UNIVERSITY PARTNER
UNIVERSITY OF WOLVERHAMPTON

iAcademy
Quality Education Improving Lives

HERALD COLLEGE KATHMANDU

Academic Year	Module	Assessment Number	Assessment Type
S20	Introductory Data Structures and Algorithms (DipIT02)	A1	Assignment Submission

[Assignment Submission]

Student Id : [NP03A190299]
Student Name : [Yogesh Shrestha]
Section : [DC8]
Module Leader : [Mr. Prakash Gautam]

Submitted on : 06-03-2020

Page No.
Date : / /

Tutorial . 4

1a-> For any positive integer n, a tree with n vertices has n-1 edges.
It is proof by Mathmatical induction.

* Base case
A tree with one vertex has 0 edge.

* Inductive Hypothesis
Suppose a tree having 'n' vertices has 'n-1' edges.

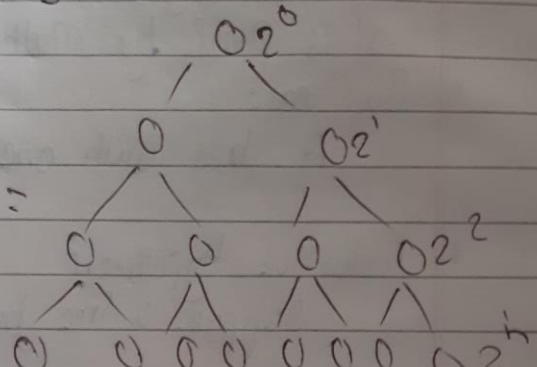
* ~~Induc~~ Indactive step
Given from Inductive hypothesis a tree having 'n' vertices has 'n-1' edges.
Add a single vertex in any leaf node of tree having n vertices. tree.
number of vertices = n+1
number of edges = n

b The # leaves in a non-empty full binary tree is one more than the # internal nodes.

Page No.
Date : / /

2→ In a perfect binary tree, T: total number of nodes with height h is $n = 2^{h+1} - 1$

→



total number of nodes =
$2^0 + 2^1 + 2^2 + \dots + 2^n$

Sum of Geometric series =

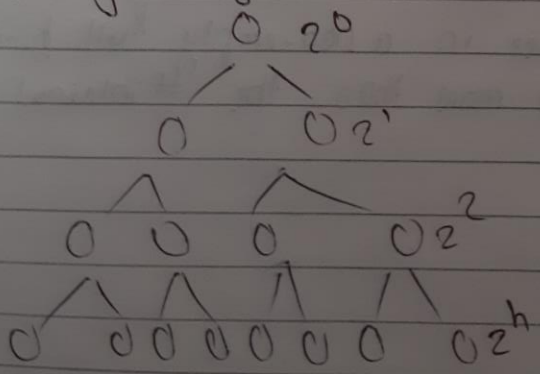
$= \frac{a(r^n - 1)}{r - 1}$

Or, $n = \frac{2^0(2^{h+1} - 1)}{2 - 1}$

Or, $n = \frac{2^{n+1} - 1}{2 - 1}$

$\therefore n = \frac{2^{h+1} - 1}{1}$

3) In a complete binary tree, T: the max # nodes with height h is: $n = 2^{h+1} - 1 = 2^{\#level} - 1$



total number of nodes $= 2^0 + 2^1 + 2^2 + 2^3 + \dots 2^h$

Or, $n = \frac{2^0(2^{h+1} - 1)}{2 - 1}$

Page No.
Date : / /

or. $n = 1 \times (2^{h+1} - 1)$
or $n = 2^{h+1} - 1$

4. In a perfect binary tree T, the height of T having total number of nodes n is : $h = \log(n+1) - 1$
total number of nodes $(n) = 2^{h+1} - 1$
taking log on both sides. we get
$n+1 = 2^h$
or $\log_2(n+1) = \log_2(2^h)$
or, $h+1 = \log_2(n+1)$
or $h = \log_2(n+1) - 1$

