04-A

栈与队列

栈接口与实现

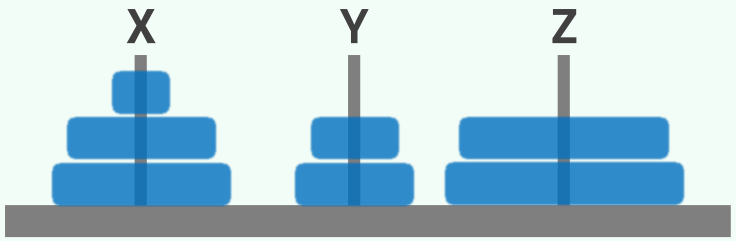
邓俊辉

deng@tsinghua.edu.cn

陛下用群臣，如积薪耳，后来者居上

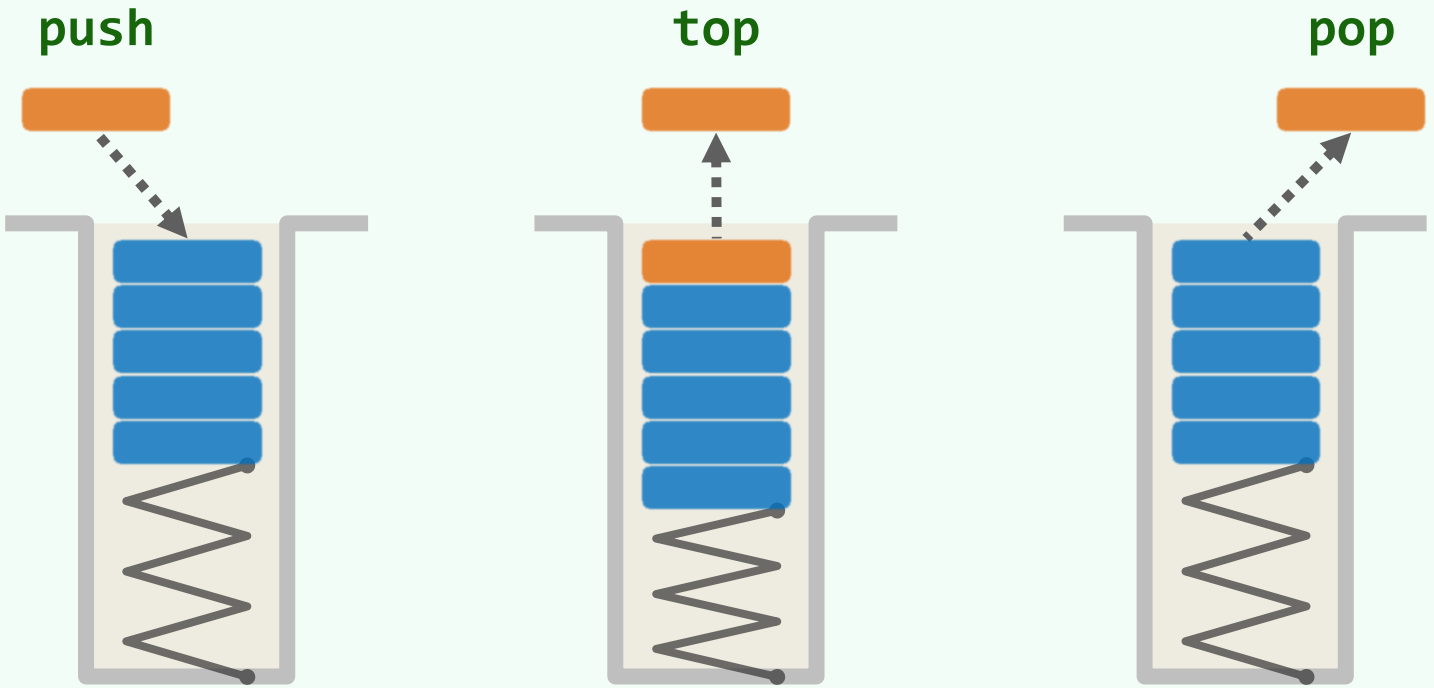
操作与接口

- 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



- 基本接口
  - size() / empty()
  - push() 入栈
  - pop() 出栈
  - top() 查顶
- 后进先出（LIFO）

  先进后出（FILO）
- 扩展接口：getMax()...



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

实现：既然属于序列的特例，故可直接基于向量或列表派生

```
template <typename T> class Stack: public Vector<T> {
public: //原有接口一概沿用
  void push( T const & e ) { insert( e ); } //入栈
  T pop() { return remove( size() - 1 ); } //出栈
  T & top() { return (*this)[ size() - 1 ]; } //取顶
}; //以向量首/末端为栈底/顶——颠倒过来呢?
```

- 确认：如此实现的栈各接口，均只需$O(1)$时间
- 课后：基于列表，派生定义栈模板类；你所实现的栈接口，效率如何?

04-B1

栈与队列

调用栈：原理与算法

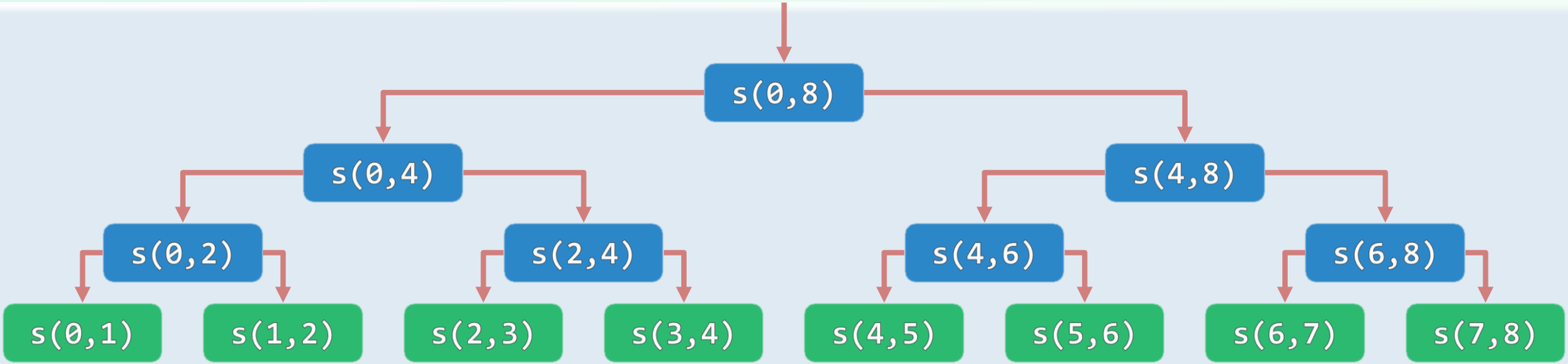
命运把我们的大脑当作一个容器，不停地把各种见解装进去、取出来，但总是现在的和最后的那个见解是可靠没错的

