算法与数据结构

4. 栈与队列

赵登阳

中国计量大学信息工程学院

2023 年 3 月 21 日

1. 栈
2. 栈与递归
3. 栈的典型应用
4. 队列
5. 队列应用

1 栈

2 栈与递归

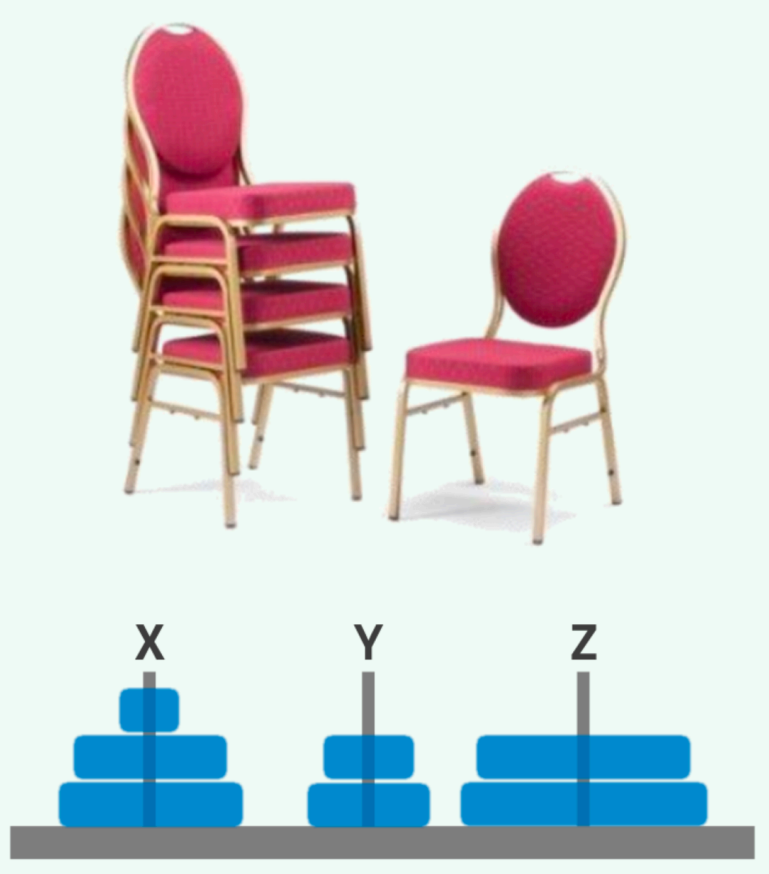
3 栈的典型应用

4 队列

5 队列应用

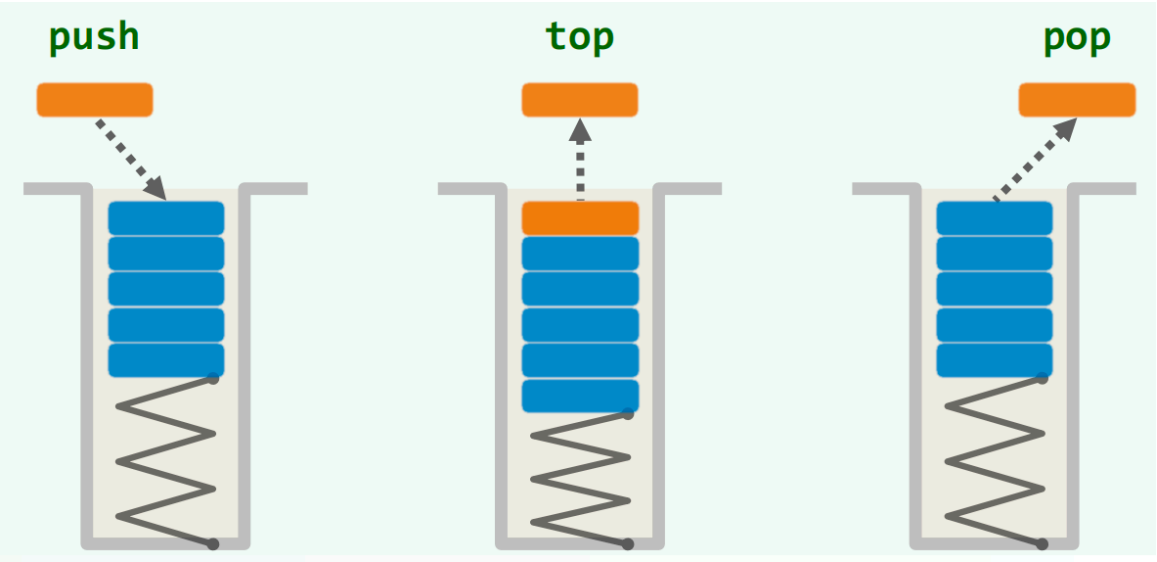
栈

- 栈（stack）是受限的序列
- 只能在栈顶（top）插入和删除
- 栈底（bottom）为盲端
- 后进先出（LIFO）
- 先进后出（FILO）



栈：基本接口

- size()
- empty()
- 入栈 push()
- 出栈 pop()
- 查顶 top()



栈：操作实例

操作	输出	栈（左侧栈顶）			
Stack()					
empty()	true				
push(5)					5
push(3)				3	5
pop()	3				5
push(7)				7	5
push(3)			3	7	5
top()	3		3	7	5
empty()	false		3	7	5

操作	输出	栈（左侧栈顶）					
push(11)				11	3	7	5
size()	4			11	3	7	5
push(6)			6	11	3	7	5
empty()	false		6	11	3	7	5
push(7)		7	6	11	3	7	5
pop()	7		6	11	3	7	5
pop()	6			11	3	7	5
top()	11			11	3	7	5
size()	4			11	3	7	5

栈：模版类

```
#include "../Vector/Vector.h"
template <typename T>
class Stack: public Vector<T> {
public:
    // 沿用原有接口：size(),empty()
    // 入栈：等效于将新元素作为向量的末元素插入
    void push( T const & e ) { insert( e ); }
    // 出栈：等效于删除向量的末元素
    T pop() { return remove( size() - 1 ); }
    // 取顶：直接返回向量的末元素
    T & top() { return (*this)[ size() - 1 ]; }
};
```

