PICT, PUNE

Subject: Data Structure And Algorithms

Roll. No 21118

Assignment No: 01

Date: 17/Feb/2021

Problem Definition

Beginning with an empty binary tree, construct binary tree by inserting the values in the order given. After constructing a binary tree perform following operations in it.

1) perform inorder, preorder & postorder traversal
2) Change a tree so that the roles of the left & right pointers are swapped at every node.
3) find the height of the tree
4) Copy this tree to another [operator =]
5) Count number of leaves, number of internal nodes
6) Erase all nodes in a binary tree

(both recursive & non-recursive implementations)

Learning Objectives:

1) Understand practical implementation & usage of non-linear data structure called binary tree for solving problems of different domain.
2) To implement general tree methods like tree traversals, tree height & do it's time & space complexity analysis

Learning Outcomes

1) Understand Binary tree as an ADT to design algorithm for specific problem.
2) Apply & analyze non-linear data structure binary tree to solve real world problems.

PICT, PUNE

Concept Related Theory:

Tree: i) A tree is a non-linear data structure unlike arrays, linked list & stacks.

ii) A tree can be empty with no nodes or a tree is a structure consisting of one node called the root & zero or one or more subtrees.

iii) Tree can be considered as an undirected graph where there is exactly one path from any node to any other node.

example:

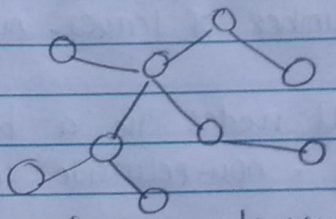


fig: represents generic tree

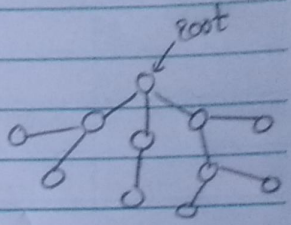


fig. represents a tree with one special node called root

Terminology:

i) Node: structure which may contain a value or condition or represent separate data structure

ii) Internal node: Any node of a tree that has child nodes.

iii) Leaf Node: Node having no child / children.

iv) Root Node: By convention trees grow downward. Topmost node in tree is called root node.

v) Ancestor: A node reachable by repeated processing from ~~parent to~~ child to parent.

Descendant: A node reachable by repeated processing from parent to child.

vi) Height of Node: Maximum number of links from

Roll. No. 21118

PICT, PUNE

given node to leaf node. &
Height of a tree is a height of root node.

vii) Level: The level of the node is the number of edges along the unique path between it & root node.

Example:

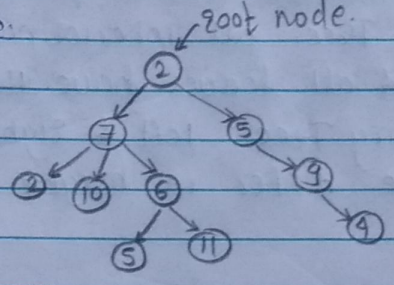


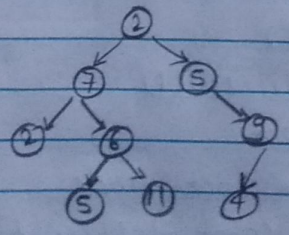
fig. A generic tree with root node.

In the diagram:
root node: 2
leaf nodes: 2, 10, 5, 11, 4
Internal nodes: 2, 7, 6, 5, 9
Height of tree: 4
Nodes according to level
level-0: 2
level-1: 7, 5
level-2: 2, 10, 6, 9
level-3: 5, 11, 4

Binary Trees:

i) A tree in which every node has at max k-childrens is known as k-ary tree. Binary tree is a special case of k-ary tree where $k=2$.

Example:



ii) Recursive definition: (non-empty) binary tree is a tuple (L,S,R), where L & R are binary trees or the empty

Rdl. No. 21118

PICT, PUNE

set S is the singleton set containing the root.

iii) Binary tree types:

Full binary tree: every node has either 0 or 2 children

Complete binary tree: every level except possibly the last, is completely filled & all nodes in the last level are as far left as possible.

Perfect Binary Tree: all interior nodes have two children & all leaves have the same depth.

Balanced Binary Tree: left & right subtrees of every node differ in height by no more than 1.

