



栈的应用

学习目标

栈的简单应用

栈的应用：表达式求值

栈和递归



栈的简单应用-进制转换

进制转换问题

【问题】 将十进制数转换为二进制数

【想法】 转换规则：除基取余，逆序排列

$$(23)_{10} = (10111)_2$$

$$\begin{array}{r|l} 2 & 23 \\ \hline & 11 \\ 2 & 11 \\ \hline & 5 \\ 2 & 5 \\ \hline & 2 \\ 2 & 2 \\ \hline & 1 \\ 2 & 1 \\ \hline & 0 \end{array} \begin{array}{l} 1 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \end{array}$$

 如何保存得到的余数，使之逆序？



用栈保存，从最后进栈的元素开始输出

在实际问题的处理过程中，有些数据具有后到先处理的特点

算法示例:将十进制正整数**n**转换成**2**进制输出。

```
void convert(int n) {  
    SeqStack<int> remstack;  
    while (n > 0) {  
        remstack.Push(n % 2);  
        n /= 2;  
    }  
    while (!remstack.Empty())  
        cout << remstack.Pop();  
    cout << endl;  
}
```

算法示例:将十进制正整数**n**转换成**base**进制**输出**。

```
void convert(int n,int base) {  
    SeqStack<int> remstack;  
    string digits = "0123456789ABCDEF";  
    while (n > 0) {  
        remstack.Push(n % base);  
        n /= base;  
    }  
    while (!remstack.Empty())  
        cout << digits[remstack.Pop()];  
}
```

The background of the slide features a large, faint sphere. Overlaid on the sphere is a complex network of thin, light blue lines connecting various nodes. Some nodes are represented by small blue dots, while others are larger, semi-transparent light blue circles. Several irregular, light blue shaded regions are scattered across the sphere's surface, creating a layered, geometric effect.

栈的简单应用-括号匹配

括号配对

问题定义：

➤ 任务：检查输入的文本中括号是否正确匹配

➤ 限定：

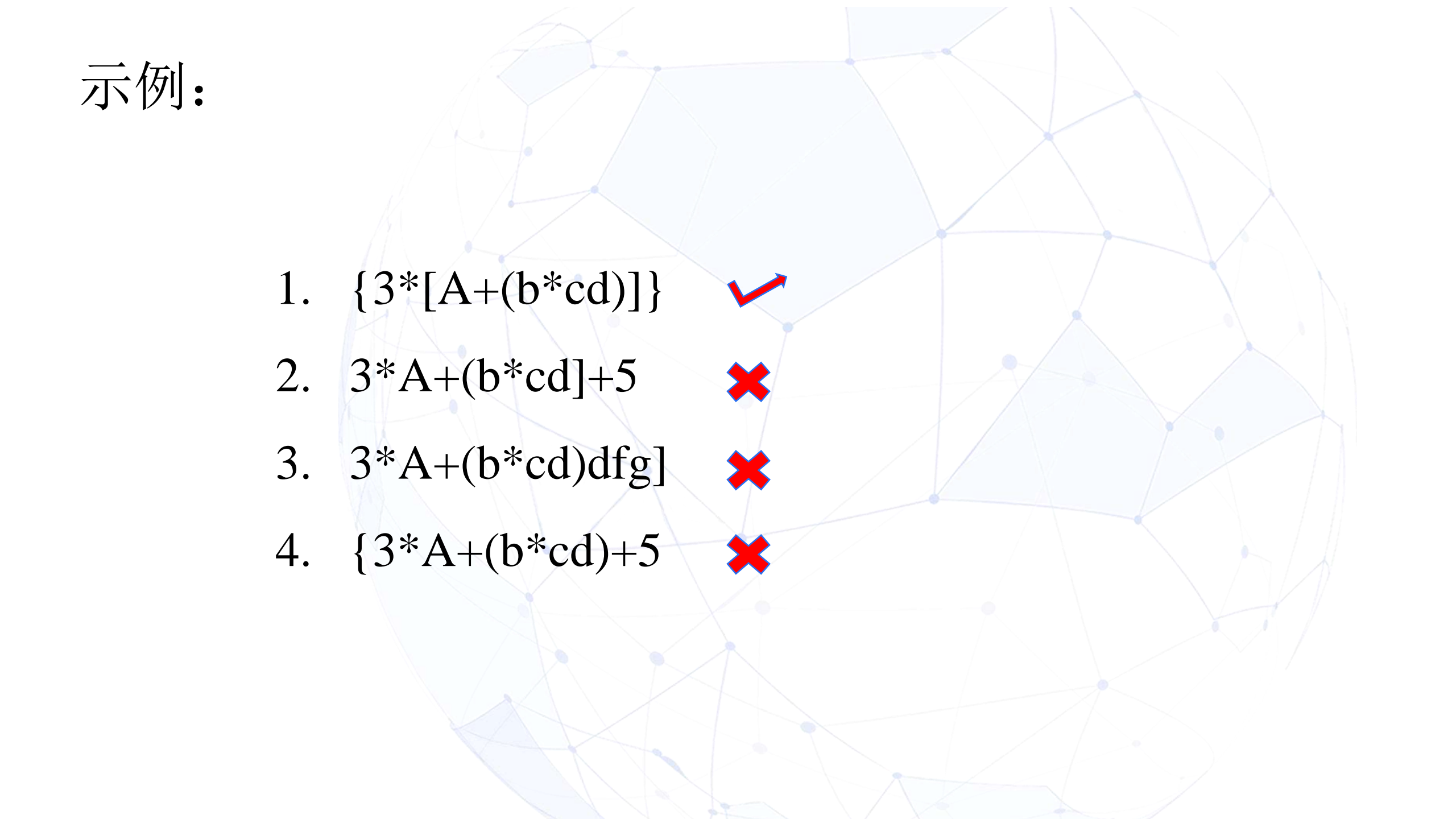
仅考虑：(), [], {}这三类括号

读入一行字符，将忽略括号外的其他所有符号

例如：{3+2*[(4+2/3)+3]*2+3/4}*4+2

– 特点：最后出现的左括号先与遇到的右括号匹配

示例：

- 
1. $\{3*[A+(b*cd)]\}$ ✓
 2. $3*A+(b*cd)+5$ ✗
 3. $3*A+(b*cd)dfg$ ✗
 4. $\{3*A+(b*cd)+5$ ✗

括号匹配算法思想

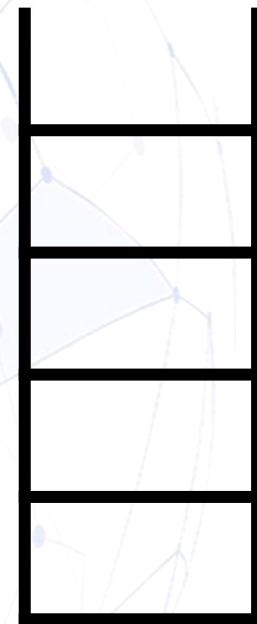
初始化一个栈

➤ 循环读入字符 x ，分情况讨论，直到读完所有字符：

- ① x 不是括号：当作普通字符，忽略；
- ② x 是左括号：入栈
- ③ x 是右括号：
 - 判断栈是否空，若为空，则右括号多余，不匹配
 - 不空，则弹出栈顶左括号，检查括号类型是否匹配，有不匹配和匹配两种情况；

➤ 进行尾部处理：输入完毕，检测此时栈是否空

- ① 空：正确匹配
- ② 不空：左括号多余，不匹配



$\{3*[A+(b*cd)]\}$


```
void BracketTest(const char *s) {  
    LinkStack<char> stack;  
    char symbol;  
    bool is_matched = true;  
    int i = 0;  
    while (s[i] != '\0') {  
        switch (s[i])  
        {  
            case '(':  
            case '[':  
            case '{':  
                stack.Push(s[i]); break;  
            case ')':  
            case ']':  
            case '}':  
                if (stack.Empty())  
                {  
                    cout << "右括号有多余"  
                    << endl;  
                    return;  
                }  
            }  
        }  
    }  
}
```

```
    else {  
        char match;  
        match = stack.GetTop();  
        stack.Pop();  
        is_matched = ((s[i] == ')') && match == '(') ||  
            (s[i] == ']') && match == '[') || (s[i] == '}') &&  
            match == '{');  
        if (!is_matched)  
        {  
            cout << "括号不匹配" << endl;  
            return;  
        }  
    }  
} //end switch  
i++;  
} //end for  
if (stack.Empty())  
    cout << "括号匹配" << endl;  
else  
    cout << "左括号多余" << endl;  
return;
```

判断字符串" $([]()[()])$ "中括号是否配对时，所需的栈的容量至少是

- ☒ A 3
- ☐ B 4
- ☐ C 5
- ☐ D 6

提交



表达式求值

表达式类型

表达式求值

表达式类型

- 表达式的三种形式:

- 中缀(infix)表示

$\langle \text{操作数A} \rangle \langle \text{运算符} \rangle \langle \text{操作数B} \rangle$, 即 $A \theta B$;

- 后缀(postfix)表示 (逆波兰式)

$\langle \text{操作数A} \rangle \langle \text{操作数B} \rangle \langle \text{运算符} \rangle$, 即 $A B \theta$;

