then break;
      x[i] ← 1.0
      U ← (U – w[i])
   }
   if (i ≤ n) the x[i] ← U/w[i]
```

```
}
```

Steps to get Optimal Solution

Step-1: Find out the Value (or Profit) per weight (v_i/w_i)

Item/ Object (i)	Value or Profit (v_i)	Weight (w_i)	Value (or Profit) per weight (v_i/w_i)
1	10	5	2
2	15	4	3.75
3	8	3	2.67
4	7	7	1
5	3	2	1.5
6	15	3	5
7	8	2	4
8	27	6	4.5

Step-2: Sort all columns as per decreasing order of (v_i/w_i)

U=M=17

Item/ Object (i)	Value or Profit (v_i)	Weight (w_i)	Value (or Profit) per weight (v_i/w_i)	Weight taken	Solution Vector (x_i)
6	15	3	5	**3 ≤ 17, take the whole item. Remaining weight U = 17-3 = 14**	1
8	27	6	4.5	**6 ≤ 14, take the whole item. Remaining weight U = 14-6 = 8**	1
7	8	2	4	**2≤ 8, take the whole item. Remaining weight U = 8-2 = 6**	1
2	15	4	3.75	**4≤ 6, take the whole item. Remaining weight U = 6-4 = 2**	1
3	8	3	2.67	**3≤ 2 is false so a fraction of the 3kg will be taken that is Fraction= U/w_i =2/3**	2/3
1	10	5	2		0
5	3	2	1.5		0
4	7	7	1		0

The solution vector or the optimal solution is
$(x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8) = (0, 1, 2/3, 0, 0, 1, 1, 1)$

b) Describe the PARTITION() algorithm of QUICK-SORT() step by step how you would get the pass1 result by taking last element as pivot on the following data. (2.5)

2, 5, 7, 5, 9, 3, 8, 6

Scheme:

- PARTITION Algorithm: 1 Mark
