DS&AA
Arif Ali MCA A

Date / /

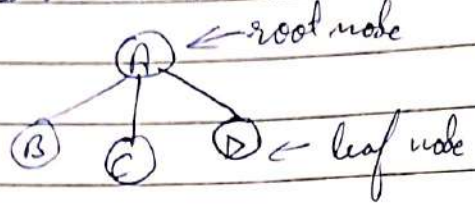
Page No.:

Assignment-3

1. Explain the following terms

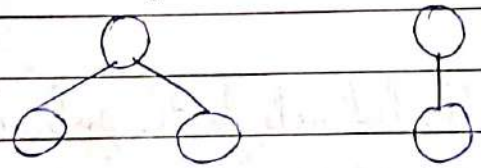
(i) Tree

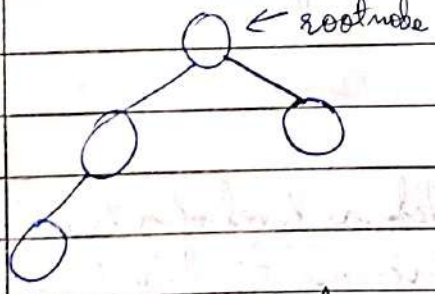
Any Hierarchical structure which doesnot have cycle is called tree.



(ii) Binary Tree

A tree node can have atmost two child such kind of tree is called Binary Tree.



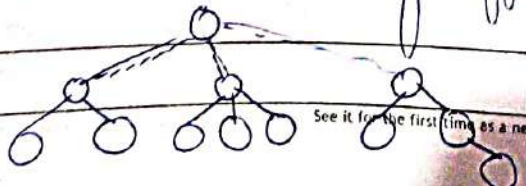


(iii) Vertex of Tree

A vertex of a Tree is called a leaf if it has no children.

(iv) Forest

It is the collection of different kinds of tree.



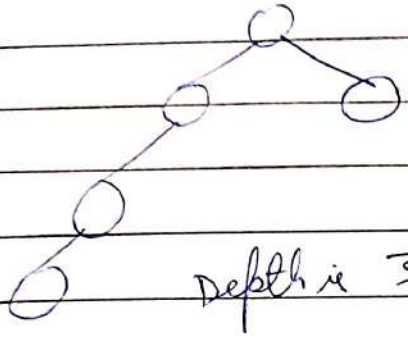
Chitra

Date ___/___/___ Page No.: ___

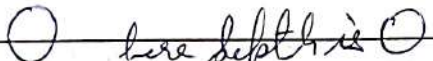
(v) Depth

In the tree, the total number of edges from the root node to the leaf node in the longest path is known as 'Depth of Tree'.

~~In tree data structure~~



Depth is 3



here depth is 0

(vi) Height

The number of edges from the leaf node to the particular node in the longest path is known as the height of that node.

(vii) Level

Each step from top to bottom is called as level of a tree. The level count starts with 0 and increments by 1 at each level or step.

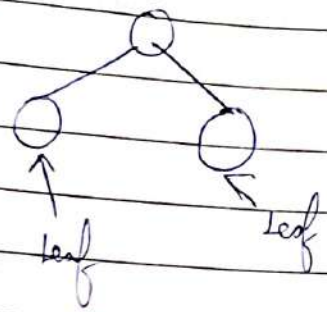
(viii) Degree of an element

The number of subtrees of a node is called its degree

Date ___/___/___ Page No.: ___

(x) Leaf

A leaf of an unrooted tree is a node of vertex degree 1.

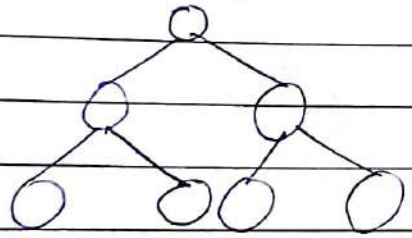


The node which does not have any child is called as Leaf.

2.

(i) Strictly Binary Tree

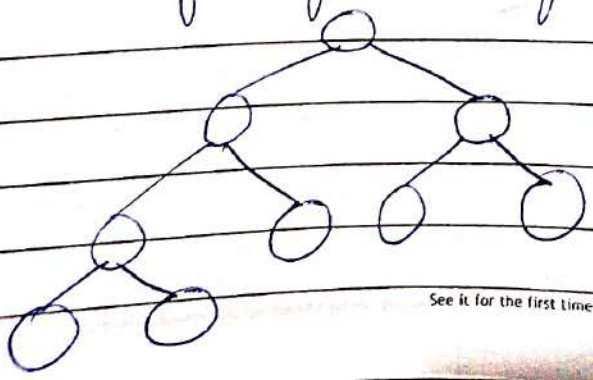
The tree can only be considered as the strictly binary tree if each node must contain either 0 or 2 children.



The strictly binary tree can also be defined as the tree in which each node must contain 2 children except the leaf nodes.