Pseudo Code

Recursive:

i) Tree Traversals:

```
Inorder(root) {
1. if (root is NULL)
       return.
2. Inorder(root->left child)
3. print data of root (or) process root node.
4. Inorder(root -> right child)

Preorder(root)
1. if(root is NULL)
       return.
2. Process root node.
3. Preorder(root->left child)
4. Preorder(root -> right child)
```

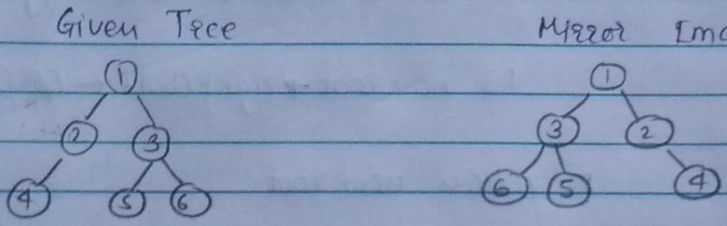
Roll No. 21118

PICT, PUNE

Postorder (root):

1. If root is NULL
   return.
2. Postorder (root->leftChild)
3. Postorder (root->rightChild)
4. Process root node.

ii) Mirror image of tree.



MirrorImage(root):

1. If root is NULL
   return
2. swap(root->leftChild, root->rightChild)
3. MirrorImage(root->leftChild)
4. MirrorImage(root->rightChild)

iii) Height of Binary Tree

Height(root): //return height of Binary Tree

1. if(root is NULL)
   return 0.
2. leftHeight = Height(root->leftChild)
3. rightHeight = Height(root->rightchild)
4. return 1+ max(leftHeight, rightHeight)

Roll. No 21118

PICT, PUNE

iv) Copying Binary Tree

```
void operator = (root):
    call copyTree method.

copyTree (new_root, old_root):
1. if (old_root is not empty):
   1.1 new_root ← new Node (old_root->data)
   1.2. new_root->leftChild ← copyTree(new_root->leftChild,
                                       old_root->leftChild)
   1.3 new_root->rightChild ← copyTree (new_root->rightChild,
                                       old_root->rightChild).
2. return new_root.
```

v) Counting Number of Leaves & Internal nodes

```
void   countNodes (root): //count total nodes in
   1. if root is NULL:          binary tree.
         return 0.
   2. leftCount ← countNodes(root->leftChild)
   3. rightCount ← countNodes (root->rightChild)
   4. return 1+ leftCount + rightCount.

countLeafNodes (root):  //count leaf nodes
   1. if root is NULL:
         return 0:
   2. leftCount ← countLeafNodes (root->leftChild).
   3. rightCount ← countLeafNodes (root->rightChild)
   4. if (root is leaf) return 1+leftCount+rightCount
      else return leftCount + rightCount;
```

Roll. No 21118

PICT, PUNE

Intermediate Nodes in tree can be calculated as
Inter nodes = total Nodes - leaf nodes.

vi) Delete Binary Tree

```
void deleteTree(root):
    1. if root is NULL:
           return.
    2. deleteTree(root->leftchild)
    3. deleteTree(root->rightchild)
    4. delete root Node.
```

Iterative / Non-recursive Implementations:

i) Tree traversals:

```
InOrder():
1. initialize empty stack. stk.
2. curr ← root Node.
3. loop until curr is not NULL or stk is not empty:
    3.1 if curr is not NULL:
        3.1.1 push curr into stack.
        3.1.2 curr ← curr->leftchild.
    3.2 else:
        3.2.1 curr ← stk.Pop
        3.2.2 process the current node
        3.2.3 curr ← curr->rightchild
```

```
Preorder():
1. initialize empty stack. stk.
2. curr ← root Node
```

3. loop until curr* is not NULL or stk is not empty:
    3.1 If curr is not NULL:
        3.1.1 process curr node
        3.1.2. push curr to stack.
        3.1.3. curr ← curr→leftchild
    3.2 else
        3.2.1 curr ← stk.pop()
        3.2.2 curr ← curr→rightchild

PostOrder(): // Using two stack.
1. create two stacks stk1 & stk2.
2. push root to stk1.
3. loop until stk1. is not empty:
    3.1 curr ← stk1.pop()
    3.2 push curr to stk2.
    3.3 If curr→leftchild is not NULL:
        3.3.1 Stk1. push (curr→leftchild)
    3.4 If curr→rightchild is not NULL:
        3.4.1 stk1. push (curr→rightchild)
4. loop while stk2. is not empty:
    4.1 curr ← stk2.pop()
    4.2 process curr Node.

