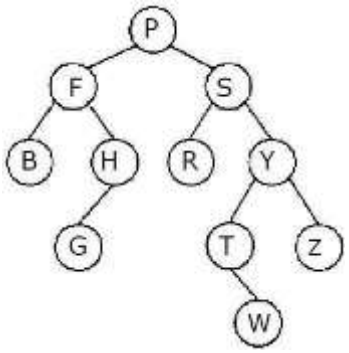
EX.NO : 8(A)	TREE REPRESENTATION AND TRAVERSALS
DATE :	

PROGRAM STATEMENT:

To write a C++ Function to perform Inorder traversal of the below given tree.



ALGORITHM:

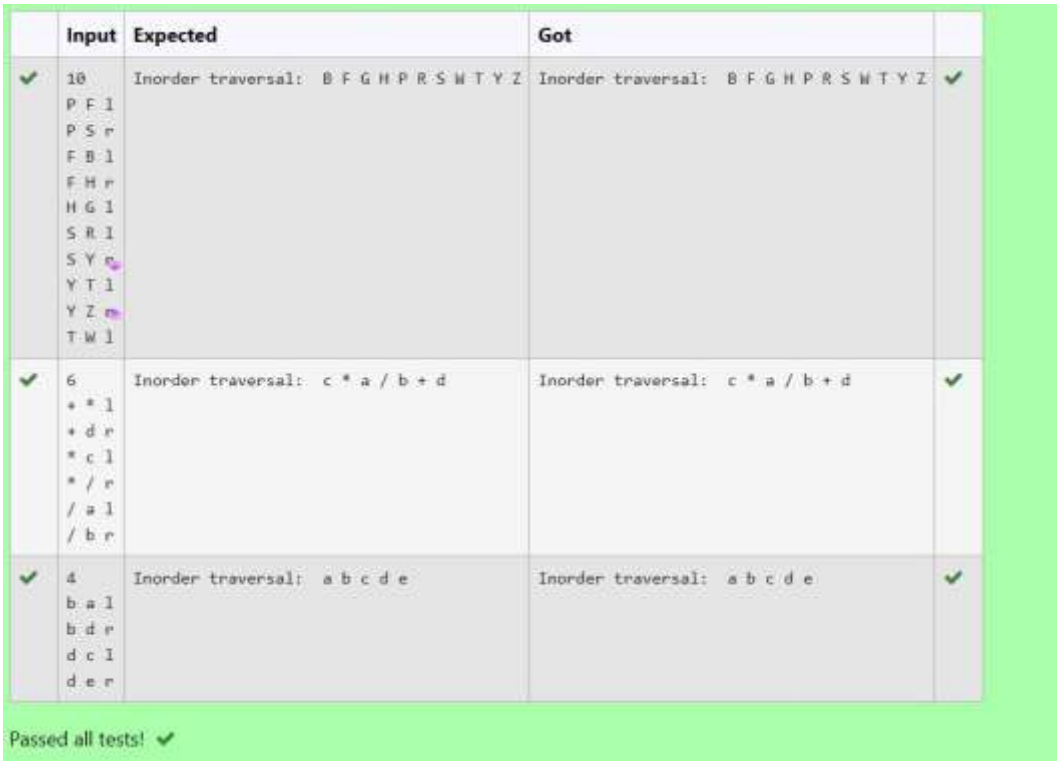
1. Start the program.
2. Define Node: Create a Node structure with data and next pointer.
3. Push: Insert a new node at the rear of the queue by updating the rear pointer.
4. Pop: Remove the front node and update the front pointer, checking for underflow.
5. Display: Print the queue from front to rear by traversing through all nodes.
6. Display Front/Rear: Print the data of the front and rear nodes.
7. Main: Perform push(), pop(), and display() operations on the queue.
8. End the program.