- 前缀(prefix)表示 (波兰式)

$\langle \text{运算符} \rangle \langle \text{操作数A} \rangle \langle \text{操作数B} \rangle$, 即 $\theta A B$;

- 中缀表达式 $a + b * (c - d) - e / f$

- 后缀表达式 $a b c d - * + e f / -$

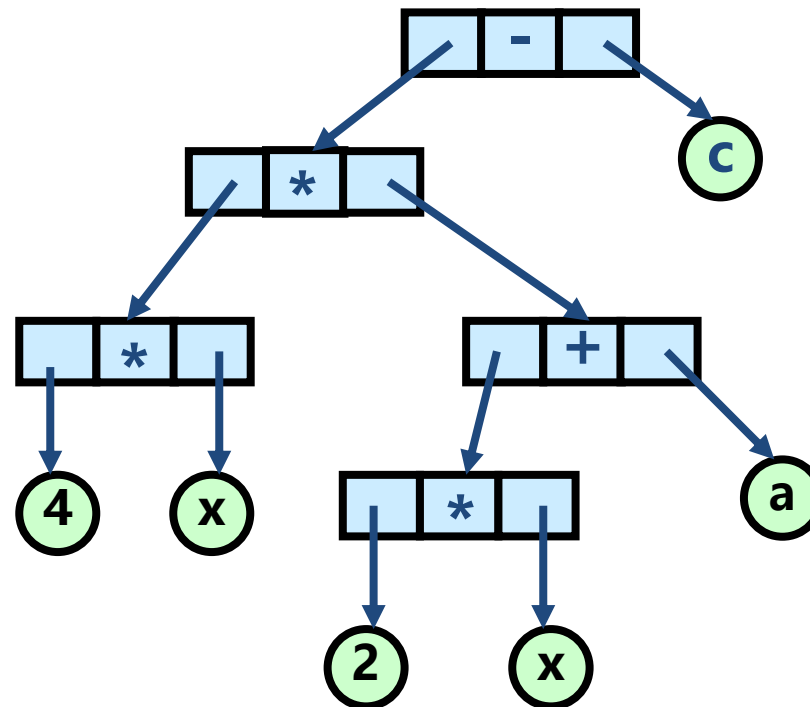
- 前缀表达式 $- + a * b - c d / e f$

中缀表达式

- 特点

- ✓ 运算符在中间
- ✓ 运算优先级可由括号改变

$$4 * x * (2 * x + a) - c$$



表达式

- 表达式定义

- ✓ 基本符号集: $\{0, 1, \dots, 9, +, -, *, /, (,)\}$

- ✓ 语法规成分集: $\{<\text{表达式}>, <\text{项}>, <\text{因子}>, <\text{常数}>, <\text{数字}>\}$

- ✓ 语法公式集

$<\text{表达式}> ::= <\text{项}> + <\text{项}> \mid <\text{项}> - <\text{项}> \mid <\text{项}>$

$<\text{项}> ::= <\text{因子}> * <\text{因子}> \mid <\text{因子}> / <\text{因子}> \mid <\text{因子}>$

$<\text{因子}> ::= <\text{常数}> \mid (<\text{表达式}>)$

表达式的定义具有递归性

$<\text{常数}> ::= <\text{数字}> \mid <\text{数字}> <\text{常数}>$

$<\text{数字}> ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

$(4 + 22) * 3 - 5$

中缀表达式的计算

1. 若有**括号**，先执行**括号内**的计算，后执行**括号外**的计算。具有多层括号时，按层次由内而外反复地脱括号，**左右括号必须配对**
2. **无括号**或同层括号时，先乘(*)、除(/)，后加(+)、减(-)
3. **同一个层次**，若有多个乘除(*)、/(/)或加减(+, -)的运算，按**自左至右次序**执行

后缀表达式

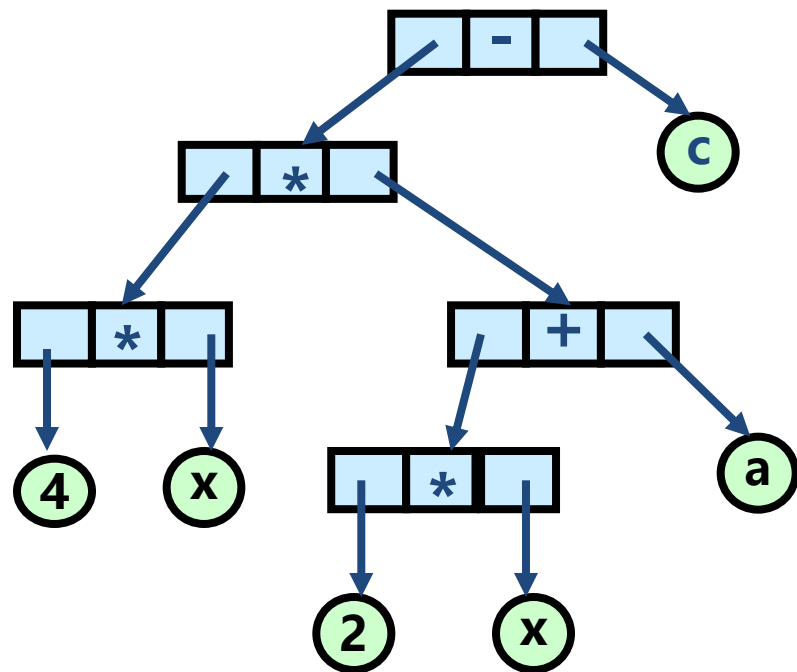
特点

运算符 在 操作数 后面
完全不需要 括号

中缀 $4 * x * (2 * x + a) - c$ 转成

$4 \ x \ * \ 2 \ x \ * \ a \ + \ * \ c \ -$

含括号的中缀表达式可以转成后缀表达式，然后用求后缀表达式的方式求值(需要设计一个转换算法)



后缀表达式的语法公式

$\langle \text{表达式} \rangle ::= \langle \text{项} \rangle \langle \text{项} \rangle +$
 $\quad \quad \quad | \langle \text{项} \rangle \langle \text{项} \rangle -$
 $\quad \quad \quad | \langle \text{项} \rangle$

$\langle \text{项} \rangle ::= \langle \text{因子} \rangle \langle \text{因子} \rangle *$
 $\quad \quad \quad | \langle \text{因子} \rangle \langle \text{因子} \rangle /$
 $\quad \quad \quad | \langle \text{因子} \rangle$

$\langle \text{因子} \rangle ::= \langle \text{常数} \rangle$ 因子这里比中缀表达式简单

$\langle \text{常数} \rangle ::= \langle \text{数字} \rangle$
 $\quad \quad \quad | \langle \text{数字} \rangle \langle \text{常数} \rangle$

$\langle \text{数字} \rangle ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$

练习

$$23 + (34 * 45) / (5 + 6 + 7)$$

提示：先处理优先级高的，脱括号，先乘除，然后加减

$$23 \ 34 \ 45 * 5 \ 6 + 7 + / +$$

中缀表达式 $(A+B)*(C-D)/(E-F*G)$ 对应的后缀表达式是

- ☐ A $A+B*C-D/E-F*G$
- ☒ B $AB+CD-*EFG*-/$
- ☐ C $AB+C*D-E/F-G*$
- ☐ D $ABCDEF+*G+*-/$

提交

后缀表达式：30 6 + 5 2 - / 的值为：

- ☐ A 3
- ☐ B 6
- ☐ C 9
- ☒ D 12

提交

$$23 + (34 * 45) / (5 + 6 + 7)$$

$$23 \ 34 \ 45 \ * \ 5 \ 6 \ + \ 7 \ + \ / \ +$$

两者异同：

- (1) 后缀表达式没有括号；
- (2) 所有操作数的出现次序完全相同
- (3) 后缀表达式运算符在操作数后面，后缀表达式的计算顺序与运算符出现次序完全相同，但中缀表达式的计算次序**不等于**运算符的出现顺序

后缀表达式求值步骤

1. 循环：依序读入表达式的符号序列（以 = 作为输入序列的结束），根据读入元素符号逐一分析处理：
 - 操作数，将其压入栈中；
 - 运算符，从栈中两次取出栈顶，按照运算符对这两个操作数进行相应计算，并将计算结果压回栈；
2. 如此继续，直到遇到符号 =，此时栈顶元素即为输入表达式的值

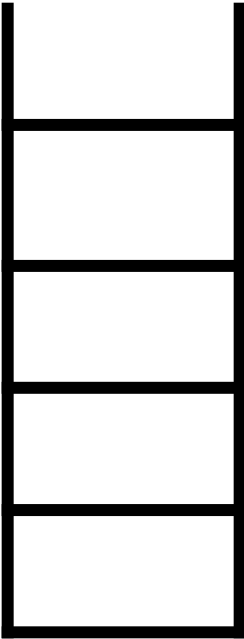
23 34 45 * 5 6 + 7 + / + =?