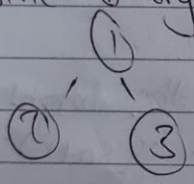
Q5. In a complete binary tree T, the height of T having total n # nodes is : $h = \log(n+1) - 1$
total number of nodes for complete binary
$n = 2^{h+1} - 1$
Taking log with base 2 on both sides
$\log_2(n+1) = \log_2(2^h)$
or $\log_2(n+1) = h+1$
or $h = \log_2(n+1) - 1$

Page No.
Date : / /

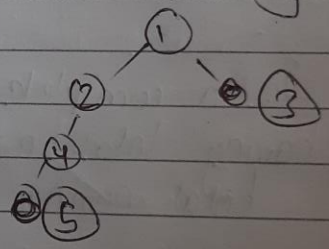
6 a) All proper binary trees are perfect : (F)
b) All perfect binary trees are complete & vice-versa. (F)
c) All complete binary tree are complete. (T)
d) All complete binary tree are balanced (T)
e) All complete binary tree are balanced & vice-versa (T)
f) Proper binary tree can never be perfect (F)
g) Perfect binary tree can always be 2-tree. (T)
h) Balanced tree can never be pathological. (F)

7 →

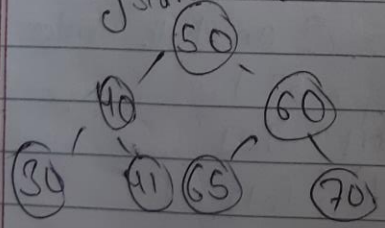
a.) Strict binary tree



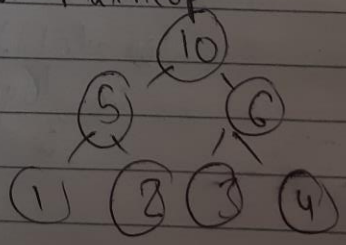
b) Unbalanced binary tree



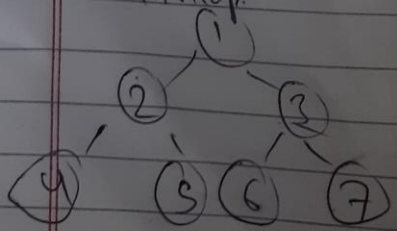
c) Binary search tree



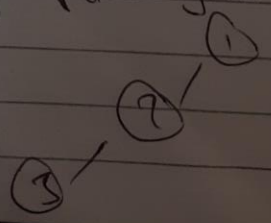
d) Maxheap



e) Minheap



g) Pathological binary tree



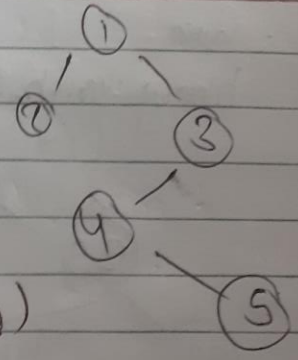
Page No.
Date : / /

8 a.

Inorder - (left, Root, Right)
: 21453

Postorder - (left, Right, Root)
: 25431

Preorder - (Root, left, Right)
: 12345

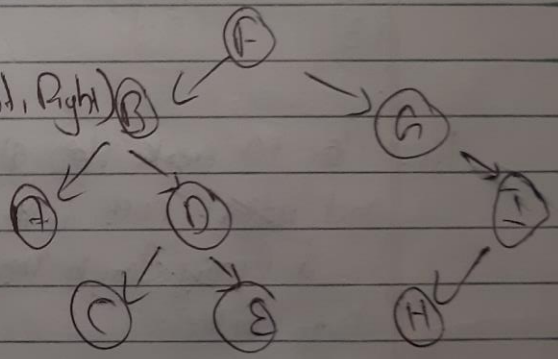


b)

Inorder - BCDEFGHI (left, Root, Right)

Preorder - FBADCEGIH
(Root, left, Right)

Postorder - ACEDBHIGF
(left, Right, Root)

