//Assignment no 4

#include <iostream>

#include <string.h>

#define max 10

using namespace std;

class node

{

public:

char data;

node \*left;

node \*right;

};

class stack

{

node \*stk[max];

int top;

public:

stack()

{

top = -1;

}

int isempty();

int isfull();

void push(node \*x);

node\* pop();

void display();

};

class expr

{

node\* root;

public:

expr()

{

root = NULL;

}

node \* getroot()

{

return(root);

}

void createexpr();

void inorder(node\* root);

};

int stack :: isempty()

{

if(top == -1)

{

return 1;

}

else

{

return 0;

}

}

int stack :: isfull()

{

if(top == max-1)

{

return 1;

}

else

{

return 0;

}

}

void stack :: push(node \*x)

{

int c = isfull();

if(c != 1)

{

stk[++top] = x;

}

else

{

cout << "\nStack is full";

}

}

node\* stack :: pop()

{

node \*t;

int j = isempty();

if(j != 1)

{

t = stk[top];

return(stk[top--]);

}

else

{

return(NULL);

}

}

void stack :: display()

{

for(int i=0; i<=top; i++)

{

cout << stk[i]->data << " ";

}

}

void expr :: createexpr()

{

node\* new1;

char str[15];

int length;

stack s;

cout << "\nEnter a prefix expression: ";

cin >> str;

length = strlen(str);

for(int i=length; i>0; i--)

{

new1 = new node;

new1->data = str[i];

new1->left = NULL;

new1->right = NULL;

if(new1->data == '+' || new1->data == '-' || new1->data == '\*' || new1->data == '/' || new1->data == '%')

{

new1->left = s.pop();

new1->right = s.pop();

s.push(new1);

}

else

{

s.push(new1);

}

}

root = s.pop();

}

void expr :: inorder(node\* root)

{

if(root != NULL)

{

inorder(root->left);

cout << root->data;

inorder(root->right);

}

}

int main()

{

int ch;

stack g;

expr e;

do

{

cout << "\n--------MENU-------";

cout << "\n1. Push";

cout << "\n2. Pop";

cout << "\n3. Display";

cout << "\n4. Expression Tree";

cout << "\n5. Exit";

cout << "\nEnter your choice: ";

cin >> ch;

switch(ch)

{

case 1:

node \*nn;

nn = new node();

cout << "\nEnter element to push: ";

cin >> nn ->data;

g.push(nn);

break;

case 2:

node \*t;

t = new node();

t = g.pop();

cout << "\nElement has been popped"<<t->data;

break;

case 3:

g.display();

break;

case 4:

e.createexpr();

e.inorder(e.getroot());

break;

case 5:

exit(0);

break;

default:

cout << "\nInvalid choice entered";

}

}while(ch != 5);

return 0;

}

output:

gescoe@gescoe-OptiPlex-3010:~/Desktop/SE-A-55$ g++ expr\_tree.cpp

gescoe@gescoe-OptiPlex-3010:~/Desktop/SE-A-55$ ./a.out

--------MENU-------

1. Push
2. Pop
3. Display
4. Expression Tree
5. Exit

Enter your choice: 1

Enter element to push: 1

--------MENU-------

1. Push
2. Pop
3. Display
4. Expression Tree
5. Exit

Enter your choice: 1

Enter element to push: 2

--------MENU-------

1. Push
2. Pop
3. Display
4. Expression Tree
5. Exit

Enter your choice: 2

Element has been popped2

--------MENU-------

1. Push
2. Pop
3. Display
4. Expression Tree
5. Exit

Enter your choice: 3

1

--------MENU-------

1. Push
2. Pop
3. Display
4. Expression Tree
5. Exit

Enter your choice: 4

Enter a prefix expression: +--a\*bc/def

a-b\*c-d/e

--------MENU-------

1. Push
2. Pop
3. Display
4. Expression Tree
5. Exit

Enter your choice: 5

[gescoe@gescoe-OptiPlex-3010](mailto:gescoe@gescoe-OptiPlex-3010):~/Desktop/SE-A-55$

### Theory for Expression Tree using Stack

#### 1. ****Introduction****

An **Expression Tree** is a binary tree used to represent arithmetic expressions. In this tree:

* The internal nodes represent operators (e.g., +, -, \*, /).
* The leaf nodes represent operands (e.g., variables, constants).
* This tree is used for evaluating mathematical expressions or converting expressions from one format to another (e.g., infix to postfix or prefix).