后缀表达式求值步骤演示

待处理的后缀表达式

23 34 45 * 5 6 + 7 + / +

栈状态的变化



| | |
|--|------|
| | 1818 |
|--|------|

计算

结果

算法3-13: 后缀表达式求值 PostFixEval(*expr*)

输入: 一个后缀表达式*expr*

输出: 后缀表达式*expr*的值。若表达式不规范, 则输出错误码 (ErrorCode)

```
1. token ← GetToken(expr) // 从表达式中取出一个元素
2. while token ≠ 表达式结尾 do
3. | if IsOperand(token) then // 如果该元素是操作数
4. | | Push(stack, token) // 则入栈
5. | else // 如果该元素是操作符
6. | | operand2 ← Top(stack)
7. | | Pop(stack)
8. | | operand1 ← Top(stack)
9. | | Pop(stack)
10. | | result ← Calculate(operand1, token, operand2) // 计算 operand1 token operand2 的值
11. | | Push(stack, result) // 计算结果入栈
12. | end
13. | token ← GetToken(expr) // 从表达式中取下一个元素
14. end
15. result ← Top(stack)
16. Pop(stack)
17. if IsEmpty(stack) = false then // 表达式不规范
18. | result ← ErrorCode
19. end
20. DestroyStack(stack)
21. return result
```

中缀表达式求解步骤？

含括号的中缀表达式比较复杂，但是中缀表达式的计算结果与其对应的后缀表达式是相同的

中缀 $4 * x * (2 * x + a) - c$ 转成

$4 \ x \ * \ 2 \ x \ * \ a \ + \ * \ c \ -$

如果将含括号的中缀表达式转成后缀表达式，然后用求后缀表达式求值的方法，该问题即可以求解

如何设计一个转换算法呢？

中缀表达式->后缀表达式需要解决的关键问题：

- (1) 如何去除中缀表达式中的括号
- (2) 如何保证按先乘除后加减运算符优先级进行运算

脱括号需要从内层到外层进行，所以可以通过栈来实现

中缀表达式 to 后缀表达式转换粗略算法

核心思路：从左到右扫描中缀表达式，用栈存放表达式中的操作符、开括号以及开括号后暂不确定计算次序其他符号

- (1) **操作数**，直接输出到后缀表达式序列；
- (2) **开括号**，将其入栈；
- (3) **闭括号**时，先判断栈是否为空，**若为空（括号不匹配）**，应作为错误进行异常处理，清栈退出；**若非空**，依次弹出栈中元素，并输出到后缀表达式序列中，直到遇到**一个开括号**为止，若没有遇到开括号，说明**括号不配对**，做异常处理，清栈退出；

中缀表达式 to 后缀表达式转换粗略算法

(4) **运算符**op (四则运算 + - * / 之一) 时

(a) 循环 当 (**栈非空 and 栈顶不是开括号 and 栈顶运算符的优先级大于等于输入的运算符的优先级**) 时,

反复 **将栈顶元素弹出, 放到后缀表达式序列中;**

(b) 将输入的**运算符**压入栈内;

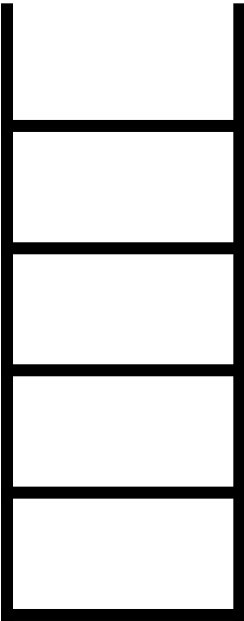
(5) 读完全部输入符号, 若栈内仍有元素, 将其全部依次弹出, 置于后缀表达式序列的尾部。若弹出元素中有开括号出现, 则说明**括号不匹配**, 做异常处理, 清栈退出

中缀表达式 to 后缀表达式

待处理中缀表达式

23 + (34 * 45) / (5 + 6 + 7)

栈状态的变化

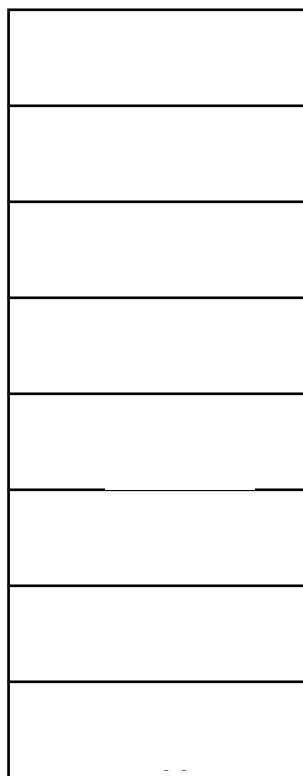


输出的后缀表达式

表达式: $5+6*(1+2)-4$

读入表达式过程: $5+6*(1+2)-4$ #

算符栈



后缀表达式:

5 6 1 2 + * + 4 -

说明1: #的作用

增设#算符, 使其优先级最低,
算符栈中初始放入#号, 表达式读完时以#结束

#的作用, 跟之前介绍的lifegame的围墙一样, sentinel

说明2: θ_1 和 θ_2 的优先级比较

| $\theta_1 \backslash \theta_2$ | + | - | * | / | (|) | # |
|--------------------------------|---|---|---|---|---|---|---|
| + | > | > | < | < | < | > | > |
| - | > | > | < | < | < | > | > |
| * | > | > | > | > | < | > | > |
| / | > | > | > | > | < | > | > |
| (| < | < | < | < | < | = | |
|) | > | > | > | > | | > | > |
| # | < | < | < | < | < | | = |

$8+5*(1*2+10)-4$

注意 θ_1 在前 θ_2 在后

说明2: θ_1 和 θ_2 的优先级比较 (2)

- 为每个算符设定优先级对应数值

| 符号 | $*/$ | $+ -$ | $($ | $)$ | $\#$ |
|-------------------|------|-------|--------------------|-----|------|
| 优先级 (priority) | 4 | 3 | 2(作为 θ_1 时) | 2 | 1 |

- $\text{Priority}[\theta_1] < \text{Priority}[\theta_2]$, 入栈, 继续读;
- $\text{Priority}[\theta_1] \geq \text{Priority}[\theta_2]$, 出栈;
- 例外1:
 - “(“作为 θ_1 时, 优先级最低;
 - “(“作为 θ_2 时, 优先级最高, 遇到“(“, 入栈, 继续读;
- 例外2:
 - θ_1 和 θ_2 是一对括号时, θ_1 出栈, θ_2 舍弃, 继续读;

详细算法描述

- (1) 初始化空栈`st`用于存放算符，初始存入“#”；初始化数组`postfix_ex`用于存放后缀表达式；
- (2) 循环（外层）从左到右依次扫描中缀表达式，根据所读到的逻辑符号`m`的不同情况分别处理：
 - ①若`m`是操作数，则将其直接加到`postfix_ex`的末尾；
 - ②否则：读取栈顶的符号，设为 $\Theta 1$ ，当前符号`m`设为 $\Theta 2$ ；
 - ③循环（内层）执行，直至 $\Theta 1$ 为“#”且 $\Theta 2$ 为“#”时结束：
 - 若 $\Theta 2$ 是左括号， $\Theta 2$ 入栈，跳出内层循环，转步骤（2）外层循环；
 - 若 $\Theta 2$ 为右括号且栈顶 $\Theta 1$ 为左括号，则出栈左括号， $\Theta 1$ 和 $\Theta 2$ 均舍弃，跳出内层循环，转步骤（2）外层循环；
 - 若 $\Theta 2$ 优先数值大，则 $\Theta 2$ 入栈，跳出内层循环，转步骤（2）外层循环；
 - 否则，即 $\Theta 1$ 的优先数值 $\geq \Theta 2$ 的优先级数值， $\Theta 1$ 比 $\Theta 2$ 级别更高或同级别但在 $\Theta 2$ 前面出现，说明 $\Theta 1$ 算符所运算的操作数已加入到`postfix_ex`中，出栈 $\Theta 1$ 并加入到`postfix_ex`中，读取新栈顶至 $\Theta 1$ 中，继续内层循环③。
- (3) 返回`postfix_ex`及其长度。

```

int trans_infix_suffix(string infix_ex[], int n, string postfix_ex[]) { // infix_ex为需转换的中缀表达式
    set<string> operators = { "+", "-", "*", "/", "(", ")", "#" }; // 算符集合
    map<string, int> priority = { {"*", 4}, {"/", 4}, {"+", 3}, {"-", 3}, {"(", 2}, {")", 2}, {"#", 1 } }; // 算符优先级字典
    SeqStack<string> st;
    int k = 0;
    st.Push("#");
    for (int i = 0; i < n; i++) {
        string m = infix_ex[i];
        if (operators.count(m) == 0) // 如果m是操作数
            postfix_ex[k++] = m; // 操作数加入后缀表达式列表
        else {
            string theta1 = st.GetTop(); // theta1为栈顶算符
            string theta2 = m; // theta2为当前算符
            while (theta1 != "#" || theta2 != "#") // 当theta1和theta2不全是#
                if (theta2 == "(") { // 左括号进栈
                    st.Push("(");
                    break;}
                // 右括号遇栈顶左括号，左括号出栈，跳出内循环，继续读下一符号
                else if (theta1 == "(" && theta2 == ")") {
                    st.Pop();
                    break;}
                // 当前算符theta2优先权高于栈顶，theta2进栈，跳出内循环，继续读下一符号
                else if (priority[theta1] < priority[theta2]) {
                    st.Push(theta2);
                    break; }
                // 栈顶优先级数值≥theta2的优先级数值，出栈栈顶加入后缀表达式
                else {
                    postfix_ex[k++] = st.Pop();
                    theta1 = st.GetTop(); // theta1为新栈顶算符，继续内层循环
                }
        }
    }
    return k;}

```

借助栈S将中缀表达式 $A-(B-C/D)*E$ 转换为后缀表达式，假设S栈初始压入“#”，则栈S的容量至少为：

- ☐ A 3
- ☐ B 4
- ☒ C 5
- ☐ D 6

提交

中缀表达式求值方法

(1) 中缀表达式 \rightarrow 后缀表达式
再调用后缀表达式求值算法

(2) 直接计算

how to ?