- 同样的，可以基于 List 模版类派生出 Stack 类

1 栈

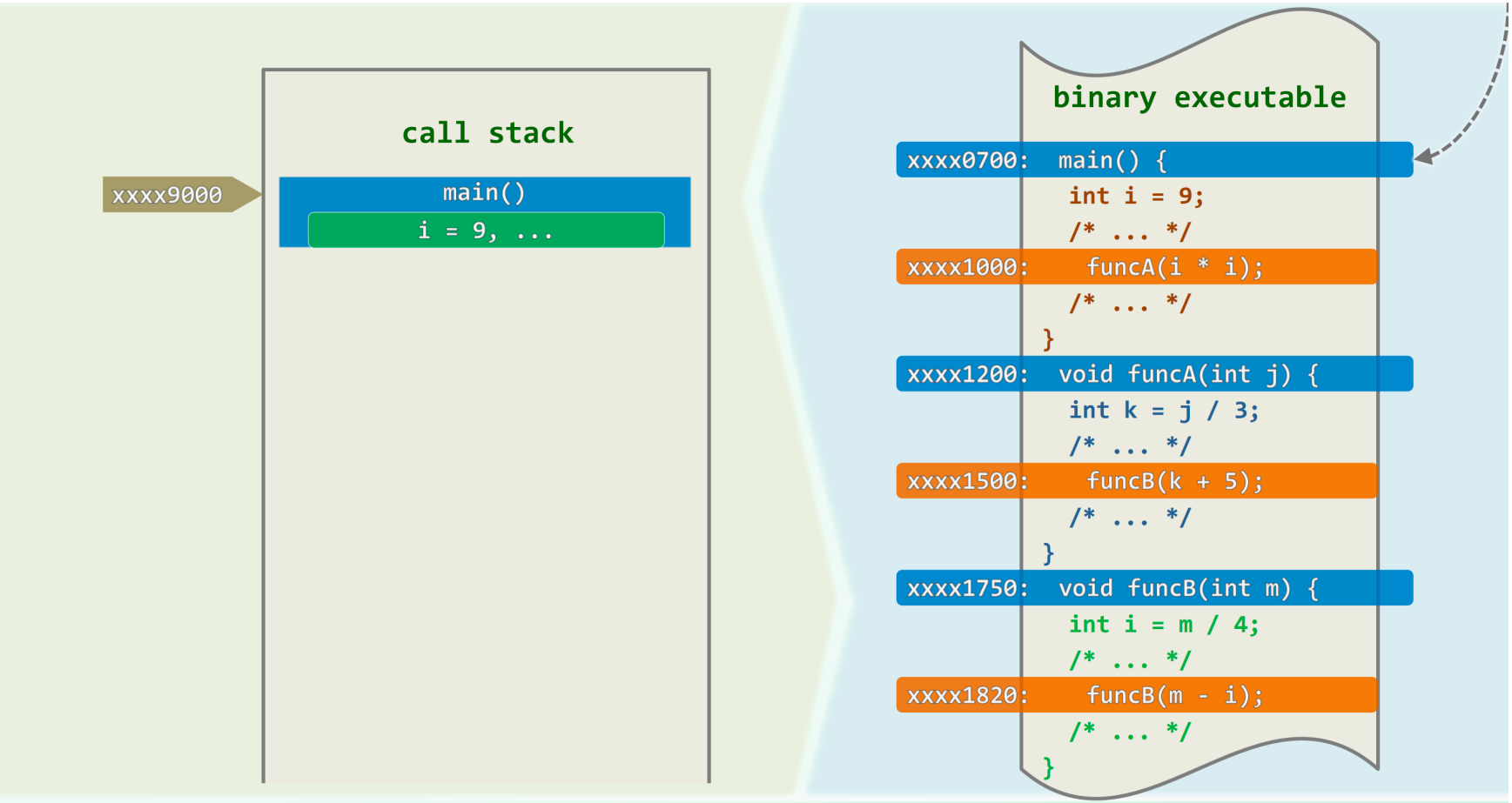
2 栈与递归

3 栈的典型应用

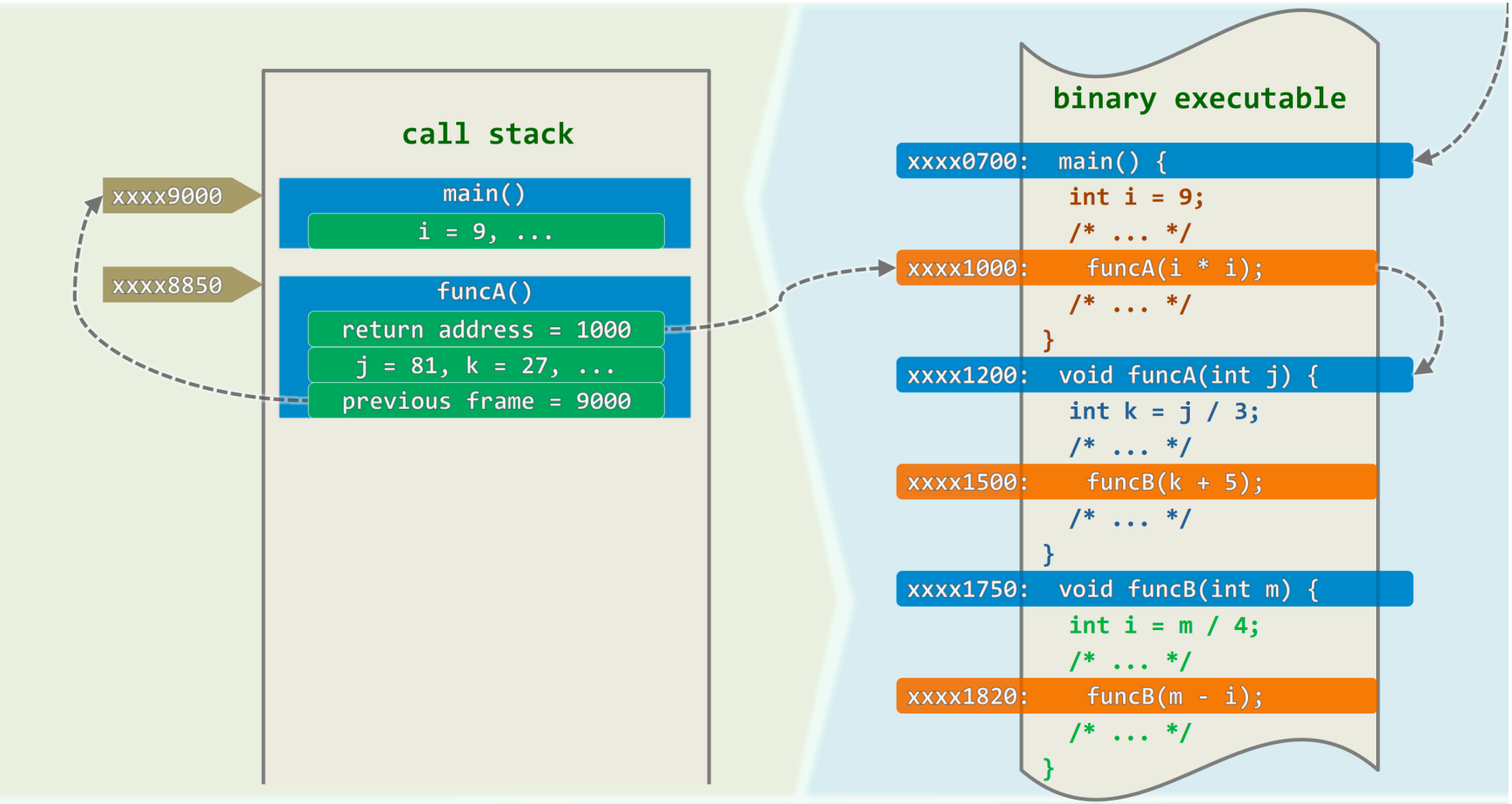
4 队列

5 队列应用

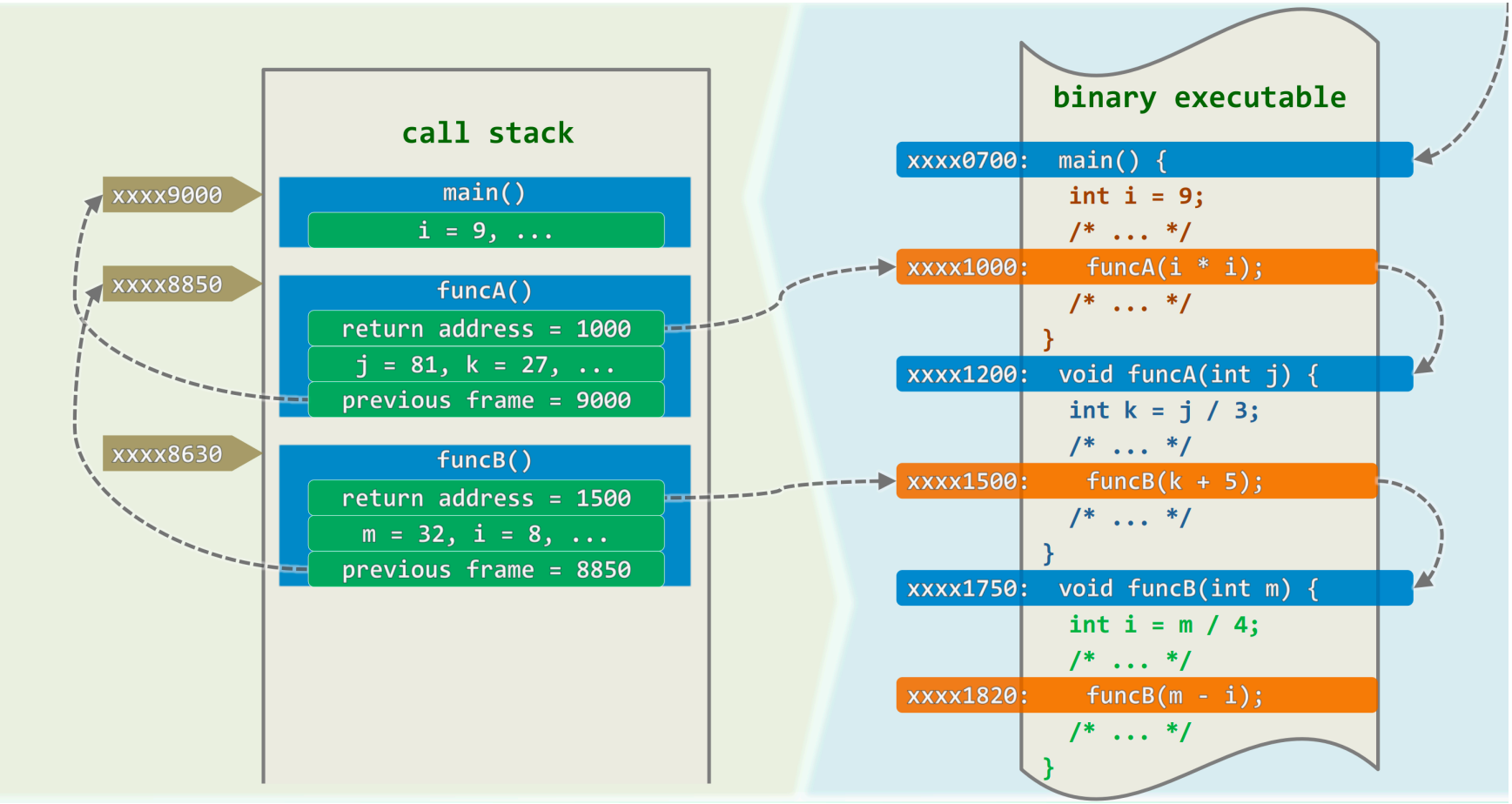
调用栈：原理与算法


call stack
xxxx9000
main()
i = 9, ...
binary executable
xxxx0700: main() {
int i = 9;
/* ... */
xxxx1000: funcA(i * i);
/* ... */
}
xxxx1200: void funcA(int j) {
int k = j / 3;
/* ... */
xxxx1500: funcB(k + 5);
/* ... */
}
xxxx1750: void funcB(int m) {
int i = m / 4;
/* ... */
xxxx1820: funcB(m - i);
/* ... */
}

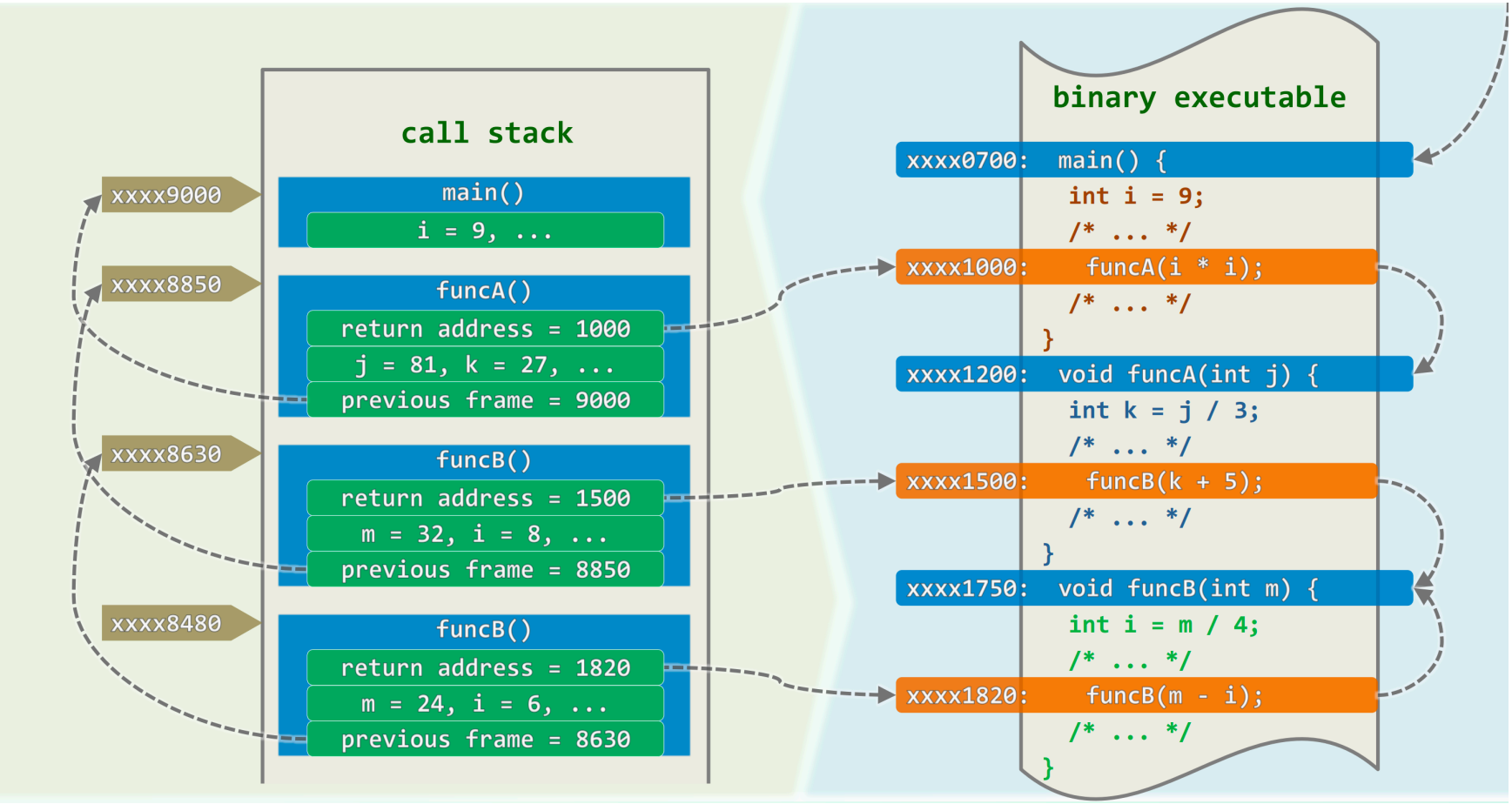
调用栈：原理与算法


call stack
xxxx9000
main()
i = 9, ...
xxxx8850
funcA()
return address = 1000
j = 81, k = 27, ...
previous frame = 9000
binary executable
xxxx0700: main() {
int i = 9;
/* ... */
xxxx1000: funcA(i * i);
/* ... */
}
xxxx1200: void funcA(int j) {
int k = j / 3;
/* ... */
xxxx1500: funcB(k + 5);
/* ... */
}
xxxx1750: void funcB(int m) {
int i = m / 4;
/* ... */
xxxx1820: funcB(m - i);
/* ... */
}

调用栈：原理与算法

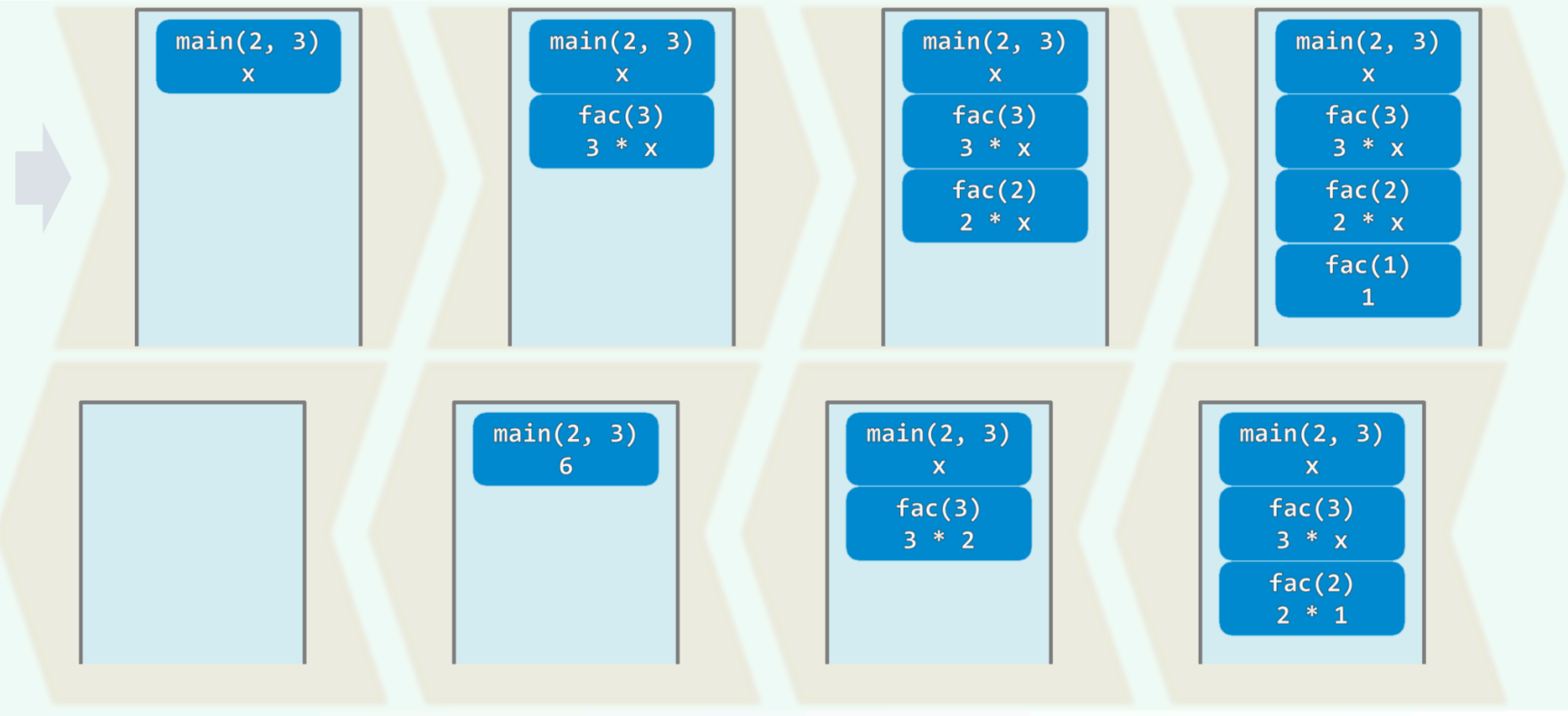

call stack
xxxx9000
main()
i = 9, ...
xxxx8850
funcA()
return address = 1000
j = 81, k = 27, ...
previous frame = 9000
xxxx8630
funcB()
return address = 1500
m = 32, i = 8, ...
previous frame = 8850
binary executable
xxxx0700: main() {
int i = 9;
/* ... */
xxxx1000: funcA(i * i);
/* ... */
}
xxxx1200: void funcA(int j) {
int k = j / 3;
/* ... */
xxxx1500: funcB(k + 5);
/* ... */
}
xxxx1750: void funcB(int m) {
int i = m / 4;
/* ... */
xxxx1820: funcB(m - i);
/* ... */
}

调用栈：原理与算法


call stack
xxxx9000
main()
i = 9, ...
xxxx8850
funcA()
return address = 1000
j = 81, k = 27, ...
previous frame = 9000
xxxx8630
funcB()
return address = 1500
m = 32, i = 8, ...
previous frame = 8850
xxxx8480
funcB()
return address = 1820
m = 24, i = 6, ...
previous frame = 8630
binary executable
xxxx0700: main() {
int i = 9;
/* ... */
xxxx1000: funcA(i * i);
/* ... */
}
xxxx1200: void funcA(int j) {
int k = j / 3;
/* ... */
xxxx1500: funcB(k + 5);
/* ... */
}
xxxx1750: void funcB(int m) {
int i = m / 4;
/* ... */
xxxx1820: funcB(m - i);
/* ... */
}