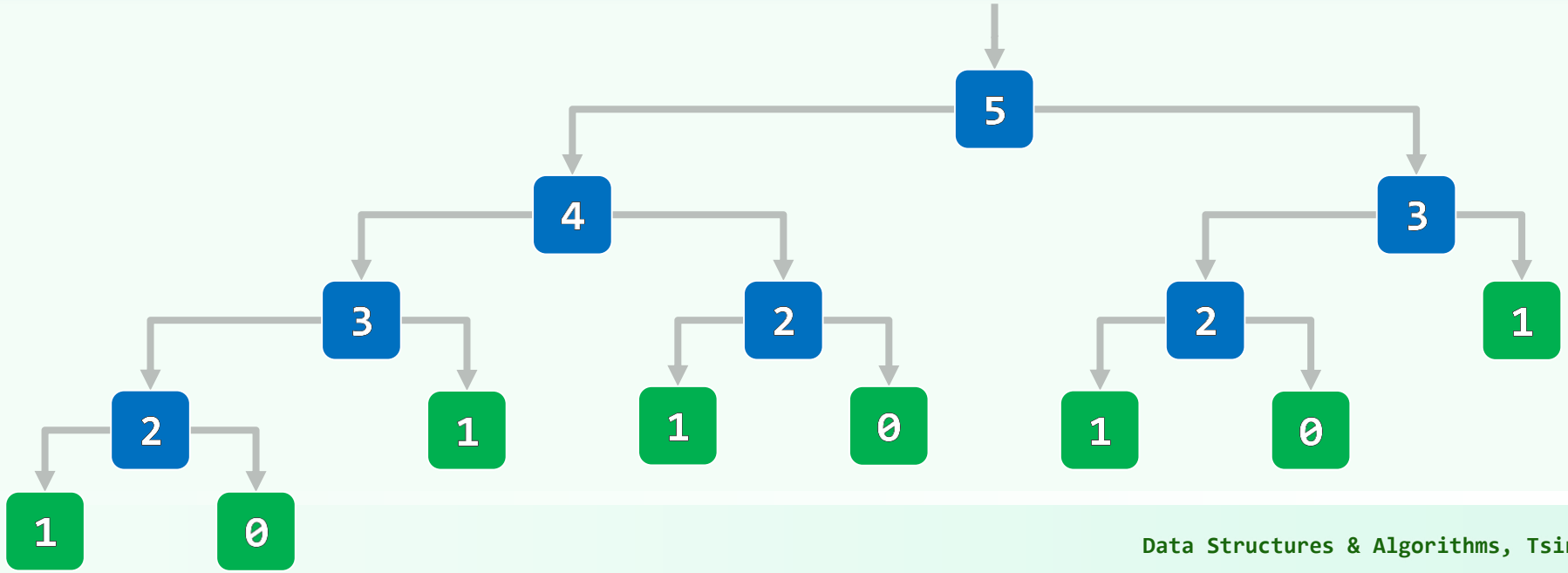
Yessiree. We do not doubt his word, an stack ourselfs into the bus like flapjacks.