(ii) Complete Binary tree

A complete binary tree is a binary tree in which all the levels are completely filled except possibly the lowest one, which is filled from the left.

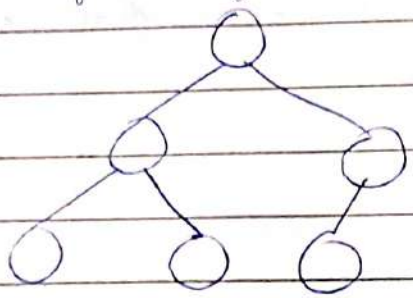


Date / /

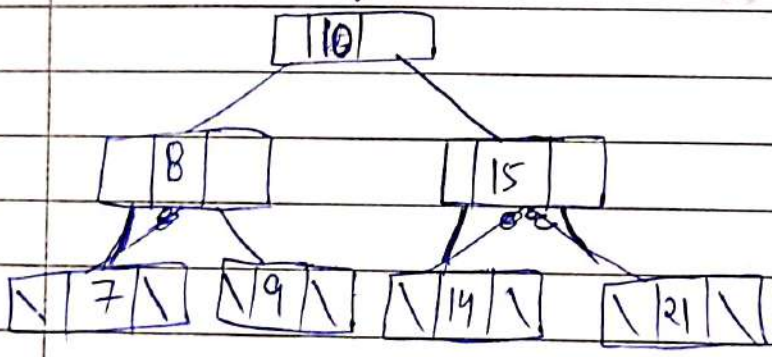
Page No.:

(iii) Almost Complete Binary Tree

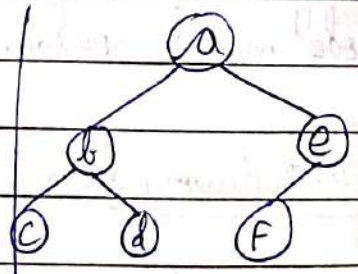
An almost complete binary tree is a special kind of binary tree where insertion takes place level by level and from left to right order at each level.