无括号中缀表达式直接求值

假设：

- 只涉及+、-、*和/四种双目运算符号（其余可自己扩展）
- 不考虑左、右括号
- 设表达式尾标记为#

表达式由三部分构成：运算符、操作数和界限符（表达式尾标记）

无括号中缀表达式直接求值 (2)

运算根据运算优先级进行，而优先级与操作数无关，两者可独立存储；

- 当读到的当前运算符优先级低于前一个时，可以计算前一个运算符，因此必须记录前一个运算符，即有一个读入运算符的逆序（栈）
- 计算某个运算符时，它所对应的操作数应该是最后两个操作数，因此必须记录操作数的逆序（栈）

$$4 + 2 * 3 - 5$$

无括号中缀表达式直接求值 (3)

(一) 规定运算符的优先级

| | | |
|---|----|----|
| # | +- | */ |
| 0 | 1 | 2 |

(二) 设定两个栈:

- 操作数栈 (OPND) —— 保存读入操作数
- 运算符栈 (OPTR) —— 保存读入运算符

无括号中缀表达式直接求值 (3)

(三) 算法:

- 初始化两个栈: 运算符栈**OPTR**和操作数栈**OPND**
- 将起始标记 ‘#’ 入**OPTR**;
- 自左至右, 循环读入表达式字符**ch**, 直到遇到尾标记 ‘#’ 为止, 按**ch**分情况进行如下处理:
 - (1) **ch**为操作数: 入操作数栈**OPND**
 - (2) **ch**为运算符: \rightarrow

(2) **ch**为运算符:

取出**OPTR**运算符栈中的栈顶运算符**ch1**, 比较**ch**与**ch1**的优先级:

◆**ch>ch1**: 将**ch**入栈**push(OPTR, ch)**, 读入下一个符号**ch**;

◆**ch<=ch1**: 先计算表达式中前一个子式

– 运算符栈出栈 $\text{theta} = \text{pop}(\text{OPTR});$

– 操作数栈出栈两次: $\text{rop} = \text{pop}(\text{OPND}); \quad \text{lop} = \text{pop}(\text{OPND});$

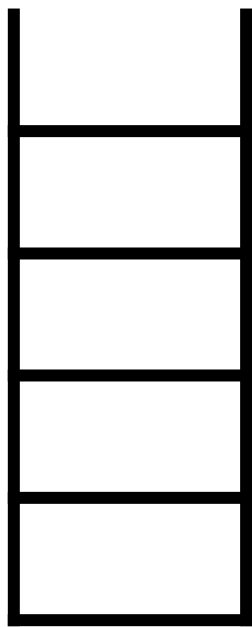
其中第一次出栈的是右操作数, 第二次是左操作数;

– 进行运算: $\text{x} = \text{operate}(\text{lop} \text{ theta} \text{ rop});$

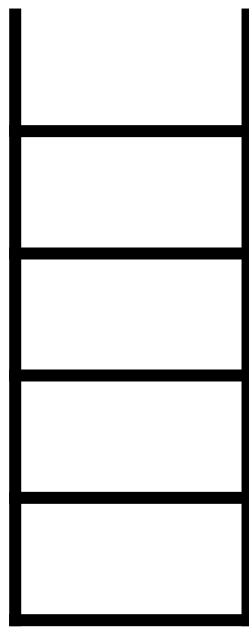
并将结果入操作数栈**push(OPND,x)**;

求表达式 #23+4/2-6#

23 + 4 / 2 - 6



OPND



OPTR

2

25

前面算法如果要考虑括号呢？

中缀表达式直接求值

例: $5 + 6 * (1 + 2) - 4 \#$

3 2 1 4

需要两个栈，一个放操作符，一个放操作数

opnd

19

#

optr



考虑括号的中缀表达式直接求解算法

- (1) 初始化opnd栈用于存放操作数，初始化空栈optr并存入#，用于存放算符；
- (2) 循环（外层）从左到右依次扫描中缀表达式，根据所遇到的逻辑符号m的不同情况分别处理：
 - ①若m是操作数，则将其入栈到opnd，跳出内层循环，转步骤（2）外层循环，继续读下一个单词；
 - ②否则，读取optr栈栈顶的符号，设为 Θ_1 ，当前符号m设为 Θ_2 ；
 - ③循环（内层）执行，直至 Θ_1 为“#”且 Θ_2 为“#”时结束：
 - 若 Θ_2 是左括号，则将其入栈到optr，跳出内层循环，转步骤（2）外层循环；
 - 若 Θ_2 为右括号且栈顶 Θ_1 为左括号，则出栈左括号， Θ_1 和 Θ_2 均舍弃，跳出内层循环，转步骤（2）外层循环；
 - 若 Θ_2 优先数值大，则 Θ_2 入栈，跳出内层循环，转步骤（2）外层循环；
 - 否则 Θ_1 从optr出栈，调用calculate完成运算，即从opnd栈依次出栈两个操作数做运算 Θ_1 ，并将运算结果入opnd栈；读取新栈顶至 Θ_1 中，继续内层循环③。

$$5 + 6 * (1 + 2) - 4$$

表达式 $3-2*(4+2*2-6*3)-5$ 求值过程中，当扫描到6时，操作数栈和算符栈从栈底到栈顶的内容依次为（ ）

- ☐ A 3,2,4,1,1; #-*(+*-
- ☐ B 3,2,8; #-*-
- ☒ C 3,2,8; #-*(-
- ☐ D 3,2,4,2,2; #-*(-

提交

The background features a large, faint sphere with a network of nodes and lines overlaid on it. The nodes are small circles in various shades of blue and purple, connected by thin, light-colored lines. Some areas of the sphere are highlighted with semi-transparent blue and purple polygons, creating a complex, interconnected geometric pattern.

应用**2**：栈与递归

什么是递归？

递归定义是一种自我嵌套定义方式，即使用被定义对象的自身来定义该对象，可用于对数学概念、实体、问题解决方案等进行定义。

递归主要应用在数学和计算机科学领域中。

- 例如，集合论对自然数的定义：0是一个自然数，每个自然数都有一个后继，这个后继也是自然数。
- 在生活中，也可以将一些概念或行为采用递归方式来定义。

某人的祖先可定义为：某人的父母以及他父亲的祖先和他母亲的祖先。

树枝、海岸线、山峰、雪花等分形图是自然界中递归的例子，俄罗斯套娃则是艺术中应用递归的例子。

递归求解问题的策略

把一个**大规模**的复杂问题层层转化为一个或多个与**原问题相同、规模较小**的子问题来求解，直到问题小到可以用非常简单直接的方式来解决，即分而治之（divide and conquer）策略。

能够用递归解决的问题应满足的条件

- (1) 必须存在递归结束的情形，即一个或多个递归出口，通常对应于最小规模或满足某个条件时无须递归可直接求解的情况，常被称为**基本情况**（base case）；
- (2) 原问题可以分解为更小规模的相同子问题，即存在一个确定的**通用递归分解规则**；
- (3) 子问题的个数必须是有限的，即保证不管多大的规模，都可以最终到达一个基本情况。

递归示例1： 阶乘函数

- 阶乘 $n!$ 的递归定义如下：

$$\text{factorial}(n) = \begin{cases} 1, & \text{if } n \leq 1 \\ n \times \text{factorial}(n-1), & \text{if } n > 1 \end{cases}$$

算法：阶乘的递归实现 $\text{Factorial}(n)$

输入：整数 $n \geq 0$

输出：整数 n 的阶乘

1. if $n \leq 1$ then
2. return 1
3. else
4. return $n \times \text{Factorial}(n-1)$
5. end

递归示例2： 求最大公约数

- 对正整数 m 和 n ，用辗转相除法求最大公约数。
 - 首先求出余数 $r=m\%n$ ；
 - 如果 $r=0$ ，则 m 和 n 的最大公约数为 n ，无须递归求解；
 - 否则 m 和 n 的最大公约数即为 n 和 r 的最大公约数。

```
int gcd(int m, int n) {  
    int r = m % n;  
    if (r == 0)  
        return n;  
    else  
        return gcd(n, r);  
}
```