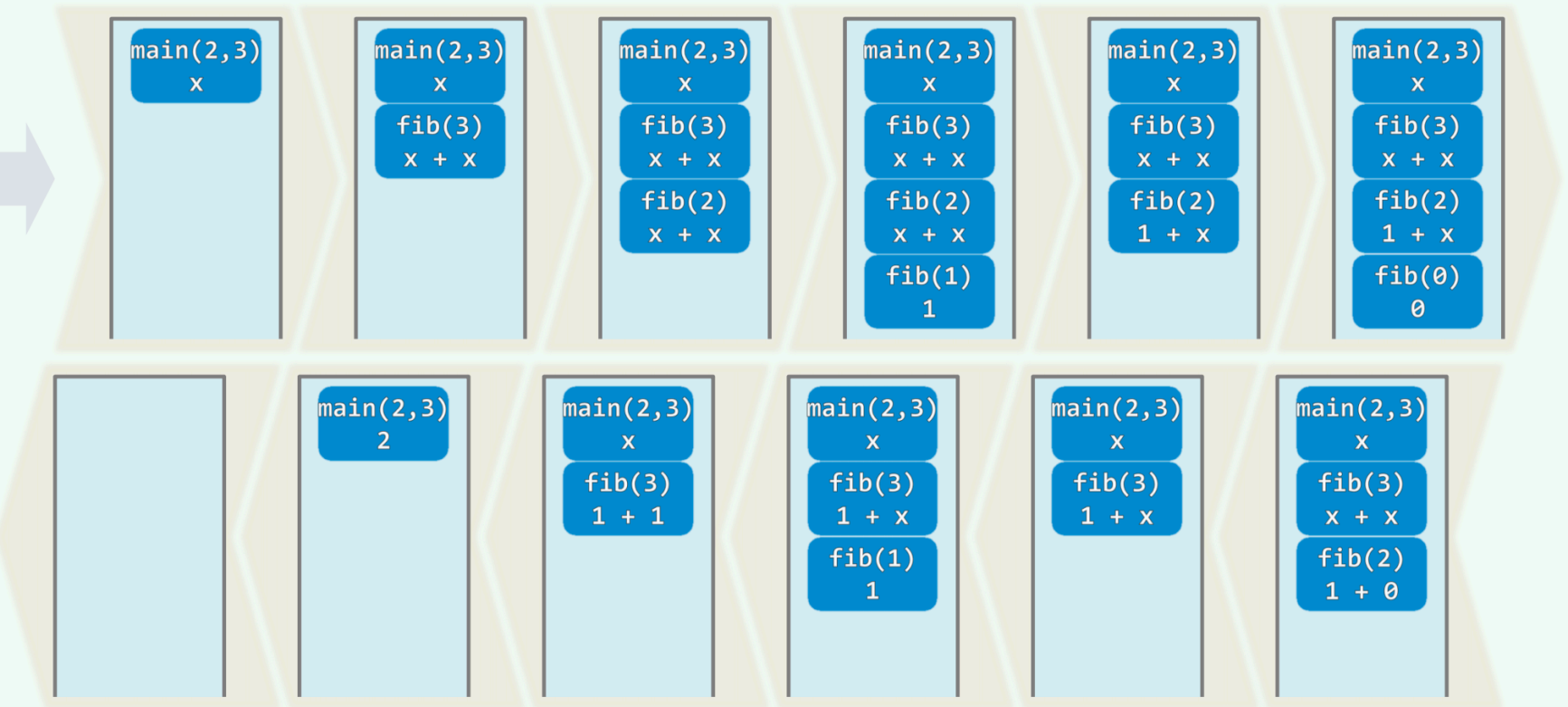
调用栈：深度与空间

```
int fac(int n) { return (n < 2) ? 1 : n * fac(n - 1); }
```



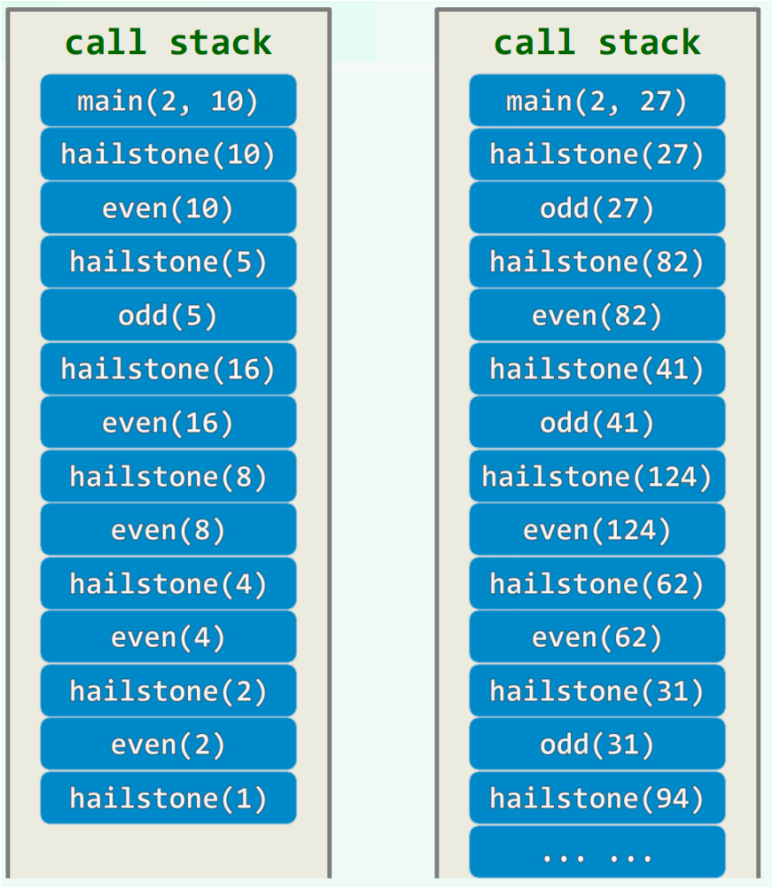
调用栈：深度与空间

```
int fib( int n ) { return (n < 2) ? n : fib(n - 1) + fib(n - 2); }
```



调用栈：深度与空间

- Hailstone 序列：
  $H(n) =$
  - $\{1\}, n \leq 1$
  - $\{n\} \cup H(n/2), n$ 为偶数
  - $\{n\} \cup H(3n+1), n$ 为奇数
- 递归算法所需的空间，主要取决于递归深度，而非递归实例总数



调用栈：消除递归

- 递归函数的空间复杂度
  - 主要取决于最大递归深度
  - 而非递归实例总数
- 隐式地维护调用栈需花费额外的时间、空间
- 为节省空间，可以
  - 显式地维护调用栈
  - 将递归算法改写为迭代版本

调用栈：消除递归

```
// 阶乘
int fac( int n ) {
    int f = 1; //O(1)空间
    while ( n > 1 )
        f *= n--;
    return f;
}

// Hailstone序列
void hailstone( int n ) { //O(1)空间
    while ( 1 < n )
        n = n % 2 ? 3*n + 1 : n/2;
}
```

调用栈：消除递归

```
// 斐波那契数列
int fib( int n )
{ //O(1)空间
    int f = 0, g = 1;
    while ( 0 < n-- )
    {
        g += f;
        f = g - f;
    }
    return f;
}
```

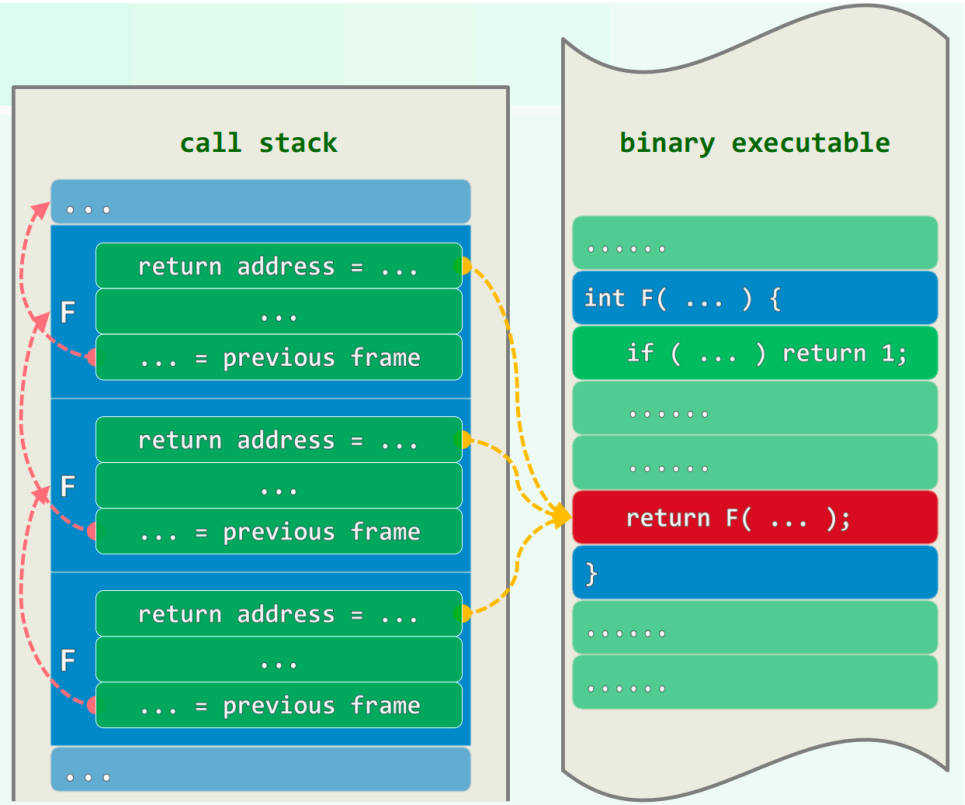
调用栈：尾递归

- 在递归实例中，作为最后一步的递归调用

```
__int64 facR(n)
{
    //base
    if (1 > n)
        return 1;
    //tail recursion
    return n * fac(n-1);
}
```

调用栈：尾递归

- 最简单的递归模式
- 一旦抵达递归基，便会
  - 引发一连串的 return（且返回地址相同）
  - 调用栈相应地连续pop
- 故不难改写为迭代形式
- 越来越多的编译器可以自动识别并代为改写
- 时间复杂度有常系数改进空间复杂度或有渐近改进



调用栈：尾递归

- 消除

fac(n) { //尾递归	fac(n) { //统一转换为迭代	fac(n) { //简捷
	int f = 1; //记录子问题的解	int f = 1;
	next: //转向标志，模拟递归调用	
if (1 > n) return 1;	if (1 > n) return f;	while (1 < n)
return n * fac(n-1);	f *= n--;	f *= n--;
	goto next; //模拟递归返回	return f;
} //O(n)时间 + O(n)空间	} //O(n)时间 + O(1)空间	} //O(n)时间 + O(1)空间

1 栈

2 栈与递归

3 栈的典型应用
- 逆序输出
- 递归嵌套
- 延迟缓冲
- 逆波兰表达式

4 队列

5 队列应用

栈的典型应用

1. 逆序输出
   - conversion
   - 输出次序与处理过程颠倒
   - 递归深度和输出长度不易预知
2. 递归嵌套
   - stack permeation + parenthesis
   - 具有自相似性的问题可递归描述，但分支位置和嵌套深度不固定
3. 延迟缓冲
   - evaluation
   - 线性扫描算法模式中，在预读足够长之后，方能确定可处理的前缀
4. 栈式计算
   - RPN
   - 基于栈结构的特定计算模式

1 栈

2 栈与递归

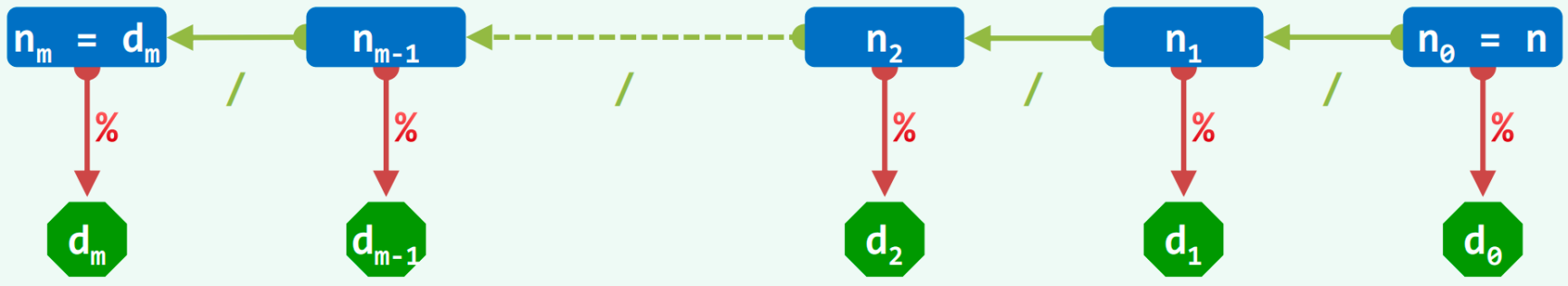
3 栈的典型应用
- 逆序输出
- 递归嵌套
- 延迟缓冲
- 逆波兰表达式

4 队列

5 队列应用

进制转换：原理

- 设 $n = (d_m \ldots d_2 d_1 d_0)_\lambda = d_m\lambda^m + \cdots + d_2\lambda^2 + d_1\lambda^1 + d_0\lambda^0$
- 令 $n_i = (d_m \ldots d_{i+2} d_{i+1} d_i)_\lambda$
- 则有 $n_{i+1} = n_i/\lambda$ 和 $d_i = n_i\%\lambda$
- 如此对 λ 反复整除、留余，即可自低而高得出 λ 进制的各位



进制转换：难点

- 位数 m 并不确定，如何正确记录并输出转换结果？具体地
  - 如何支持足够大的 m，同时空间也不浪费？
  - 自低而高得到的数位，如何自高而低输出？
- 若使用向量，则扩容策略必须得当
- 若使用列表，则多数接口均被闲置
- 使用栈，既可满足以上要求，亦可有效控制计算成本

进制转换：递归实现

```
void convert(Stack<char>& S, __int64 n, int base)
{
    static char digit[] = "0123456789ABCDEF";
    if (0 < n) {
        S.push(digit[n % base]);
        convert(S, n / base, base);
    }
}
```

进制转换：迭代实现

```
void convert(Stack<char>& S, __int64 n, int base)
{
    char digit[] = "0123456789ABCDEF";
    while ( n > 0 )
    {
        S.push( digit[ n % base ] );
        n /= base;
    }
}
```

1 栈

2 栈与递归

3 栈的典型应用
- 逆序输出
- 递归嵌套
- 延迟缓冲
- 逆波兰表达式

4 队列

5 队列应用

括号匹配

```
( a [ i - 1 ] [ j + 1 ] ) + a [ i + 1 ] [ j - 1 ] ) * 2
```

```
( a [ i - 1 ] [ j + 1 ]   + a [ i + 1 ] [ j - 1 ] ) * 2
```

- 除了各种括号，其余符号均可暂时忽略

```
(   [       ] [       ] )     [       ] [       ] )
```

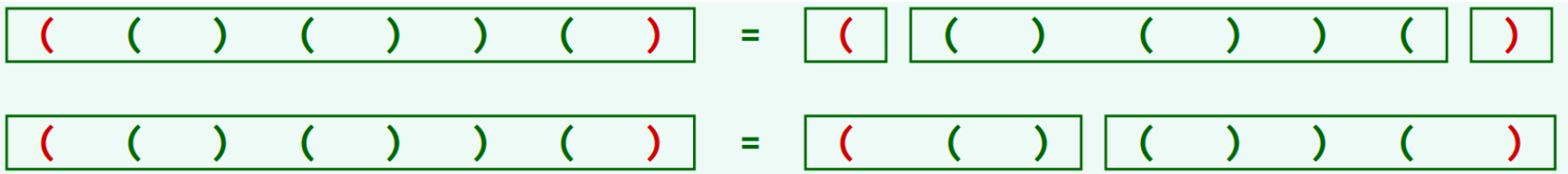
```
(   [       ] [       ]       [       ] [       ] )
```