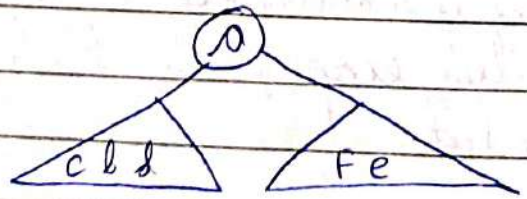


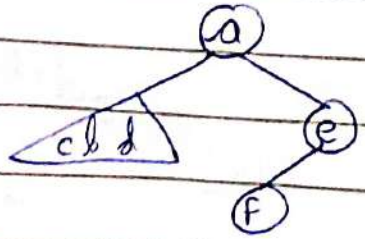
Representation of Tree in Memory



4. Construct Binary tree from
Inorder: c b d a f e
Postorder: c d b f e a

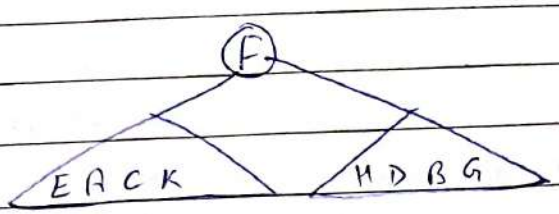


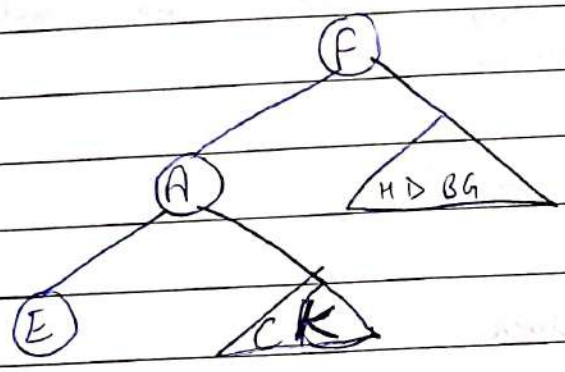


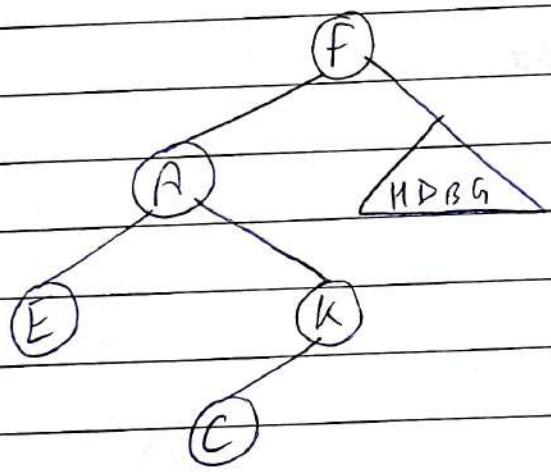


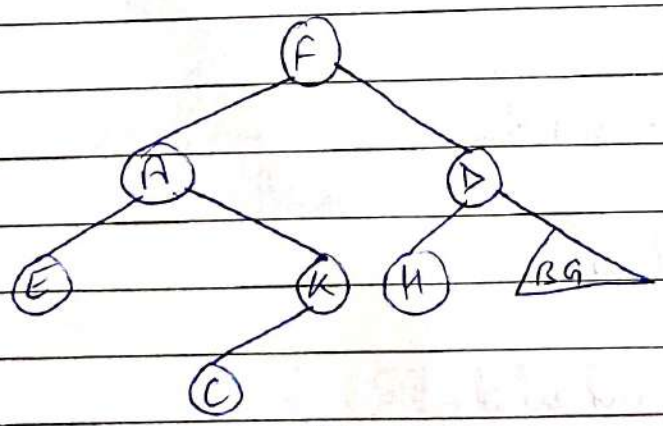
Date ___/___/_____ Page No.: _____

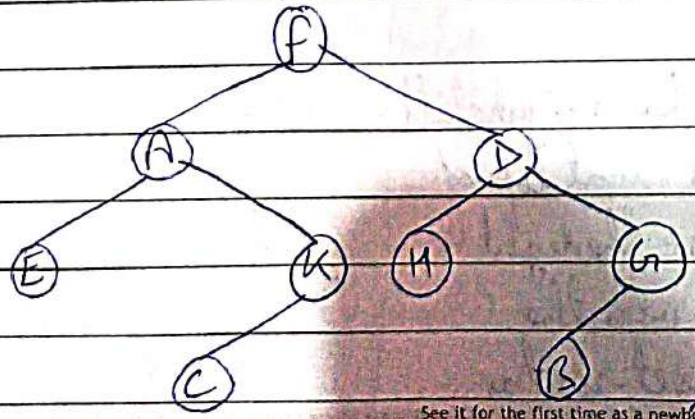
5. Inorder: E A C K F H D B G
Preorder: F A E K C D H G B











Chitra

See it for the first time as a newborn child that has no name."

6.

Non Recursive algorithm to traverse the tree in Inorder.

1. Create an empty stack
2. Initialize current node as root
3. Push the current node to S and set current = current -> left until current is = NULL.
4. If current is NULL and stack is not empty then
   (a) Pop the top item from stack.
   (b) Print the popped item, set current = popped_item -> right
   (c) goto step 3.
5. If current is NULL and stack is empty then we are done.

Non recursive Postorder traversal

1. Create an empty stack
2. Do following while root is not NULL
   (a) Push root's right child and then root to stack
   (b) Set root as root's left child.
3. Pop an item from stack and set it as root.
   (a) If the popped item has a right child and the right child is at top of stack, then remove the right child from stack, push the root back and set root as root's right child.
   (b) Else print root's data & set root as NULL
4. Repeat step 2.1 and 2.2 while stack is not empty.

8.

Page No.:

Algorithm for insertion and deletion in BST.

```
void insert (struct BST **root, int key)
{
  BST *temp;
  temp = (struct BST *)malloc(sizeof(struct BST *));
  if(*root==NULL)
        *root = temp
  if(key < root->info)
        insert(&((*root)->left), key);
  else if(key >= root->key)
        insert(&((*root)->right), key);
}
void deletenode(struct BST **root, int key, struct BST **proot)
{
    if(*root==NULL)
        printf("Tree empty"); return;
    if(key < (*root)->info)
        deletenode(&((*root)->left), key, root);
    else if(key > (*root)->info)
        deletenode(&((*root)->right), key, root);
    else
    {
        struct BST *temp = *root;
        if((*root)->left==NULL && (*root)->right==NULL) //leaf node
        {  if(key < (*proot)->info)
              (*proot)->left = NULL;
           else
              (*proot)->right = NULL;
        }
```

```
else if((*root)->left && (*root)->right)
{
    struct BST *min;
    min = minvaluenode(temp->right);
    temp->info = min->info;
    deletenode(&(temp->right), min->info, &temp);
}
else
{
    struct BST *child;
    if(temp->left != NULL)
        child = temp->left;
    else
        child = temp->right;
    if(temp->key < (*proot)->key)
        (*proot)->left = child;
    else
        (*proot)->right = child;
}
free(temp);
}
}
```

9. Algorithm to traverse threaded binary tree in inorder.

```
inorder ()
Begin
    temp = root
    repeat infinitely, do
    p = temp
    temp = right of temp
    if right flag of p is false, then
        while left flag of temp is not null, do
            temp = left of temp.
```