PROGRAM:

```
Void traverseInOrder(struct node *dis)
{
if(dis!=NULL){
   traverseInOrder(dis->left);
   cout<<" "<<dis->data;
   traverseInOrder(dis->right);
}

}
```

OUTPUT :



	Input	Expected	Got	
✔	10 P F l P S r F B l F H r H G l S R l S Y r Y T l Y Z r T W l	Inorder traversal: B F G H P R S W T Y Z	Inorder traversal: B F G H P R S W T Y Z	✔
✔	6 + * l + d r * c l * / r / a l / b r	Inorder traversal: c * a / b + d	Inorder traversal: c * a / b + d	✔
✔	4 b a l b d r d c l d e r	Inorder traversal: a b c d e	Inorder traversal: a b c d e	✔

Passed all tests! ✔

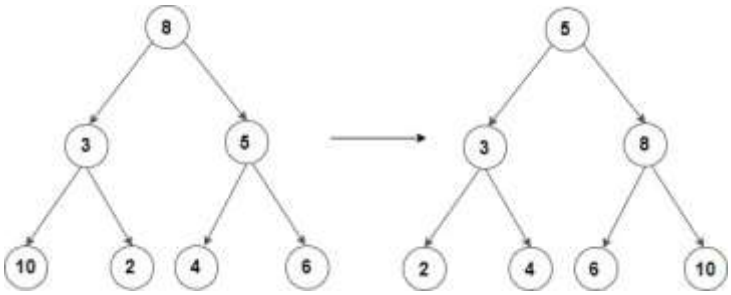
RESULT:

Thus, the C++ program to perform Inorder traversal of the below given tree is created successfully.

EX.NO : 8(B)	BINARY SEARCH TREE
DATE :	

PROGRAM STATEMENT:

To write a C++ function to convert a binary tree into a binary search tree.



ALGORITHM:

1. Start the program.
2. Define Function: Create a function convertBST that takes the root of the BST, an InOrderArray, and an index i as input.
3. Traverse Left Subtree: Recursively call convertBST on the left child of the current node.
4. Assign Data: Assign the value from InOrderArray[i] to the current node's data, then increment i.
5. Traverse Right Subtree: Recursively call convertBST on the right child of the current node.
6. End the program.

PROGRAM:

```
void convertBST(node*root,vector<int> InOrderArray,int& i)
{
if(root)
{ convertBST(root->left,InOrderArray,i);
  root->data = InOrderArray[i++];
  convertBST(root->right,InOrderArray,i);
  }
}
```

OUTPUT :

	Input	Expected	Got	
✔	6 8 3 l 3 10 l 3 2 r 8 5 r 5 4 l 5 6 r	Inorder 2 3 4 5 6 8 10 Before conversion 10 3 2 8 4 5 6 After conversion 2 3 4 5 6 8 10	Inorder 2 3 4 5 6 8 10 Before conversion 10 3 2 8 4 5 6 After conversion 2 3 4 5 6 8 10	✔
✔	6 5 7 l 5 9 r 7 1 l 7 6 r 9 10 l 9 11 r	Inorder 1 5 6 7 9 10 11 Before conversion 1 7 6 5 10 9 11 After conversion 1 5 6 7 9 10 11	Inorder 1 5 6 7 9 10 11 Before conversion 1 7 6 5 10 9 11 After conversion 1 5 6 7 9 10 11	✔
✔	4 1 12 l 1 9 r 12 5 l 12 6 r	Inorder 1 5 6 9 12 Before conversion 5 12 6 1 9 After conversion 1 5 6 9 12	Inorder 1 5 6 9 12 Before conversion 5 12 6 1 9 After conversion 1 5 6 9 12	✔

Passed all tests! ✔

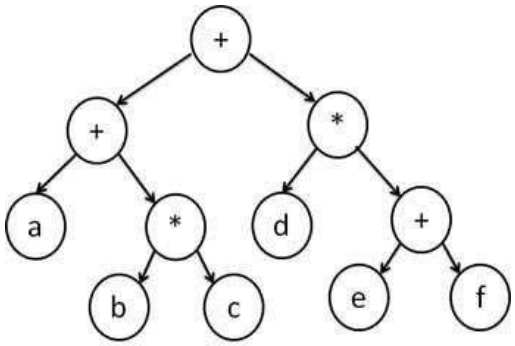
RESULT:

Thus, the C++ program to convert a binary tree into a binary search tree is created successfully.