11) Height of Binary Tree:
1. IF root is NULL return 0.
2. Create a new queue qu. & initialize counter to 0.
3. Push root ~~qu~~ to the qu.
4. do while qu is not empty:
    4.1 increment the counter by 1.
    4.2 Lib curr ← qu.front() & pop front from qu.

Roll. No 21118

PICT, PUNE

4.3 If curr->leftchild is not null push it to the rear of qu.

4.4 If curr->rightchild is not NULL push it to the rear of qu.

5. return the value of counter.

III) Mirror image of binary tree

```
MirrorImage(root):
1. If root is null return.
2. create an empty queue qu.
3. push root to the qu.
4. while qu is not empty:
   4.1 curr ← qu.front
       pop front from qu.
   4.2 swap(curr->leftchild, curr->rightchild)
   4.3 If curr->leftchild is not null push it
       to the rear of qu.
   4.4 If curr->rightchild is not null push it
       to the rear of qu.
```

IV) Copying Binary tree:

operator overloading

```
void operator=(Node* root): // take root of
    this->root = copyTree(root)   old tree as
                                  argument.

copyTree(old_root): // take root of old tree
1. If old_root is NULL return NULL   return root of newly created tree.
2. Create two queue qu1 & qu2.
```

Roll. No. 21118

PICT, PUNE

3. new_root ← new Node(dd_root→data)
4. push new_root to qu1 & old_root to qu2.
5. repeat until qu2 is not empty:
    5.1 old_curr ← qu2.pop()
    5.2 new_curr ← qu1.pop()
    5.3 if old_curr→leftchild is not NULL:
        5.3.1 new_curr→leftchild ← new Node(old_curr→leftchild→data)
        5.3.2 push old_curr→leftchild to qu2.
        5.3.3 push new_curr→leftchild to qu1.
    5.4 if old_curr→rightchild is not NULL:
        5.4.1 new_curr→rightchild ← new Node(old_curr→rightchild→data)
        5.4.2 push old_curr→rightchild to qu2.
        5.4.3 push new_curr→rightchild to qu1.
5. return new_root.

Y) Counting nodes in Binary Tree

countLeafNodes(root) : //return count of leaf nodes
1. If root is NULL return 0.
2. create an empty queue qu.
3. push root to the qu.
4. initialize counter to 0.
5. while qu is not empty do the following:
    5.1 curr ← qu.front(), pop from queue.
    5.2 ~~initialize~~ increment the counter if curr node is leaf.
    5.3 if curr→leftchild is not NULL push it to qu.
    5.4 if curr→rightchild is not NULL push it to qu.

Roll. No. 21118

PICT, PUNE

6. Return the counter value

countInternalNodes(root): //return count of internal
1. if root is NULL return 0. nodes.
2. Create an empty queue qu.
3. push root to the qu.
4. Initialize counter to 0.
5. while qu is not empty do the following:
   5.1 curr ← qu.front(), pop from queue.
   5.2 IF curr node is not leaf increment the counter.
   5.3 If curr→leftchild is not NULL push it to the queue.
   5.4 If curr→rightchild is not NULL push it to the queue.
6. return counter value.

viz Delete Binary Tree.

DeleteTree(root):
1. If root is NULL.
2. create an empty queue, qu
3. push root to the qu.
4. while qu is not empty do following:
   5.1 curr ← qu.front(), pop from queue.
   5.2 If curr→leftchild is not NULL push it to the queue.
   5.3 If curr→rightchild is not NULL push it to the queue.
   5.4 release memory allocated to curr.

PICT, PUNE

Roll No. 21118

Recursive way of creating Binary Tree

CreateTreeRec(*): // return root of created tree.

1. Ask the user for data (@@ -1 if user don't want any data).
2. x ← data by user
3. If (x is -1) return NULL.
4. create newNode with data field as x.
5. Tell user to enter data for leftchild of newNode & call createTreeRec()
   Also assign return value to leftchild. i.e.
   newNode->leftchild ← createTreeRec().
6. Repeat step 5 for the rightchild.
7. return newNode.

Iterative way of creative binary tree:

CreateTreeIt(): // return root of created tree.

1. Ask user for data of root (@@ -1 if user don't want any data)
2. x ← data by user.
3. if (x is -1) return NULL:
4. new-root ← new Node(x)
5. create a queue. qu.
6. push root to qu.
5. do while qu is not empty:
   5.1 curr ← qu.front(); pop from qu.
   5.2 ask data for leftchild of curr.
   5.2.1 if (data != -1)

Roll. No. 21118

curr→leftchild ← newnode with data.

