Graphical user interface, text, application

Description automatically generated

1. Write algorithm for

a. Infix expression to prefix

1. First, reverse the infix expression given in the problem.
2. Scan the expression from left to right.
3. Whenever the operands arrive, print them.
4. If the operator arrives and the stack is found to be empty, then simply push the operator into the stack.
5. If the incoming operator has higher precedence than the TOP of the stack, push the incoming operator into the stack.
6. If the incoming operator has the same precedence with a TOP of the stack, push the incoming operator into the stack.
7. 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.
8. 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.
9. When we reach the end of the expression, pop, and print all the operators from the top of the stack.
10. If the operator is ')', then push it into the stack.
11. If the operator is '(', then pop all the operators from the stack till it finds ) opening bracket in the stack.
12. If the top of the stack is ')', push the operator on the stack.
13. At the end, reverse the output.

.

b. Infix expression to postfix.

1. Print the operand as they arrive.
2. If the stack is empty or contains a left parenthesis on top, push the incoming operator on to the stack.
3. If the incoming symbol is '(', push it on to the stack.
4. If the incoming symbol is ')', pop the stack and print the operators until the left parenthesis is found.
5. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
6. If the incoming symbol has lower precedence than the top of the stack, pop and print the top of the stack. Then test the incoming operator against the new top of the stack.
7. If the incoming operator has the same precedence with the top of the stack then use the associativity rules. If the associativity is from left to right then pop and print the top of the stack then push the incoming operator. If the associativity is from right to left then push the incoming operator.
8. At the end of the expression, pop and print all the operators of the stack.

c. Write code for balance bracket.

1. First, we declare a character stack.
2. Convert input string into a character array.
3. Traverse the input string(By traversing the character array).
   1. We push the current character to stack if it is a starting bracket**('(' or '{' or '[')**.
   2. We pop the current character from the stack if it is a closing bracket. If the popped character doesn't match with the starting bracket, brackets are not balanced.
4. Once the traversing is finished and there are some starting brackets left in the stack, the brackets are not balanced.

3. 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(do using priority queue)

<https://www.geeksforgeeks.org/connect-n-ropes-minimum-cost/>

3.Implement binary search tree

1. Search (root, item)
2. Step 1 - if (item = root → data) or (root = NULL)
3. return root
4. else if (item < root → data)
5. return Search(root → left, item)
6. else
7. return Search(root → right, item)
8. END if
9. Step 2 - END

Code

// Binary Search Tree operations in C

#include <stdio.h>

#include <stdlib.h>

struct node {

int key;

struct node \*left, \*right;

};

// Create a node

struct node \*newNode(int item) {

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

temp->key = item;

temp->left = temp->right = NULL;

return temp;

}

// Inorder Traversal

void inorder(struct node \*root) {

if (root != NULL) {

// Traverse left

inorder(root->left);

// Traverse root

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

// Traverse right

inorder(root->right);

}

}

// Insert a node

struct node \*insert(struct node \*node, int key) {

// Return a new node if the tree is empty

if (node == NULL) return newNode(key);

// Traverse to the right place and insert the node

if (key < node->key)

node->left = insert(node->left, key);

else

node->right = insert(node->right, key);

return node;

}

// Find the inorder successor

struct node \*minValueNode(struct node \*node) {

struct node \*current = node;

// Find the leftmost leaf

while (current && current->left != NULL)

current = current->left;

return current;

}

// Deleting a node

struct node \*deleteNode(struct node \*root, int key) {

// Return if the tree is empty

if (root == NULL) return root;

// Find the node to be deleted

if (key < root->key)

root->left = deleteNode(root->left, key);

else if (key > root->key)

root->right = deleteNode(root->right, key);

else {

// If the node is with only one child or no child

if (root->left == NULL) {

struct node \*temp = root->right;

free(root);

return temp;

} else if (root->right == NULL) {

struct node \*temp = root->left;

free(root);

return temp;

}

// If the node has two children

struct node \*temp = minValueNode(root->right);

// Place the inorder successor in position of the node to be deleted

root->key = temp->key;

// Delete the inorder successor

root->right = deleteNode(root->right, temp->key);

}

return root;

}

// Driver code

int main() {

struct node \*root = NULL;

root = insert(root, 8);

root = insert(root, 3);

root = insert(root, 1);

root = insert(root, 6);

root = insert(root, 7);

root = insert(root, 10);

root = insert(root, 14);

root = insert(root, 4);

printf("Inorder traversal: ");

inorder(root);

printf("\nAfter deleting 10\n");

root = deleteNode(root, 10);

printf("Inorder traversal: ");

inorder(root);

}