

Project documentation

Data structure course design

——Solving arithmetic expressions

Author name: Xuedi Liu

Number: 1752985

instructor: Ying Zhang

College/Major: School of Software Engineering /Software Engineering

1. Analysis
 - 1.1 Project background analysis
 - 1.2 Project function requirements
2. Design
 - 2.1 Data structure design
 - 2.2 Class structure design
 - 2.3 Member and operational design
 1. Node
 2. Stack
3. Realization
 - 3.1 Monocular operator preprocessing
 1. Pre-processing flowchart and description
 2. Preprocessing core code
 3. Pre-processing result demonstration
 - 3.2 Infix expression is converted to a suffix expression
 1. Conversion flowchart and description
 2. Conversion function core code
 3. Suffix expression
 - 3.3 Implementation of suffix expression evaluation
 1. Evaluation diagram and description
 2. Evaluation core code
 3. Evaluation diagram
 - 3.4 Implementation of other detail functions and their code implementation
 1. Operator precedence comparison
 2. Overall system design
4. Test
 - 4.1 Function test
 1. Ordinary four arithmetic test
 2. Four arithmetic operations with parentheses
 3. Operation with power/surplus
 4. Operation that includes a monocular operator
 5. Operation containing a floating point number
 - 4.2 Boundary test
 1. The input expression does not contain an operator
 2. The remainder operation contains floating point numbers and needs to be rounded up
 - 4.3 Error test
 1. The number of left and right parentheses in the arithmetic expression does not match
 2. The divisor in the expression is 0

■ **Operating Environment:**

- Unix executables: running on **Unix** platforms
 - Linux executables: running on **Linux** platforms
 - exe executable file: Windows Console Application, running on 64-bit Windows platform
- **Code hosting platform:** Github

1. Analysis

1.1 Project background analysis

The arithmetic expression we usually use is an infix expression, such as $1+3*2$. The characteristic of the infix expression is that the binary operator is always **placed between the two operands associated with it**. This kind of expression is better understood by people, but the computer is very cumbersome to process. The order of operations is often determined by the content of the expression, and it is not regular. So in order to facilitate the computer to solve the value of the infix expression, we consider converting the infix expression into a format that is easy for the computer to understand, and then performing the evaluation of the expression.

1.2 Project function requirements

One expression that is clearly computationally advantageous compared to the infix expression is the suffix expression (aka inverse Polish expression). The characteristics of the suffix expression are: each operator is placed after its operand, and the expression does not contain parentheses. The order of the operators is related to the order of calculation, with the above infix expression $1+2*3$ as For example, the conversion to the suffix expression is $123*+$. For the generated suffix expression, we can easily evaluate the number before the operator based on the read operator.

2. Design

2.1 Data structure design

From the analysis, the main operation we need to perform is:

1. Convert the infix expression to a suffix expression
2. Find the value of the suffix expression

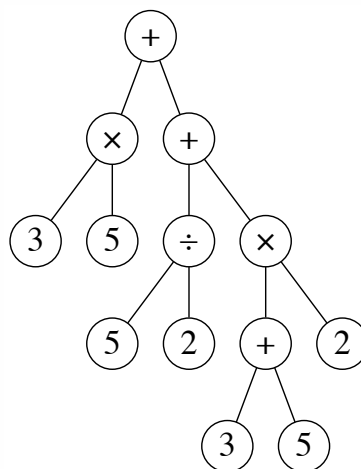
But because not all expressions consist of simple single digits with binocular operators, in addition to the monocular operator given in the question, this problem also needs to process a step:

0. Preprocess the infix expression

Among them, the following parts need special treatment.:

1. Dealing with leading monocular operator "+"、 "-"
2. Dealing with multi-digit
3. Dealing with fractional (including rounding when doing fractional operations on decimals)