The provided code implements a program to build and display an **Expression Tree** using a **stack** for **prefix expressions**. The prefix expressions are read, and their corresponding expression tree is created. After constructing the tree, the program uses **inorder traversal** to display the expression in a readable form.

#### 2. ****Key Concepts****

* **Prefix Expression**: The operator comes before its operands. Example: +ab.
* **Infix Expression**: The operator comes between the operands. Example: a + b.
* **Postfix Expression**: The operator comes after its operands. Example: ab+.

#### 3. ****Stack****

A **stack** is used to store nodes of the expression tree. Stacks follow the **Last In First Out (LIFO)** principle, meaning the last element added is the first one to be removed. In this program, the stack helps to manage the operands and operators as the expression is processed.

* **Push operation**: Adds a node to the stack.
* **Pop operation**: Removes a node from the stack.

#### 4. ****Expression Tree Construction****

The program processes a **prefix expression** and constructs the corresponding expression tree. The steps are as follows:

1. Read the prefix expression.
2. Process each symbol from right to left.
3. If the symbol is an operator (+, -, \*, /), pop two nodes from the stack, create a new node with the operator as its data, and set the popped nodes as the left and right children of this new node. Then push this new node onto the stack.
4. If the symbol is an operand (like a number or a variable), create a node with the operand as its data and push it onto the stack.
5. After processing all symbols, the final node left on the stack is the root of the expression tree.

#### 5. ****Inorder Traversal****

Once the expression tree is built, an **inorder traversal** is used to print the expression in a human-readable format. Inorder traversal visits the left subtree, the node, and the right subtree in this order.

#### 6. ****Algorithm for Expression Tree Construction****

##### ****Step 1: Create Expression Tree from Prefix Expression****

1. Read the prefix expression from right to left.
2. For each symbol:
   * If the symbol is an operand, create a new node and push it onto the stack.
   * If the symbol is an operator, pop two nodes from the stack, create a new node for the operator, and make the popped nodes the left and right children of the new node. Push the new node onto the stack.
3. After processing all symbols, the root of the tree is the last node remaining in the stack.

##### ****Step 2: Inorder Traversal****

1. Traverse the left subtree.
2. Visit the node and print its data.
3. Traverse the right subtree.

#### 7. ****Algorithm for the Operations****

##### ****Expression Tree Creation (from Prefix)****

1. Create an empty stack.

2. Read the prefix expression from right to left.

3. For each character in the expression:

- If the character is an operand (e.g., `a`, `b`, `1`, etc.):

- Create a new node with this character and push it onto the stack.

- If the character is an operator (`+`, `-`, `\*`, `/`):

- Pop two nodes from the stack.

- Create a new node with the operator as its data.

- Set the two popped nodes as the left and right children of this new node.

- Push the new node onto the stack.

4. The last node in the stack is the root of the expression tree.

##### ****Inorder Traversal Algorithm****

1. If the current node is NULL, return.

2. Recursively traverse the left subtree.

3. Print the current node's data.

4. Recursively traverse the right subtree.

#### 8. ****Time Complexity****

* **Expression Tree Construction**: O(n), where **n** is the number of symbols in the prefix expression. Each symbol is processed once.
* **Inorder Traversal**: O(n), where **n** is the number of nodes in the expression tree. Each node is visited once.

#### 9. ****Code Explanation****

Here is a step-by-step explanation of the code:

* **Node Class**: Represents each node of the tree, which has data (character) and two pointers: left and right.
* **Stack Class**: Implements a stack to store nodes. It has functions like push, pop, isempty, and display to manage the stack.
* **Expr Class**: Handles the creation of the expression tree from a prefix expression. It has the function createexpr() for constructing the tree and inorder() for displaying the expression.
* **Main Function**: The main menu allows the user to perform stack operations (push, pop, display) and create the expression tree.

### Code Algorithm

1. Initialize an empty stack.

2. Create an expression tree object.

3. Display a menu to the user for operations like pushing, popping, displaying, and creating an expression tree.

4. When the user chooses the expression tree option:

- Prompt for a prefix expression.

- Build the tree using the `createexpr()` method.

- Display the tree using the `inorder()` method to print the expression in infix notation.

5. Loop until the user exits.

### Conclusion

This code demonstrates the use of stacks to build an expression tree from a prefix expression. It allows for the manipulation of the tree, such as pushing and popping nodes, and provides the ability to traverse the tree in an inorder fashion to print the expression in a readable format. This is a simple and effective way to represent and evaluate mathematical expressions.