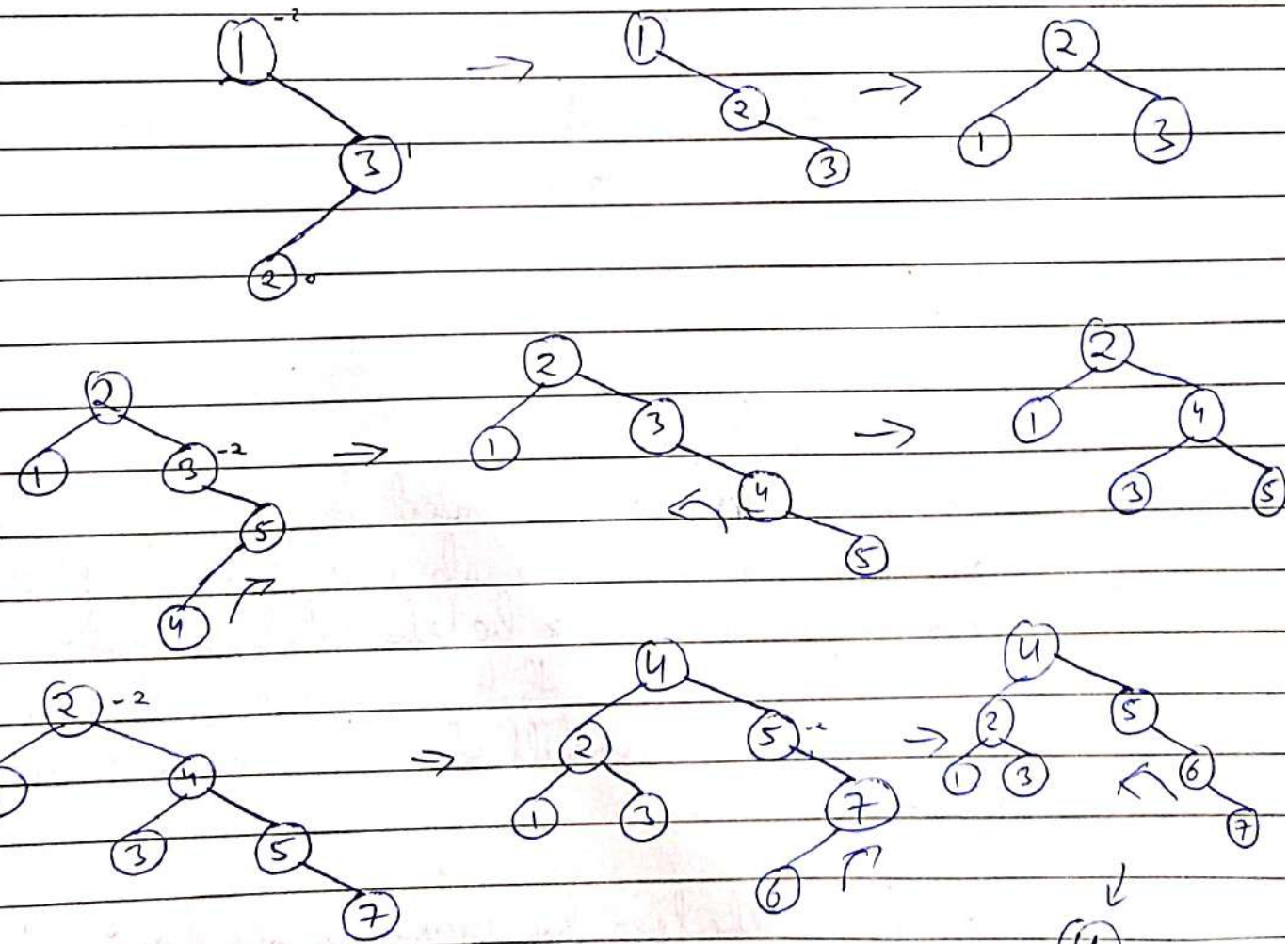
Date ___/___/___

Page No.: ___

done
end if
if temp and root are same, then
break
end if
print key of temp
done
End