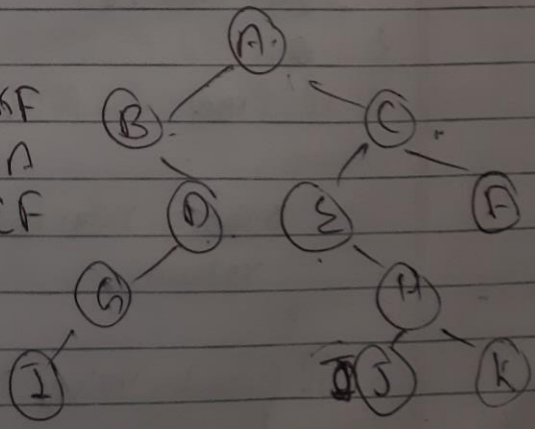


c)

Pre-order - ABDGJCEHJKF

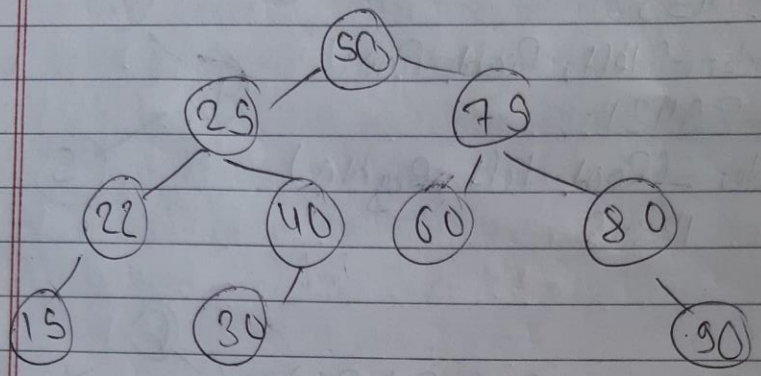
Post order : JGDBJKHEFCA

Inorder - BJGDAEJHKCF

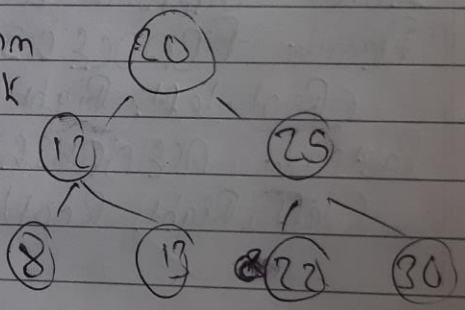


Page No.
Date : / /

9:- . Given.
binary search tree: 50, 25, 75, 22, 40, 60, 80, 90, 15, 30



10:- b. To create algorithm and insert node with key k in a Binary Search Tree, it is Given below:



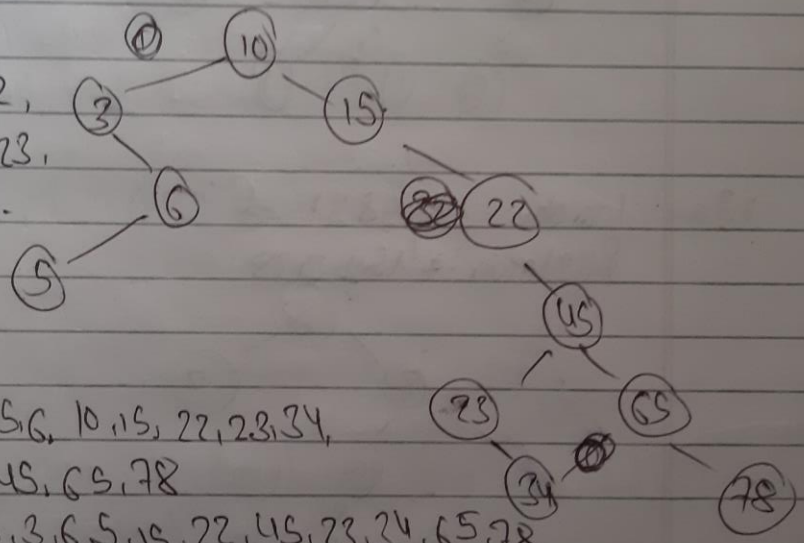
→ Insert (root, k)

```
{
if (root == NULL)
 {
  root = new node(k)
  return;
 }
if (root.value > k)
 {
  if root.left == NULL)
  { root.left = new node (k)}
  else { insert (root.left, (k))}
  }
```