For pre-processed infix expressions, convert them to suffix expressions. There are two common methods. We learned in the data structure course. For a binary tree, if its leaf node stores a number and the other nodes store operators, then the binary traversal sequence of the binary tree will get an infix expression (need to be in a certain In some places, the parentheses are manually added to make the expression logically correct. Correspondingly, the pre-order traversal sequence of the binary tree corresponds to the prefix expression of the arithmetic expression, and the post-order traversal sequence corresponds to the suffix expression. Taking the expression $3 \times 5 + 5 \div 2 + (3 + 5) \times 2$ as an example, it is expressed as a binary tree, as shown in the following figure:



The process of expressing the infix expression as a binary tree is as follows:

1. Find the last operator in the expression, the operator with the lowest priority, as the root node.
2. In the left part of the operator found in step 1, find the operator that last performed the operation, as the left child of the root node of step 1; the same operation is performed on the right part.

3. If the left or right part of the operator found in step 1 contains only the operand and does not contain an operator, the operand is used as the corresponding leaf node.
4. Repeat the above process until the binary tree contains all the operands and operators of the expression.

After the binary tree of the expression is established, the post-order traversal (left subtree -> right subtree -> root node) of the binary tree is started from the leaf node, and the suffix expression corresponding to the infix expression is obtained.

The traversal of the binary tree is divided into recursive method and non-recursive method. For the binary traversal and post-order traversal of the binary tree, we can use the **Stack** this data structure to achieve the non-recursive traversal of the binary tree. Similarly, in order to facilitate the conversion of expressions, we can directly use the stack to implement the infix expression into a postfix expression, thus avoiding the construction of the binary tree. This process is similar to the process of calculating an infix expression, using a stack save operator, and a queue structure for saving the converted postfix expression. The process is as follows:

1. Scans the expression from start to finish. If an operator is encountered, it compares the priority with the top element of the operator stack: if the current operator priority is higher than the priority of the top element, it is pushed into the operator stack; otherwise The pop-up operator top element is pushed into the result queue until the top element's priority is lower than the current operator's priority, and the current operator is pushed into the operator stack.
2. If an operand is encountered, it is directly pushed into the result stack.
3. If you encounter a left parenthesis, push it into the operator stack
4. If the right parenthesis is encountered, the top element of the operator stack is popped into the result queue in turn, until the left parenthesis is encountered, and the left parenthesis is popped up into the operator stack.

For the calculation of the suffix expression, the implementation idea is:

1. If it is an operand, push it into the Results Stack.
2. If it is an operator, take two elements from the result stack and do the calculation.(note that the order of the elements from the stack is the opposite of the order of the operands)

After traversing the suffix expression, the elements in the result stack are the final result.

2.2 Class structure design

A stack is a special container for storing data objects. The data elements are arranged in a linear logical order, so the first and last elements can also be defined. However, although the stack structure also supports the insertion and deletion of objects, its scope of operation is limited to a specific end of the stack. That is to say, if a new element can only be inserted from one end, then the existing element can only be deleted from this end. The other end of the operation is prohibited, called the blind end.

As an abstract data type, the common operations required by the stack are the stacking of the elements, the stacking of the top elements of the stack, and the return of the values of the top elements of the stack. The stack implementation can be based on a singly linked list: setting a sentinel header node, each time the push operation is equivalent to inserting a new node after the header node, and

popping the node after deleting the header node.

Since this problem requires a stack structure of multiple data types, the data structure is implemented by a template class.