递归算法的关键

设计递归算法的关键是获得一个清晰而准确的问题求解方案的递归定义。

所有递归算法都包含两大部分：

- (1) **递归出口**：不用递归实现的基本情况的直接处理。
- (2) **递归规则**：将特定规模的问题求解分解为一个或多个小规模同等问题的求解。

递归函数特点

- (1) 直接或者间接调用自己;
- (2) 在结构上, 递归函数包含如if...else等形式的分支语句;
- (3) 除了少数情况下在函数中通过输入或删除元素等方法修改问题规模之外, 大多情况下递归函数都含有用来表示问题规模的入口参数。在执行过程中, 随着递归函数的调用和返回, 该参数表示的规模通常从大变小, 再从小变大反复变化。
- (4) 较为简短, 少量的代码就可完成解题过程所需要的重复计算。

一个递归算法必须包括_____。

- ☐ A 递归部分
- ☒ B 终止条件和递归部分
- ☐ C 迭代部分
- ☐ D 终止条件和迭代部分

提交

以下哪项不是递归函数的特点？

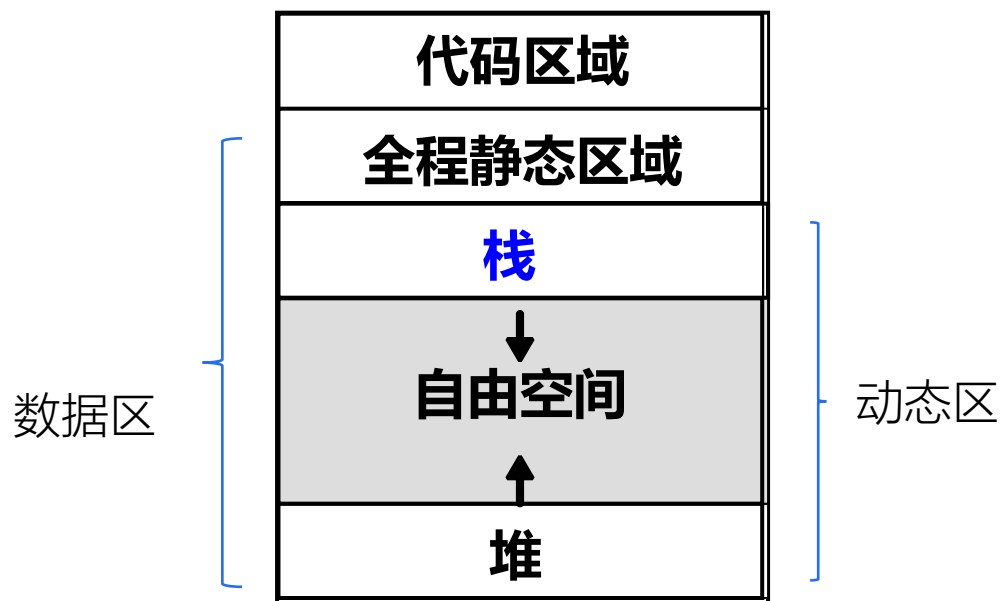
- ☐ A 有一个基本结束条件
- ☐ B 能够不断减小问题规模
- ☒ C 对函数运行结果进行缓存
- ☐ D 函数直接或间接调用自身

提交

函数调用与递归的实现

● 程序运行时环境

- ✓ 目标计算机上用来管理存储器并保存执行过程所需的信息的寄存器及存储器的结构



- **运行栈 (stack)** 用于分配**后进先出 LIFO**的数据, 比如, 函数调用
- **堆 (heap)** 用于不符合LIFO的数据, 比如指针所指向空间的分配

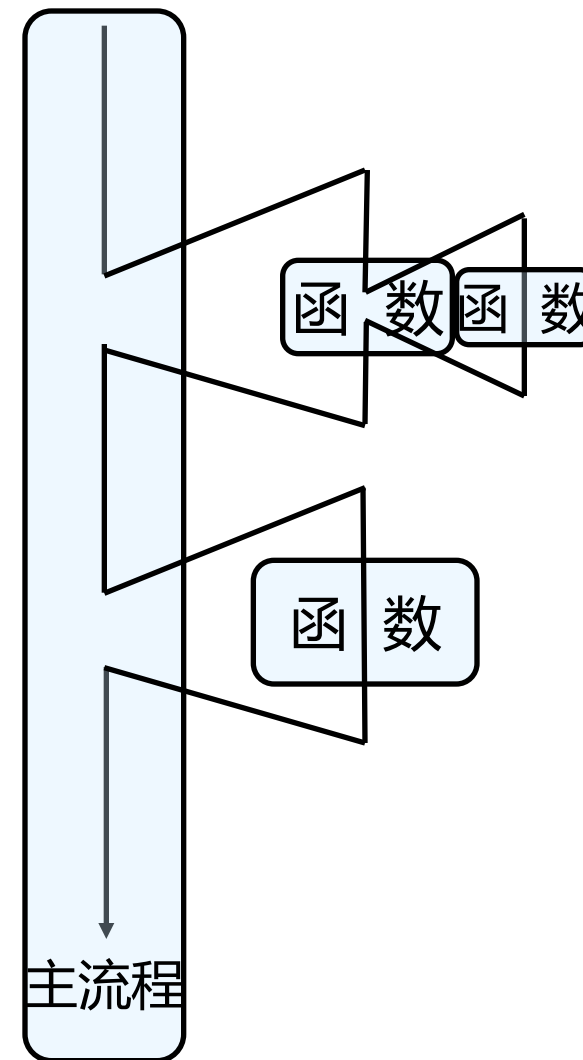
函数调用及返回的步骤

● 调用

- ✓ 保存调用信息（参数，返回地址）
- ✓ 分配数据区（局部变量）
- ✓ 控制转移给被调函数的入口

● 返回

- ✓ 保存返回信息
- ✓ 释放数据区
- ✓ 控制转移到上级函数（主调用函数）



```

void A(int s,int t);
void B(int d);
→ void main(){
    int m,n;
    m=0;n=1;
    ...
    → A(m,n);
    → 1:...
    m+=n;
    → }

```

```

→ void A(int s,int t){
    int i;
    ...
    → B(i);
    → 2:....
    → }
→ void B(int d){
    int x,y;
    ...
    → }

```

存储每个函数的参数、局部变量和返回地址等信息的空间称为**调用记录**（invocation record）或**活动记录**、**栈帧**

| |
|--------------------|
| 自变量（参数）空间 |
| 用作簿记信息空间，如返回地址等 |
| 用作局部变量空间 |
| 用作临时变量空间（比如匿名对象变量） |

调用记录存储在哪里？

运行栈中的活动记录

- 每调用一个函数，执行一次进栈操作，将被调函数的活动记录入栈，即每个新的活动记录都分配在栈的顶部
- 每次从函数返回时，执行一次出栈操作，释放本次活动记录，恢复到上次调用所分配的数据区
- 被调函数的变量地址全部采用相对于栈顶的相对地址来表示

```

void A(int s,int t);
void B(int d);
void main(){
    int m,n;
    m=0;n=1;

    ...
    A(m,n);
1:...
    m+=n;
}

```

```

void A(int s,int t){
    int i;

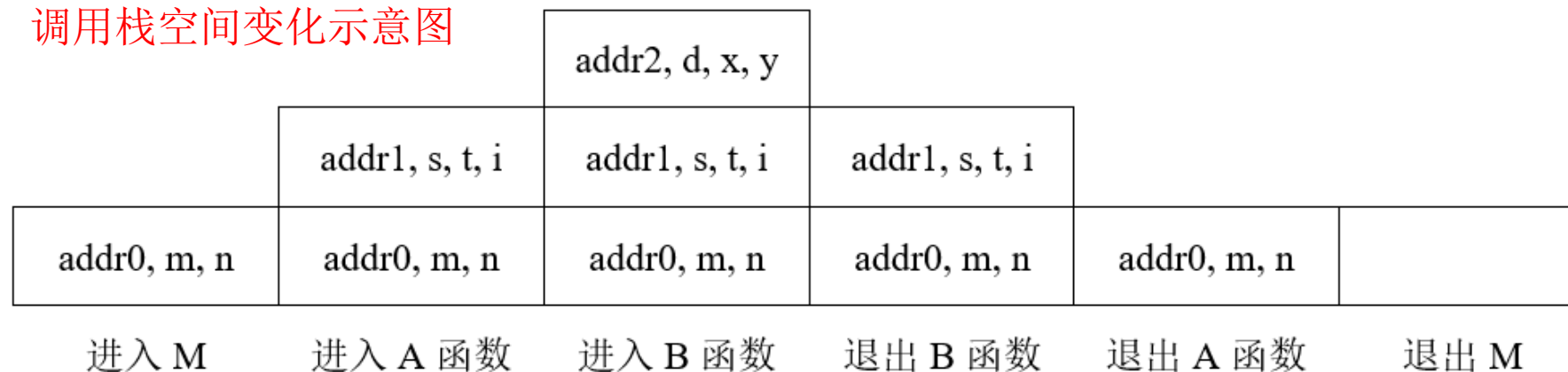
    ...
    B(i);
2:....
}
void B(int d){
    int x,y;

    ...
}

```

Last In, First Out (LIFO)
(后进先出)