Page No.
Date : / /

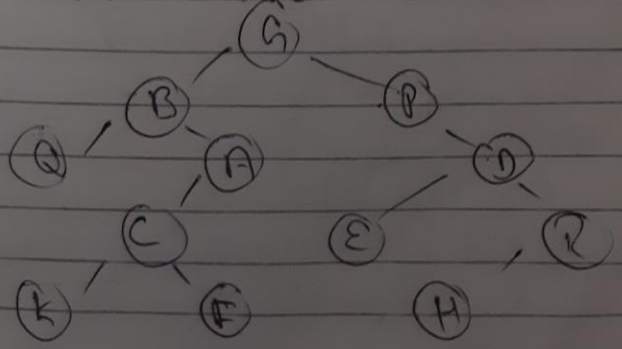
```
if (root.value < k)
{ if (root.right == NULL)
  { root.right = new node (k) }
  else {insert (root.right, (k) }
  }
```

11-> Given,
10, 3, 15, 22,
6, 45, 65, 23,
78, 34, 5.



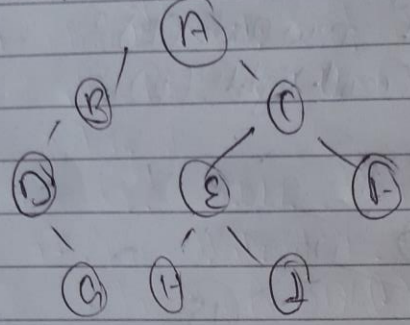
Inorder = 3, 5, 6, 10, 15, 22, 23, 34, 45, 65, 78
Preorder = 10, 3, 6, 5, 15, 22, 45, 23, 34, 65, 78
Post order = 5, 6, 3, 34, 23, 78, 65, 45, 22, 15, 10

12-> Preorder : GBADCKFPDERH &
Inorder: GDKCFAGBEDHR

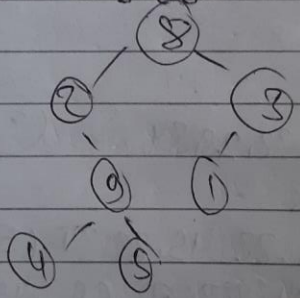


Page No.
Date : / /

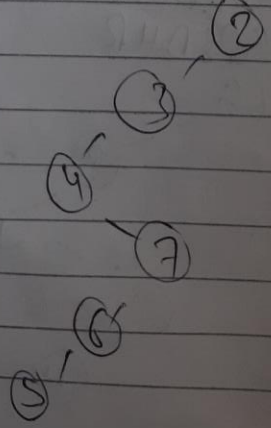
b) Inorder: DGBAHEICF
Postorder: GDBHIEFCA



13a. Inorder: 2495831
Postorder: 4592381

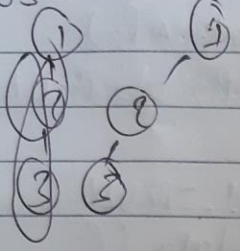


b. Inorder: 4,5,6,7,3,2
Preorder: 2,3,4,7,6,5

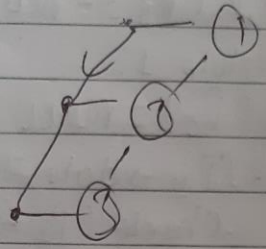


Page No.
Date : / /

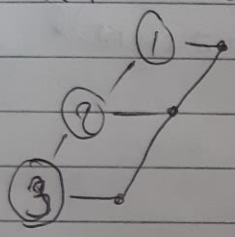
(c) Preorder: 123
Postorder: 321



Check (using flag) (Preorder)



Check (Post order)



321

~~14.~~ ~~Inorder:~~

16. → An algorithm to search a data in BST.