2.3 Member and operational design

1. Node

```
template<typename T>
struct Node {
    T data; //Node data
    Node<T> *next; //Point to the next pointer
};
```

2. Stack

Private members:

```
private:
    Node<T> *head; //Head pointer
    int _size; //Stack size
```

Public operation:

```
public:
    Stack();

    int size(); //Return the size of the stack
    bool empty(); //Returns whether the stack is empty
    void push(T e); //Insert an element
    //↑Stepping on the pit: Although it is better to pass the pointer in theory,
    but according to the code habit, we generally use the value to pass the pit: here,
    although the pointer is better in theory, but according to the code habit, we
    generally use the value to pass
    void pop(); //Pop-up top element
    T top(); //Returns the top element value of the stack
```

Core function:

Element stack:

```

template<typename T>
void Stack<T>::push(T e) {
    auto n = new Node<T>;
    n->data = e;
    n->next = this->head->next;
    this->head->next = n;
    this->_size++;
}

```

Stack top element popping:

```

template<typename T>
void Stack<T>::pop() {
    if (!this->empty()) {
        auto p = head->next;
        head->next = head->next->next;
        delete p;
    }
    this->_size--;
}

```

Return to the top element of the stack:

```

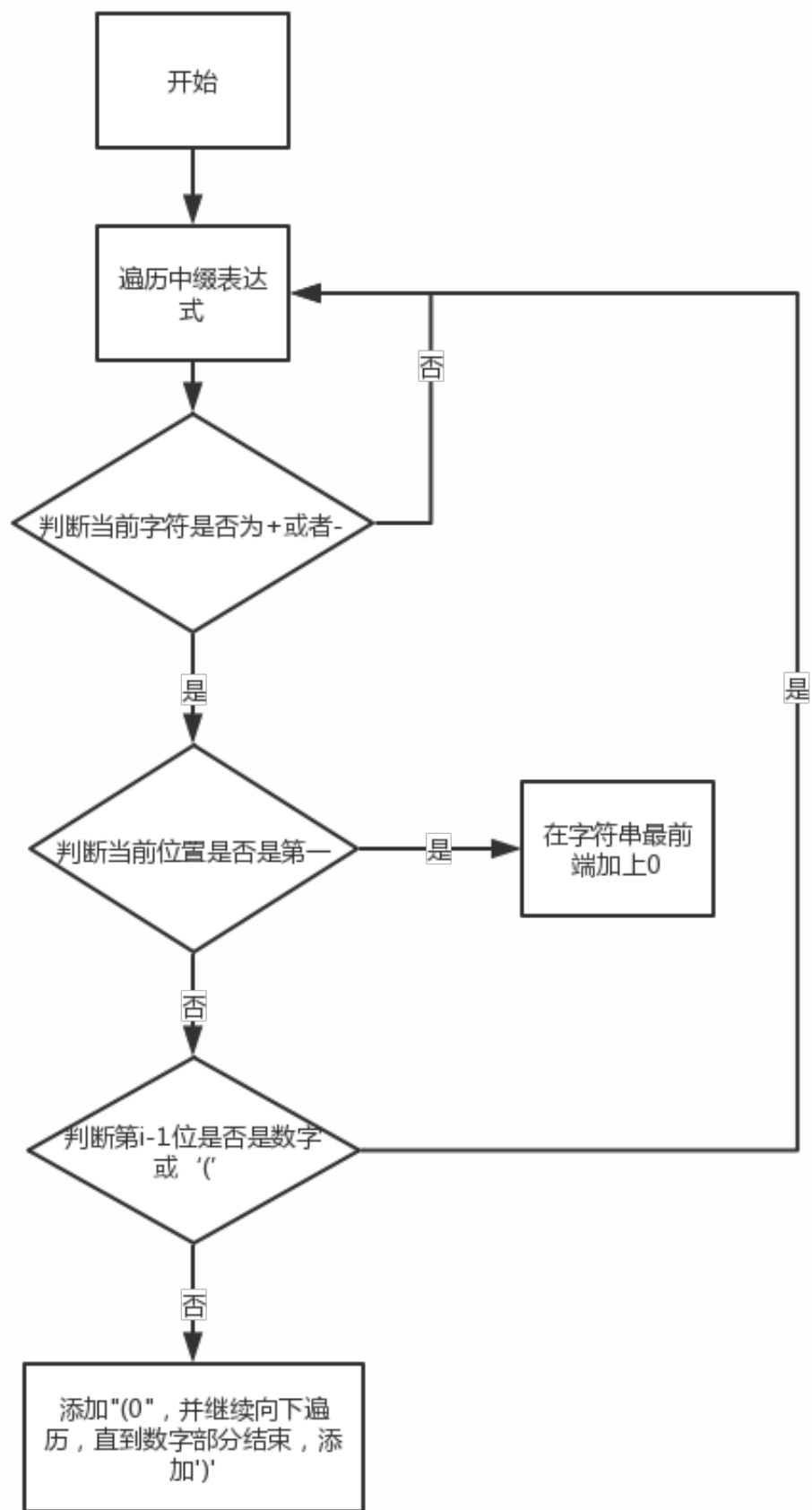
template<typename T>
T Stack<T>::top() {
    if (!this->empty())
        return this->head->next->data;
}

