TRF LEVEL 2

PROGRAMMING DOMAIN

TASK – 1

Task 1.1

1.a. Algorithm for infix expression to prefix

Iterate the given expression from left to right, one character at a time

Step 1: First reverse the given expression

Step 2: If the scanned character is an operand, put it into prefix expression.

Step 3: If the scanned character is an operator and operator's stack is empty, push operator into operators' stack.

Step 4: If the operator's stack is not empty, there may be following possibilities.

If the precedence of scanned operator is greater than the top most operator of operator's stack, push this operator into operator 's stack.

If the precedence of scanned operator is less than the top most operator of operator's stack, pop the operators from operator's stack untill we find a low precedence operator than the scanned character.

If the precedence of scanned operator is equal then check the associativity of the operator. If associativity left to right then simply put into stack. If associativity right to left then pop the operators from stack until we find a low precedence operator.

If the scanned character is opening round bracket ('('), push it into operator's stack.

If the scanned character is closing round bracket (')'), pop out operators from operator's stack until we find an opening bracket ('(').

Repeat Step 2,3 and 4 till expression has character

Step 5: Now pop out all the remaining operators from the operator's stack and push into postfix expression.

Step 6: Exit

b. **Algorithm for infix expression to postfix**

Step 1 : Scan the Infix Expression from left to right.

Step 2 : If the scanned character is an operand, append it with final Infix to Postfix string.

Step 3 : Else,

Step 3.1 : If the precedence order of the scanned(incoming) operator is greater than the precedence order of the operator in the stack (or the stack is empty or the stack contains a '(' or '[' or '{'), push it on stack.

Step 3.2 : Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)

Step 4 : If the scanned character is an '(' or '[' or '{', push it to the stack.

Step 5 : If the scanned character is an ')'or ']' or '}', pop the stack and and output it until a '(' or '[' or '{' respectively is encountered, and discard both the parenthesis.

Step 6 : Repeat steps 2-6 until infix expression is scanned.

Step 7 : Print the output

Step 8 : Pop and output from the stack until it is not empty.

c. Code for balance bracket

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

bool areBracketsBalanced(string expr)
{
    stack<char> temp;
    for (int i = 0; i < expr.length(); i++)
{
        if (temp.empty())
{

            temp.push(expr[i]);
        }
        else if ((temp.top() == '(' && expr[i] == ')')
                 || (temp.top() == '{' && expr[i] == '}')
```

```
                || (temp.top() == '[' && expr[i] == ']'))
{
            temp.pop();
        }
        else
{
            temp.push(expr[i]);
        }
    }
    if (temp.empty())
 {
        return true;
    }
    return false;
}

int main()
{
    string expr = "{()}[]";

    if (areBracketsBalanced(expr))
        cout << "Balanced";
    else
        cout << "Not Balanced";
    return 0;
}
```

2. Given "n" ropes of different lengths, connect them into a single rope with minimum cost. Assume that the cost to connect two ropes is the same as the sum of their lengths. (Hint: Use Priority Queue)

```
#include <bits/stdc++.h>

using namespace std;

int minCost(int arr[], int n)
{

    priority_queue<int, vector<int>, greater<int> > pq(
        arr, arr + n);


    int res = 0;


    while (pq.size() > 1) {

        int first = pq.top();
```

```
        pq.pop();
        int second = pq.top();
        pq.pop();

        res += first + second;
        pq.push(first + second);
    }

    return res;
}


int main()
{
    int len[] = { 4, 3, 2, 6 };
    int size = sizeof(len) / sizeof(len[0]);
    cout << "Total cost for connecting ropes is "
         << minCost(len, size);
    return 0;
}
```

3. Implement binary search tree