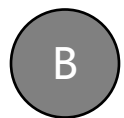
调用栈空间变化示意图



栈是实现函数间相互调用所必须的结构。



对



错

提交

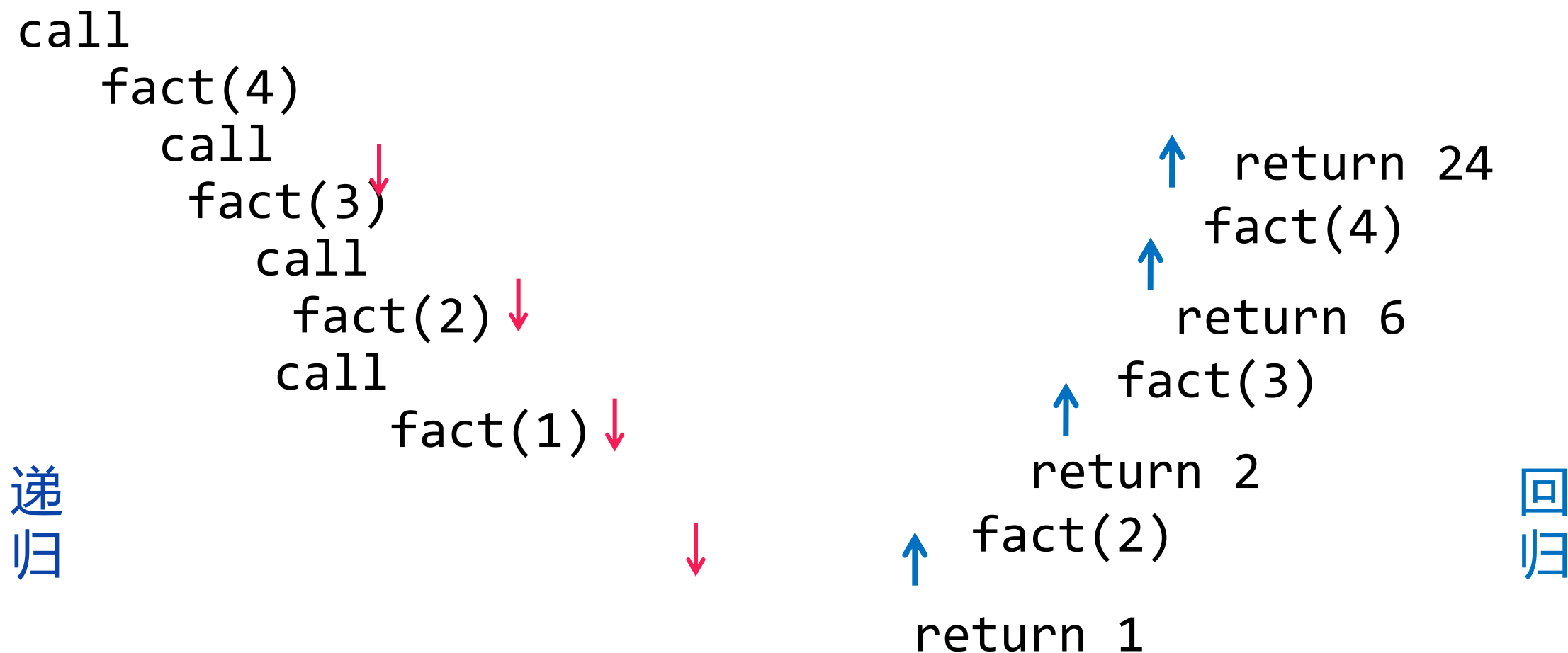
递归函数的调用

1. 递归函数的调用与普通函数的调用类似，不过每次调用的是自己
2. 递归调用时，被调函数的局部变量等数据不能预先分配到固定单元，而须每调用一次就分配一份，以存放运行当前所用的数据，当返回时随即释放。

递归函数中同一个局部变量，在不同的递归层次被分配不同的存储空间，放在内部栈的不同位置

3. 递归的深度决定递归函数在运行栈中活动记录的数目

递归函数执行过程图解：4!



```
int fun(int n){
    if (n <= 0)
        return 0;
    else
        return fun(n - 1) + n;
}
```

当以fun(2)调用该递归函数，递归返回阶段前，调用栈中**自栈底到栈顶**保存的记录依次对应的是

- ☒ A fun(2) -> fun(1) -> fun(0)
- ☐ B fun(0) -> fun(1) -> fun(2)
- ☐ C fun(2) -> fun(0) -> fun(1)
- ☐ D fun(1) -> fun(0) -> fun(2)

提交

递归算法的非递归实现

- 以阶乘为例，非递归方式

- ✓ 建立迭代

- 一般通过循环实现

- ✓ 递归转换为非递归

- 自己编写程序模拟运行栈)

阶乘的迭代实现

// 使用循环迭代方法，计算阶乘n!的一种程序

```
long fact (long n) {  
    int m = 1;  
    int i;  
    if (n > 1)  
        for ( i = 1; i <= n; i++ )  
            m = m * i;  
    return m ;  
}
```

阶乘的一种非递归实现

```
long fact(long n) {           // 使用栈方法，计算阶乘n!的程序
    Stack s;
    int m = 1;

    while (n > 1)              // ?
        s.push(n--);
    while (!isEmpty(s))        // ?
        m *= s.pop(s);
    return m;
}
```

与函数调用原理相同，只不过将由系统负责的保存工作信息变为由程序自己保存

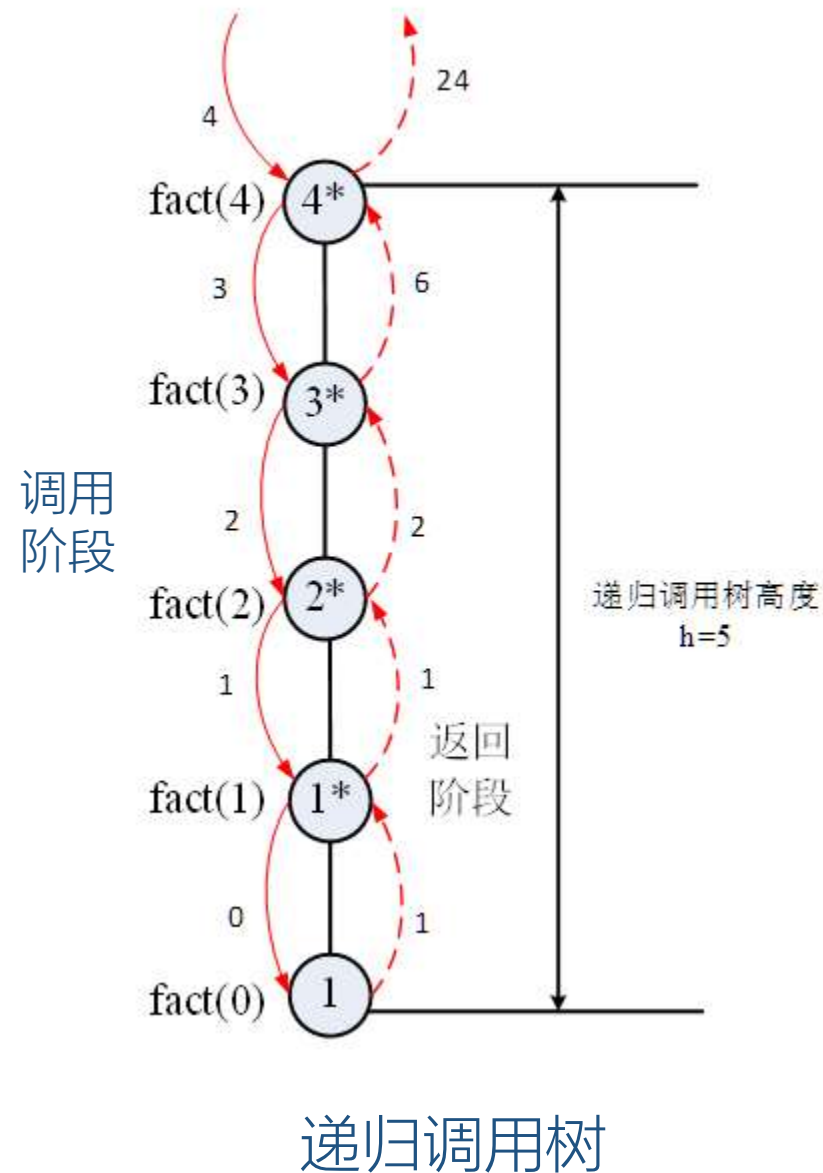
递归算法的性能

阶乘递归函数时间效率

```
int fact(int n){  
    if (n==0)  
        return 1;  
    return n*fact(n-1);  
}
```