- 从简单入手，先来考查只有一种括号的情况

括号匹配：由外而内的思路

1. 平凡：无括号的表达式是匹配的
2. 减治？S 匹配，仅当（S）匹配
3. 分治？S1 和 S2 均匹配，仅当 S1+S2 匹配

- 然而，根据以上性质，却不易直接应用已知的策略
- 究其根源在于，2 和 3 均为充分性，比如反例：



括号匹配：递归实现

1. 表达式 S 可以分解为如下形式：
   S = S0 + "(" + S1 + ")" + S2 + S3
2. 其中，S0 和 S3 不含括号，且 S1 中左右括号数目相等
3. 则 S 匹配当且仅当 S1 和 S2 均匹配

```
void trim(const char exp[], int& lo, int& hi)
{
    // 查找第一个和最后一个括号
    while((lo<=hi) && (exp[lo]!='(') &&
          (exp[lo]!=')')) lo++;
    while((lo<=hi) && (exp[hi]!='(') &&
          (exp[hi]!=')')) hi--;
}
```

括号匹配：递归实现

```
int divide(const char exp[], int lo, int hi)
{
    int mi = lo;
    // crc为[lo,mi]内左右括号数目之差
    int crc = 1;
    // 逐个检查字符，直到左右括号数目相等或者越界
    while((0 < crc) && (++mi < hi)) {
        if(exp[mi] == ')')  crc--;
        if(exp[mi] == '(')  crc++;
    }
    // 若mi<=hi，则为合法切分点
    // 否则，意味着局部不匹配
    return mi;
}
```

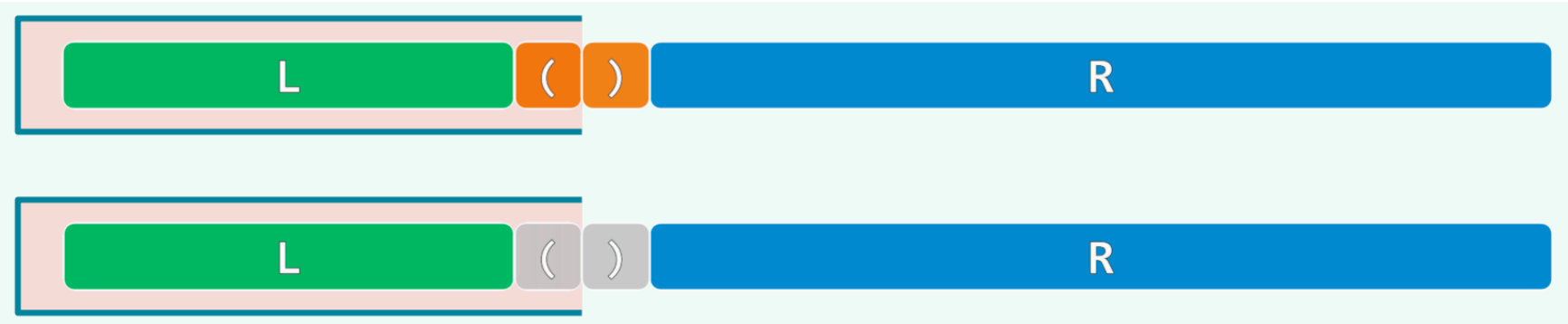
括号匹配：递归实现

```
bool paren(const char exp[], int lo, int hi)
{
    trim(exp, lo, hi); if(lo > hi) return true;
    if(exp[lo] != '(') return false;
    if(exp[hi] != ')') return false;
    int mi = divide(exp, lo, hi);
    // 切分点不合法，意味着局部以至整体不匹配
    if(mi > hi) return false;
    return paren(exp, lo+1, mi-1) &&
           paren(exp, mi+1, hi);
}
```

- 最坏情况下，divide() 需要线性时间
- 递归深度为 $\mathcal{O}(n)$，算法共需 $\mathcal{O}(n^2)$ 时间

括号匹配：由内而外的思路

- 颠倒之前思路：消去一对紧邻的左右括号，不影响全局的匹配判断
- 亦即："L+()+R"匹配，仅当"L+R"匹配
- 那么如何找到这对括号？再者，如何使问题的这种简化得以持续进行？
- 顺序扫描表达式，用栈记录已扫描的部分
- 反复迭代：凡遇"("，则进栈；凡遇")"，则出栈



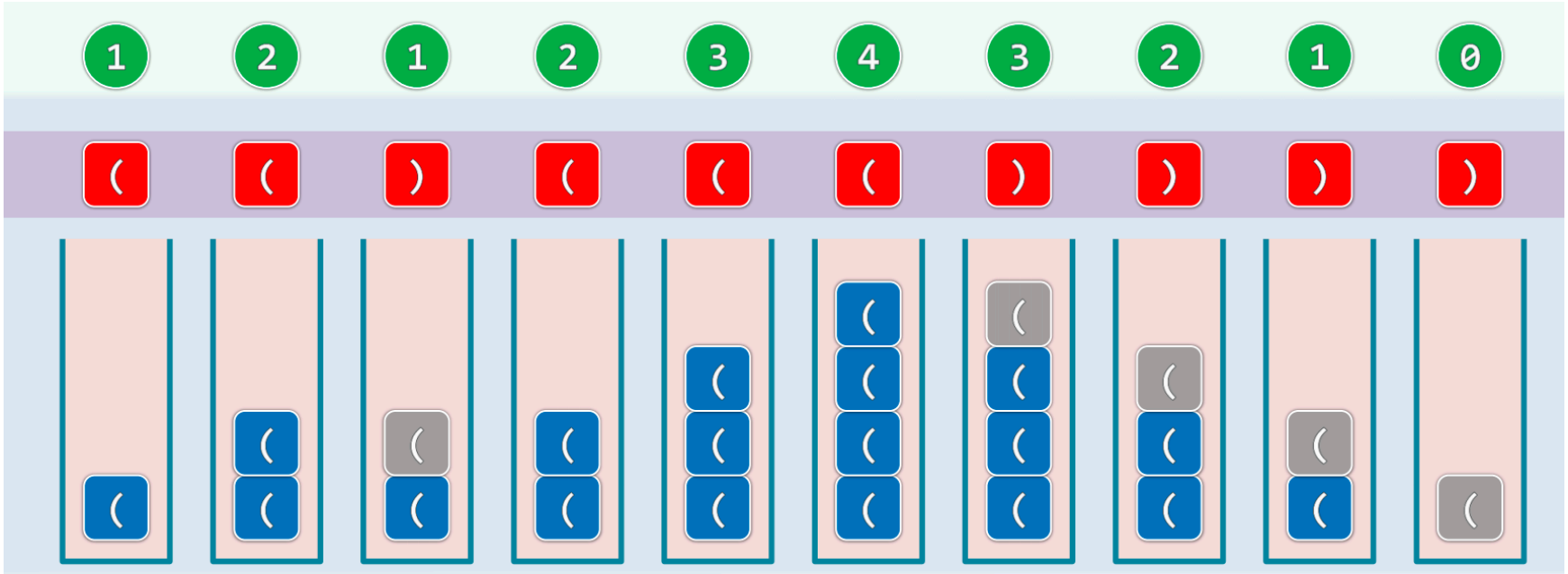
括号匹配：迭代实现

```
// 针对只有括号的序列
bool paren(const char exp[], int lo, int hi ) {
    Stack<char> S;
    for ( int i = lo; i < hi; i++ )
        if ( '(' == exp[i] )
            S.push( exp[i] ); //遇左括号：则进栈
        else if ( ! S.empty() )
            S.pop();
        else return false;
    return S.empty(); //最终栈空，当且仅当匹配
}
```

- 算法共需 $\mathcal{O}(n)$ 时间

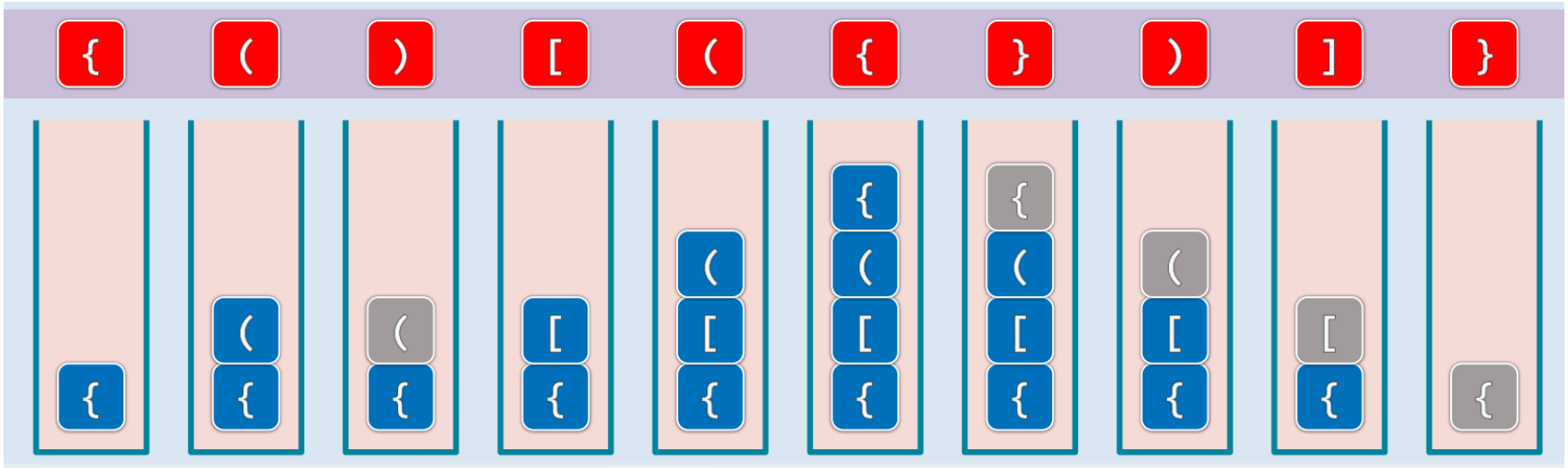
括号匹配：迭代实现

- 实际上，若仅考虑一种括号，只需一个计数器足矣：S.size()
- 一旦转负，则为失配（右括号多余）；最后归零，即为匹配（否则左括号多余）



多类括号匹配

- 以上思路及算法，可否推广至多种括号并存的情况？
- 甚至，只需约定“括号”的通用格式，而不必事先固定括号的类型与数目
- 比如：<body>|</body>, <font>|</font>, ...

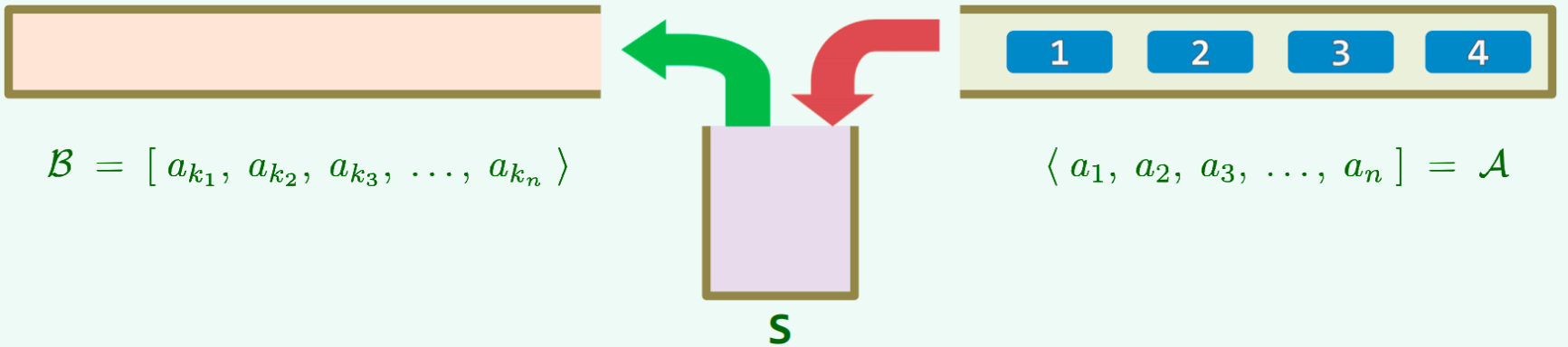


多类括号匹配：迭代实现

```
bool paren(const char exp[], int lo, int hi ) {
    Stack<char> S;
    for( int i = lo; i < hi; i++ )
        switch (exp[i]) {
            case '(': case '[': case '{':
                S.push(exp[i]); break;
            case ')':
                if(S.empty() || ('(' != S.pop()))
                    return false; break;
            case ']': ...
            case '}': ...
            default: break; // 非符号字符一律忽略
        }
    return S.empty();
}
```

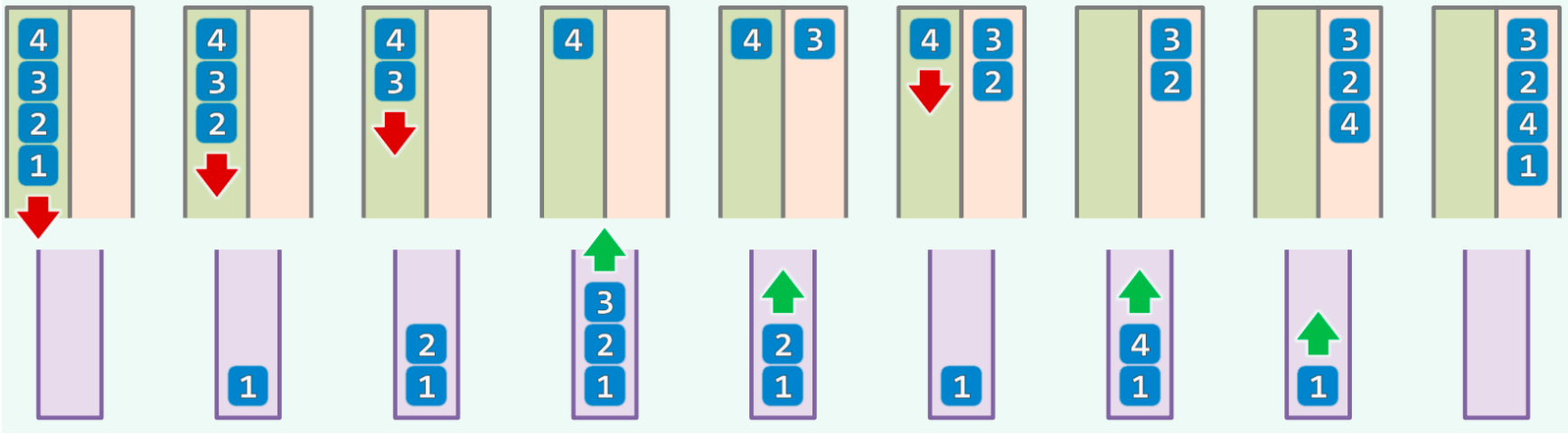
栈混洗

- 考察栈 A=<$a_1, a_2, a_3, \ldots, a_n$]，B=S=$\Phi$
- 只允许
  - 将 A 的顶元素弹出并压入 S，即 S.push(A.pop())
  - 或将 S 的顶元素弹出并压入 B，即 B.push(S.pop())
- 若经一系列以上操作后，A 中元素全部转入 B 中
  B = $[a_{k_1}, a_{k_2}, a_{k_3}, \ldots, a_{k_n}>$
- 则称为 A 的一个栈混洗



