***// EXPRESSION TREE***

#include<stdlib.h>

#include<stdio.h>

#include<string.h>

typedef struct Node

{

struct Node \*right,\*left;

char data;

} Node;

typedef struct st\_Node

{

Node \*t;

struct sNode \*next;

} st;

*// Create Binary Expression tree from a string in postfix form of length len*

Node\* createExpressionTree(char\* a,int len);

*// Print the preorder traversal of the Binary Expression Tree*

void preorder(Node \*);

*// Print the postorder traversal of the Binary Expression Tree*

void postorder(Node \*);

*// Print the inorder traversal of the Binary Expression Tree*

void inorder(Node \*);

*// Free the space used by the Binary Expression Tree*

void freeTree(Node \* root);

int main()

{

char postfix[100];

int len;

Node \*root;

int loop = 1,code;

while(loop)

{

scanf("%d",&code);

switch(code)

{

case 1:

scanf("%s",postfix);

len=strlen(postfix);

root = createExpressionTree(postfix,len);

break;

case 2:

inorder(root);

printf("\n");

break;

case 3:

preorder(root);

printf("\n");

break;

case 4:

postorder(root);

printf("\n");

break;

default:

loop = 0;

break;

}

}

freeTree(root);

return 0;

}

void push(st\*\* top\_ref, Node \*t)

{

st\* new\_tNode = (st\*) malloc(sizeof(st));

if(new\_tNode == NULL)

return;

new\_tNode->t = t;

new\_tNode->next = (\*top\_ref);

(\*top\_ref) = new\_tNode;

}

int isEmpty(st \*top)

{ return (top == NULL)? 1 : 0; }

Node \*pop(st\*\* top\_ref)

{

Node \*res;

st \*top;

if(isEmpty(\*top\_ref))

return;

else

{

top = \*top\_ref;

res = top->t;

\*top\_ref = top->next;

free(top);

return res;

}

}

int isOperator(char c)

{

if(c=='\*'||c=='+'||c=='-'||c=='/'||c=='^'||c=='%')

return 1;

return 0;

}

Node\* new\_node(char data)

{

Node \*temp = (Node\*)malloc(sizeof(Node));

temp->data = data;

temp->left = NULL;

temp->right = NULL;

return temp;

}

Node\* createExpressionTree(char\* a,int len)

{

Node \*t, \*t1, \*t2;

st \*s = NULL;

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

{

if(!isOperator(a[i]))

{

t = new\_node(a[i]);

push(&s, t);

}

else

{

t = new\_node(a[i]);

t1 = pop(&s);

t2 = pop(&s);

t->right = t1;

t->left = t2;

push(&s, t);

}

}

return pop(&s);

}

void inorder(Node \*root)

{

if (root != NULL)

{

inorder(root->left);

printf("%c", root->data);

inorder(root->right);

}

}

void preorder(Node \*root)

{

if (root != NULL)

{

printf("%c", root->data);

preorder(root->left);

preorder(root->right);

}

}

void postorder(Node \*root)

{

if (root != NULL)

{

postorder(root->left);

postorder(root->right);

printf("%c", root->data);

}

}

void freeTree(Node \* root)

{

*// not mandatory*

}

struct et

{

char value;

et\* left, \*right;

};

// A utility function to check if 'c'

// is an operator

bool isOperator(char c)

{

if (c == '+' || c == '-' ||

c == '\*' || c == '/' ||

c == '^')

return true;

return false;

}

// Utility function to do inorder traversal

void inorder(et \*t)

{

if(t)

{

inorder(t->left);

printf("%c ", t->value);

inorder(t->right);

}

}

// A utility function to create a new node

et\* newNode(int v)

{

et \*temp = new et;

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

temp->value = v;

return temp;

};

// Returns root of constructed tree for given

// postfix expression

et\* constructTree(char postfix[])

{

stack<et \*> st;

et \*t, \*t1, \*t2;

// Traverse through every character of

// input expression

for (int i=0; i<strlen(postfix); i++)

{

// If operand, simply push into stack

if (!isOperator(postfix[i]))

{

t = newNode(postfix[i]);

st.push(t);

}

else // operator

{

t = newNode(postfix[i]);

// Pop two top nodes

t1 = st.top(); // Store top

st.pop(); // Remove top

t2 = st.top();

st.pop();

// make them children

t->right = t1;

t->left = t2;

// Add this subexpression to stack

st.push(t);

}

}

// only element will be root of expression

// tree

t = st.top();

st.pop();

return t;

}

// Driver program to test above

int main()

{

char postfix[] = "ab+ef\*g\*-";

et\* r = constructTree(postfix);

printf("infix expression is \n");

inorder(r);

return 0;

}