Name	Shubhan Singh
UID no.	2022300118
Experiment No.	5

Program 1	
PROBLEM STATEMENT :	*Construct binary tree from preorder and inorder traversal.* *Display the level order traversal of the resultant tree.*
ALGORITHM:	Algorithm for **create_node**: • Input: An integer value **val**. • Output: A new TreeNode with the given value **val**, initialized with NULL left and right children. 1. Create a new TreeNode pointer `root`. 2. Allocate memory for `root` using `malloc` to store a TreeNode structure 3. Set `root->val` to the given value `val`. 4. Set `root->left` and `root->right` to NULL. 5. Return the `root` TreeNode. Algorithm for **buildTree**: • Input: Two integer arrays **preorder** and **inorder** along with their sizes **preorderSize** and **inorderSize**. • Output: The root of the constructed binary tree. 1. If `preorderSize` is 0, return NULL, indicating an empty tree. 2. If `preorderSize` is 1, create a TreeNode with `preorder[0]` as value and return it. 3. Initialize pointers `left`, `right`, and `centre` to NULL. 4. Initialize `porder`, `iorder`, and `root` as `preorder[0]`. 5. Decrement `preorderSize` (for 0-based indexing). 6. While `preorderSize` is greater than or equal to 0: a. Set `porder` to `preorder[preorderSize]`. b. Set `iorder` to `inorder[preorderSize]`. c. If `iorder` is equal to `porder`: i. Create a TreeNode `centre` with value `porder`. ii. Set `centre->right` to `right`.

	iii. Update `right` to `centre`. iv. Decrement `preorderSize` and continue to the next iteration. d. Create a TreeNode `centre` with value `iorder`. e. If `iorder` is equal to `root`: i. Set `centre->right` to `right`. ii. Decrement `preorderSize` and break from the loop. f. Create a TreeNode `left` with value `porder`. g. Set `centre->right` to `right`. h. Set `centre->left` to `left`. i. Update `right` to `centre`. j. Decrement `preorderSize` by 2. 7. If `preorderSize` is still greater than or equal to 0: a. Recursively call `buildTree` with adjusted `preorder` and `inorder arrays for the left subtree. b. Set the left child of `centre` to the result of the recursive call. 8. Return `centre` as the root of the constructed binary tree. Algorithm for **displaytree**: • Input: The root of a binary tree, its size, the value of the last element, and a flag **nullatend** to determine if "NULL" should be printed at the end. • Output: Prints the tree in level-order traversal format. 1. Create an empty queue `q` of TreeNode pointers. 2. Enqueue the `root` into `q`. 3. Print "[" to start the representation. 4. While `q` is not empty: a. Dequeue the front element `temp` from `q`. b. If `temp` is not NULL, print its value, followed by a space. c. If `temp` has the same value as `lastelement`, break the loop. d. If `temp` is NULL, print "NULL" followed by a space. e. Enqueue `temp->left` and `temp->right` into `q` if they exist. 5. If `nullatend` is 1, print "NULL" followed by a space. 6. Print "]" to close the representation.