```

3. Realization

3.1 Monocular operator preprocessing

1. Pre-processing flowchart and description



2. Preprocessing core code

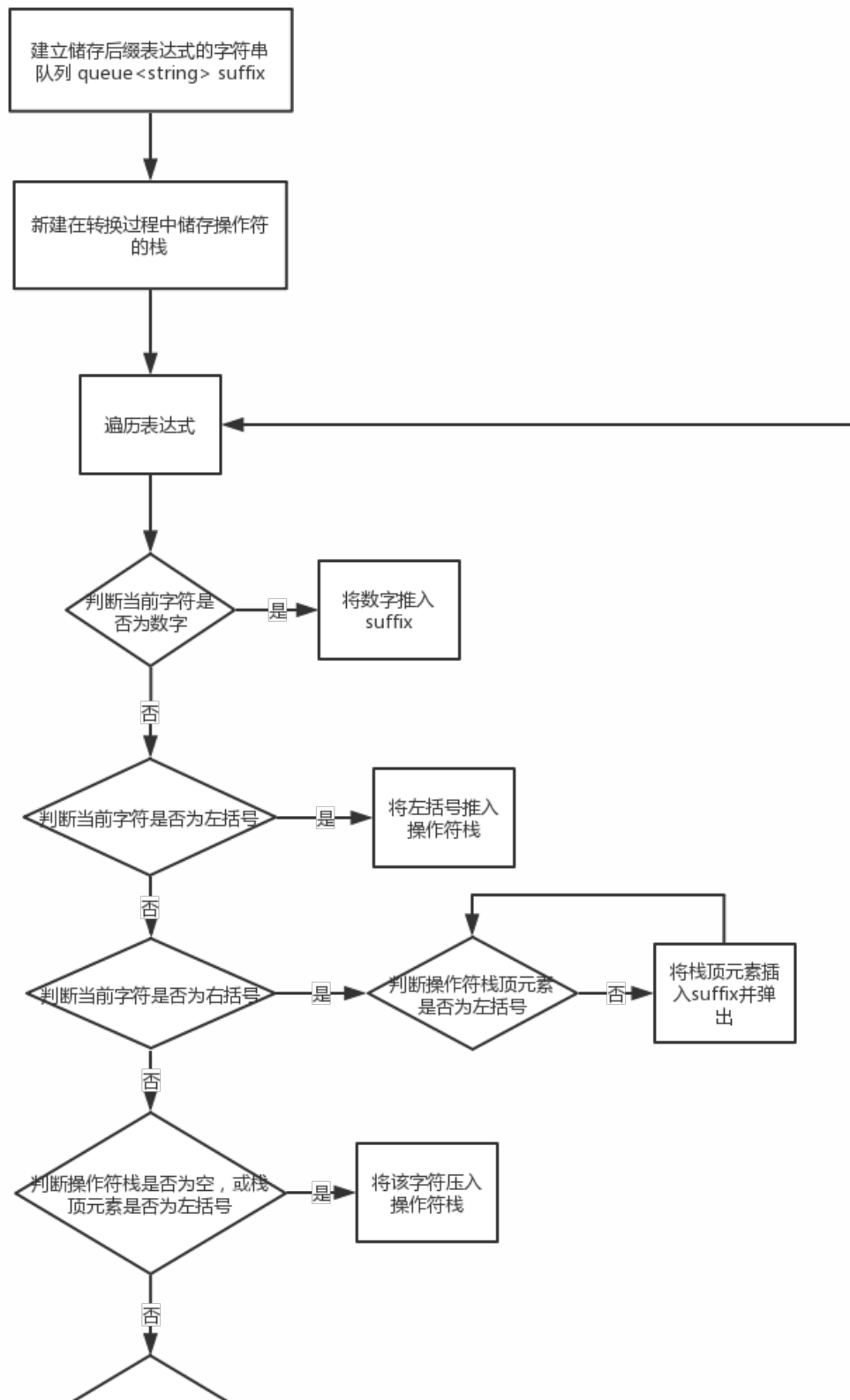
```
//Process the unary operator and return the processed string
string monoOpProcess(string s) {
    for (int i = 0; i < s.size(); ++i) {
        if (s[i] != '+' && s[i] != '-') {
            continue;
        }
        if (i == 0) {
            s.insert(i, 1, 0 + '0');
        } else if (!isNumber(s[i - 1]) && s[i - 1] != '(') {
            s.insert(i, "(0");
            for (i = i + 3; i <= s.size(); ++i) {
                if (isNumber(s[i]) || s[i] == '.') {
                    continue;
                } else {
                    s.insert(i, ")");
                    break;
                }
            }
        }
    }
    return s;
}
```

