#include <iostream>

#include <string>

#include <string.h>

#include <stack>

#include <math.h>

using namespace std;

bool isOperator(char x) //Kiểm tra nó là toán tử

{

return x == '+' || x == '-' || x == '\*' || x == '/' || x == '%' || x == '^' || x == 'r' || x == '!';

}

bool isOperand(char x) //Kiểm tra nó là toán hạng

{

if ((x >= '0' && x <= '9') || x == '.')

return true;

return false;

}

int getPriority(char x) //Kiểm tra thứ tự ưu tiên của toán tử

{

if (x == '+' || x == '-') return 1;

if (x == '\*' || x == '/' || x == '%') return 2;

if (x == '^' || x == 'r' || x == '!') return 3;

return 0;

}

string toPostfix(string& str) //Đưa ra dạng Tiền tố hậu tố

{

stack<char> operatorStack; //Stack dùng để chứa các toán tử ( ) + - \* / [ ] { } % ! ^ r

string result = "#"; //Tạo một chuỗi trống, chuỗi này sẽ là chuỗi được trả về

for (int i = 0; i < str.size(); i++)

{

if (str[i] == '{') //Nếu là dấu mở ngoặc thì thêm vào stack

{

operatorStack.push(str[i]);

}

else

{

if (isOperand(str[i]))

{

result = result + str[i];

}

else

{

if (isOperator(str[i]))

{

if (str[i] == 'r')

{

result = result + " 2";

}

result += " ";

while (!operatorStack.empty() && getPriority(operatorStack.top()) >= getPriority(str[i]))

{

result = result + operatorStack.top() + " "; //Nếu là \* / > + -

operatorStack.pop();

}

operatorStack.push(str[i]);

if (str[i] == '!')

{

result = result + "1 ";

}

}

else

{

//[

if (str[i] == '[')

{

operatorStack.push(str[i]);

}

else

{

if (isOperand(str[i]))

{

result = result + str[i];

}

else

{

if (isOperator(str[i]))

{

if (str[i] == 'r')

{

result = result + " 2";

}

result += " ";

while (!operatorStack.empty() && getPriority(operatorStack.top()) >= getPriority(str[i]))

{

result = result + operatorStack.top() + " ";

operatorStack.pop();

}

operatorStack.push(str[i]);

if (str[i] == '!')

{

result = result + "1 ";

}

}

else

{

if (str[i] == '(')

{

operatorStack.push(str[i]);

}

else

{

if (isOperand(str[i]))

{

result = result + str[i];

}

else

{

if (isOperator(str[i]))

{

if (str[i] == 'r')

{

result = result + " 2";

}

result += " ";

while (!operatorStack.empty() && getPriority(operatorStack.top()) >= getPriority(str[i]))

{

result = result + operatorStack.top() + " ";

operatorStack.pop();

}

operatorStack.push(str[i]);

if (str[i] == '!')

{

result = result + "1 ";

}

}

else

{

if (str[i] == ')')

{

result += " ";

while (!operatorStack.empty() && operatorStack.top() != '(')

{

result = result + operatorStack.top() + " ";

operatorStack.pop();

}

operatorStack.pop();

}

}

}

}

if (str[i] == ']')

{

result += " ";

while (!operatorStack.empty() && operatorStack.top() != '[')

{

result = result + operatorStack.top() + " ";

operatorStack.pop();

}

operatorStack.pop();

}

}

}

}

//(

if (str[i] == '(')

{

operatorStack.push(str[i]);

}

else

{

if (isOperand(str[i]))

{

result = result + str[i];

}

else

{

if (isOperator(str[i]))

{

if (str[i] == 'r')

{

result = result + " 2";

}

result += " ";

while (!operatorStack.empty() && getPriority(operatorStack.top()) >= getPriority(str[i]))

{

result = result + operatorStack.top() + " ";

operatorStack.pop();

}

operatorStack.push(str[i]);

if (str[i] == '!')

{

result = result + "1 ";

}

}

else

{

if (str[i] == ')')

{

result += " ";

while (!operatorStack.empty() && operatorStack.top() != '(')

{

result = result + operatorStack.top() + " ";

operatorStack.pop();

}

operatorStack.pop();

}

}

}

}

//{

if (str[i] == '}')

{

result += " ";

while (!operatorStack.empty() && operatorStack.top() != '{')

{

result = result + operatorStack.top() + " ";

operatorStack.pop();

}

operatorStack.pop();

}

}

}

}

}

while (!operatorStack.empty()) //Khi xét hết biểu thức rồi, thì trên stack còn bao nhiêu ký tự, đem hết xuống result.