```
#include <iostream>
using namespace std;
struct Node {
   int data;
   Node *left;
   Node *right;
};
Node* create(int item)
{
   Node* node = new Node;
   node->data = item;
   node->left = node->right = NULL;
   return node;
}
void inorder(Node *root)
{
   if (root == NULL)
      return;
   inorder(root->left); //traverse left subtree
   cout<< root->data << "   "; //traverse root node
```

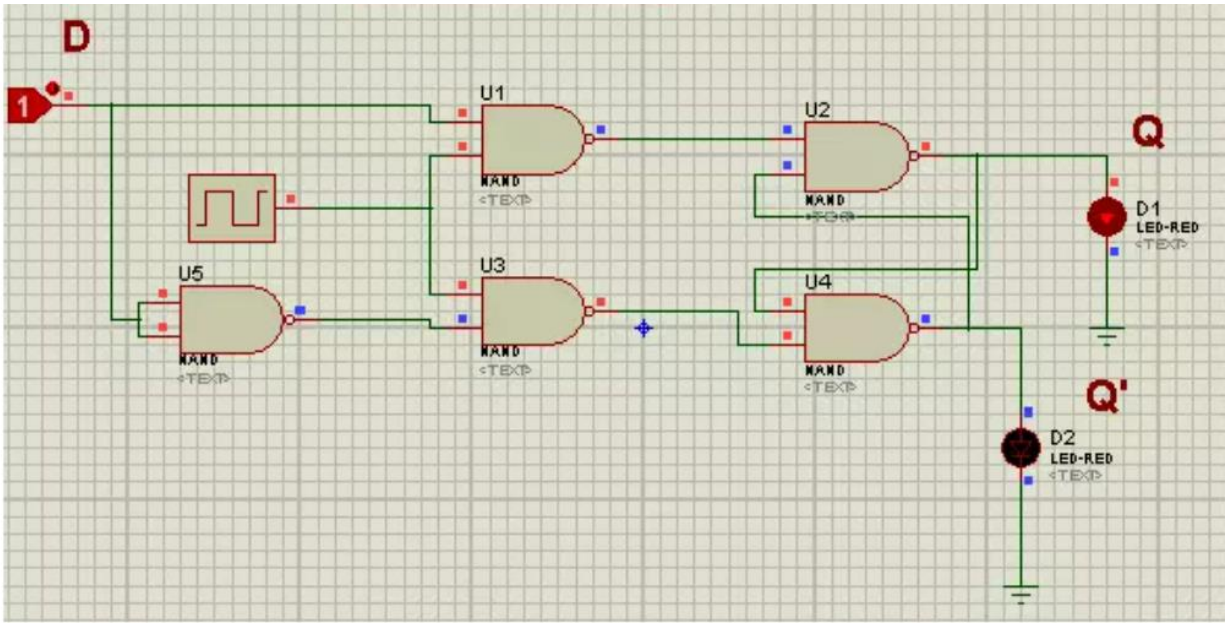
```
    inorder(root->right); //traverse right subtree
}
Node* findMinimum(Node* cur) /*To find the inorder successor*/
{
    while(cur->left != NULL) {
        cur = cur->left;
    }
    return cur;
}
Node* insertion(Node* root, int item) /*Insert a node*/
{
    if (root == NULL)
        return create(item); /*return new node if tree is empty*/
    if (item < root->data)
        root->left = insertion(root->left, item);
    else
        root->right = insertion(root->right, item);
    return root;
}
void search(Node* &cur, int item, Node* &parent)
{
    while (cur != NULL && cur->data != item)
    {
        parent = cur;
        if (item < cur->data)
            cur = cur->left;
        else
            cur = cur->right;
    }
}
void deletion(Node*& root, int item) /*function to delete a node*/
{
    Node* parent = NULL;
    Node* cur = root;
    search(cur, item, parent); /*find the node to be deleted*/
    if (cur == NULL)
        return;
```

```
    if (cur->left == NULL && cur->right == NULL) /*When node has no children*/
    {       if (cur != root)
        {
            if (parent->left == cur)
                parent->left = NULL;
            else
                parent->right = NULL;
        }
        else
            root = NULL;
        free(cur);
    }
    else if (cur->left && cur->right)
    {
        Node* succ  = findMinimum(cur->right);
        int val = succ->data;
        deletion(root, succ->data);
        cur->data = val;
    }
    else
    {
        Node* child = (cur->left)? cur->left: cur->right;
        if (cur != root)
        {
            if (cur == parent->left)
                parent->left = child;
            else
                parent->right = child;
        }
        else
            root = child;
        free(cur);
    }
}
int main()
{
  Node* root = NULL;
```

```
  root = insertion(root, 45);
  root = insertion(root, 30);
  root = insertion(root, 50);
  root = insertion(root, 25);
  root = insertion(root, 35);
  root = insertion(root, 45);
  root = insertion(root, 60);
  root = insertion(root, 4);
  printf("The inorder traversal of the given binary tree is - \n");
  inorder(root);
  deletion(root, 25);
  printf("\nAfter deleting node 25, the inorder traversal of the given binary tree is - \n
");
  inorder(root);
  insertion(root, 2);
  printf("\nAfter inserting node 2, the inorder traversal of the given binary tree is - \n"
);
  inorder(root);
  return 0;
}
```

Task 1.2

Design D flip-flops using logic gates



Submitted By:
Aryan Naik
Programming Domain
Roll No 11