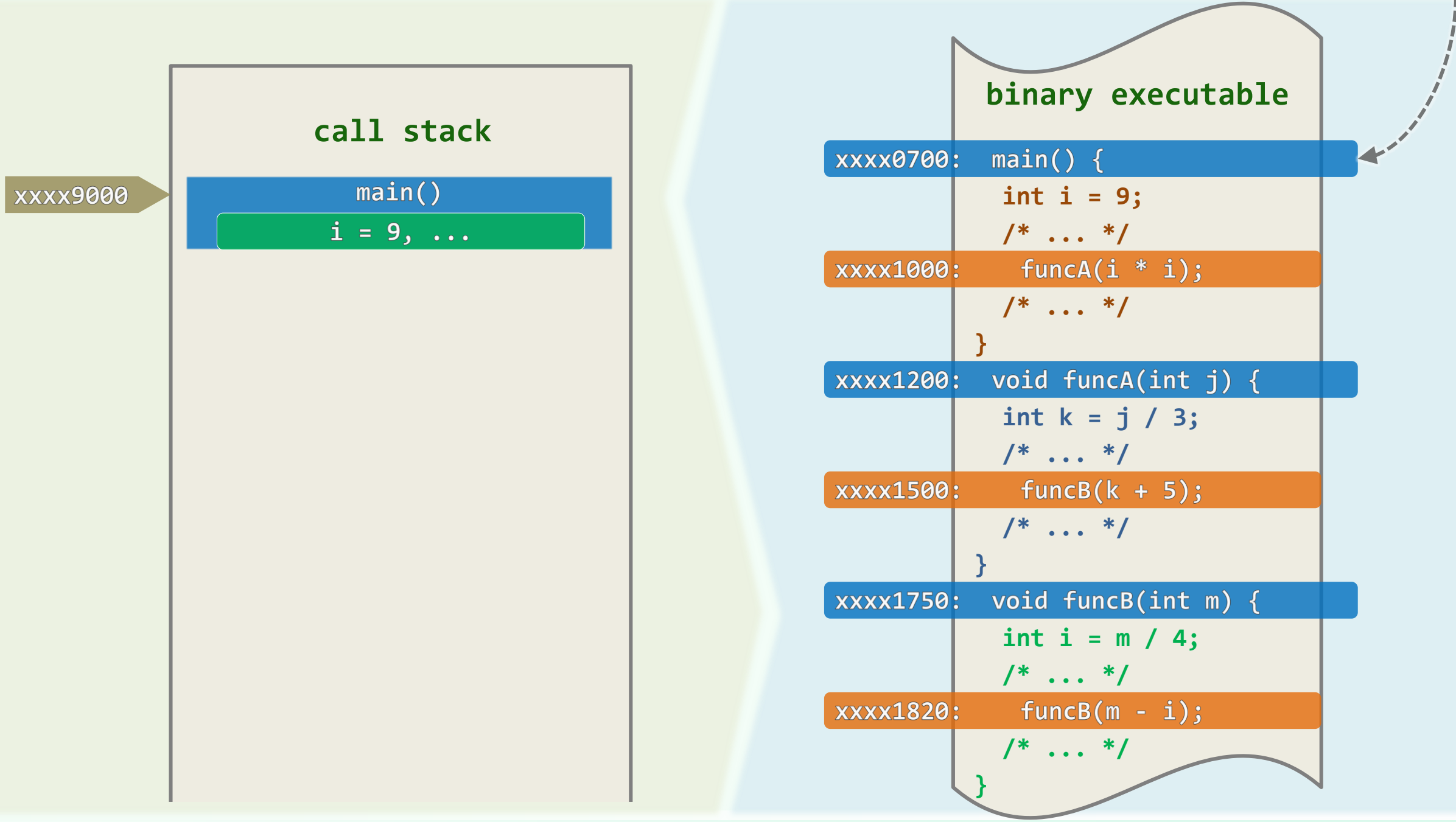
邓俊辉

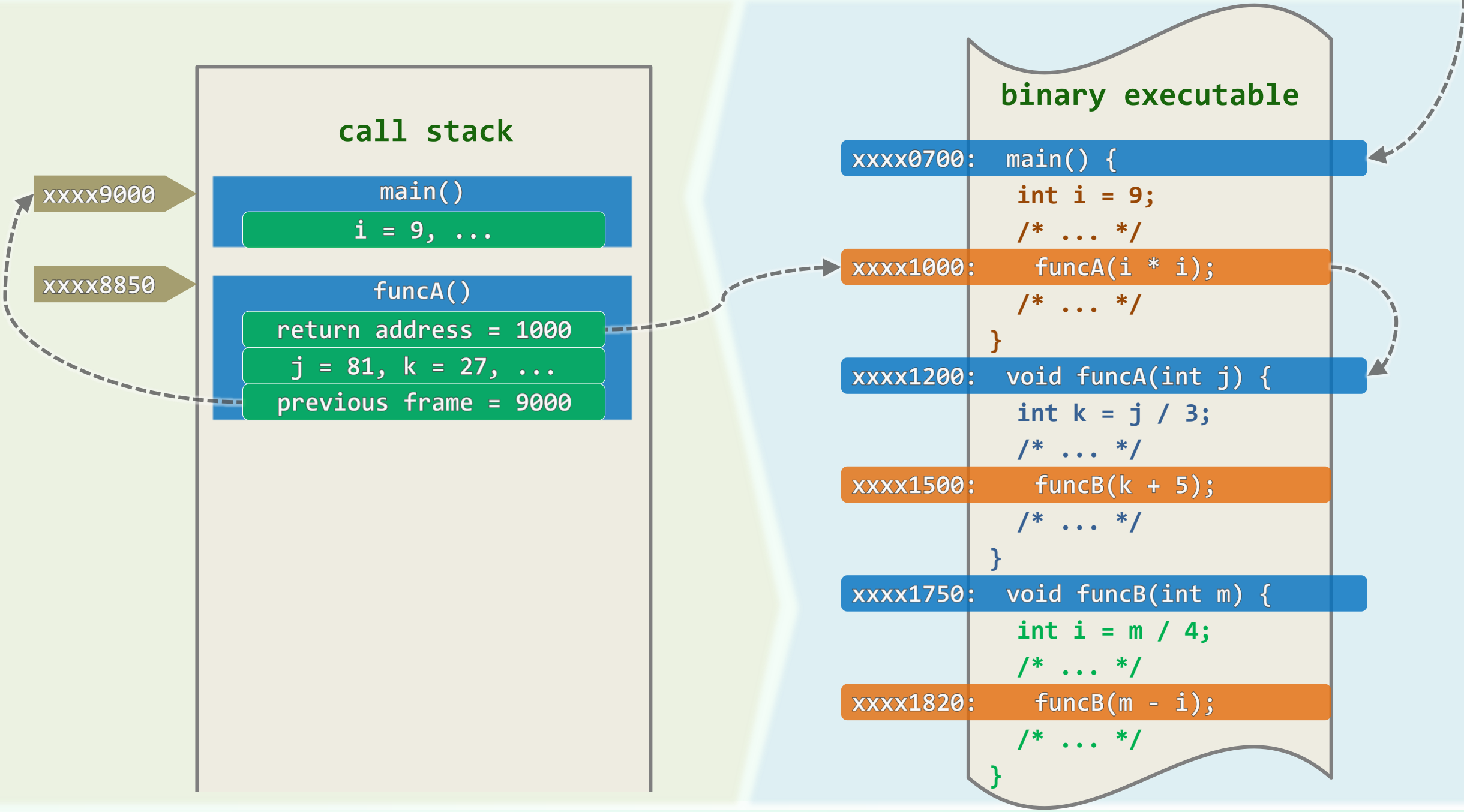
deng@tsinghua.edu.cn

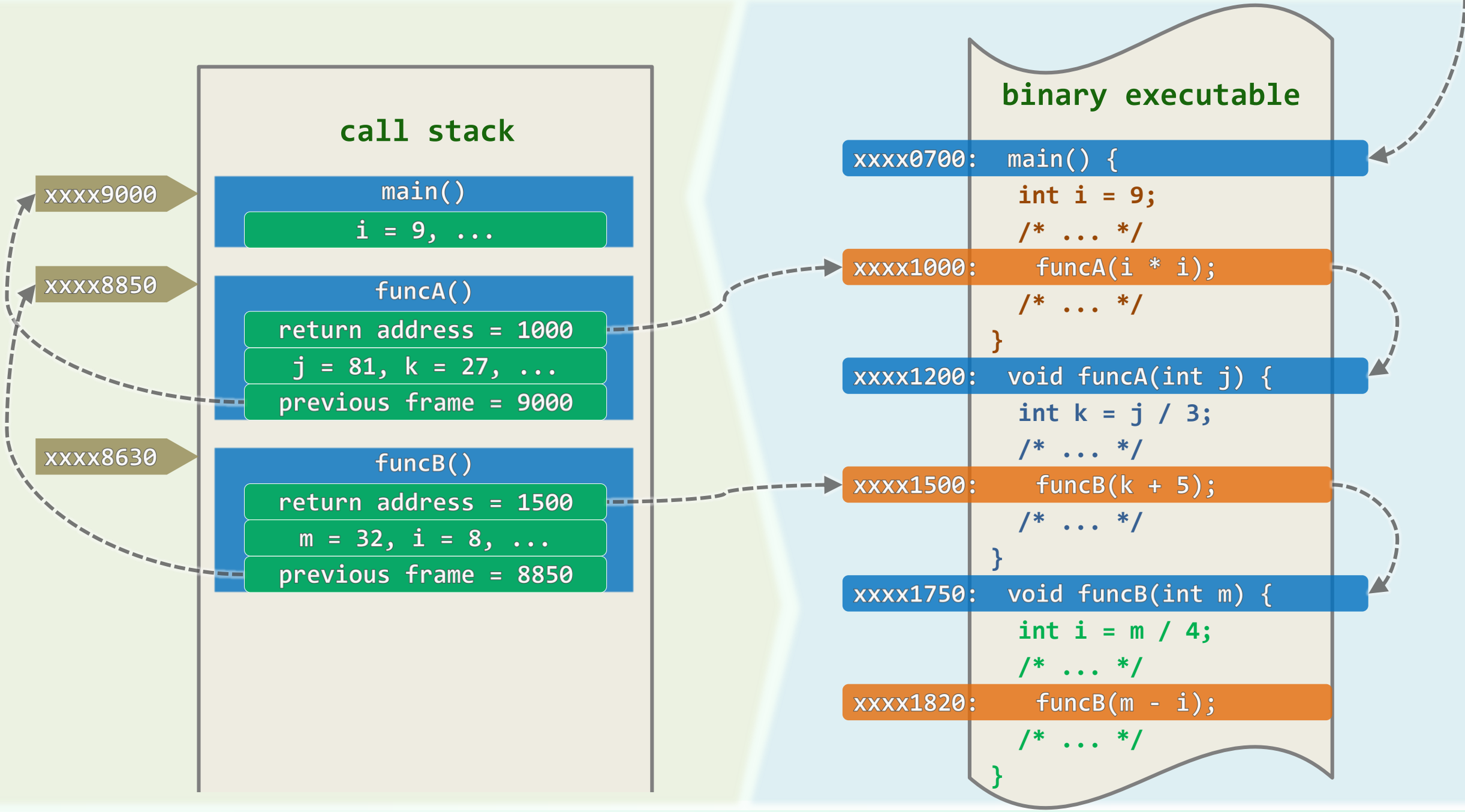
函数调用树：如何实现？Theseus的线团 + 粉笔

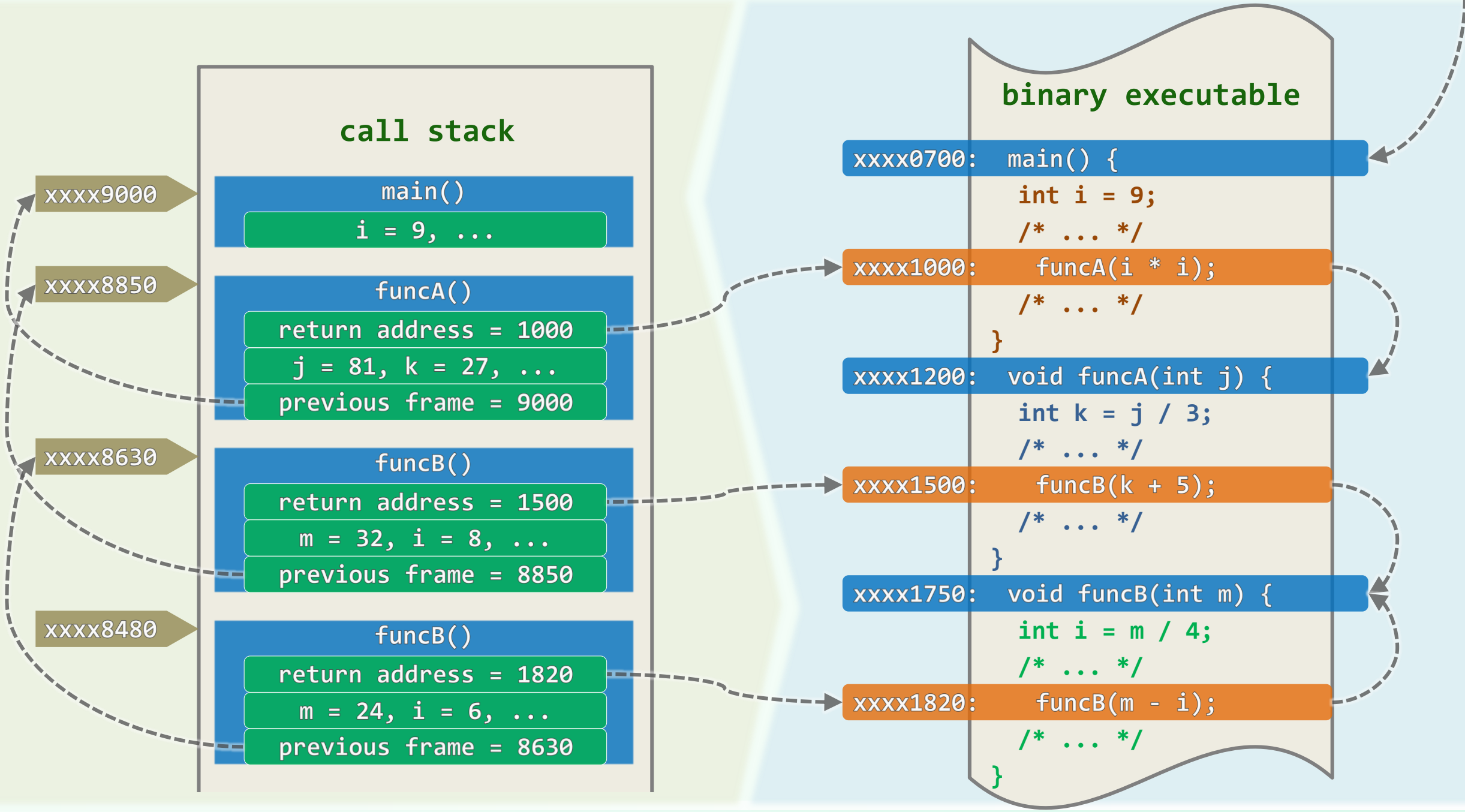





call stack
xxxx9000
main()
i = 9, ...
binary executable
xxxx0700: main() {
int i = 9;
/* ... */
xxxx1000: funcA(i * i);
/* ... */
}
xxxx1200: void funcA(int j) {
int k = j / 3;
/* ... */
xxxx1500: funcB(k + 5);
/* ... */
}
xxxx1750: void funcB(int m) {
int i = m / 4;
/* ... */
xxxx1820: funcB(m - i);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