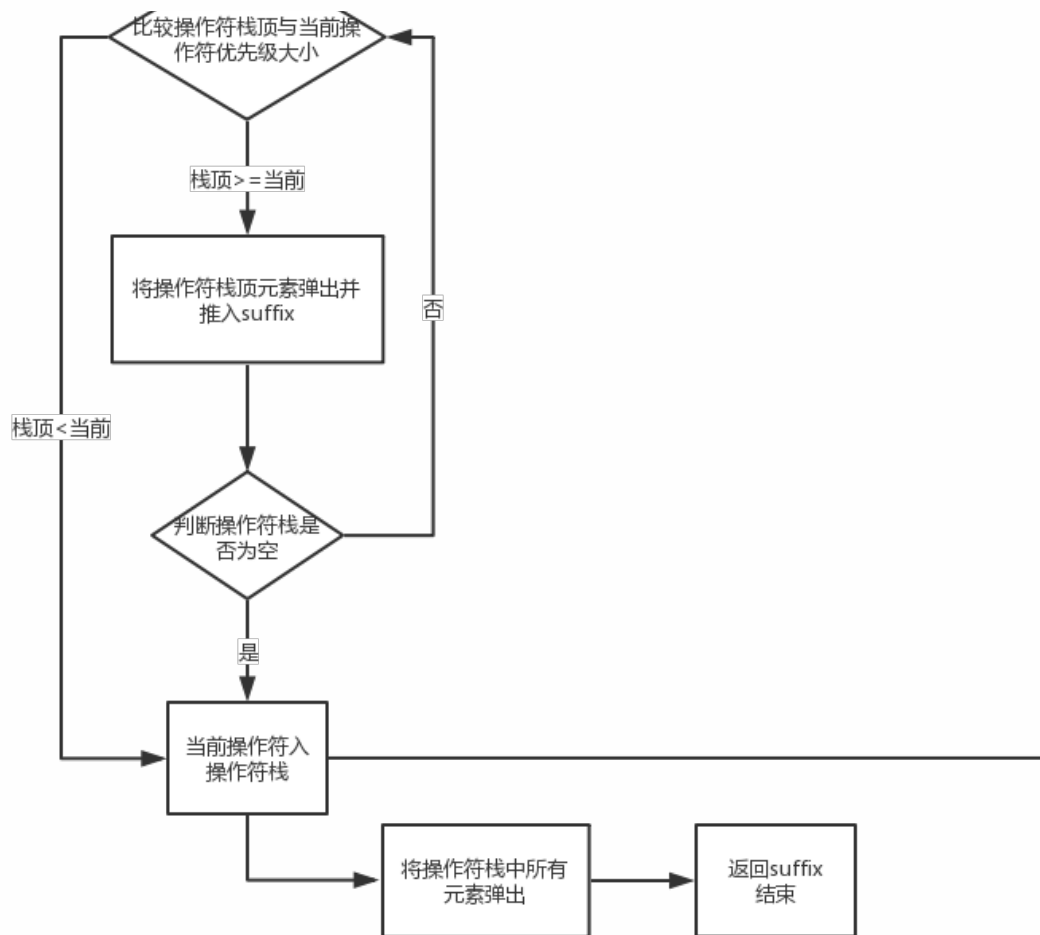
3. Pre-processing result demonstration

```
输入表达式：
-2*(3+5)+2^3/4=
处理后的表达式为：0-2*(3+5)+2^3/4
是否继续(y/n)?
y
输入表达式：
2^4/8-/(+2+8)%3=
处理后的表达式为：2^4/8-((0+2)+8)%3
```