{

result = result + " " + operatorStack.top();

operatorStack.pop();

}

return result;

}

//đến đây xử lý được xong chuỗi, tức là đưa chuỗi về dạng hậu tố, sau đây ta sẽ đưa chúng vào cây

struct TNode

{

string data;

TNode\* Left;

TNode\* Right;

};

TNode\* CreateTNode(string x) //Tạo Node

{

TNode\* p = new TNode;

if (p == NULL)

return NULL;

else

{

p->data = x;

p->Left = p->Right = NULL;

return p;

}

}

struct Tree

{

TNode\* Root; //cây nhị phân này chỉ cần nắm bắt nút gốc, các nút còn lại thì dựa vào nút gốc

Tree()

{

this->Root = NULL;

}

};

void InOrder(TNode\*& R) //In ra màn hình theo thứ tự Left-Node-Right

{

if (R)

{

InOrder(R->Left);

cout << R->data << " ";

InOrder(R->Right);

}

}

void PostOrder(TNode\*& R) //In ra màn hình theo thứ tự Left-Right-Node

{

if (R)

{

PostOrder(R->Left);

PostOrder(R->Right);

cout << R->data << " ";

}

}

void PreOrder(TNode\*& R) //In ra màn hình theo thứ tự Node-Left-Right

{

if (R)

{

cout << R->data << " ";

PreOrder(R->Left);

PreOrder(R->Right);

}

}

void addTree(TNode\*& R, string str) //Thêm các kí tự trong chuỗi kí tự vào cây nhị phân

{

str = toPostfix(str); //Gán lại chuỗi mới sau khi chạy sắp xếp trong stack

stack<TNode\*> stackNode;

int i = 1;

while (i < str.size())

{

if (isOperand(str[i])) //Nếu gặp toán hạng thì tạo Node và cho vào stack

{

string s = "";

s += str[i];

i++;

while (isOperand(str[i]))

{

s += str[i];

i++;

}

TNode\* p = CreateTNode(s);//Node lá

stackNode.push(p);

}

else

{

if (isOperator(str[i]))

{

string s = "";

s += str[i];

TNode\* p = CreateTNode(s);

p->Right = stackNode.top();

stackNode.pop();

p->Left = stackNode.top();

stackNode.pop();

stackNode.push(p);

i++;

}

else

i++;

}

}

R = stackNode.top();

}

double GT(int x) //Hàm tính giai thừa

{

double Giaithua = 1;

for (int i = 1; i <= x; i++)

{

Giaithua = Giaithua \* i;

}

return Giaithua;

}

double toNumberbyTree(TNode\*& R) //Tính cây biểu thức

{

if (R)

{

if (isOperand(R->data[R->data.size() - 1]))

{

double num = stod(R->data);

return num;

}

if (isOperator(R->data[0]))

{

if (R->data[0] == '+')

return toNumberbyTree(R->Left) + toNumberbyTree(R->Right);

if (R->data[0] == '-')

return toNumberbyTree(R->Left) - toNumberbyTree(R->Right);

if (R->data[0] == '\*')

return toNumberbyTree(R->Left) \* toNumberbyTree(R->Right);

if (R->data[0] == '/')

return (toNumberbyTree(R->Left) / toNumberbyTree(R->Right));

if (R->data[0] == '%')

return int(toNumberbyTree(R->Left)) % int(toNumberbyTree(R->Right));

if (R->data[0] == '^')

return pow(toNumberbyTree(R->Left), toNumberbyTree(R->Right));

if (R->data[0] == '!')

return GT(toNumberbyTree(R->Left));

if (R->data[0] == 'r')

return sqrt(toNumberbyTree(R->Right));

}

}

}

double toNumberbyPostfix(string str)

{

str = toPostfix(str);

stack<double> stackNumber;

int i = 1;

while (i < str.size())

{

if (isOperand(str[i]))

{

string s = "";

s += str[i];

i++;

while (isOperand(str[i]))

{

s += str[i];

i++;

}

double num = stod(s);

stackNumber.push(num);

}

else

{

if (isOperator(str[i]))

{

if (str[i] == '+')

{

double num = stackNumber.top();

stackNumber.pop();

num = stackNumber.top() + num;

stackNumber.pop();

stackNumber.push(num);

}

if (str[i] == '-')

{

double num = stackNumber.top();

stackNumber.pop();

num = stackNumber.top() - num;

stackNumber.pop();

stackNumber.push(num);

}

if (str[i] == '\*')

{

double num = stackNumber.top();

stackNumber.pop();

num = stackNumber.top() \* num;

stackNumber.pop();

stackNumber.push(num);

}

if (str[i] == '/')

{

double num = stackNumber.top();

stackNumber.pop();

num = stackNumber.top() / num;

stackNumber.pop();

stackNumber.push(num);

}

if (str[i] == '%')

{

double num = stackNumber.top();

stackNumber.pop();

num = int(stackNumber.top()) % int(num);

stackNumber.pop();

stackNumber.push(num);

}

if (str[i] == '^')

{

double num = stackNumber.top();

stackNumber.pop();

num = pow(stackNumber.top(), num);

stackNumber.pop();

stackNumber.push(num);

}

if (str[i] == '!')

{

stackNumber.pop();

double num = GT(stackNumber.top());

stackNumber.pop();

stackNumber.push(num);

}

if (str[i] == 'r')

{

double num = sqrt(stackNumber.top());

stackNumber.pop();

stackNumber.pop();

stackNumber.push(num);

}

i++;

}

else

i++;

}

}

return stackNumber.top();

}

int main()

{

Tree Tr;

string str;

cout << "Input:\n";

getline(cin, str);

addTree(Tr.Root, str);

cout << "\nPost-fix: \n" << toPostfix(str);

cout << endl;

cout << "\nIn-Order: \n";

InOrder(Tr.Root); //LNR

cout << "\nPre-Order: \n";

PreOrder(Tr.Root); //NLR

cout << "\nPost-Order: \n";

PostOrder(Tr.Root); //LRN

cout << "\nKQ by Tree: \n" << toNumberbyTree(Tr.Root);

cout << "\nKQ by Stack: \n" << toNumberbyPostfix(str);

return 0;

}