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);
/* ... */
}


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);
/* ... */
}


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);
/* ... */
}

04-B2

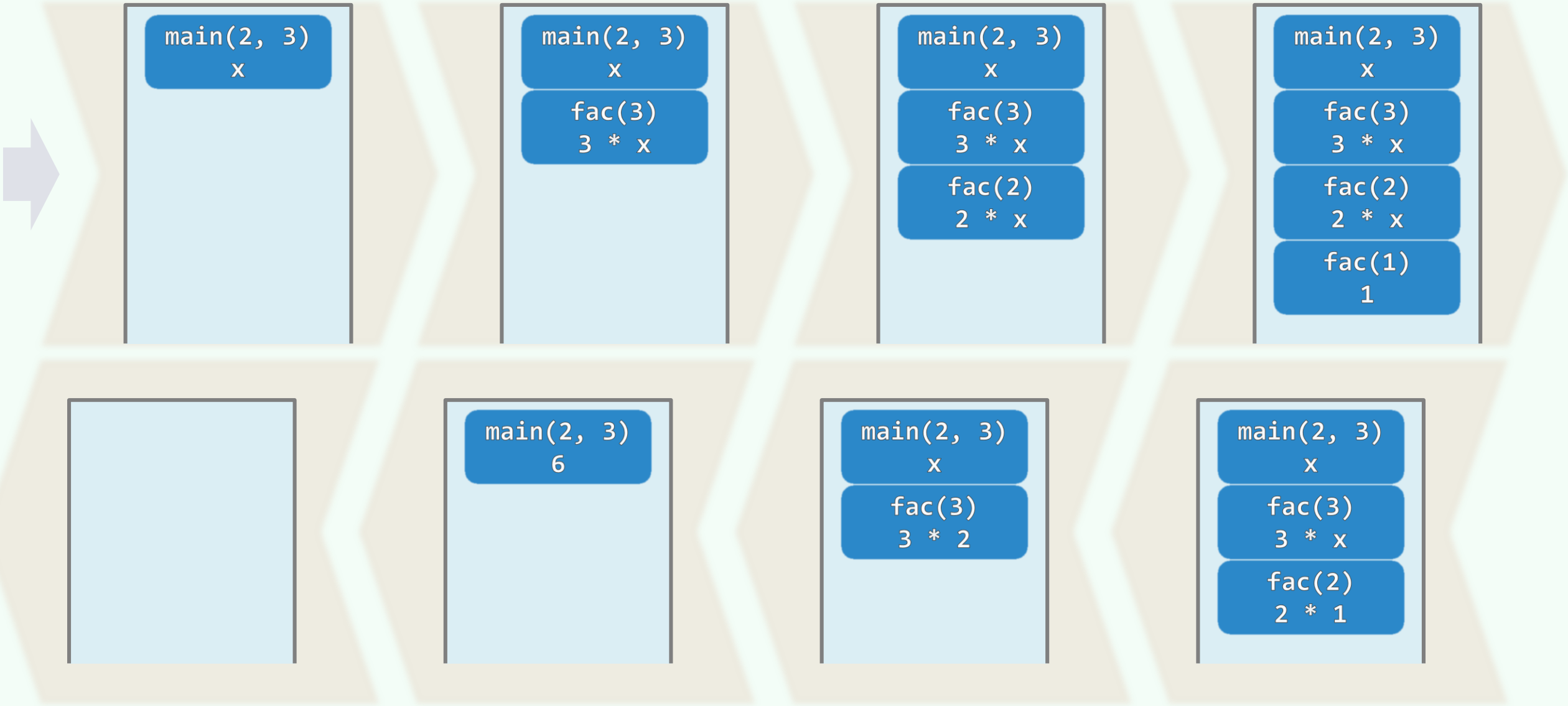
栈与队列

调用栈：深度与空间

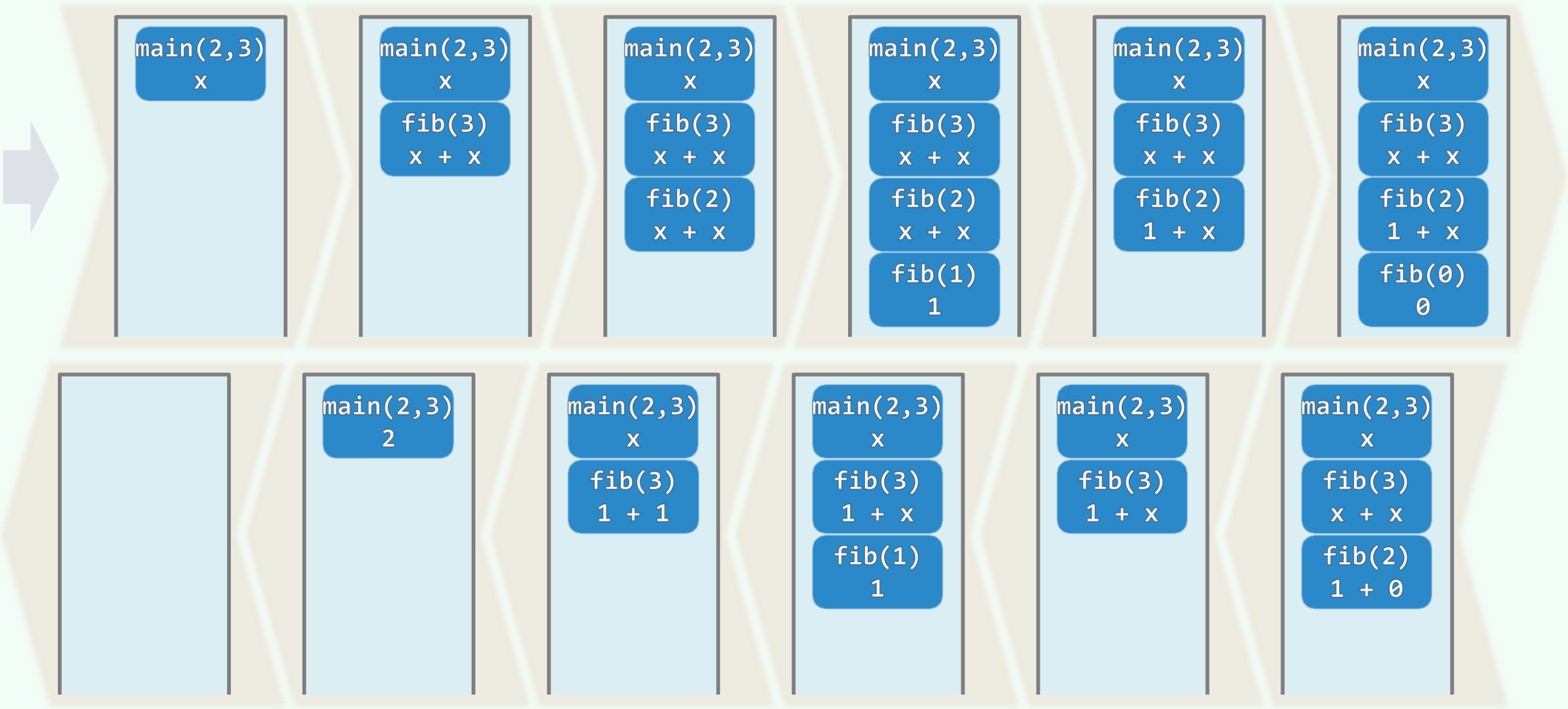
邓俊辉

deng@tsinghua.edu.cn

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



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



空间复杂度

```
❖ hailstone(int n) {
      if ( 1 < n )