3.2 Infix expression is converted to a suffix expression

1. Conversion flowchart and description





2. Conversion function core code

```

queue<string> getSuffix(string s) {
    queue<string> suffix; //Generated suffix expression
    auto op = new Stack<char>;
    for (int i = 0; i < s.size(); ++i) {
        if (isNumber(s[i])) {
            string num = string(1, s[i]);
            int j = i + 1;
            for (; j < s.size(); ++j) {
                if (!isNumber(s[j]) && s[j] != '.') {
                    break;
                }
                num = num + s[j];
                i = j;
            }
            suffix.push(num);
        } else if (s[i] == '(') {
            op->push(s[i]);
        } else if (s[i] == ')') {
            while (op->top() != '(') {
                suffix.push(string(1, op->top()));
                op->pop();
            }
        }
    }
}

```

```

        op->pop();//Put the left parenthesis too
    } else {
        if (op->empty() || op->top() == '(') {
            op->push(s[i]);
        } else {
            while (cmp(op->top(), s[i])) {
                suffix.push(string(1, op->top()));
                op->pop();
                if (op->empty()) {
                    break;
                }
            }
            op->push(s[i]);
        }
    }
}
while (!op->empty()) {
    suffix.push(string(1, op->top()));
    op->pop();
}
return suffix;
}

```

3. Suffix expression

输入表达式:

$-2*(3+5)+2^3/4=$

处理后的表达式为: $0-2*(3+5)+2^3/4$

转化后的后缀表达式: $0235+*-23^4/+$

是否继续(y/n)?