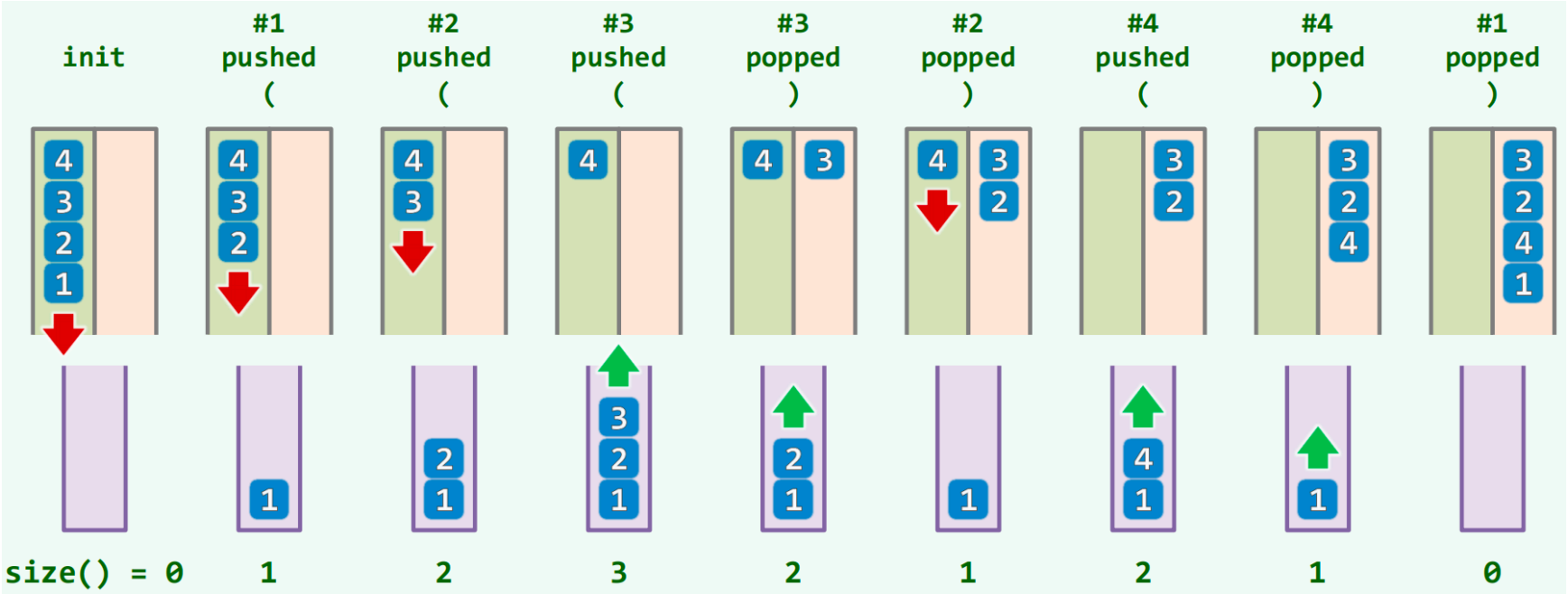
栈混洗

- 同一输入序列，可有多种栈混洗：[1, 2, 3, 4 >，[4, 3, 2, 1 >，[3, 2, 4, 1 > ..
- 由 n 次 push 和 n 次 pop 构成的任何操作序列，只要满足“任一前缀中的 push 不少于 pop”这一限制，该序列也必然对应某个栈混洗。



栈混洗和括号匹配

- 每一栈混洗，都对应于栈 S 的 n 次 push 与 n 次 pop 操作构成的某一序列；反之亦然
- n 个元素的栈混洗，等价于 n 对括号的匹配；二者的组合数，也自然相等



1 栈

2 栈与递归

3 栈的典型应用
- 逆序输出
- 递归嵌套
- 延迟缓冲
- 逆波兰表达式

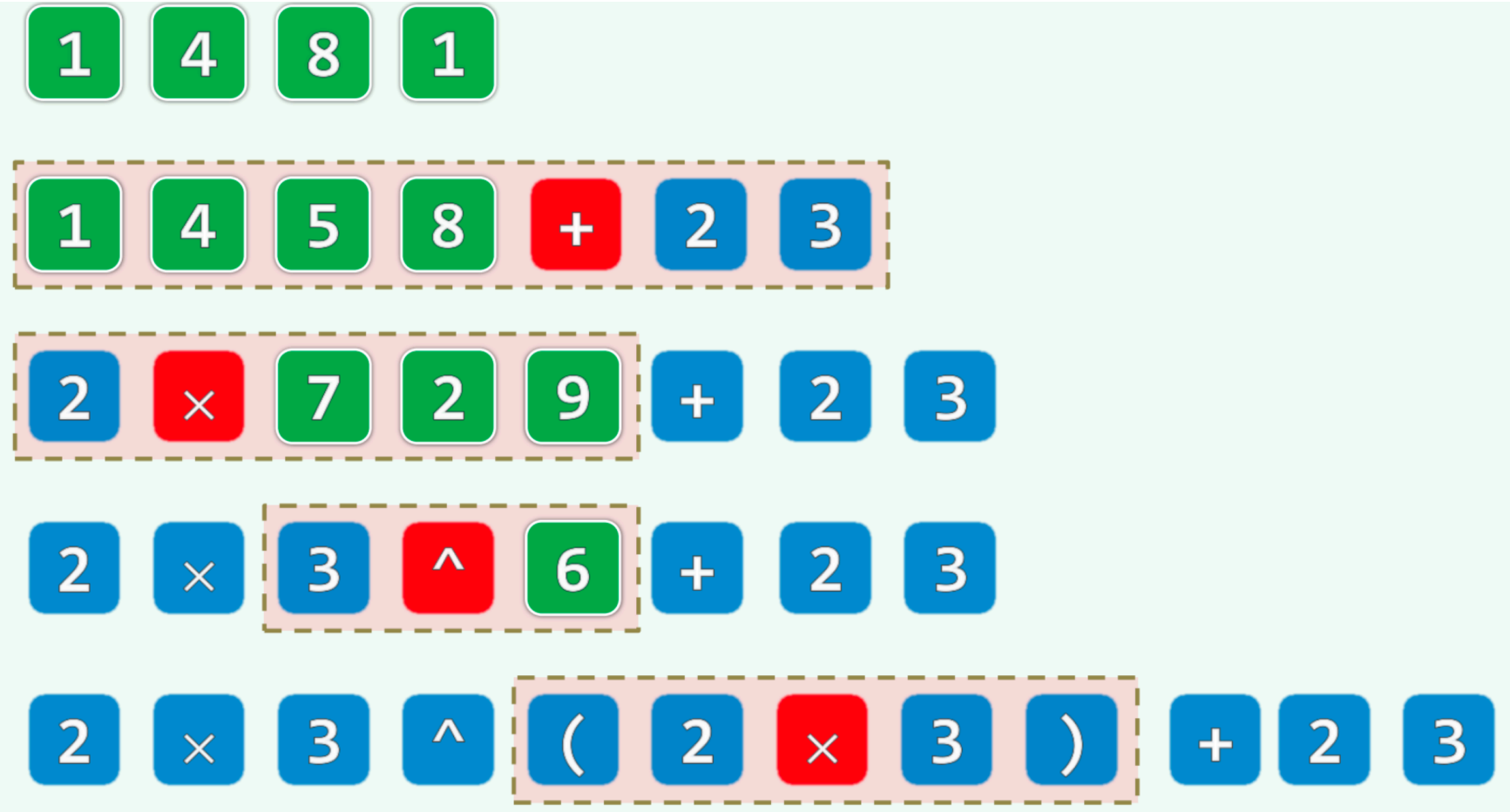
4 队列

5 队列应用

中缀表达式求值：减而治之

- 思路：优先级高的局部执行计算，并被代以其数值；运算符渐少，直至得到最终结果
- 定义：str(v)：数值 v 对应的字符串（名），
  val(S)：符号串 S 对应的数值（实）
- 设表达式：$S = S_L + S_0 + S_R$
- S_0 可优先计算，且 $\mathrm{val}(S_0) = v_0$
- 则有递推化简关系
  $\mathrm{val}(S) = \mathrm{val}(\ S_L + \mathrm{str}(v_0) + S_R\ )$

中缀表达式求值：实例

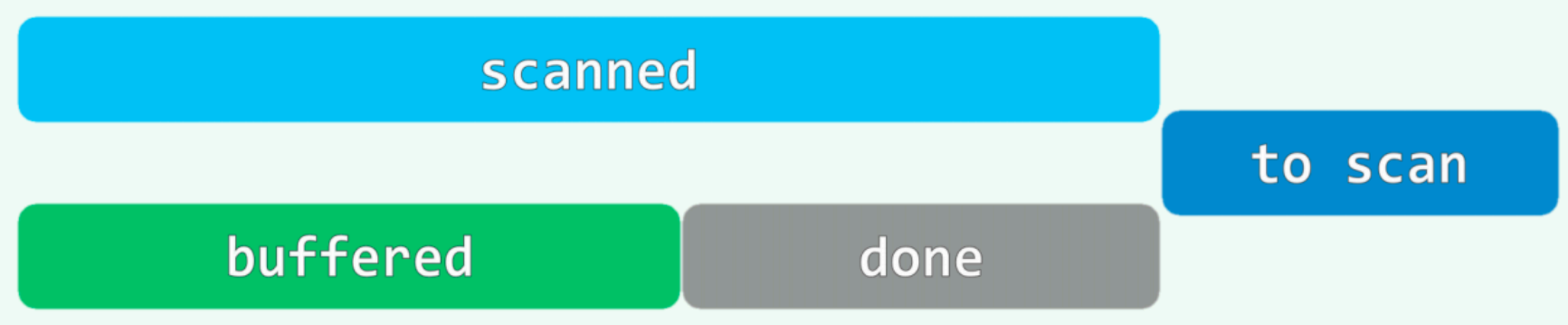


中缀表达式求值：优先级

- 难点：如何高效地找到可优先计算的S_0（亦即，其对应的运算符）？
- 与括号匹配迭代版类似，但亦不尽相同
  - 不能简单地按“左先右后”次序处理各运算符
  - 此时，需要考虑更多因素...
- 约定俗成的优先级：
  1 + 2 * 3 ^ 4 !
- 可强行改变次序的括号：
  (((1 + 2) * 3) ^ 4) !

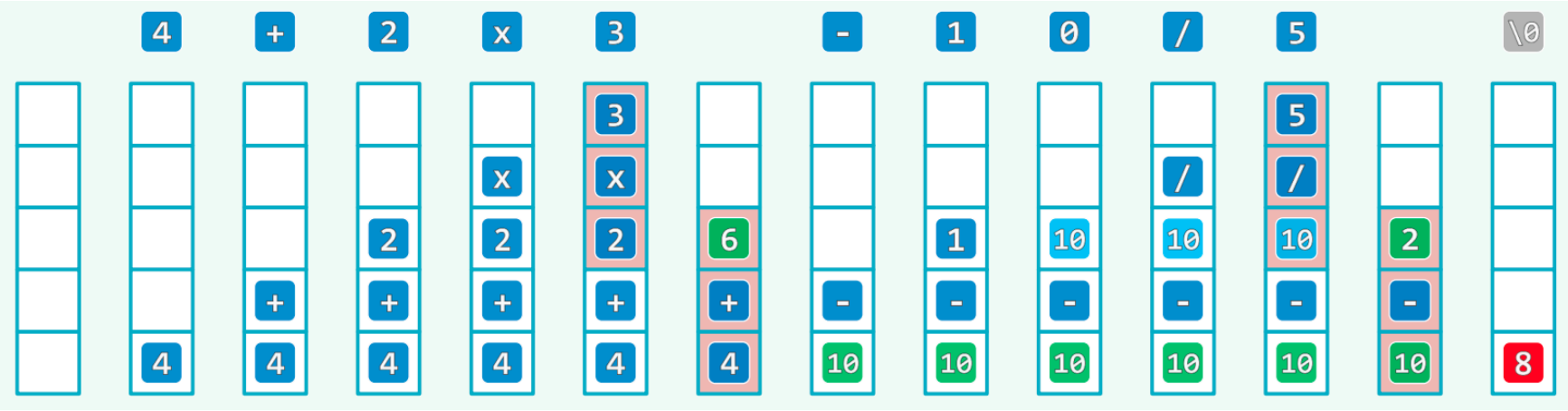
中缀表达式求值：延迟缓冲

- 仅根据表达式的前缀，不足以确定各运算符的计算次序
- 只有获得足够的后续信息，才能确定其中哪些运算符可以执行
- 体现在求值算法的流程上，为处理某一前缀，必须提前预读并分析更长的前缀
- 为此，需借助某种支持延迟缓冲的机制



中缀表达式求值：栈 + 线性扫描

- 自左向右扫描表达式，用栈记录已扫描的部分，及中间结果
- If (栈的顶部存在可优先计算的子表达式)
  - Then 令其退栈并计算；计算结果进栈
  - Else 当前字符进栈，转入下一字符
- 栈内最终所剩的那个元素，即表达式之值