         n % 2  ?  odd( n ) : even( n );
  }
❖ even( int n ) { hailstone( n / 2 ); }
  odd( int n ) { hailstone( 3*n + 1 ); }
❖ main( int argc, char* argv[] )
  { hailstone( atoi( argv[1] ) ); }
```

❖ 可见，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

04-B3

栈与队列

调用栈：消除递归

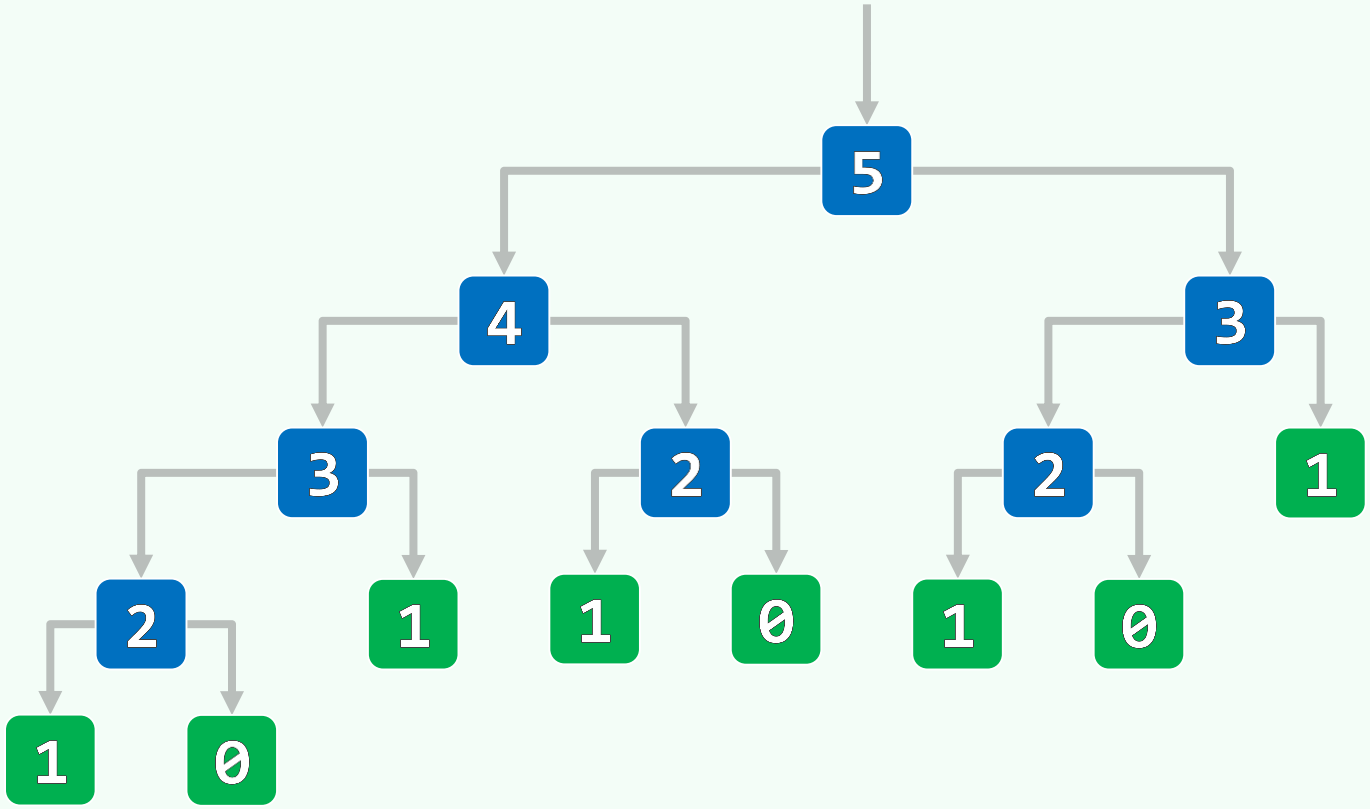
邓俊辉

deng@tsinghua.edu.cn

动机 + 方法

- ❖ 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
- ❖ 为隐式地维护调用栈

  需花费额外的时间、空间
- ❖ 为节省空间，可
  - 显式地维护调用栈
  - 将递归算法改写为迭代版本...