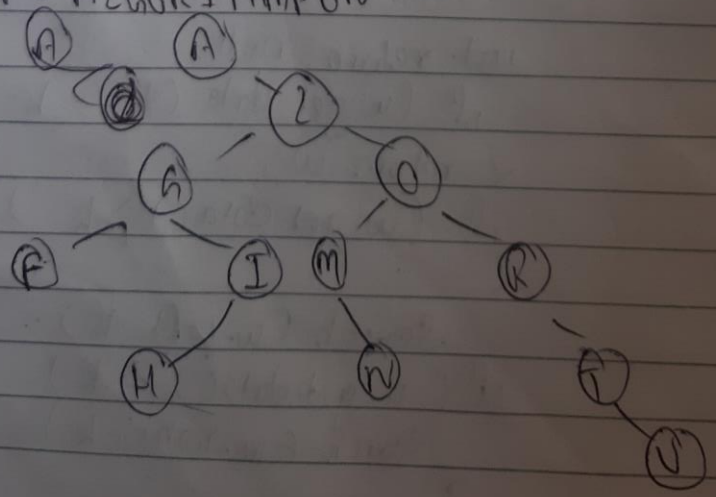
```
Search (w.root, k) {
    if (w = NULL)
    {
        ~~rot~~. return null;
        if (w.get data () == k)
        { return w;
        if (w.get data() > k)
        {
            Search (w.left, k)
        if (w.get data() < k)
            Search (w.right, k)
        }
```

Page No.
Date : / /

17.→ Given
Two binary trees. -- . O(n)

```
int check(node *P1, node *P2)
{ if (P1 == NULL && P2 == NULL)
   return 1;
  if (P1 != NULL || P2 != NULL)
  if (P1->data == P2->data &&
     check(P1->left, P2->left) &&
     check(P1->right, P2->right))
  {
     return 1;
  }
  return 0;
}
```

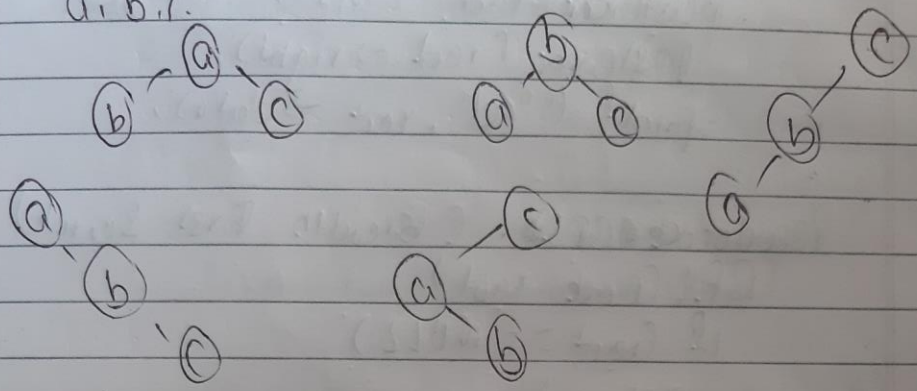
14-→. Inorder : ABCDEFGHIJKLMNOPQRSTUVWXYZ
Preorder : ALGORITHMFUN



Page No.
Date : / /

15→ Given,

a, b, c.



19→ Pseudocode DFS (Depth First Search)

```
Void Inorder (struct node *root)
{
  if (root == NULL)
  { return;
  }
  inorder (root -> left);
    printf ("%d", root -> data);
    inorder (root -> right);
}
Void Preorder (struct node *root)
{
  if (root == NULL)
  return;
printf Preorder ("%d", root -> data);
    Preorder ( root -> left);
    Preorder (root -> right);
}
```

Page No.
Date : / /

```
void Postorder (struct node *root)
    postorder (root -> left);
    postorder (root -> right);
    printf ("%d", root -> data);
```

Pseudocode BFS (Breadth First Search)

```
BFS (Node root)
  if (root == NULL)
    return 1
Queue q
  q.enqueue(root)
  while (!q is. Empty)
  Node x = q. dequeue()
    Print x
      if (x.left != null)
        q. enqueue (x.left)
      if (x.right != null)
      q. enqueue (x.right)
```