- Representation of intermediate steps of pass-1 : 1.5 Mark

Answer:

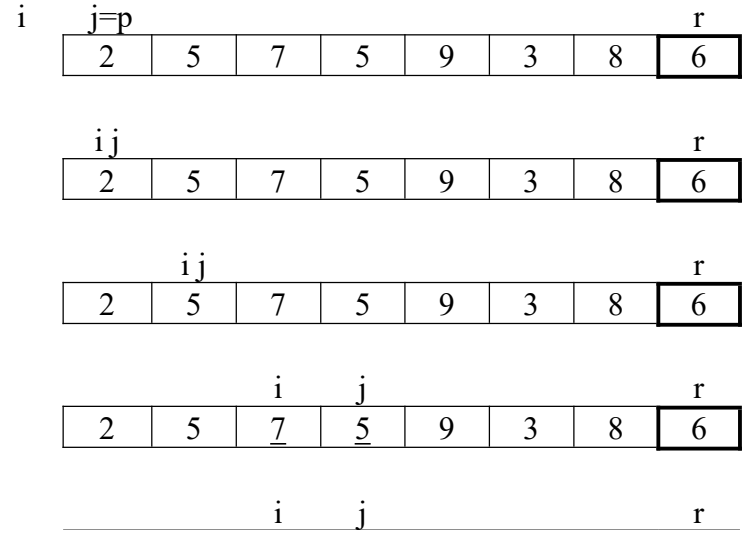
PARTITION Algorithm:

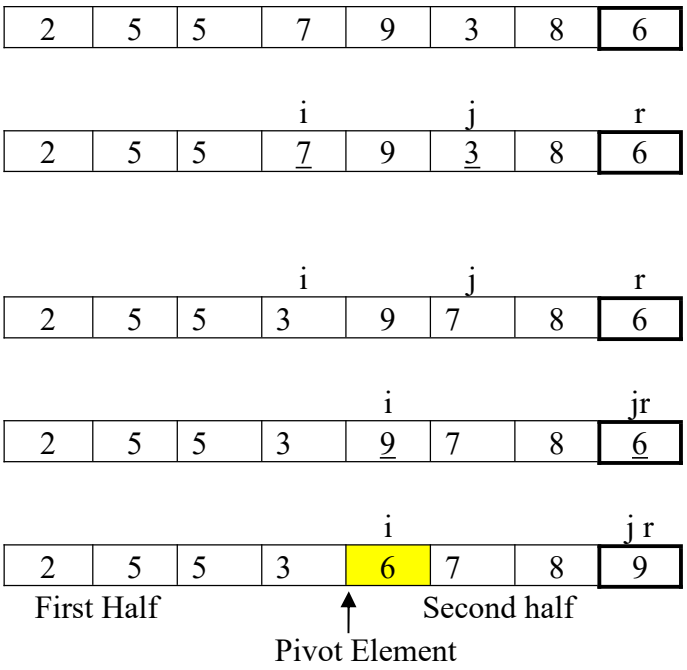
```
PARTITION(A,p,r)
{
    x ← A[r]
    i ← p-1
    for j ← p to r-1
    {
       if A[j] ≤ x
       {
           i ← i+1
           A[i] ↔A[j]
        }
    }
     i ← i+1
     A[i] ↔A[r]
     return i
}
```

Representation of intermediate steps of pass-1

Given Data: 2, 5, 7, 5, 9, 3, 8, 6

Underlined means swapping of data





Q7 a) Write pseudocode for the procedure HEAP-EXTRACT-MIN. Explain with the given min-heap level order traversal A={10, 20, 15, 22, 30, 18, 17, 40, 28}. **(2.5)**

Scheme:

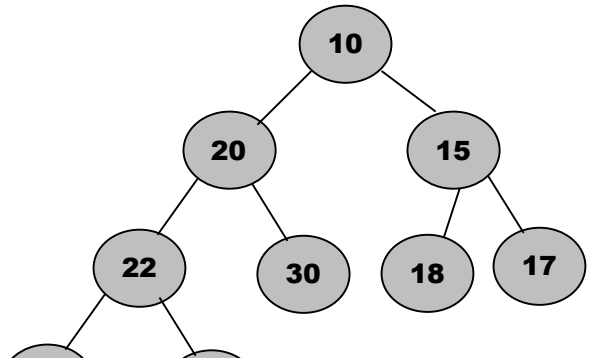
- Correct HEAP-EXTRACT-MIN algorithm – 1 mark
- Explanation of HEAP-EXTRACT-MIN algorithm on the given data by showing correct diagrams -1.5 marks

Answer:

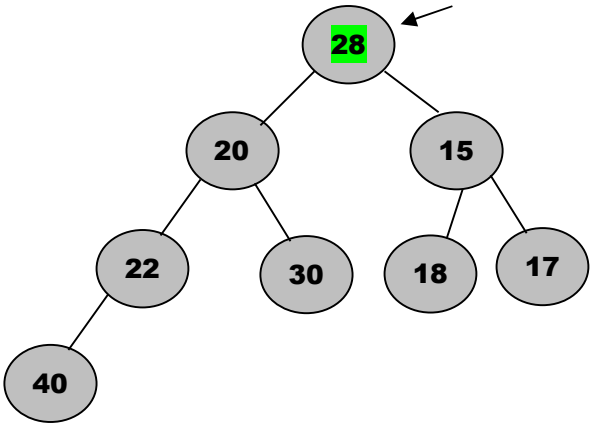
```
//This algorithm assumes the array representing min-heap starts at index 1
HEAP-EXTRACT-MIN(A)
{
  if heap-size[A] < 1
      error "Heap Underflow"
 min←A[1]
 A[1]←A[heap-size[A]]
 heap-size[A]← heap-size[A]-1
 MIN-HEAPIFY(A,1)
 return min
}
```

Explanation

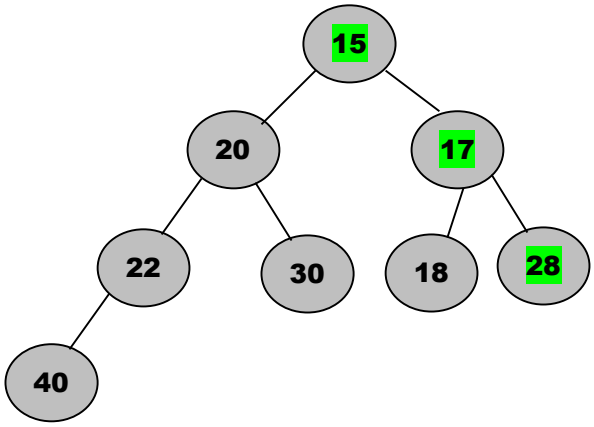
The given level order traversal A={10, 20, 15, 22, 30, 18, 17, 40, 28} yield the following min-heap tree.