实例

- 通常，消除递归只是在常数意义上优化空间
- 但也可能有实质改进

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

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

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

栈与队列

调用栈：尾递归

04-B4

《星期评论》问我“女子解放从那里做起？”我的答案是：
“女子解放当从女子解放做起。此外更无别法。”

邓俊辉
deng@tsinghua.edu.cn

定义

- 在递归实例中，作为最后一步的递归调用
- fac(n) {

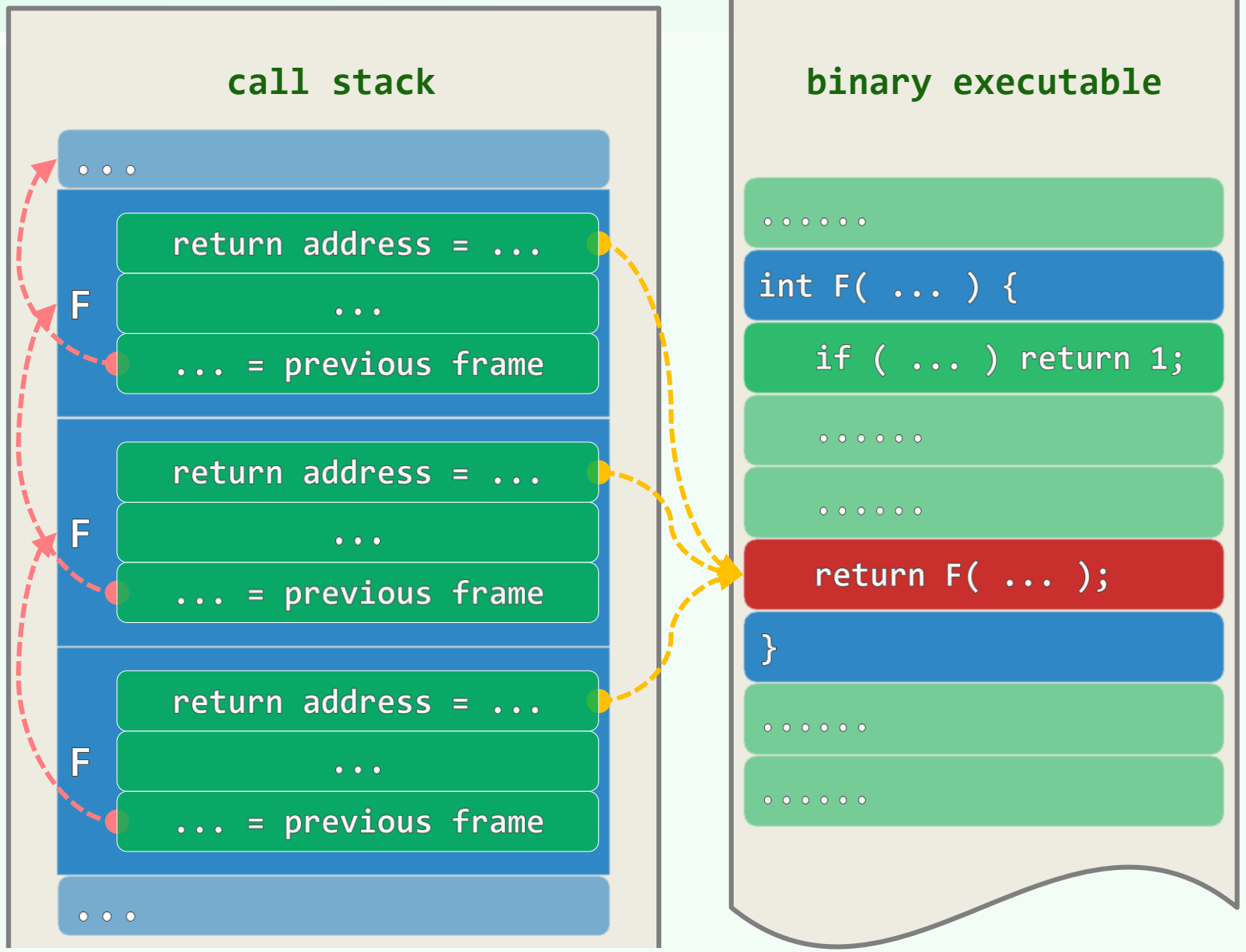
```
    if (1 > n) return 1; //base
    return n * fac( n-1 ); //tail recursion
}
```

- 严格地讲，上例在递归返回前的最后一步是乘法

  但这不是本质问题

性质

- 系最简单的递归模式
- 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



消除

```
fac(n) { //尾递归
   if (1 > n) return 1;
   return n * fac( n-1 );
} //O(n)时间 + O(n)空间
```

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

```
fac(n) { //简捷
   int f = 1;
   while (1 < n)
      f *= n--;
   return f;
} //O(n)时间 + O(1)空间
```

栈与队列

进制转换

04-C

Hickory, Dickory, Dock

The mouse ran up the clock

wan, twan, tetbera, metbera, pimp,

setbera, letbera, bovera, dovera, dick,

wanadick, twanadick, tetberadick, metberadick, pimpdick,

setberadick, letberadick, boveradick, doveradick, bumfit,

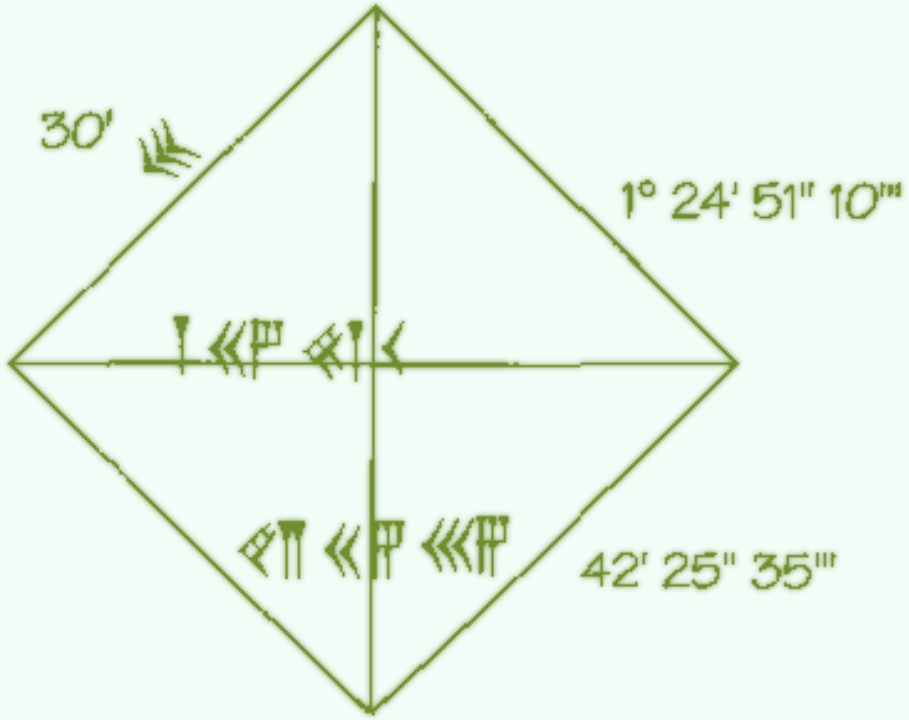
wanabumfit, ...