Page No.
Date : / /

20) Soln

Build maxheap (Array n)
{
int i;
for($i = \lfloor \frac{n-1}{2} \rfloor$ to 0)
{max-heapify (Array, n, i)}

Time complexity $= O(n \log n)$

max-heapify (Array, n, largest (i))
{
int largest = i;
int l = 2*i + 1;
int r = 2*i + 2;
if (l ≤ n and A[l] > A[largest])
{largest = l}
if (r ≤ n and A[r] > A[largest])
{largest = r}
if (largest != i)
{Swap (A[i] and A[largest])
max-heapify (Array, n, largest)
}

Time Complexity $= O(\log n)$

Page No.
Date : / /

21.) ~~World~~ Pseudocode to build mini-heap and to illustrate Min-heapify procedure.

```
build-minheap (Array, n)
{ int i;
  for (i = ⌊(n-1)/2⌋ to 0)
  { min-heapify (Array, n, i) }

Min-heapify (Array, n, i){
  int larger = i;
  int left = 2*i + 1;
  int Right = 2*i + 2;

if (left ≤ n and A[left] < A[larger])
      larger = left;
if (Right ≤ n and A[Right] < A[larger])
      larger = Right;
  if (larger != i)
  {
    Swap (A[i] and A[larger])
    min-heapify (Array, n, larger)
  }
```

Page No.
Date : / /

22) a) getMini(Array){
return Array[0];
}

b) Extract Min (Array, n){
Array[0] = Array[n-1]
n = n-1;
Minheapify (Array, n, 0);

c) Insert (Array, key){
~~Heapify = 0~~
HeapSize = HeapSize + 1;
int i = HeapSize;
A[i] = key;
while (i > 1 and A[i] < A[i])
{
Swap (A[i] and A[i])
i = i/2
}

d. decrease Key (Array, key){
int i = HeapSize;
A[i] = key;
while (i > 1 and A[i] < A[i/2])
{ Swap (A[i] and A[i/2])
i = i/2
}