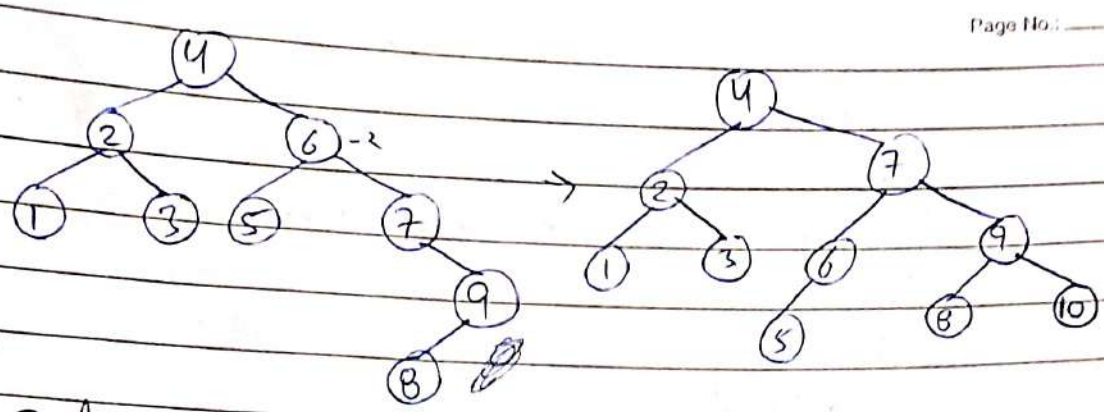
10. Draw AVL for following data

1, 3, 2, 5, 4, 7, 6, 9, 8, 10, 9



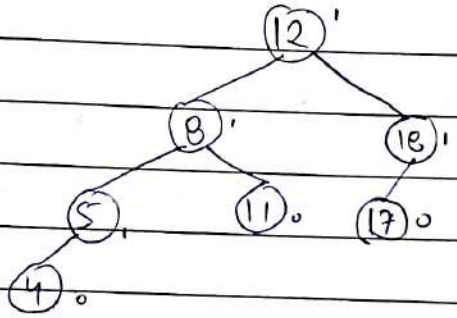
Date ___/___/___

Page No.: _____



11. Define AVL tree.
Give algorithm to insert a node in AVL Tree.

AVL tree is self-balancing Binary Search Tree where the difference between heights of left and right subtree cannot be more than one for all nodes.



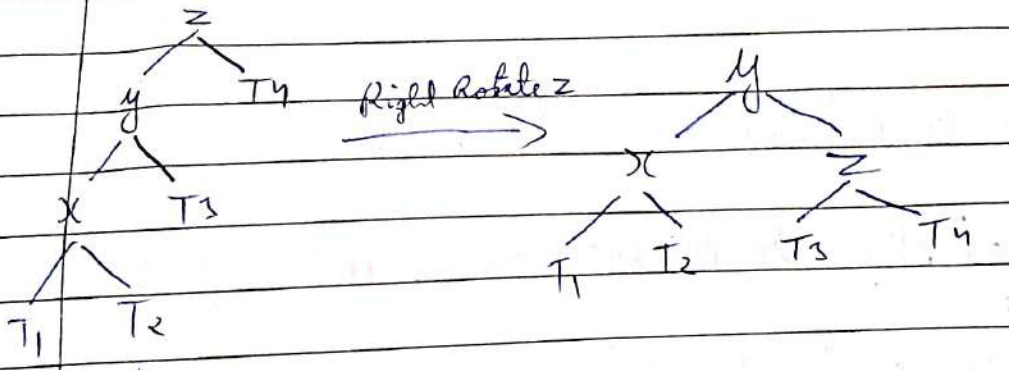
Let newly inserted node be w

1. Perform standard BST insert for w.
2. Starting from w, travel up and find the first unbalanced node. Let z be the unbalanced node, y be the child of z that comes on the path from w to z and x be the grandchild of z that comes on the path from w to z.

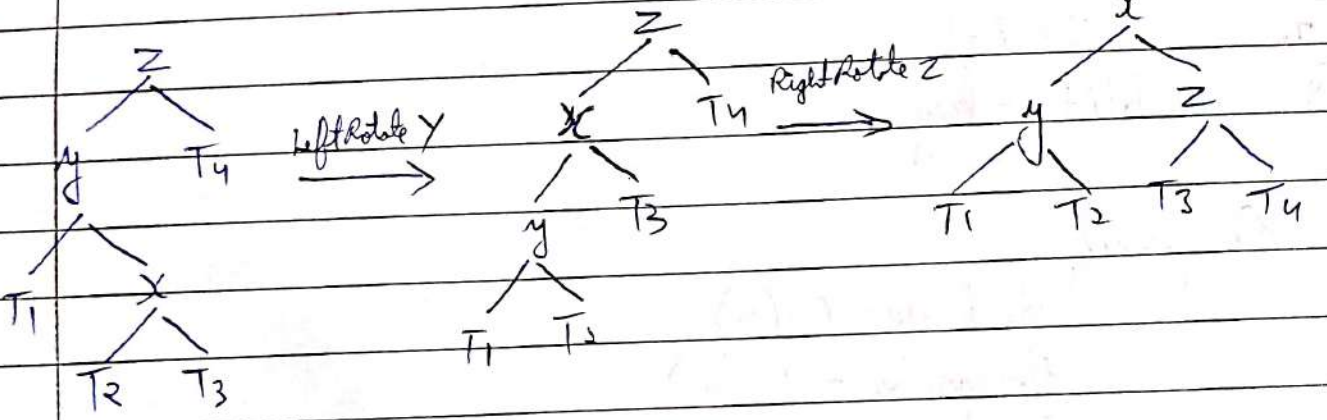
3. Re-balance the tree by performing appropriate operations rotation on the subtree rooted with z.
   (a) Left Left case
   (b) Left Right case
   (c) Right Right case (d) Right Left Case