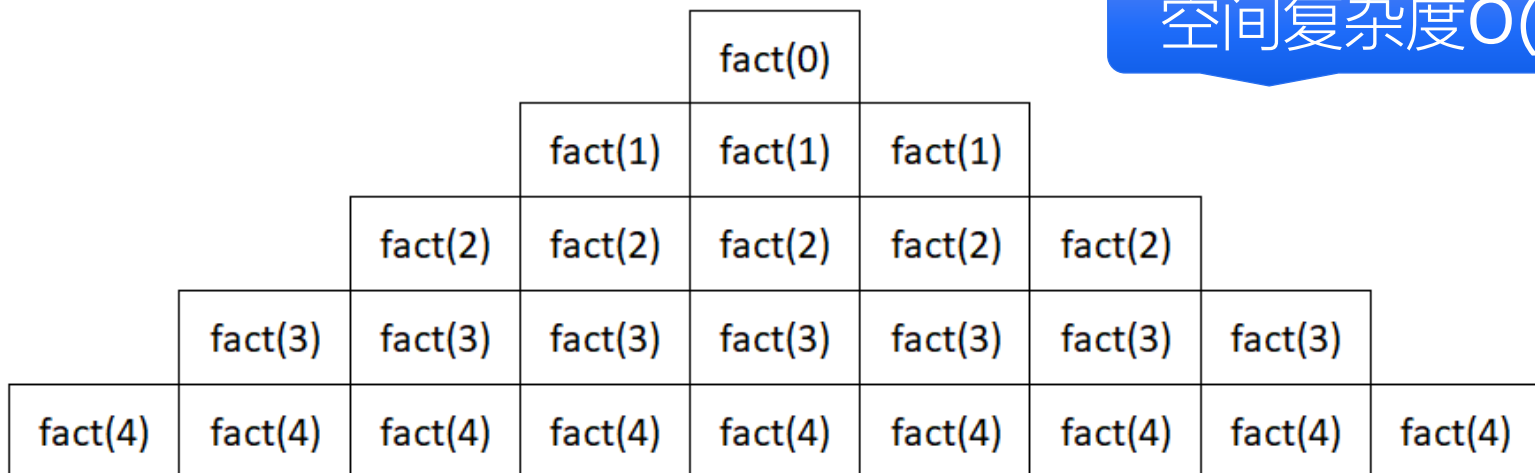
时间复杂度 $O(n)$

- 递归算法的**时间效率**与**递归调用树的规模**相关。
- 递归调用的最大层次数称为**递归深度**。运行`fact(4)`时，递归调用的最大层次为5，即递归深度为5。
- 递归深度**与对应**递归调用树的高度**一致。



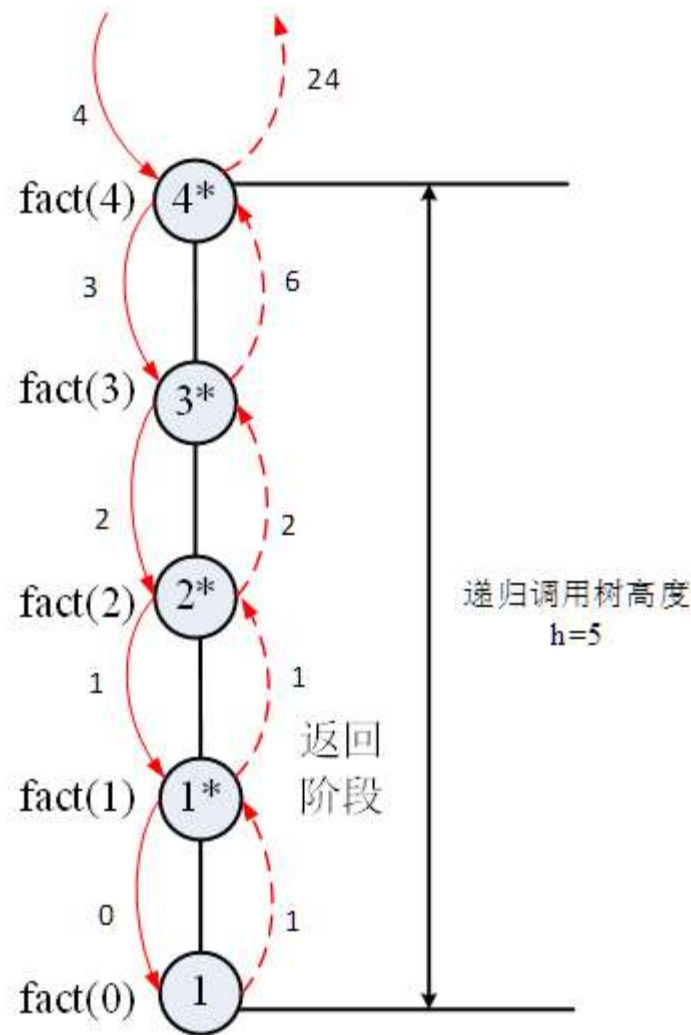
阶乘递归函数空间效率

- 递归函数的空间效率与递归深度（递归调用树的高度）相关，fact递归函数的空间效率为 $O(n)$ 。
- 如果递归深度过大，则会导致调用栈所占容量过大发生溢出。



空间复杂度 $O(n)$

递归调用阶段



递归调用树

递归算法的性能

- 可用递归调用树直观表示出算法的运行过程;
- 算法的时间性能与调用树的规模（结点个数）相关;
- 算法的空间性能与调用树的高度相关。

队列应用：车厢重排

问题描述

一列挂有 n 节车厢的货运列车途径 n 个车站，计划在行车途中将各节车厢停放在不同的车站。假设 n 个车站的编号从 1 到 n ，货运列车按照从第 n 站到第 1 站的顺序经过这些车站，且将与车站编号相同的车厢卸下。

货运列车的各节车厢以随机顺序入轨，为方便列车在各个车站卸掉相应的车厢，则须重排这些车厢，使得各车厢从前往后依次编号为 1 到 n ，这样在每个车站只需卸掉当前最后一节车厢即可。

车厢重排可通过转轨站完成。一个转轨站包含 1 个入轨 (I)， 1 个出轨 (O) 和 k 个位于入轨和出轨之间缓冲轨 (H_i)。

请设计合适的算法来实现火车车厢的重排。

队列应用：车厢重排

各缓冲轨中车厢按编号进入出轨



队列应用：车厢重排

本质：将一个无序序列转换成一个以队列方式组织的有序序列

转换过程采用缓冲轨存储尚未确定输出次序的车厢，满足递增或增减的特性即可，以栈或队列组织均可

若将每个缓冲轨看成一个队列

转化为：将一个长度为 n 的随机序列（车厢进入入轨），通过 k 个缓冲队列，输出到一个有序队列（出轨）

队列应用：车厢重排

重排规则：

1. 一个车厢从 入轨 移至 **出轨** 或 **缓冲轨**；
2. 一个车厢 只有在**其编号恰是下一个待输出编号**时，可 移到**出轨**；
3. 一个车厢移到某个**缓冲轨**，**仅当其编号大于该缓冲轨中队尾车厢的编号**，若多个缓冲轨满足这一条件，则选择**队尾车厢编号最大的**，**否则选择一个空缓冲轨**，若无空缓冲轨则无法重排

队列应用：车厢重排

示例： 9节车厢的货车，使用3个容量为3的缓冲轨

初始状态： 9节车厢均在入轨排队



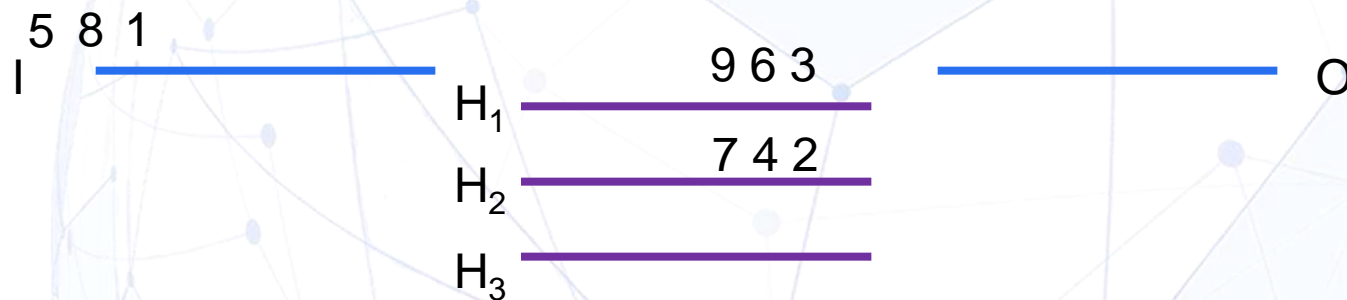
入轨的车厢按序分别进入缓冲轨 H_1 、 H_2



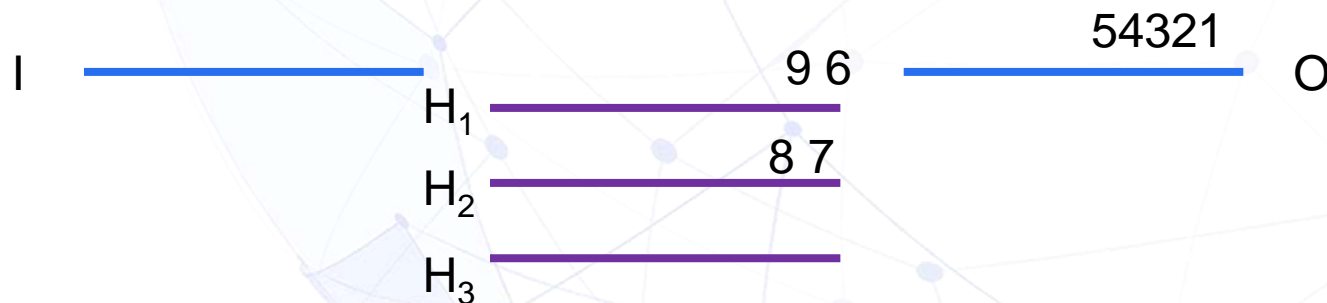
考虑1个问题：为啥缓冲轨道队列，每次都是选大于队尾编号最大的那个队列入队呢？

队列应用：车厢重排(2)