中缀表达式求值：主算法

```
double evaluate( char* S, char* RPN ) {
    Stack<double> opnd; //运算数栈
    Stack<char> optr; //运算符栈
    optr.push('\0'); //哨兵
    while ( ! optr.empty() )
    { //逐个处理各字符，直至运算符栈空
        if ( isdigit( *S ) ) //若为运算数
            readNumber( S, opnd ); //读入
        else //若为运算符，比较与栈顶的优先级
            switch( priority( optr.top(), *S ) )
            { /* 分别处理 */ }
    }
    return opnd.pop(); //弹出并返回最后的计算结果
}
```

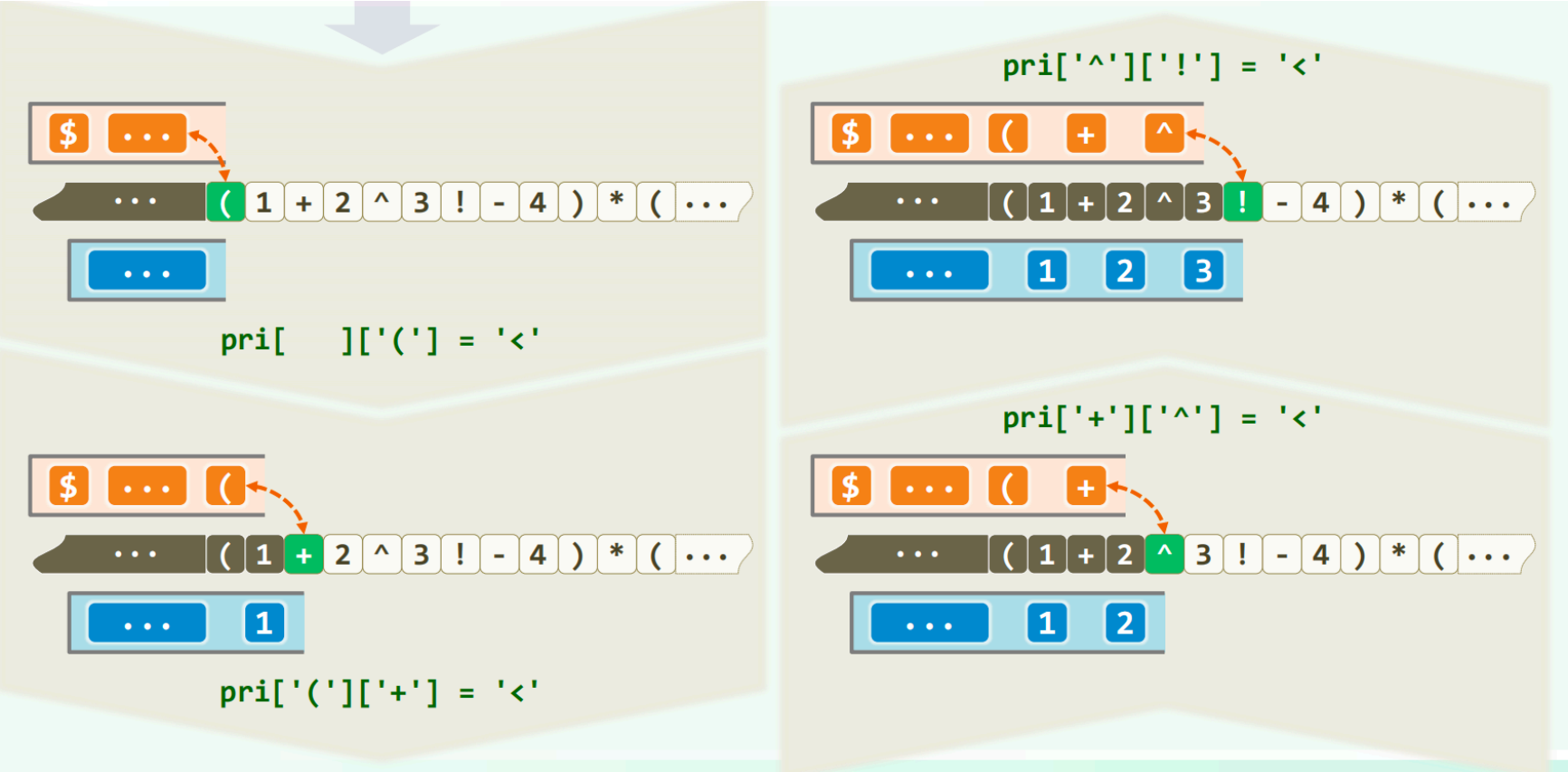
中缀表达式求值：优先级表

```
const char pri[N_OPTR][N_OPTR] = { //运算符优先等级 [栈顶][当前]
   /* --  + */  '>', '>', '<', '<', '<', '<', '<', '>', '>',
   /*  |  - */  '>', '>', '<', '<', '<', '<', '<', '>', '>',
   /* 栈  * */  '>', '>', '>', '>', '<', '<', '<', '>', '>',
   /* 顶  / */  '>', '>', '>', '>', '<', '<', '<', '>', '>',
   /* 运  ^ */  '>', '>', '>', '>', '>', '<', '<', '>', '>',
   /* 算  ! */  '>', '>', '>', '>', '>', '>', ' ', '>', '>',
   /* 符  ( */  '<', '<', '<', '<', '<', '<', '<', '=', ' ',
   /*  |  ) */  ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',
   /* -- \0 */  '<', '<', '<', '<', '<', '<', '<', ' ', '='
   //           +    -    *    /    ^    !    (    )    \0
   //           |-------------- 当前运算符 --------------|
```

'<': 静待时机: 算法

```
switch( priority( optr.top(), *S ) ) {
    case '<': //栈顶运算符优先级更低
        optr.push( *S );
        S++; break; //计算推迟, 当前运算符进栈
    case '=':
        /* ...... */
    case '>': {
        /* ...... */
        break;
        } //case '>'
} //switch
```

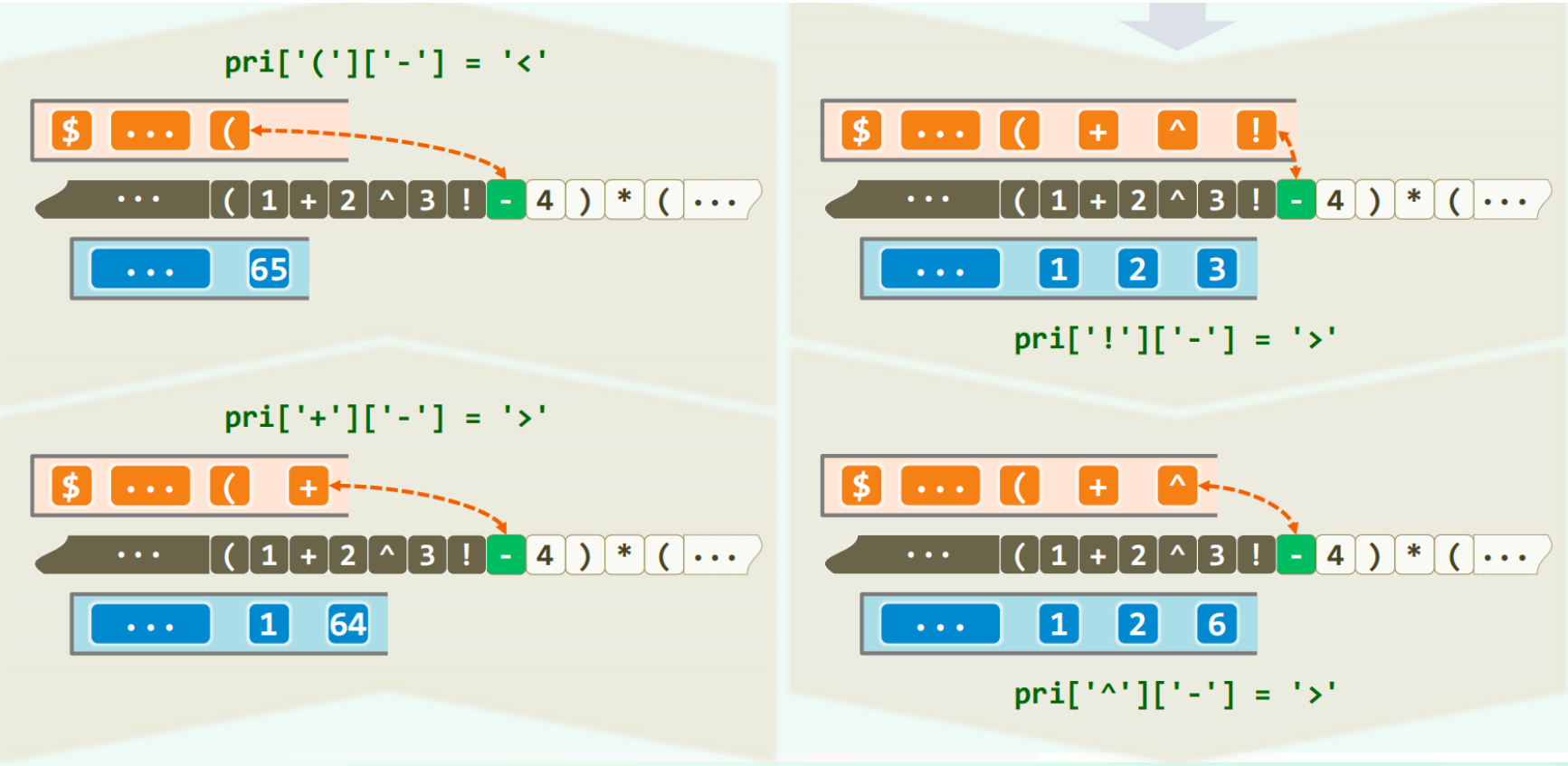
'<'：静待时机：实例



'>': 时机已到：算法

```
switch( priority( optr.top(), *S ) ) {
    /* ...... */
    case '>': {
        char op = optr.pop();
        if ( '!' == op ) //一元运算符
            opnd.push( calcu( op, opnd.pop() ) );
        else { //二元运算符
            double opnd2 = opnd.pop();
            double opnd1 = opnd.pop();
            //实施计算，结果入栈
            opnd.push(calcu( opnd1, op, opnd2 ));
        } //为何不简写？
        break;
    } //case '>'
} //switch
```

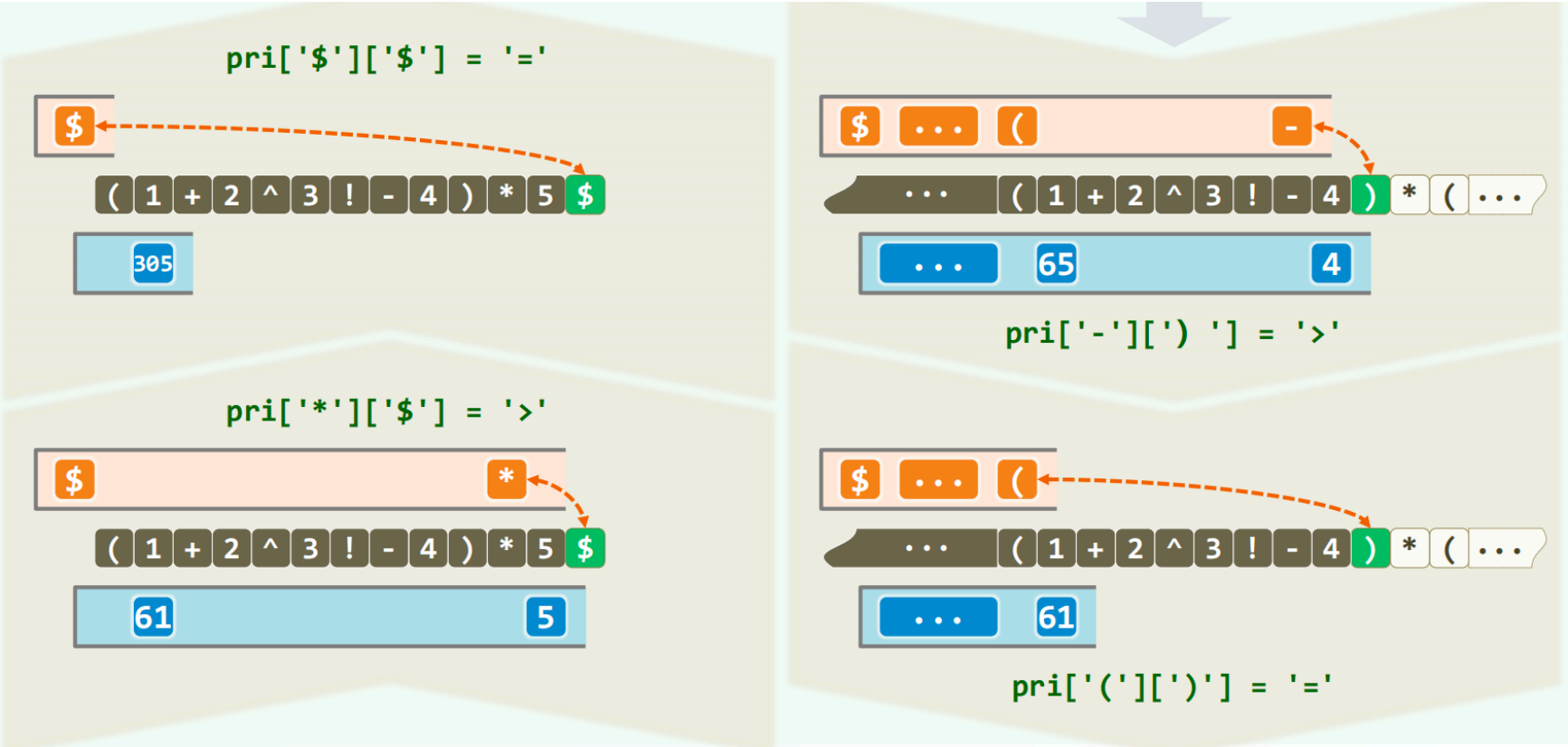
'>'：时机已到：实例



'=': 终须了断: 算法

```
switch( priority( optr.top(), *S ) ) {
    case '<':
    /* ...... */
    case '=': //当前为右括号, 或尾部哨兵'\0'
        optr.pop();
        S++; break; //脱括号并接收下一个字符
    case '>': {
    /* ...... */
    break;
    } //case '>'
} //switch
```