Date __/__/____ Page No.: _____

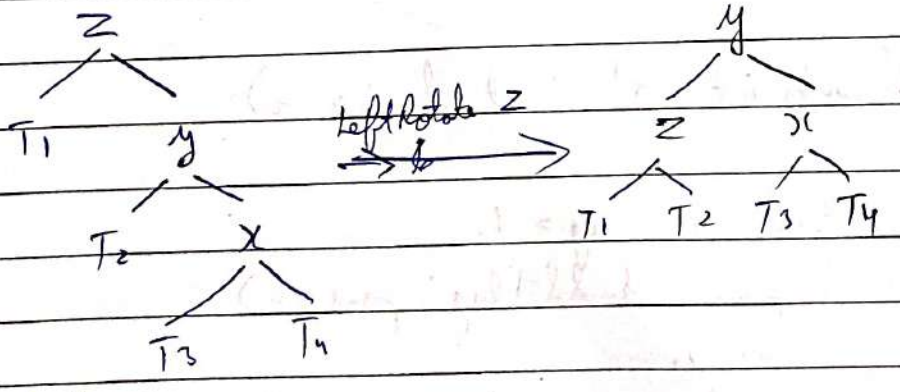
Left Left Case



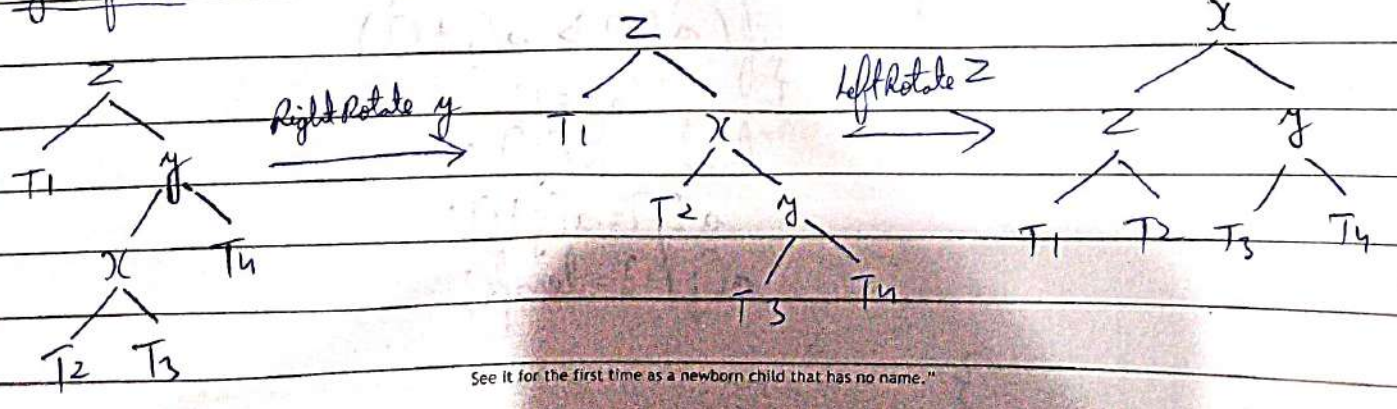
Left Right Case



Right Right case



Right Left Case



Date ___/___/___

Page No.: _____

12.

(i) Insertion Sort

```
Insertion(A)
1. for j ← 1 to A.length - 1 ~ //Right to Left
2.     key = A[j]
3.     //Insert A[j] into the sorted segment A[1---j-1]
4.     i ← j-1
5.     while i >= 0 & A[i] > key
6.         A[i+1] = A[i]
7.         i = i-1
8.     A[i+1] = key
```

Complexity

Best Case - $O(n)$

Average Case - $O(n^2)$

Worst Case - $O(n^2)$

(ii) Bubble Sort

```
void bubbleSort(int a[], int l, int r)
{
    int pass, temp, j, flag = 1;
    for(pass = l; pass < r && flag; pass ++)
    {
        flag=0;
        for (j = l; j <= r-pass; j ++)
            if(a[j] > a[j+1])
            {
                flag = 1;
                temp = a[j];
                a[j]=a[j+1];
                a[j+1] = temp;
            }
    }
}
```

Chitra

"If we have no peace, it is because we have forgotten that we belong to each other." —Mother Teresa

Date ___/___/___ Page No.: ___

Complexity

Best case $O(n)$
Average Case $O(n^2)$
Worst Case $O(n^2)$

(iii) Selection Sort

```
int min(int a[], int l, int r)
{
    int m, mi, i;
    mi = l;
    m = a[l];
    for(i = l+1; i <= r; i++)
    {
        if(m > a[i])
        {
            m = a[i];
            mi = i;
        }
    }
    return mi;
}
void selectionSort(int a[], int l, int r)
{
    int i, mi, temp;
    for(i = l; i < r; i++)
    {
        mi = min(a, i, r);
        if(i != mi)
        {
            temp = a[i];
            a[i] = a[mi];
            a[mi] = temp;
        }
    }
}
```