$2^4/8-(+2+8)\%3$

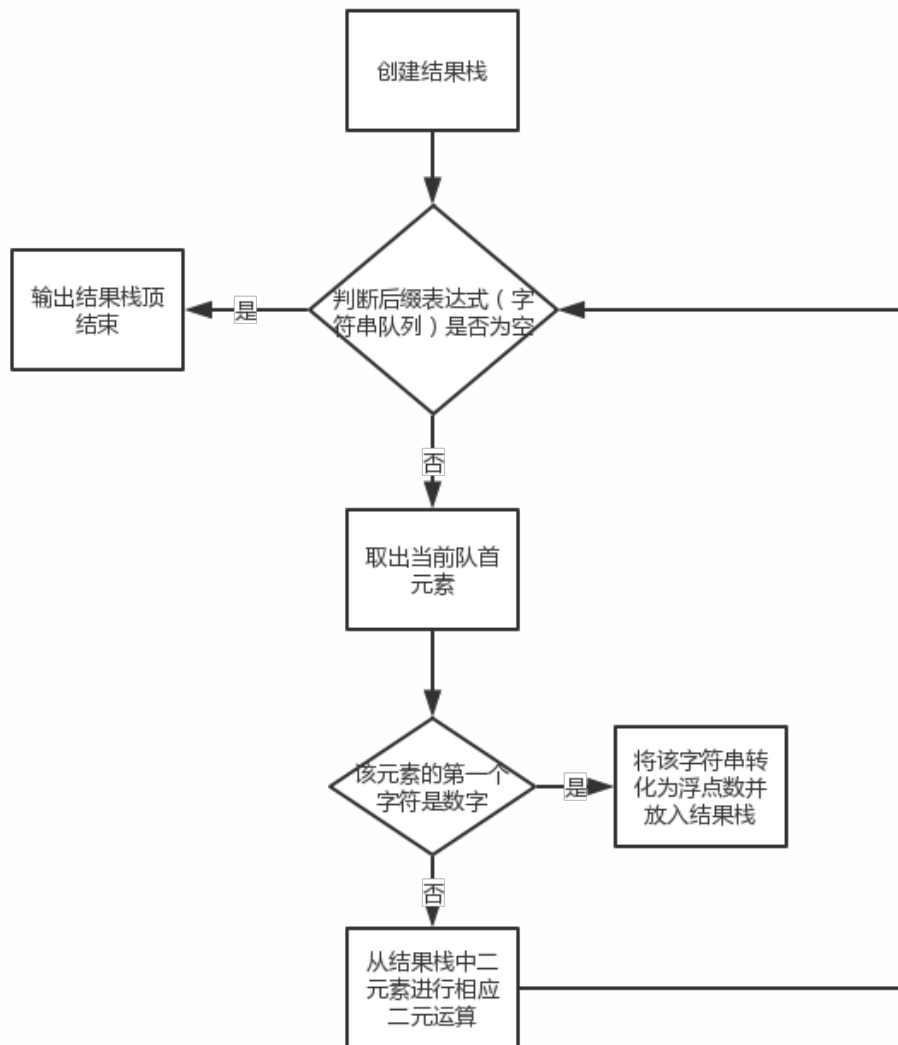
输入表达式:

处理后的表达式为: $^4/8-((0+2)+8)\%$

转化后的后缀表达式: $4^8/02+8+\%-$

3.3 Implementation of suffix expression evaluation

1. Evaluation diagram and description



It is important to note that since the data structure of the stack follows the **principle of last in, first out**, the binary operation of the two numbers taken from the top of the stack should be noted.

2. Evaluation core code

```
double calculate(double a, double b, string op) {  
    if (op == "+") {  
        return a + b;  
    } else if (op == "-") {  
        return a - b;  
    }  
}
```

```

    } else if (op == "*") {
        return a * b;
    } else if (op == "/") {
        if (b == 0) {
            cout << "除数不能为0，您输入的表达式有误" << endl;
            exit(-1);
        } else {
            return a / b;
        }
    } else if (op == "%") {
        return (int(a) % int(b));
    } else if (op == "^") {
        return pow(a, b);
    }
}

```

```

double getAns(queue<string> s) {
    Stack<double> ans; //结果栈
    while (!s.empty()) {
        string temp = s.front();
        s.pop();
        if (isnumber(temp[0])) {
            //当前取出是数字
            ans.push(atof(temp.c_str()));
        } else if (temp == "(" || temp == ")") {
            cout << "您输入的表达式括号不匹配" << endl;
            exit(-1);
        } else {
            double b = ans.top();
            ans.pop();
            double a = ans.top();
            ans.pop();
            ans.push(calculate(a, b, temp));
        }
    }
    return ans.top();
}

```

3. Evaluation diagram

```

输入表达式:
2^4/8-(+2+8)^3=
1
是否继续(y/n)?
y
输入表达式:
~2*(3+5)+2^3/4=
-14

```

3.4 Implementation of other detail functions and their code implementation

1. Operator precedence comparison

Data structure used: **map**

Set the operator to the map key, giving different values for operators of different priorities.