EX.NO : 8(C)	EXPRESSION TREE
DATE :	

PROGRAM STATEMENT:

To construct an expression tree from the given postfix expression. Generate the inorder and preorder traversal of the given expression tree below



ALGORITHM:

1. Start the program.
2. Define Node Class: Create a node class with attributes value, left, right, and next. Initialize the node with a constructor to set value and set left and right to NULL.
3. Constructor: The parameterized constructor initializes the value and sets left and right pointers to NULL. The default constructor sets left and right pointers to NULL.
4. Friend Classes: Declare stack and expression_tree as friend classes so they can access private members of the node class. 5. End the program

PROGRAM:

```
class node
{
public:
char value;
node*left;
node*right;
node*next=NULL;
node(char c)
   {
   this->value=c;
   left=NULL;
   right=NULL;
```

```
    }
    node()
    {
      left=NULL;
      right=NULL;
    }
    friend class stack;
    friend class expression_tree;
};
```

OUTPUT :

	Input	Expected	Got	
✔	abc*+def+*+	The Inorder Traversal of Expression Tree: a + b * c + d * e + f The Preorder Traversal of Expression Tree: + + a * b c * d + e f	The Inorder Traversal of Expression Tree: a + b * c + d * e + f The Preorder Traversal of Expression Tree: + + a * b c * d + e f	✔
✔	abc*+def+*+	The Inorder Traversal of Expression Tree: a + b * c + d * e + f The Preorder Traversal of Expression Tree: + + a * b c * d + e f	The Inorder Traversal of Expression Tree: a + b * c + d * e + f The Preorder Traversal of Expression Tree: + + a * b c * d + e f	✔
✔	ABC*+D/	The Inorder Traversal of Expression Tree: A + B * C / D The Preorder Traversal of Expression Tree: / + A * B C D	The Inorder Traversal of Expression Tree: A + B * C / D The Preorder Traversal of Expression Tree: / + A * B C D	✔
✔	ab+cde+**	The Inorder Traversal of Expression Tree: a + b * c * d + e The Preorder Traversal of Expression Tree: * + a b * c + d e	The Inorder Traversal of Expression Tree: a + b * c * d + e The Preorder Traversal of Expression Tree: * + a b * c + d e	✔

Passed all tests! ✔

RESULT:

Thus, the C++ program to Generate the inorder and preorder traversal of the given expression tree below is created successfully.

EX.NO : 8(D)	AVL TREE
DATE :	

PROGRAM STATEMENT:

To write a C++ code to perform LL & RR rotation in an AVL Tree while inserting elements.

ALGORITHM:

1. Start the program.
2. Insert Node: Define the function insert() to insert a new key in the AVL tree. If the tree is empty, create a new node with the given key.
3. Normal BST Insertion: Traverse the tree recursively and insert the node at the appropriate position based on the comparison of the key with the current node's key.
4. Update Height: After insertion, update the height of the ancestor node based on the maximum height between the left and right child.
5. Check Balance Factor: Calculate the balance factor of the current node to check if it has become unbalanced. If unbalanced, handle it with appropriate rotations (left or right).
6. End the program.

PROGRAM:

```
`Node* insert(Node* node, int key) {
    if (node == NULL) return(newNode(key));
    if (key< node->key)
            node->left = insert(node->left, key);
    else if (key> node->key)
          node->right =insert(node->right, key);
    else
           returnnode;
    node->height = 1 + max(height(node->left), height(node->right));
    int balance=getBalance(node);
  // Left Left Case
  if(balance > 1 && key< node->left->key)
          returnrightRotate(node);

  // Right Right Case
  if(balance <-1 && key> node->right->key)
          returnleftRotate(node);

  // Left Right Case
  if(balance > 1 &&key> node->left->key)
```

```
{
        node->left = leftRotate(node->left);
        returnrightRotate(node);
}

// Right Left Case
if (balance < -1 && key< node->right->key)
{ node->right =rightRotate(node-
        >right);
        returnleftRotate(node);
        returnnode();
}
```

OUTPUT :

	Input	Expected	Got	
✔	5 10 20 15 30 5	Preorder traversal of the constructed AVL tree is 15 10 5 20 30	Preorder traversal of the constructed AVL tree is 15 10 5 20 30	✔
✔	6 10 20 30 40 50 25	Preorder traversal of the constructed AVL tree is 30 20 10 25 40 50	Preorder traversal of the constructed AVL tree is 30 20 10 25 40 50	✔
✔	7 12 8 18 5 11 17 4	Preorder traversal of the constructed AVL tree is 12 8 5 4 11 18 17	Preorder traversal of the constructed AVL tree is 12 8 5 4 11 18 17	✔
✔	9 12 8 18 5 11 17 4 7 2	Preorder traversal of the constructed AVL tree is 12 5 4 2 8 7 11 18 17	Preorder traversal of the constructed AVL tree is 12 5 4 2 8 7 11 18 17	✔

Passed all tests! ✔

RESULT:

Thus, the C++ program to perform LL & RR rotation in an AVL Tree while inserting elements is created successfully.