Complexity

Best Case $O(n^2)$
Average Case $O(n^2)$
Worst Case $O(n^2)$

(iv) Quick Sort

Page No.:

```
QuickSort(A, p, r)
1. if p < r
2.    then q ← Partition(A, p, r)
3. QuickSort(A, p, q-1);
4. QuickSort(A, q+1, r);
```

```
Partition(A, p, r)
1.  x ← A[r]
2.  i ← p-1
3.  for j ← p to r-1
7.     do if A[j] <= x
5.        then i ← i+1
6.           exchange A[i] ← A[j]
7.  exchange A[i+1] ← A[r+1]
8.  return i+1
```

Complexity

Best Case	$O(n \log n)$
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

(V) Heap Sort

```
MaxHeapify(A, i)
1. l = left(i)   // l = 2*i
2. r = right(i)
3. if l ≤ A.heapsize && A[l] > A[i]
4.         largest = l
5. else
6.         largest = i
7. if (r ≤ A.heapsize && A[r] > A[largest])
8.         largest = r
9. if larges ≠ i
       exchange A[i] with A[largest]
       MaxHeapify(A, largest)
```

```
BuildMaxHeap
1. A.heapsize = A.length
2. for i = A.length/2 down to 1
3. MaxHeapify(A, i)
```

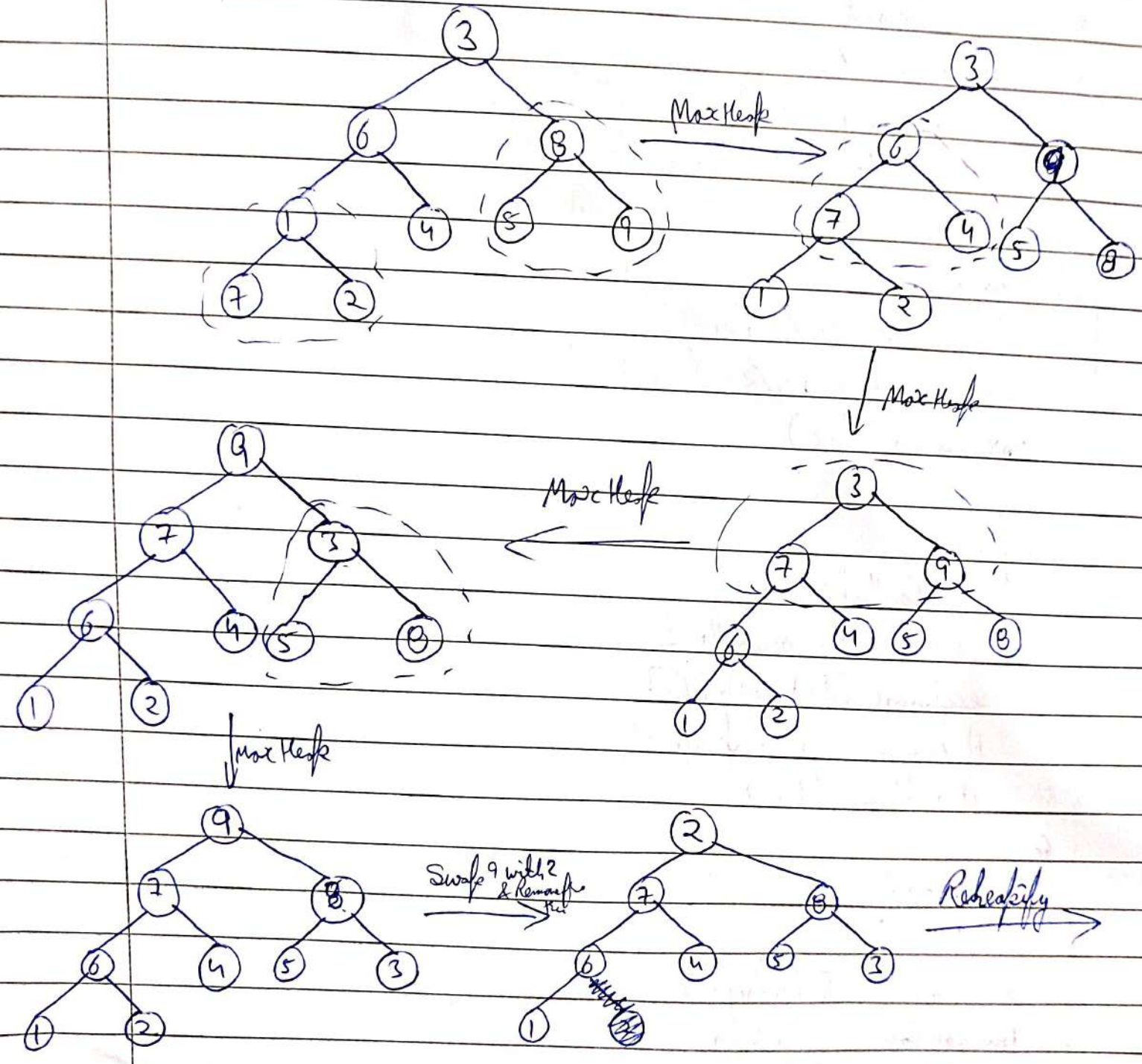
```
HeapSort
1. BuildMaxHeap(A)
2. for i = A.length down to 2
3.     exchange A[1] with A[i]
4.     A.heapsize = A.heapsize - 1;
5.     MaxHeapify(A, 1)
```