Core code:

```
map<char, int> optionWeight = {
    {'+', 1},
    {'-', 1},
    {'*', 2},
    {'/', 2},
    {'%', 3},
    {'^', 4}
}; //Operator priority
```

```
//When the current operator's priority is higher than or lower than the top-of-stack
operator, the stack top stack is required.
//compare stack-top with now
bool cmp(char a, char b) {
    return optionWeight[a] - optionWeight[b] >= 0;
}
```

2. Overall system design

```
string s; //表达式
while (1) {
    cout << "输入表达式: \n";
    cin >> s;
    s.pop_back(); //删除等号做处理
    queue<string> suffix; //后缀表达式
    s = monoOpProcess(s); //处理单目运算符
    suffix = getSuffix(s); //转后缀表达式
    cout << getAns(suffix) << endl;
    cout << "是否继续(y/n)?" << endl;
    char op;
    cin >> op;
    if (op == 'y') {
        continue;
    } else if (op == 'n') {
        break;
    }
}
```


	}
}	

4. Test

4.1 Function test

1. Ordinary four arithmetic test

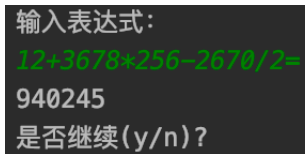
Test case:

$$12+3678*256-2670/2=$$

Expected outcome:

940245

Experimental result:



输入表达式:
 $12+3678*256-2670/2=$
940245
是否继续(y/n)?

2. Four arithmetic operations with parentheses

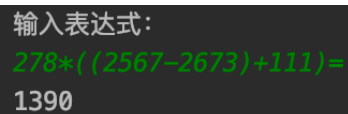
Test case:

$$278*((2567-2673)+111)=$$

Expected outcome:

1390

Experimental result:



输入表达式:
 $278*((2567-2673)+111)=$
1390

3. Operation with power/surplus

Test case:

$$3^3\%9+114514=$$

Expected outcome:

114514

Experimental result:

```
输入表达式:  
3^3%9+114514=  
114514  
是否继续(y/n)?
```

4. Operation that includes a monocular operator

Test case:

$-3^3\%9+-114514=$

Expected outcome:

-114514

Experimental result:

```
输入表达式:  
-3^3%9+-114514=  
-114514  
是否继续(y/n)?
```

5. Operation containing a floating point number

Test case:

$1.1919*114.514/7777=$

Expected outcome:

0.0175504

Experimental result:

```
输入表达式:  
1.1919*114.514/7777=  
0.0175504  
是否继续(y/n)?
```

4.2 Boundary test

1. The input expression does not contain an operator

Test case:

114514=

Expected outcome:

114514

Experimental result:

输入表达式:

114514=

114514

2. The remainder operation contains floating point numbers and needs to be rounded up

Test case:

114514.111%3.222=

Expected outcome:

1

Experimental result:

输入表达式:

114514.111%3.222=

1

是否继续(y/n)?

4.3 Error test

1. The number of left and right parentheses in the arithmetic expression does not match

Test case:

$278 * (((2567 - 2673) + 111) =$

Expected outcome:

您输入的表达式括号不匹配

Experimental result:

```
输入表达式:  
278*(( (2567-2673)+111)=  
您输入的表达式括号不匹配
```

2. The divisor in the expression is 0

Test case:

$11111 / 0 =$

$2678 \% 0 =$

Expected outcome:

除数不能为0，您输入的表达式有误

Experimental result:

```
输入表达式:  
11111/0=  
除数不能为0，您输入的表达式有误
```

```
输入表达式:  
2678%0=  
除数不能为0，您输入的表达式有误
```