TRF LEVEL 2

TASK 1

Name: Soham Akotkar

Domain: Electrical

Roll no:34

Task 1.1

1c.Algorithm for infix expression to prefix.

* First, reverse the infix expression given in the problem.
* Scan the expression from left to right and print the operands.
* If the operator arrives and the stack is found to be empty, then simply push the operator into the stack.
* If the incoming operator has higher precedence or same precedence than the TOP of the stack, push the incoming operator into the stack.
* If the incoming operator has lower precedence than the TOP of the stack, pop, and print the top of the stack. Test the incoming operator against the top of the stack again and pop the operator from the stack till it finds the operator of a lower precedence or same precedence.
* If the incoming operator has the same precedence with the top of the stack and the incoming operator is ^, then pop the top of the stack till the condition is true. If the condition is not true, push the ^ operator.
* When we reach the end of the expression, pop, and print all the operators from the top of the stack.
* If the operator is ')', then push it into the stack.
* If the operator is '(', then pop all the operators from the stack till it finds ) opening bracket in the stack.
* If the top of the stack is ')', push the operator on the stack.
* At the end, reverse the output.

1b.Algorithm for Infix expression to postfix expression.

* Scan the infix expression from left to right.
* If the scanned character is an operand, output it.
* Else,
  + If the precedence and associativity of the scanned operator are greater than the precedence and associativity of the operator in the stack (or the stack is empty or the stack contains a ‘(‘, then push it.
  + ‘^’ operator is right associative and other operators like ‘+’,’- ‘,’\*’ and ‘/’ are left-associative. Check especially for a condition when both,  operator at the top of the stack and the scanned operator are ‘^’. In this condition, the precedence of the scanned operator is higher due to its right associativity. So it will be pushed into the operator stack.
  + 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 the scanned character is an ‘(‘, push it to the stack.
* If the scanned character is an ‘)’, pop the stack and output it until a ‘(‘ is encountered, and discard both the parenthesis.
* Repeat above steps until the infix expression is scanned.
* Print the output

1c.Code for balance bracket.

Code in python:

def areBBalanced(expr):

    stack = []

    for char in expr:

        if char in ["(", "{", "["]:

             stack.append(char)

        else:

            if not stack:

                return False

            current\_char = stack.pop()

            if current\_char == '(':

                if char != ")":

                    return False

            if current\_char == '{':

                if char != "}":

                    return False

            if current\_char == '[':

                if char != "]":

                    return False

    if stack:

        return False

    return True

if \_\_name\_\_ == "\_\_main\_\_":

    expr = "{()}[]"

    if areBBalanced(expr):

        print("Balanced")

    else:

        print("Not Balanced")

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.

**import** heapq

**def** minCost(arr, n):

    heapq.heapify(arr)

  res **=** 0

**while**(len(arr) > 1):

        first **=** heapq.heappop(arr)

        second **=** heapq.heappop(arr)

        res **+=** first **+** second

        heapq.heappush(arr, first **+** second)

**return** res

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    lengths **=** [4, 3, 2, 6]

    size **=** len(lengths)

  print("Total cost for connecting ropes is " **+**

    str(minCost(lengths, size)))

3.Implement binary search tree

#include <stdio.h>

#include <stdlib.h>

**struct** node {

**int** key;

**struct** node \*left, \*right;

};

**struct** node\* newNode(**int** item)

{

**struct** node\* temp

        = (**struct** node\*)**malloc**(**sizeof**(**struct** node));

    temp->key = item;

    temp->left = temp->right = NULL;

**return** temp;

}

**void** inorder(**struct** node\* root)

{

**if** (root != NULL) {

        inorder(root->left);

**printf**("%d \n", root->key);

        inorder(root->right);

    }

}

**struct** node\* insert(**struct** 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);

**return** node;

}

**int** main()

{

**struct** node\* root = NULL;

    root = insert(root, 50);

    insert(root, 30);

    insert(root, 20);

    insert(root, 40);

    insert(root, 70);

    insert(root, 60);

    insert(root, 80);

    inorder(root);

**return** 0;

}