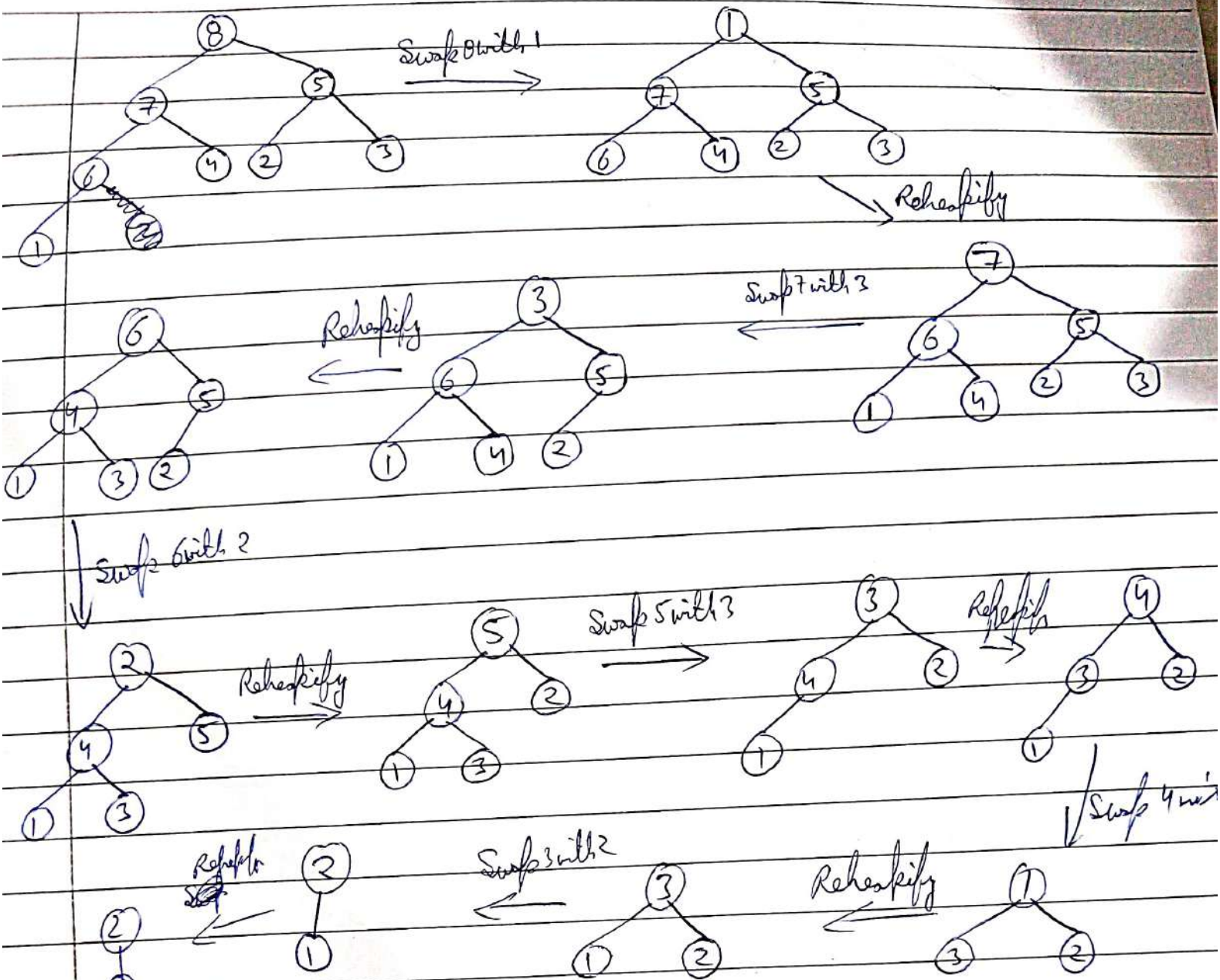
Complexity

Best Case	O(n log n)
Average Case	O(n log n)
Worst Case	O(n log n)

14. 3, 6, 8, 1, 4, 5, 9, 7, 2

Sort the data using heap sort.




Swap 8 with 1
Reheapify
Swap 7 with 3
Reheapify
Swap 6 with 2
Swap 5 with 3
Reheapify
Swap 4 with
Swap 3 with 2
Reheapify
Swap 2 with

1	2	3	4	5	6	7	8	9