车厢1直接进入出轨



车厢2、3、4按序分别从H₂、H₁、H₂出队进入出轨；
车厢8进入缓冲轨H₂，车厢5直接进入出轨



队列应用：车厢重排

各缓冲轨中车厢按编号进入出轨



考虑1个问题：

缓冲轨道的数量 k ，最多要多少才能保证肯定能重排？ k 的最小值呢？

队列应用：车厢重排

算法详见教材

算法：车厢重排 $\text{TrainCarriageScheduling}(in_track, out_track, n, k)$

输入：入轨的车厢序列 in_track ；车厢数量 $n \geq 0$ ；缓冲轨数量 $k > 0$

输出：按序重排车厢的出轨序列 out_track ；若任务不可能完成则退出



队列应用：车厢重排

算法：车厢重排 $\text{TrainCarriageScheduling}(in_track, out_track, n, k)$

输入：入轨的车厢序列 in_track ；车厢数量 $n \geq 0$ ；缓冲轨数量 $k > 0$

输出：按序重排车厢的出轨序列 out_track ；若任务不可能完成则退出

```
1. for  $i \leftarrow 1$  to  $k$  do
2. |  $\text{InitQueue}(buffer[i])$ 
3. end
4.  $\text{InitQueue}(out\_track)$ 
5.  $next\_out \leftarrow 1$ 
6. for  $i \leftarrow 1$  to  $n$  do
7. | if  $in\_track[i] = next\_out$  then
8. | |  $\text{EnQueue}(out\_track, i)$ 
9. | |  $next\_out \leftarrow next\_out + 1$ 
10. | else
11. | | for  $j \leftarrow 1$  to  $k$  do // 考察每一缓冲轨队列
12. | | |  $front\_crg \leftarrow \text{GetFront}(buffer[j])$  // 查看队列  $j$  的头元素
13. | | | if  $front\_crg \neq \text{NIL}$  且  $in\_track[front\_crg] = next\_out$  then
14. | | | |  $\text{EnQueue}(out\_track, front\_crg)$ 
15. | | | |  $\text{DeQueue}(buffer[j])$ 
16. | | | |  $next\_out \leftarrow next\_out + 1$ 
17. | | | | break
18. | | | end
19. | | end
```



队列应用：车厢重排

```
20. | | if  $j > k$  then // 若入轨和缓冲轨的队首元素中没有编号为 $next\_out$ 的车厢
21. | | |  $max\_rear \leftarrow 0$ 
22. | | |  $max\_buffer \leftarrow -1$ 
23. | | | for  $j \leftarrow 1$  to  $k$  do // 考察每一缓冲轨队列的队尾
24. | | | |  $rear\_crg \leftarrow \text{GetRear}(buffer[j])$ 
25. | | | | if  $rear\_crg \neq \text{NIL}$  且  $in\_track[i] > in\_track[rear\_crg]$  then
26. | | | | | if  $in\_track[rear\_crg] > max\_rear$  then
27. | | | | | |  $max\_rear \leftarrow in\_track[rear\_crg]$  // 最大队尾元素值
28. | | | | | |  $max\_buffer \leftarrow j$  // 最大队尾元素所在的队列编号
29. | | | | | end
30. | | | | end
31. | | | end
32. | | | if  $max\_buffer \neq -1$  then
33. | | | |  $\text{EnQueue}(buffer[max\_buffer], i)$ 
34. | | | else
35. | | | | for  $j \leftarrow 1$  to  $k$  do
36. | | | | | if  $\text{IsEmpty}(buffer[j]) = \text{true}$  then
37. | | | | | | break
38. | | | | | end
39. | | | | end
```



队列应用：车厢重排

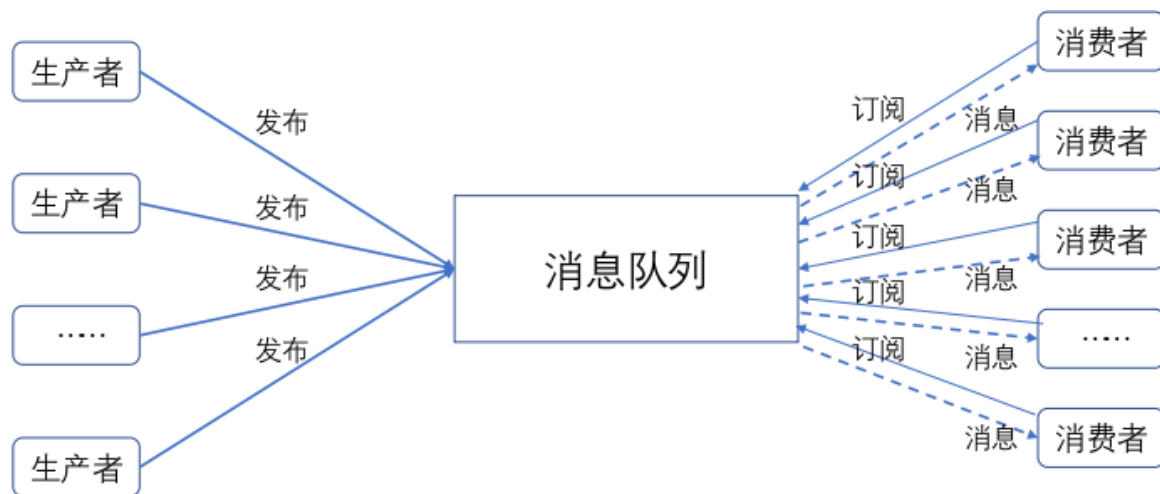
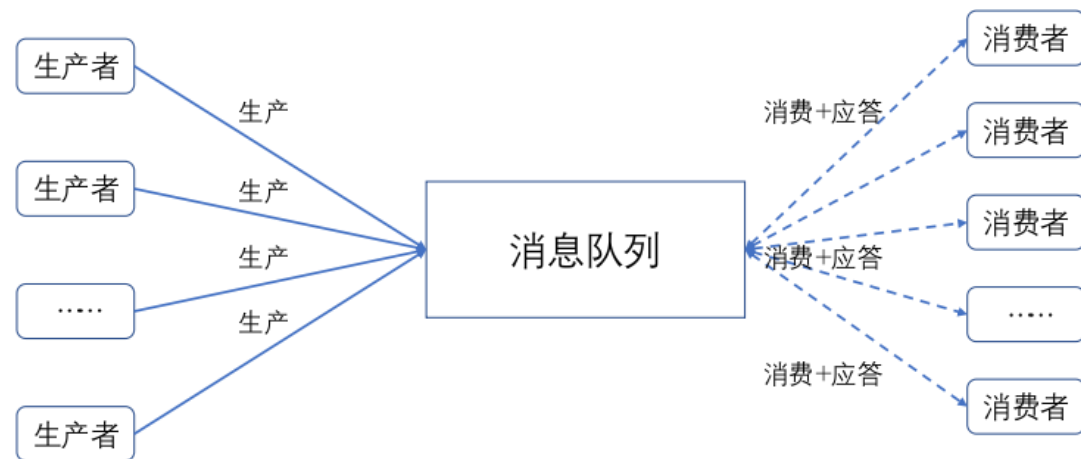
```
40. | | | | if  $j \leq k$  then
41. | | | | | EnQueue(buffer[j], i)
42. | | | | else
43. | | | | | 任务不可能完成，退出
44. | | | | end
45. | | | end
46. | | end
47. | end
48. end
49. while  $next\_out \leq n$  do
50. | for  $j \leftarrow 1$  to  $k$  do // 考察每一缓冲轨队列
51. | | if front_crg ≠ NIL 且  $in\_track[front\_crg] = next\_out$  then
52. | | | EnQueue(out_track, front_crg)
53. | | | DeQueue(buffer[j])
54. | | |  $next\_out \leftarrow next\_out + 1$ 
55. | | | break
56. | | end
57. | end
58. end
59. for  $i \leftarrow 1$  to  $k$  do
60. | DestroyQueue(buffer[i])
61. end
```

单调队列

- 一种元素具有严格单调性的队列，满足两个约束：
 1. 队列元素从头到尾的严格单调性。
 2. 队列元素先进先出，当然队首元素比尾部元素要先进。
- 根据元素的递增或递减可组织为
 - ✓ 单调递增队列
 - ✓ 单调递减队列
- 单调队列的主要操作基本与队列相同，**区别**在于单调队列的元素入队时，需要**单调性检查**，通过删除队列中不满足单调性的元素来保持队列的单调性
- 单调队列的元素间兼有先进先出的公平性和单调性，可用于解决滑动窗口类问题

消息队列

- 进程或线程之间通信的一种常用方式
- 为进程/线程提供一个临时存储消息的轻量级缓冲区，通常采用先进先出的存储方式，包括
 - 请求、恢复、错误消息、明文信息或控制权等
- 一个消息包含
 - 消息头：用于存储消息类型、目的地id、源id、消息长度和控制信息等信息
 - 消息体



课堂小结

栈的常见应用

