IT DS 201 LAB

SUBMITTED BY ADITYA SINGH 2K19/EP/005

Program 18 : Write a program to implement insertion in the AVL Tree.

CODE

```
#include<bits/stdc++.h>
using namespace std;

class Node{
    public:
        int key;
        Node *left;
        Node *right;
        int height;
};

int height(Node *N){
    if (N == NULL) return 0;
    return N->height;
}

Node* newNode(int key){
    Node* node = new Node();
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1;
    return(node);
}
```

```
Node *rightRotate(Node *y){
    Node *x = y->left;
    Node *T2 = x->right;

    x->right = y;
    y->left = T2;

    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;

    return x;
}
```

```
Node *leftRotate(Node *x){
    Node *y = x->right;
    Node *T2 = y->left;

    y->left = x;
    x->right = T2;

    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(height(y->left), height(y->right)) + 1;

    return y;
}
```

```
int getBalance(Node *N){
    if (N == NULL) return 0;
    return height(N->left) - height(N->right);
}
```

```
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
        return node;

    node->height = 1 + max(height(node->left), height(node->right));

    int balance = getBalance(node);

    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    if (balance > 1 && key > node->left->key){
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    if (balance < -1 && key < node->right->key){
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    return node;
}
```

```
void preOrder(Node *root){
    if(root != NULL){
        cout << root->key << " ";
        preOrder(root->left);
        preOrder(root->right);
    }
}
```

ALGORITHM

1) Perform the normal BST insertion.
2) The current node must be one of the ancestors of the newly inserted node. Update the height of the current node.
3) Get the balance factor (left subtree height – right subtree height) of the current node.
4) If the balance factor is greater than 1, then the current node is unbalanced and we are either in the Left-Left case or Left-Right case. To check whether it is left-left case or not, compare the newly inserted key with the key in the left subtree root.
5) If the balance factor is less than -1, then the current node is unbalanced and we are either in Right-Right case or Right-Left case. To check whether it is a Right-Right case or not, compare the newly inserted key with the key in the right subtree root.

INPUT/OUTPUT

```
int main(){
    Node *root = NULL;

    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);

    /* The constructed AVL Tree :
            30
            / \
          20 40
         / \   \
       10 25  50
    */
    cout << "Preorder traversal of the constructed AVL tree is \n";
    preOrder(root);

    return 0;
}
```

```
Preorder traversal of the constructed AVL tree is
30 20 10 25 40 50 [Finished in 5.8s]
```

Program 19 : Write a program to implement Queue Data Structure using Stack.

CODE

```
#include <bits/stdc++.h>
using namespace std;

struct Queue {
    stack<int> s1, s2;

    void enQueue(int x){
        s1.push(x);
    }

    bool empty() {
      return s1.empty() && s2.empty();
    }

    int deQueue(){
        if (s1.empty() && s2.empty()) {
            cout << "Q is empty";
            exit(0);
        }
        if (s2.empty()) {
            while (!s1.empty()) {
                s2.push(s1.top());
                s1.pop();
            }
        }
        int x = s2.top();
        s2.pop();
        return x;
    }
};
```

ALGORITHM

enQueue(q, x)
 1) Push x to stack1 (assuming size of stacks is unlimited).
Here time complexity will be O(1)

deQueue(q)
 1) If both stacks are empty then error.
 2) If stack2 is empty -
  While stack1 is not empty, push everything from stack1 to stack2.
 3) Pop the element from stack2 and return it.
Here time complexity will be O(n).

INPUT/OUTPUT

```
int main(){
    Queue q;
    q.enQueue(1);
    q.enQueue(4);
    q.enQueue(3);
    q.enQueue(9);
    q.enQueue(5);

    while(!q.empty()){
      cout << q.deQueue() << '\n';
    }

    return 0;
}
```

```
1
4
3
9
5
[Finished in 1.5s]
```

END