5.2 Repeat step 5.1 for rightchild of curr.

6. return new.root.

Time & space complexity Analysis:

Recursive:

Method	Time Complexity	Auxiliary Space.
1) Tree traversals (All three) n→ number of nodes in tree.	Since in algorithm we are visiting each node $\theta(n)$ time is required	O(height of tree) space is required for recursion stack
2) Mirror image of tree. n→no. of nodes in tree.	we are visiting each node hence $\theta(n)$ time is required	for recursion stack O(height of tree) space is required.
3) Height of Binary Tree	Again since we are visiting each node $\theta(n)$ time is required.	for recursion stack O(height of tree) space is required.
4) Copying Binary Tree	In this method also since we need to visit all nodes $\theta(n)$ time is required.	for recursion stack O(height of tree) space is required
5) Count Nodes (Both internal & leaf)	We have to visit each node & check if it is internal / leaf. This requires $\theta(n)$ time. Checking for leaf node is const time operation.	for recursive stack O(height of tree) space is required.
6) Delete a tree.	We need to visit each	for recursive stack

Roll. No. 21118

PICT, PUNE

	node & release it's memory. Releasing memory is const time operation. Visiting all nodes take $\theta(n)$ time.	O(height of tree) space will be required.

Iterative Methods: (for all operation: n = total nodes in tree, h = height of tree)

Method	Time complexity	Aux. Space.
1) Tree traversals.		
a) Inorder	Since we are visiting all nodes $\theta(n)$ time will be required	In worst case all nodes will be pushed on stack. (eg. skew trees) Aux. space required will be $\theta(h)$
b) Preorder	Same as inorder $\theta(n)$ time will be required.	Same as inorder $\theta(h)$ ~~time~~ aux. space will be required.
c) postorder (two stack method)	The time required will be $\theta(n)$ since we visiting all nodes	The space required will be $\theta(n)$. At the end of algorithm second stack contain all the nodes.
2) Mirror image.	we are visiting each node & swapping pointers hence $\theta(n)$ time will be required.	In worst case the space occupied by the queue is $O(n)$. which is only additional space algorithm requires.
3) Height of tree.	Since we are doing level order traversal	The space will be $O(n)$ since it's just

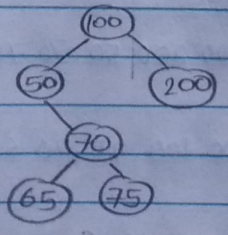
Roll. No. 21118

PICT, PUNE

	O(n) time will be required.	level order traversal with few extra variables.
4) Copying binary Tree.	O(n) time will be required.	O(n) space will be required for aux. data structures.
5) Counting nodes (both leaf nodes & internal nodes)	O(n) time will be required. (level order traversal)	Space required will be O(n) by the queue data structure.
6) Delete Binary Tree.	O(n) time will be required (level order Traversal)	Algorithm requires O(n) auxiliary space.

Testcases:

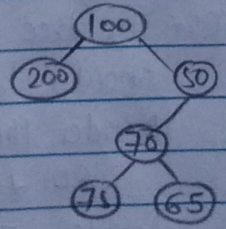
① Tree structure: generic binary tree.



Testcase	Expected output	Actual Output
1) Iterative creation	The tree traversals should be	The tree traversals are
	Preorder: 100 50 70 65 75 200	Preorder: 100 50 70 65 75 200
	Inorder: 50 65 70 75 100 200	Inorder: 50 65 70 75 100 200

Roll No. 21118

PICT, PUNE

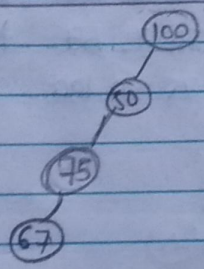
	Postorder: 65 75 70 50 200 100.	Postorder: 65 75 70 50 200 100
2) Height of Tree.	By recursive method: 4 The Tree height is 4	By iterative method: 4 By recursive method: 4
3) Copying a tree.	The tree traversals of copied tree & original tree should be same ie same as in testcase1	copying tree by iterative method: Preorder: 100 50 70 65 75 200 Inorder: 50 65 70 75 100 200 Postorder: 65 75 70 50 200 100
		Copying tree by recursive method: Preorder: 100 50 70 65 75 200 Inorder: 50 65 70 75 100 200 Postorder: 65 75 70 50 200 100
4) Mirror image.	Traversals of Mirrored tree are: Preorder: 100 200 50 70 75 65 Inorder: 200 100 75 70 65 50 Postorder: 200 75 65 70 50 100. 	Mirror Image by iterative method: Preorder: 100 200 50 70 75 65 Inorder: 200 100 75 70 65 50 Postorder: 200 75 65 70 50 100
		Mirror Image by recursive Method: Preorder: 100 200 50 70 75 65 Inorder: 200 100 75 70 65 50 Postorder: 200 75 65 70 50 100.