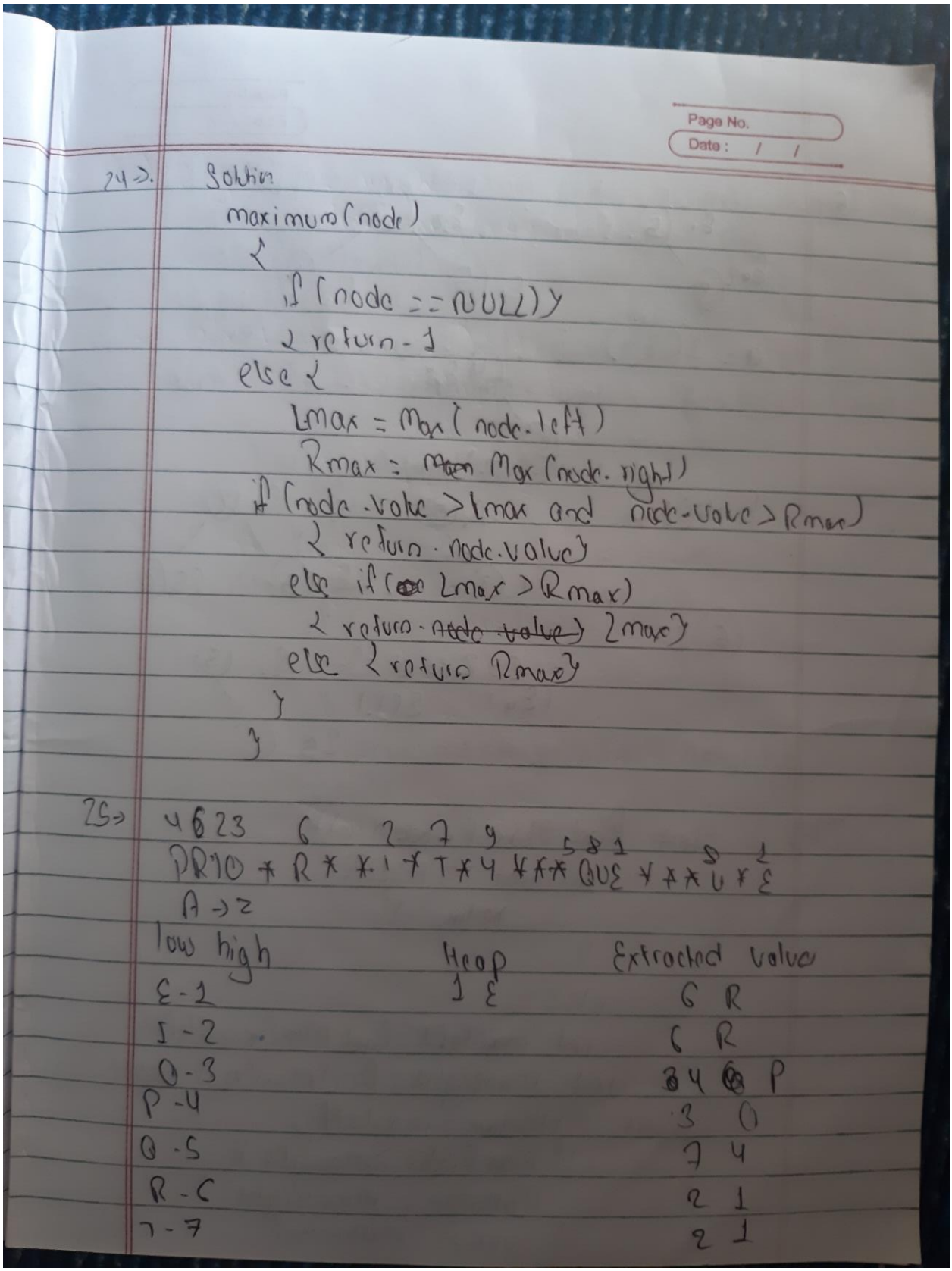
Page No.
Date : / /

23.) a) getMax(){
return Array[a];
}

b) extractMax(Array, n){
Array[a] = Array[n-1]
n = n-1;
Maxheapify(Array, n, a);

c) insert(Array, key){
Heapsize = Heapsize + 1;
int i = Heapsize;
A[i] = key;
while (i > 1; and A[i/2] < A[i])
{
Swap(A[i] and A[i/2])
i = i/2
}
}

d. decreasekey(Array, key){
int i = Heapsize;
A[i] = key;
while (i > 1; and A[i/2] < A[i])
{ Swap [A[i] and A[i/2])
i = i/2
}}



Page No.
Date : / /

24->. Solution

```
maximum(node)
{
  if (node == NULL){
  return -1
else {
  Lmax = Max(node.left)
  Rmax = Max(node.right)
if (node.value > Lmax and node.value > Rmax)
  { return node.value}
else if (Lmax > Rmax)
  { return node.value} {max}
else {return Rmax}
  }
}
```