'=': 终须了断：实例



+ - * / !: 芸芸众生

```
const char pri[N_OPTR][N_OPTR] = { //运算符优先等级 [栈顶][当前]
   /* --  + */  '>', '>', '<', '<', '<', '<', '<', '>', '>',
   /*  |  - */  '>', '>', '<', '<', '<', '<', '<', '>', '>',
   /* 栈  * */  '>', '>', '>', '>', '<', '<', '<', '>', '>',
   /* 顶  / */  '>', '>', '>', '>', '<', '<', '<', '>', '>',
   /* 运  ^ */  '>', '>', '>', '>', '>', '<', '<', '>', '>',
   /* 算  ! */  '>', '>', '>', '>', '>', '>', ' ', '>', '>',
   /* 符  ( */  '<', '<', '<', '<', '<', '<', '<', '=', ' ',
   /*  |  ) */  ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',
   /* -- \0 */  '<', '<', '<', '<', '<', '<', '<', ' ', '='
   //            +    -    *    /    ^    !    (    )    \0
   //            |-------------- 当前运算符 --------------|
```

'(': 我不下地狱，谁下地狱

```
const char pri[N_OPTR][N_OPTR] = { //运算符优先等级 [栈顶][当前]
    /* --  + */  '>', '>', '<', '<', '<', '<', '<', '>', '>',
    /*  |  - */  '>', '>', '<', '<', '<', '<', '<', '>', '>',
    /* 栈  * */  '>', '>', '>', '>', '<', '<', '<', '>', '>',
    /* 顶  / */  '>', '>', '>', '>', '<', '<', '<', '>', '>',
    /* 运  ^ */  '>', '>', '>', '>', '>', '<', '<', '>', '>',
    /* 算  ! */  '>', '>', '>', '>', '>', '>', ' ', '>', '>',
    /* 符  ( */  '<', '<', '<', '<', '<', '<', '<', '=', ' ',
    /*  |  ) */  ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',
    /* -- \0 */  '<', '<', '<', '<', '<', '<', '<', ' ', '='
    //            +    -    *    /    ^    !    (    )    \0
    //            |-------------- 当前运算符 --------------|
```

')'：死线已至（然后满血复活）

```
const char pri[N_OPTR][N_OPTR] = { //运算符优先等级 [栈顶][当前]
    /* --  + */  '>', '>', '<', '<', '<', '<', '<', '>', '>',
    /*  |  - */  '>', '>', '<', '<', '<', '<', '<', '>', '>',
    /* 栈  * */  '>', '>', '>', '>', '<', '<', '<', '>', '>',
    /* 顶  / */  '>', '>', '>', '>', '<', '<', '<', '>', '>',
    /* 运  ^ */  '>', '>', '>', '>', '>', '<', '<', '>', '>',
    /* 算  ! */  '>', '>', '>', '>', '>', '>', ' ', '>', '>',
    /* 符  ( */  '<', '<', '<', '<', '<', '<', '<', '=', ' ',
    /*  |  ) */  ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',
    /* -- \0 */  '<', '<', '<', '<', '<', '<', '<', ' ', '='
    //            +    -    *    /    ^    !    (    )    \0
    //            |-------------- 当前运算符 --------------|
```

'\0'：从创世纪，到世界末日

```
const char pri[N_OPTR][N_OPTR] = { //运算符优先等级 [栈顶][当前]
   /* --  + */  '>', '>', '<', '<', '<', '<', '<', '>', '>',
   /*  |  - */  '>', '>', '<', '<', '<', '<', '<', '>', '>',
   /* 栈  * */  '>', '>', '>', '>', '<', '<', '<', '>', '>',
   /* 顶  / */  '>', '>', '>', '>', '<', '<', '<', '>', '>',
   /* 运  ^ */  '>', '>', '>', '>', '>', '<', '<', '>', '>',
   /* 算  ! */  '>', '>', '>', '>', '>', '>', ' ', '>', '>',
   /* 符  ( */  '<', '<', '<', '<', '<', '<', '<', '=', ' ',
   /*  |  ) */  ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ',
   /* -- \0 */  '<', '<', '<', '<', '<', '<', '<', ' ', '='
   //            +    -    *    /    ^    !    (    )    \0
   //            |-------------- 当前运算符 --------------|
```

中缀表达式求值：实例

表达式	运算符栈	操作数栈	注解
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$		表达式起始标识入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ (		左括号入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ (	0	操作数0入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ (!	0	运算符'!'入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ (	1	运算符'!'出栈执行
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ (+	1	运算符'+'入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ (+	1 1	操作数1入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ (	2	运算符'+'出栈执行
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$	2	左括号出栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ *	2	运算符'*'入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ *	2 2	操作数2入栈

中缀表达式求值：实例

表达式	运算符栈	操作数栈	注解
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ * ^	2 2	运算符'^'入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ * ^ (	2 2	左括号入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ * ^ (	2 2 3	操作数3入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ * ^ (!	2 2 3	运算符'!'入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ * ^ (	2 2 6	运算符'!'出栈执行
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ * ^ (+	2 2 6	运算符'+'入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ * ^ (+	2 2 6 4	操作数4入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ * ^ (	2 2 10	运算符'+'出栈执行
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ * ^	2 2 10	左括号出栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ *	2 1024	运算符'^'出栈执行
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$	2048	运算符'*'出栈执行

中缀表达式求值：实例

表达式	运算符栈	操作数栈	注解
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ -	2048	运算符'-'入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ - (	2048	左括号入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ - (	2048 5	操作数5入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ - (!	2048 5	运算符'!'入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ - (	2048 120	运算符'!'出栈执行
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ - (-	2048 120	运算符'-'入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ - (-	2048 120 67	操作数67入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ - (	2048 53	运算符'-'出栈执行
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ - (-	2048 53	运算符'-'入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ - (- (	2048 53	左括号入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ - (- (	2048 53 8	操作数8入栈

中缀表达式求值：实例

表达式	运算符栈	操作数栈	注解
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ - (- (+	2048 53 8	运算符'+'入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ - (- (+	2048 53 8 9	操作数9入栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ - (- (	2048 53 17	运算符'+'出栈执行
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ - (-	2048 53 17	左括号出栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ - (	2048 36	运算符'-'出栈执行
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$ -	2048 36	左括号出栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$	$	2012	运算符'-'出栈执行
(0!+1)*2^(3!+4)-(5!-67-(8+9))$		2012	表达式起始标识出栈
(0!+1)*2^(3!+4)-(5!-67-(8+9))$			返回唯一的元素2012

1 栈

2 栈与递归

3 栈的典型应用
- 逆序输出
- 递归嵌套
- 延迟缓冲
- 逆波兰表达式

4 队列

5 队列应用

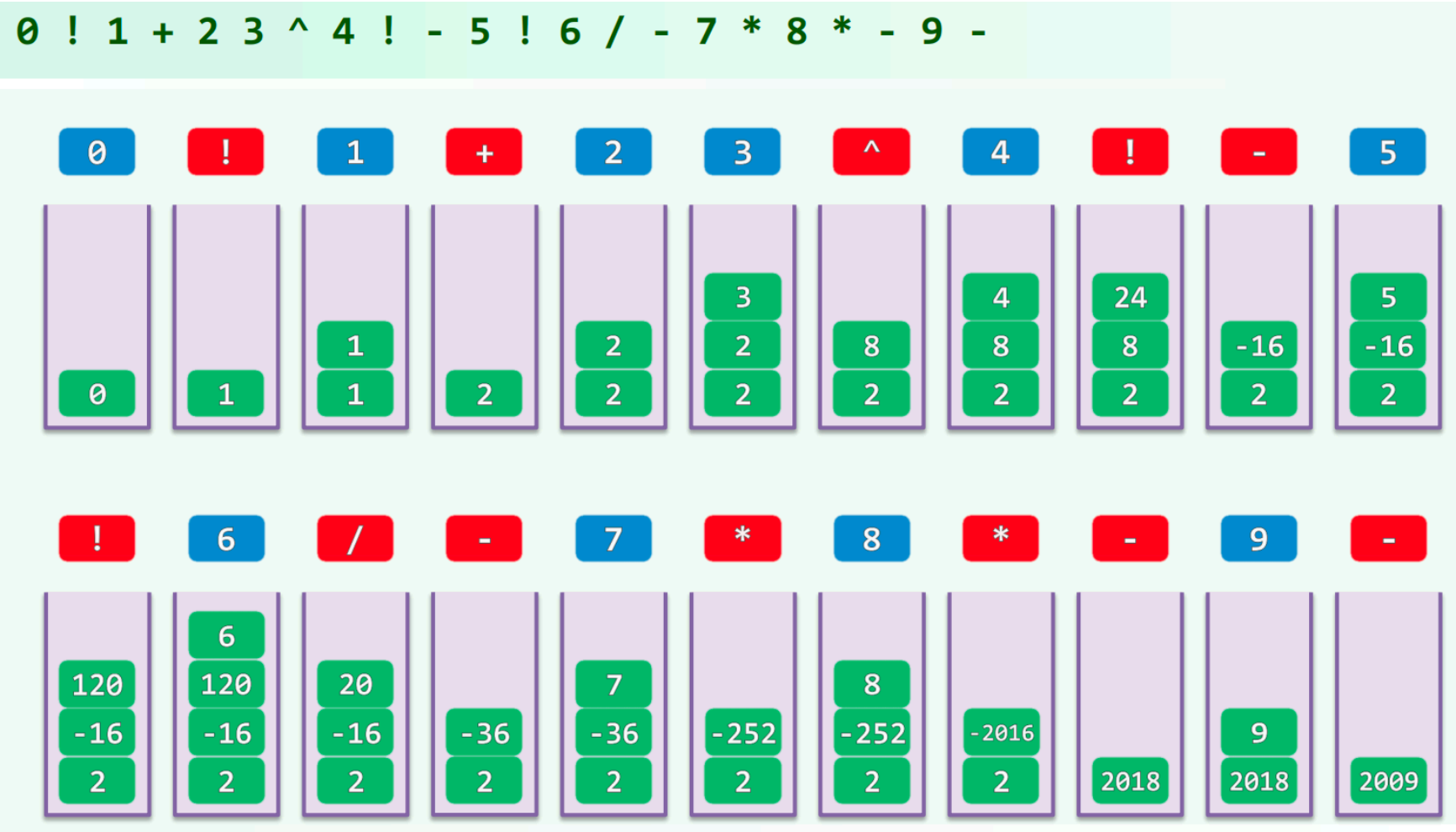
定义

- 逆波兰表达式（Reverse Polish Notation）
- 只有获得足够的后续信息，才能确定其中哪些运算符可以执行
- 在由运算符（operator）和操作数（operand）组成的表达式中
- 不使用括号（parenthesis-free），即可表示带优先级的运算关系
- 相对于日常使用的中缀式（infix），RPN 亦称作后缀式（postfix）
- 作为补偿，须额外引入一个起分隔作用的元字符（比如空格）

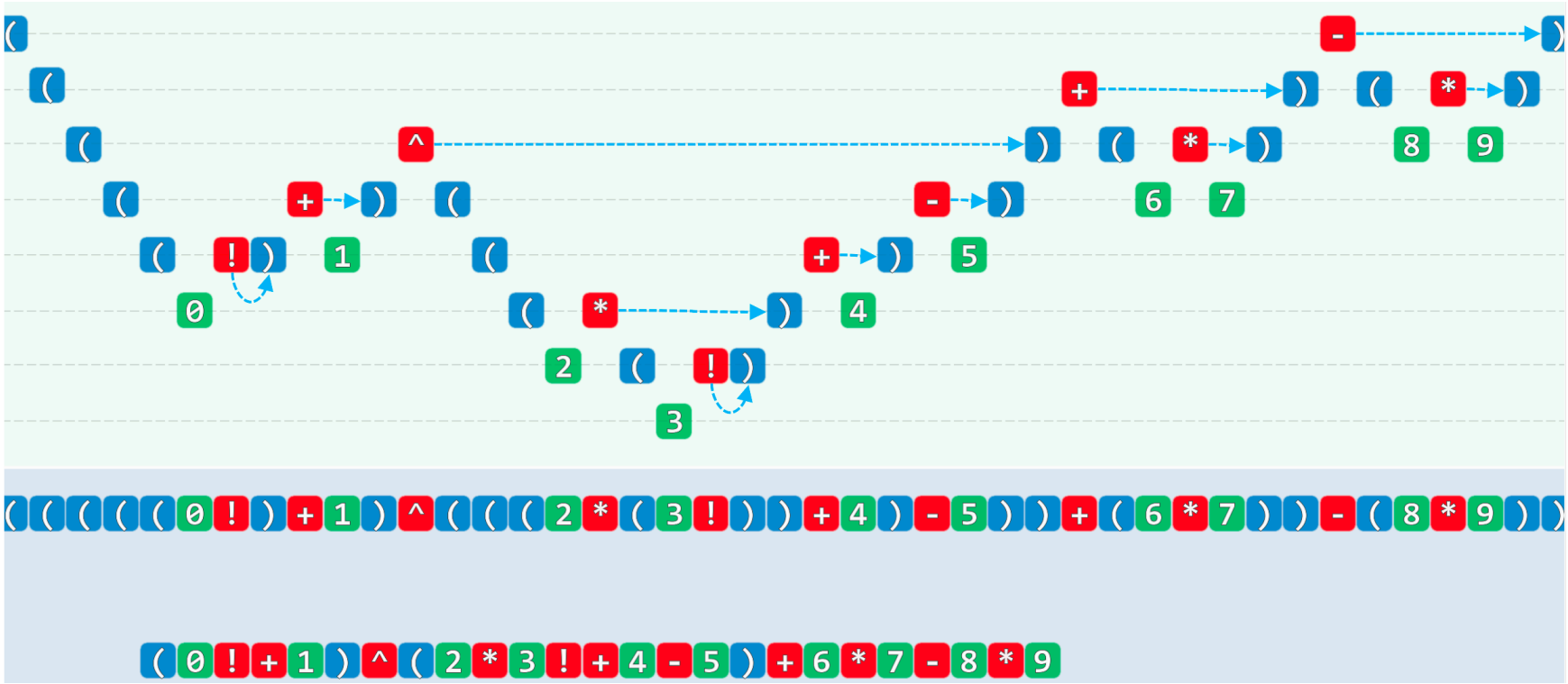
求值

```
rpnEvaluation(expr)
输入：RPN表达式expr（假定语法正确）
输出：表达式数值
{
    引入栈S，用以存放操作数；
    while(expr尚未扫描完毕) {
        从expr中读入下一个元素x;
        if (x是操作数) x压入S;
        else { // x是运算符
            从栈S中探出运算符x所需数目的操作数；
            对探出的操作数实施x运算，
            并将运算结果重新压入S;
        }
    }
    返回栈顶；
}
```