Roll No. 21118

PICT, PUNE

5) Counting Nodes	In given binary tree count of nodes is as follow: Total Nodes : 6 Leaf Nodes : 3 Internal Nodes : 3	By Iterative Method: Total Nodes: 6 Leaf Nodes: 3 Internal Nodes: 3 By Recursive Method: Total Nodes: 6 Leaf Nodes: 3 Internal Nodes: 3

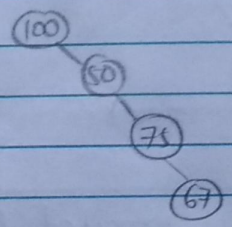
(B) Tree structure:- Skew tree.



Testcase.	Expected Output	Actual Output
1. Creating Tree.	Tree traversals should be i) Preorder: 100 50 75 67 ii) Inorder: 67 75 50 100 iii) Postorder: 67 75 50 100	After Iterative creation traversals are same as expected. After recursive creation traversals are same as expected.
2) Height of Tree	Tree height is 4	By iterative method, height of tree is 4 By Recursive method height of tree is 4
3) Copying Tree	The tree traversals of copied tree should	By iterative method, tree traversals of copied tree

PICT, PUNE

	be same as original tree (same as in testcase-1)	Preorder: 100 50 75 67 Inorder: 67 75 50 100 Postorder: 65 75 50 100 By recursive Method: preorder: 100 50 75 67 Inorda: 67 75 50 100 Postorda: 65 75 50 100
4) Mirror Image.	The traversals of mirror image of tree are: Preorder: 100 50 75 67 Inorder: 100 50 75 67 Postorder: 67 75 50 100	By both iterative & recursive methods, mirror tree has same traversals as expected.
5) Counting Nodes.	For given Binary tree, count of node is as follows: Total Nodes: 4 Leaf Nodes: 1 Internal nodes: 3	By both recursive & iterative method the count of nodes is same as expected



6) Tree structure: tree with single node.

Testcase	Expected Output	Actual Output
1) Creation	After creating tree	By both iterative &

Roll No 21118

PICT, PUNE

	Following should be tree traversals Preorder: 100 Inorder: 100 Postorder: 100	recursive methods tree traversal are printed as expected.
7) Height of tree	Height of tree is 1	By both iterative & recursive methods height of tree is 1.
8) Copying tree	Copied tree should have same traversal printed as original tree (Testcase-1)	Copied Binary tree is having same traversal printed as original tree.
9) Mirror Image	Tree traversal of mirrored tree are: Preorder: 100 Inorder: 100 Postorder: 100	The mirror tree has same traversals as expected by both iterative & recursive methods
5) Counting Nodes	Node count is as follow: Total Nodes: 1 Leaf Nodes: 1 Internal Nodes: 0	The node count is as expected by both recursive & iterative methods.

① Tree structure: empty tree.

Testcases	Expected Output	Actual Output
1) Creation.	After creation, while displaying tree program should print message saying	Programs prints message "Empty Tree" when created in both ways (iterative & recursive)

PICT, PUNE

	empty tree.	
2) Height of Tree	Height of tree should be 0.	By both recursive & iterative methods height is coming 0.
3) Copying Tree.	The copied tree should also be empty tree.	For copied tree, by both iterative & recursive methods programs shows empty tree.
4) Mirror image.	The mirror image should be empty tree	for mirror image, by both iterative & recursive methods programs displays message "Empty Tree"
5) Counting Nodes	Count for all type of nodes in tree is 0.	By both iterative & recursive methods count of nodes (all types is coming 0).

Real Time Applications of Trees:

1) Trees are used to represent hierarchical data.
   eg. folder structure, organization structure, XML/HTML content, inheritance in oop etc.
2) The specific types of trees called Binary Search Trees has lots of application like database manag. in computer science.
3) Trees can be used to create priority queue data structure which is used in offices where tasks are performed according to priority.
4) In computer networks, Minimum spanning trees are used

Roll. No.
21118

PICT, PUNE

5) Trees are also used in compilers

Conclusion:

Through this assignment, I have learned operations & applications of non-linear data structure called trees.