Applying HEAP-EXTRACT-MIN on the above min-heap will return the minimum value that is on the root (i.e. 10) and this node can be replaced by the last element (i.e. 28) a now apply MIN-HEAPIFY on this node that is MIN-HEAPIFY(A,1)



After applying MIN-HEAPIFY(A,1) that is nothing but to re-build the heap, becomes



b) Show that there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h in any n-element heap. **(2.5)**

Scheme:

- Any valid proof – 2.5 marks
- Proof with taking some sample nodes – 1 Mark

Answer:

Height of a node - Longest distance from a leaf to that node
Depth of a node – Distance from root to that node
If the heap is not a full binary tree (bottom level is not full), then the nodes at a given level (depth) don't all have the same height. For example, although all nodes at depth d have height 0, nodes at depth d−1 can have either height 0 or height 1.

Proof By induction on h

Basis: Show that it is true for h = 0 (i.e.No of leaf nodes$\leq \lceil n/2^{h+1} \rceil = \lceil n/2 \rceil$

The tree leaves (nodes at height 0) are at depths d and d−1. They consist of

• all nodes at depth d, and

• the nodes at depth d−1 that are not parents of depth-d nodes.

Let x be the number of nodes at depth d that is, the number of nodes in the bottom (possibly incomplete) level.

Rest (n-x) nodes above the bottom level form a complete binary tree, and a complete binary tree has an odd number of nodes (1 less than a power of 2). Thus if n is odd, x is even, and if n is even, x is odd.

Proving the base case:

a) **If x is even**, there are x/2 nodes at depth d−1 that are parents of depth d nodes, hence $2^{d-1}-x/2$ nodes at depth d−1 that are not parents of depth-d nodes. Thus,
Total no. of height-0 nodes= nodes at depth d + nodes at depth d-1
$= x + (2^{d-1}-x/2)$
$= 2^{d-1}+x/2$
$= \lceil (2^d+x-1)/2 \rceil$ (as x is even)
$= \lceil \mathbf{n/2} \rceil$
($n = 2^d + x - 1$ because the complete tree down to depth d−1 has $2^d - 1$ nodes and depth d has x nodes.)
Hence proved.

b) **If x is odd**, by an argument similar to the even case, we see that
Total no. of height-0 nodes= nodes at depth d + nodes at depth d-1
$= x + (2^{d-1}-(x+1)/2)$
$= 2^{d-1}+(x-1)/2$
$= (2^d+x-1)/2$
$= n/2$
$= \lceil \mathbf{n/2} \rceil$ (because x odd ⇒n even)
Hence proved.

Inductive step: Show that if it is true for height h−1, it is true for h.

Let n_h be the number of nodes at height h in the n-node tree T .

Consider the tree T1 formed by removing the leaves of T . It has $n1 = n-n_0$ nodes.

We know from the base case that $n_0 = \lceil n/2 \rceil$ so $n1 = n-n_0 = n-\lceil n/2 \rceil = \lfloor n/2 \rfloor$

Now, the nodes at height h in T would be at height h − 1 if the leaves of the tree were removed that is, they are at height h−1 in T1. Letting $n1_{h-1}$ denote the number of nodes at height h − 1 in T1, we have

$n_h = n1_{h-1}$

By induction, we can bound $n1_{h-1}$:

$n_h = n1_{h-1} \leq \lceil n1/2^h \rceil = \lceil \lfloor n/2 \rfloor /2^h \rceil \leq \lceil (n/2)/2^h \rceil = \lceil n/2^{h+1} \rceil$ Hence proved.

==========THE END==========