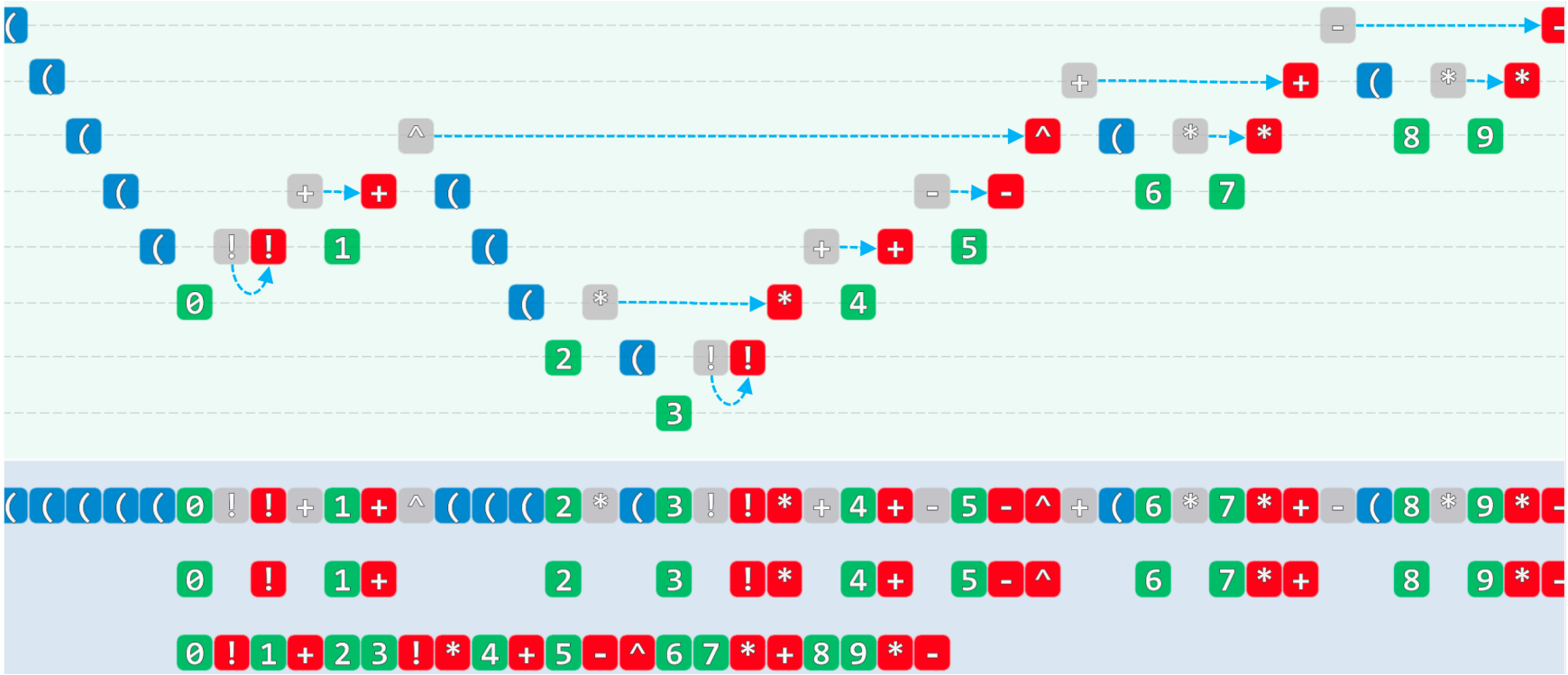
求值案例


0 ! 1 + 2 3 ^ 4 ! - 5 ! 6 / - 7 * 8 * - 9 -
0 ! 1 + 2 3 ^ 4 ! - 5
0 1 1 1 2 2 2 2 3 2 2 8 2 4 8 2 24 8 2 -16 2 5 -16 2
! 6 / - 7 * 8 * - 9 -
120 -16 2 6 120 -16 2 20 -16 2 -36 2 7 -36 2 -252 2 8 -252 2 -2016 2 2018 9 2018 2009

手工转换：添加括号



手工转换：以运算符替换右括号，清除左括号



自动转换

```
double evaluate( char* S, char* RPN ) { //RPN转换
    /* ..................................... */
    while ( ! optr.empty() ) {
        if ( isdigit( * S ) ) {
            readNumber( S, opnd );
            append( RPN, opnd.top() ); }
        else //若当前字符为运算符
            switch(priority(optr.top(), *S)) {
                /* ................... */
                case '>': {
                    char op = optr.pop();
                    append( RPN, op );
                /* ................... */
                } //case '>'
                /* ................... */
```

1 栈

2 栈与递归

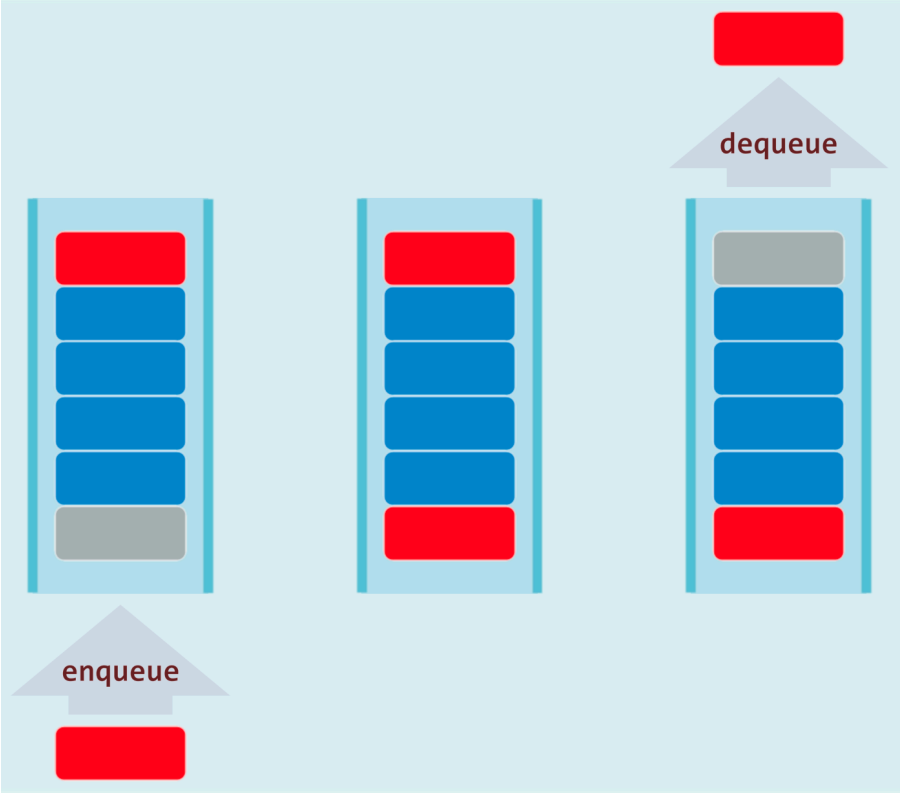
3 栈的典型应用

4 队列

5 队列应用

操作与接口

- 队列（queue）也是受限的序列
- 只能在队尾插入（查询）：enqueue() / rear()
- 只能在队头删除（查询）：dequeue() / front()
- 先进先出（FIFO）
- 后进后出（LILO）



实例

操作	输出	队列（右侧为队头）		
Queue()				
empty()	true			
enqueue(5)		5		
enqueue(3)		3	5	
dequeue()	5	3		
enqueue(7)		7	3	
enqueue(3)		3	7	3
front()	3	3	7	3
empty()	false	3	7	3

操作	输出	队列（右侧为队头）					
enqueue(11)		11	3	7	3		
size()	4	11	3	7	3		
enqueue(6)		6	11	3	7	3	
empty()	false	6	11	3	7	3	
enqueue(7)		7	6	11	3	7	3
dequeue()	3	7	6	11	3	7	
dequeue()	7	7	6	11	3		
front()	3	7	6	11	3		
size()	4	7	6	11	3		

实现

```
template <typename T>
class Queue: public List<T> {
public: //原有接口一概沿用
    void enqueue( T const & e )
    { insertAsLast( e ); } //入队
    T dequeue()
    { return remove( first() ); } //出队
    T & front()
    { return first()->data; } //队首
};
```

1 栈

2 栈与递归

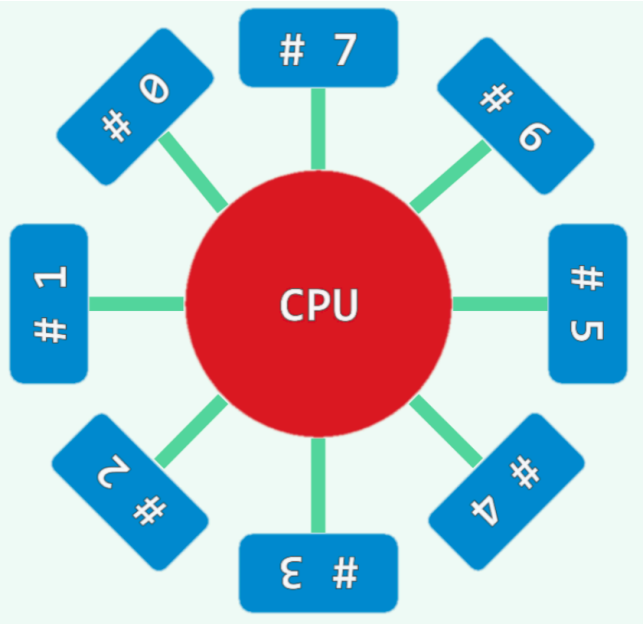
3 栈的典型应用

4 队列

5 队列应用

资源循环分配

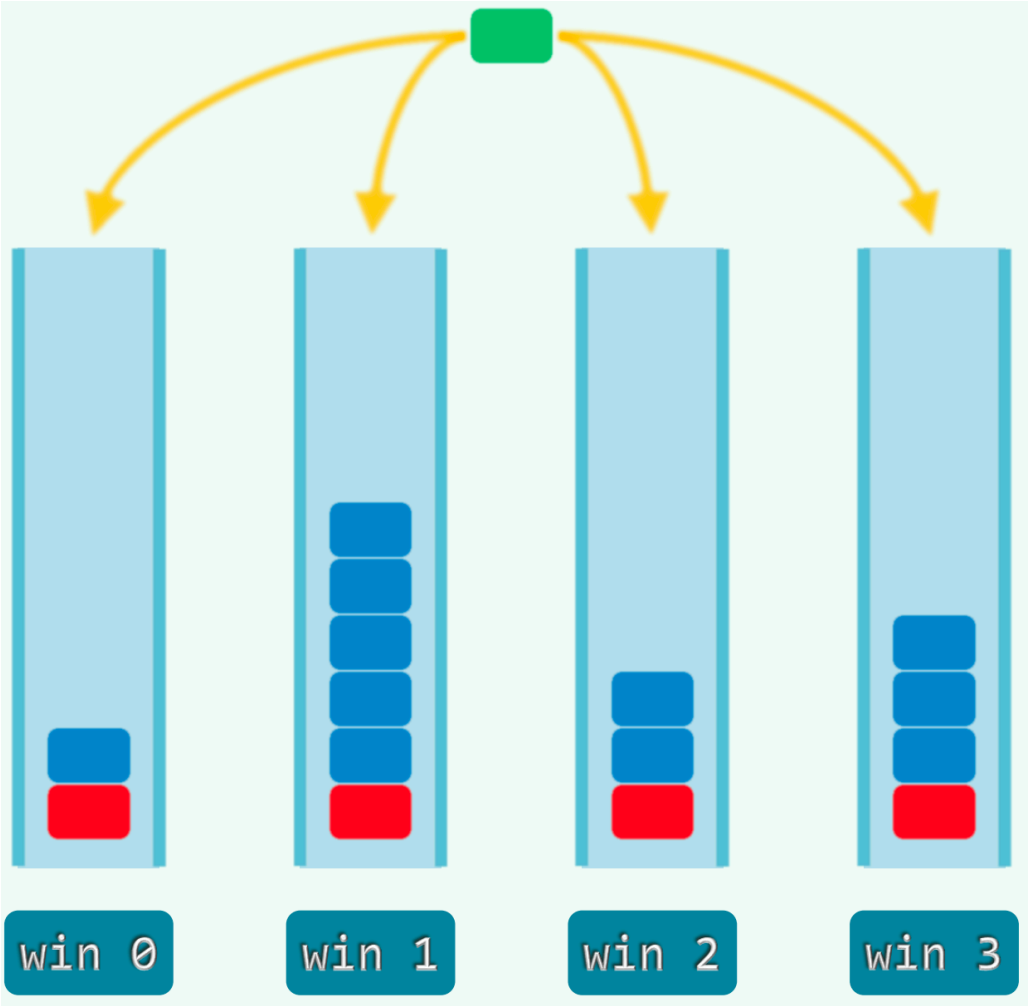
- 一组客户（client）共享同一资源时，如何兼顾公平与效率？
- 多个应用程序共享 CPU
- 实验室成员共享打印机



资源循环分配

```
RoundRobin { //循环分配器
    //共享资源的所有客户组成队列
    Queue Q( clients );
    //在服务关闭之前，反复地
    while ( ! ServiceClosed() ) {
        //令队首的客户出队，并
        e = Q.dequeue();
        //接受服务，然后重新入队
        serve( e );
        Q.enqueue( e );
    }
}
```

银行服务模拟：模型



银行服务模拟：模型

- 提供 n 个服务窗口
- 任一时刻，每个窗口至多接待一位顾客，其他顾客排队等候
- 顾客到达后，自动地选择和加入最短队列（的末尾）

```
struct Customer { //顾客类
    int window; //所属窗口（队列）
    unsigned int time; //服务时长
};
```

银行服务模拟：实现

```
void simulate( int nWin, int servTime ) {
    Queue<Customer>* windows =
        new Queue<Customer>[nWin];
    for ( int now = 0; now < servTime; now++ ) {
        if(rand() % (1 + nWin)) {
            Customer c; c.time = 1 + rand() % 98;
            c.window = bestWindow(windows, nWin);
            windows[c.window].enqueue(c);
        }
        for ( int i = 0; i < nWin; i++ )
            if ( ! windows[ i ].empty() )
                if(--windows[i].front().time<=0)
                    windows[ i ].dequeue();
    } //for
    delete [] windows; //释放所有队列
}
```

银行服务模拟：实现

```
int bestWindow(Queue<Customer> w, int nWin) {
    int minSize = w[0].size();
    int optWin = 0;
    for(int i=1; i<nWin; i++)
        if(minSize > w[i].size())
        {
            minSize = w[i].size();
            optWin = i;
        }
    return optWin;
}
```