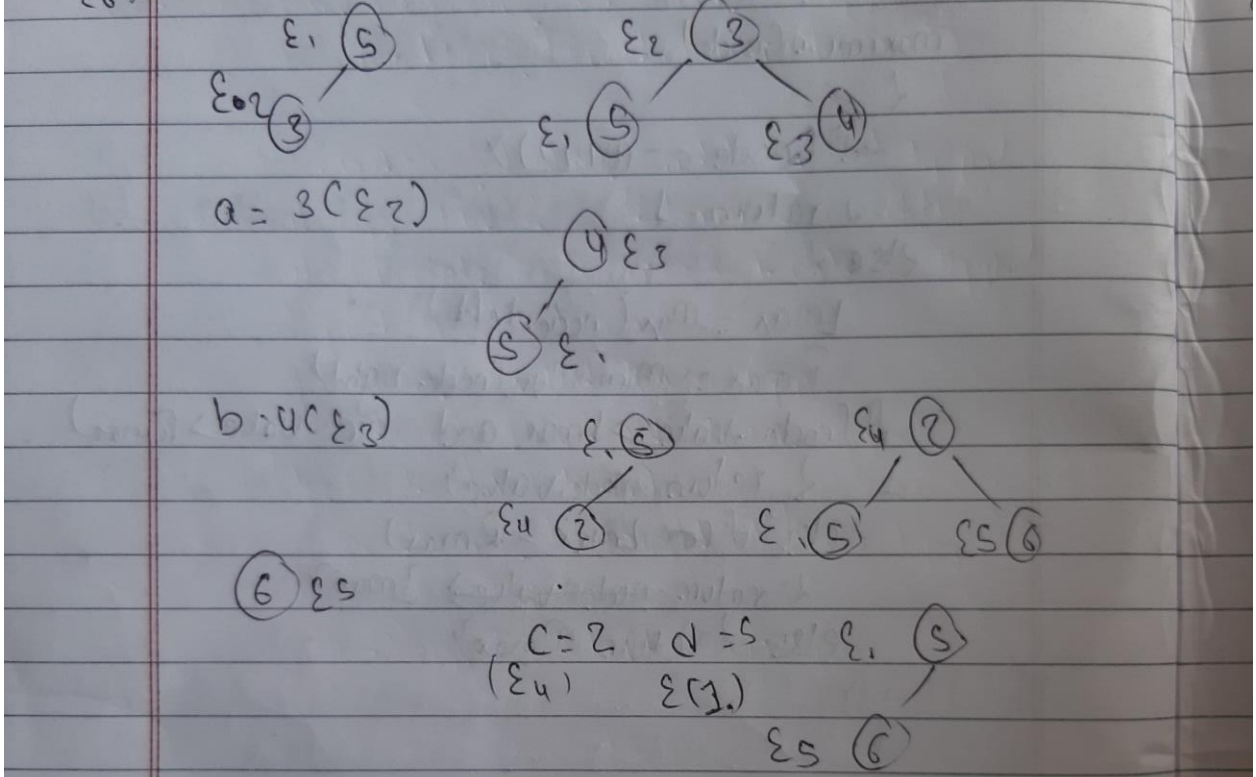
25-> 4 6 23 6 2 7 9 5 8 1 PRIO * R * * I * T * Y * * * QUE * * * U * E

A -> Z

low high	Heap	Extracted value
E-1	I E	G R
I-2		G R
O-3		3 4 G P
P-4		3 O
Q-5		7 4
R-6		2 1
7-7		2 1

Page No.
Date : / /

26.



27.

```
int findMax(node root){
    if(root == NULL){
        return -1;
    }
    else{
        int maxLeft = findMax(root.left);
        int maxRight = findMax(root.right);
        int mathmax = maxLeft;
        if(maxRight > maxLeft){
            mathmax = maxRight;
        }
        return mathmax;
```

Page No.
Date : / /

28.->

```
void printRange(int lower, int upper){
    printRange1(root, lower, upper);}

void printRange1(NodeP, int lower, int upper)
{ if (p != null) {    // terminal condition
  if (P.data >= lower && P.data <= upper)
  { print(P.data)
    PrintRange1(P.left, lower, upper)
    PrintRange1(P.right, lower, upper);}
  else if (P.data < lower)
  { printRange1(P.right, lower, upper);}
  else
  { printRange1(P.left, lower, upper);}
}
```

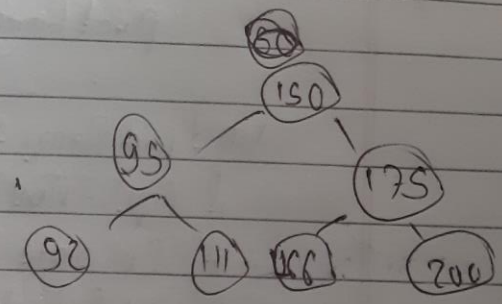
Page No.
Date : / /

29-> Implement a method that prints all nodes in a certain level.

Invoking printLevel(2) on the binary tree above should

~~Print: 92 111 166 92 111~~

Print: 92 111 166 200




```
void printLevel(int level){
    printLevel(Node root, int level)
}

void print(Node n, int level){
    if(n == null){
        return;
    }
    if(level == 0){ print(n->data); }
    else
    {
        print(n->left, level-1)
        print(n->right, level-1)
    }
}
```