PROGRAM:

```
// Given two integer arrays preorder and inorder
// where preorder is the preorder traversal of a binary tree and
// inorder is the inorder traversal of the same tree, construct and
return the binary tree.


// Input: preorder = [3,9,20,15,7], inorder = [9,3,15,20,7]
// Output: [3,9,20,null,null,15,7]
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>

typedef struct TreeNode {
      int val;
      struct TreeNode *left;
      struct TreeNode *right;
  } TreeNode;

typedef struct Queue
{
    int front;
    int rear;
    unsigned size;
    TreeNode** array;
    bool prev;
}que;

struct Queue* initialize_queue(unsigned size){
    que* queue=malloc(sizeof(que));
    queue->size=size;
    queue->array=malloc(size*sizeof(TreeNode*));
    queue->rear=0;
    queue->front=0;
    queue->prev=true;
}

int isEmpty(struct Queue* queue){
    if(queue->front==queue->rear){
```

```
        if(queue->prev){
            return 1;
        }
    }
    return 0;
}

int isFull(struct Queue* queue){
    if(queue->front==queue->rear){
        if(!(queue->prev)){
            return 1;
        }
    }
    return 0;
}

void enqueue(struct Queue *queue, TreeNode* item){
    if(!isFull(queue)){
        queue->array[queue->rear]=item;
        queue->rear++;
        queue->rear%=queue->size;
        queue->prev=false;
    }
    else{
        printf("Queue if full!\n");
    }
}

TreeNode* dequeue(struct Queue* queue){
    if(!isEmpty(queue)){
        queue->prev=true;
        TreeNode* retval=queue->array[queue->front];
        queue->front++;
        queue->front%=queue->size;
        return retval;
    }
    else{
        printf("Queue is empty!\n");
```

```
        return 0;
    }
}


// complete this function
TreeNode* create_node(int val);
// complete this function
TreeNode* buildTree(int* preorder, int preorderSize, int* inorder,
int inorderSize);

TreeNode* create_node(int val){
    TreeNode* root=malloc(sizeof(TreeNode));
    root->val=val;
    root->left=root->right=NULL;
    return root;
}

TreeNode* buildTree(int* preorder, int preorderSize, int* inorder,
int inorderSize){
    if(preorderSize==0){return NULL;}
    if(preorderSize==1){
        TreeNode* root=create_node(preorder[0]);
        return root;
    }
    TreeNode* right=NULL;
    TreeNode* left;
    TreeNode* centre;
    int porder,iorder,root=preorder[0];
    preorderSize--;//for 0 based indexing
    while(preorderSize>=0){//constructs right subtree with root
        porder=preorder[preorderSize];
        iorder=inorder[preorderSize];
        if(iorder==porder){
            centre=create_node(porder);
            centre->right=right;
            right=centre;
            preorderSize--;
```

```
            continue;
        }
        centre=create_node(iorder);
        if(iorder==root){
            centre->right=right;
            preorderSize--;
            break;
        }
        left=create_node(porder);
        centre->right=right;
        centre->left=left;
        right=centre;
        preorderSize-=2;
    }
    if(preorderSize>=0){

left=buildTree((++preorder),preorderSize+1,inorder,preorderSize+1);
        centre->left=left;
    }
    return centre;
}

void displaytree(TreeNode* root,int size,int lastelement,int
nullatend){//level order traversal
    printf("[ ");
    que* q=initialize_queue(size);
    TreeNode* temp,left,right;
    enqueue(q,root);
    while(!isEmpty(q)){
        temp=dequeue(q);
        if(temp!=NULL){
            printf("%d ",temp->val);
            if(temp->val==lastelement){
                break;
            }
        }
        else
        printf("NULL ");
```

```
        if(temp!=NULL)
        enqueue(q,temp->left);
        if(temp!=NULL)
        enqueue(q,temp->right);
    }
    if(nullatend){
        printf("NULL ");
    }
    printf("]");
}

int main(){
    int size;
    printf("Enter number of nodes in tree: ");
    scanf("%d",&size);
    int inorder[size];
    int preorder[size];
    printf("Enter preorder traversal:\n");
    for(int i=0;i<size;i++){
        scanf("%d",&preorder[i]);
    }
    printf("Enter inorder traversal:\n");
    for(int i=0;i<size;i++){
        scanf("%d",&inorder[i]);
    }
    TreeNode* root = buildTree(preorder,size,inorder,size);
    int last,nullatend=0;
    if(inorder[size-1]==preorder[size-1]){
        last=inorder[size-1];
    }
    else{
        last=preorder[size-1];
        nullatend=1;
    }
    displaytree(root,size,last,nullatend);
    return 0;
}
```

RESULT:

```
PS C:\Users\shubh\OneDrive - Bharatiya Vidya Bhavans Sardar Patel Institute O
tute Of Technology\C programs\data structures\" ; if ($?) { gcc binary_tree_f
Enter number of nodes in tree: 5
Enter preorder traversal:
3 9 20 15 7
Enter inorder traversal:
9 3 15 20 7
[ 3 9 20 NULL NULL 15 7 ]
```

```
PS C:\Users\shubh\OneDrive - Bharatiya Vidya Bhavans Sardar Patel
tute Of Technology\C programs\data structures\" ; if ($?) { gcc b
Enter number of nodes in tree: 6
Enter preorder traversal:
3 15 16 5 20 7
Enter inorder traversal:
16 15 5 3 7 20
[ 3 15 20 16 5 7 NULL ]
```

Theory :	A binary tree is a fundamental data structure used in computer science and is composed of nodes that have at most two children, referred to as the left child and the right child. Binary trees are widely used in various applications and algorithms, including searching, sorting, and data representation. Here's an overview of binary trees, along with explanations of three common tree traversal techniques: pre-order, post-order, and level-order traversals. **Binary Tree Structure:** A binary tree is a hierarchical structure consisting of nodes connected by edges. Each node in a binary tree has at most two child nodes, the left child and the right child. The top node is called the root, and nodes with no children are referred to as leaf nodes. Binary trees can be used to represent a wide range of hierarchical structures, including family trees, directory structures, and expression parsing trees. **Key Terminology:** • **Root:** The topmost node in a binary tree. • **Parent:** A node that has one or more child nodes. • **Child:** A node directly connected to a parent. • **Leaf:** A node with no children. • **Subtree:** A tree formed by a node and all its descendants. • **Depth:** The level at which a node is located in the tree, with the root at level 0. • **Height:** The length of the longest path from a node to a leaf. **Tree Traversal Techniques:** **Pre-order Traversal:** • In a pre-order traversal, you start at the root node and visit nodes in the following

	order: root, left subtree, right subtree. • This traversal is often used for creating a copy of the tree or expression evaluation. **Post order traversal:** • In a post-order traversal, you visit nodes in the following order: left subtree, right subtree, root. • Post-order is useful for deleting a tree, as it ensures that you visit and delete all child nodes before the parent node. **Level-order Traversal:** • In a level-order traversal, you visit nodes level by level, starting from the root and moving down through each level from left to right. • This traversal is commonly implemented using a queue data structure. • Level-order traversal is useful for tasks like breadth-first search.
CONCLUSION:	1. Binary trees and these traversal techniques are fundamental in computer science and are used in various algorithms and data structures. Understanding these concepts is crucial for solving a wide range of problems involving hierarchical data. 2. A binary tree can be reconstructed from 2 of its traversals, by use of a recursive function.