邓俊辉

deng@tsinghua.edu.cn

进制转换

- 给定任一10进制非负整数，将其转换为λ进制表示形式
  - $12345_{(10)} = 30071_{(8)}$
  - printf("%d | %I64d | %b | %o | %x" , n);
- 巴比伦楔形文字 (Babylonic cuneiform) 中的60进制...
- 1°24'51''10'''
  = 1 + 24/60 + 51/60^2 + 10/60^3
  = 1.41421296296296... //正方形的对角线
- 误差 ~ |1°24'51''10''' - sqrt(2)|
  < 0.000,000,6 = 0.6 x 10^{-6} //即便边长为1km，误差亦不足1mm

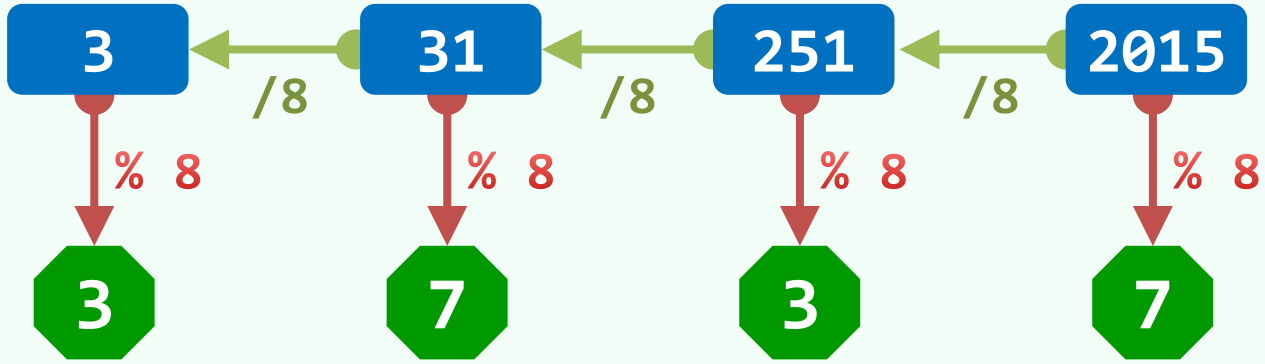


短除法：整商 + 余数

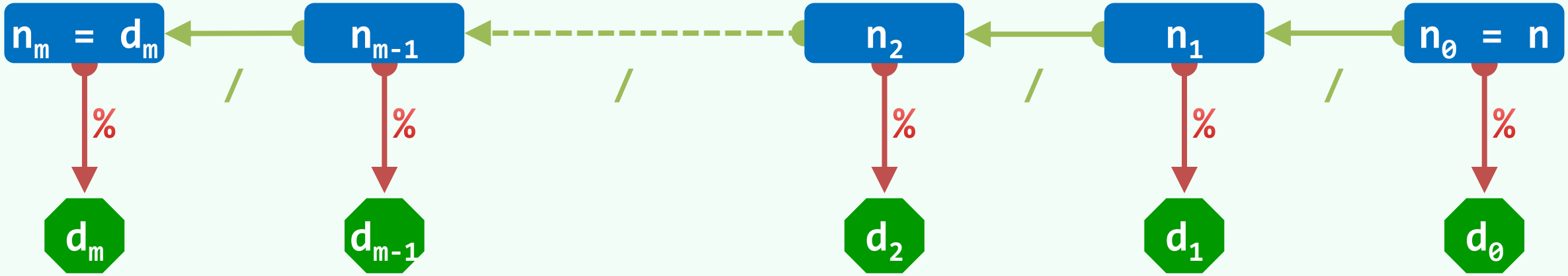
❖ 设：$n = (d_m \dots d_2 d_1 d_0)_\lambda = d_m \cdot \lambda^m + \cdots + d_2 \cdot \lambda^2 + d_1 \cdot \lambda^1 + d_0 \cdot \lambda^0$

令：$n_i = (d_m \dots d_{i+2} d_{i+1} d_i)_\lambda$

❖ 则有：$n_{i+1} = n_i \,/\, \lambda$ 和 $d_i = n_i \,\%\, \lambda$



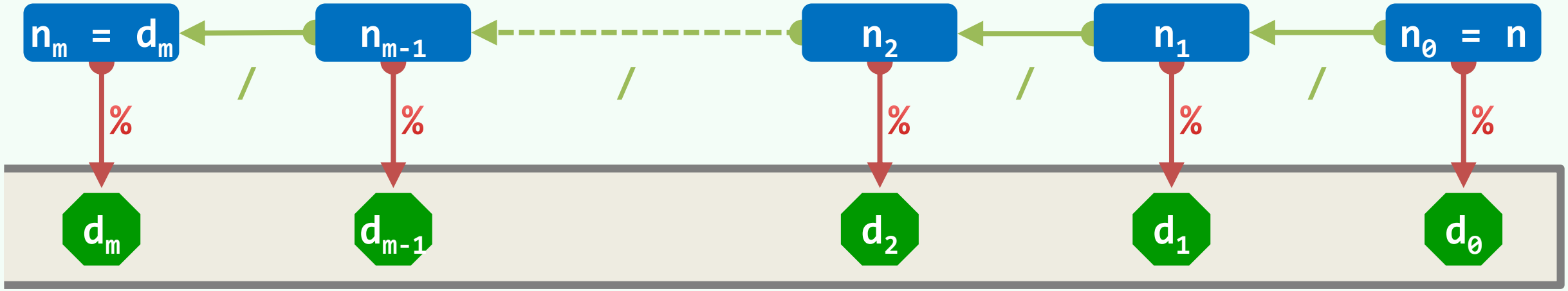
❖ 如此对λ反复整除、留余，即可自低而高得出λ进制的各位



难点 + 算法

❖ 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

实现

```
void convert( Stack<char> & S, __int64 n, int base ) {
   char digit[] = "0123456789ABCDEF"; //数位符号，如有必要可相应扩充
   while ( n > 0 ) //由低到高，逐一计算出新进制下的各数位
      {  S.push( digit[ n % base ] ); n /= base;  } //余数入栈，n更新为除商
} //新进制下由高到低的各数位，自顶而下保存于栈S中
main() {
  Stack<char> S; convert( S, n, base ); //用栈记录转换得到的各数位
  while ( ! S.empty() ) printf( "%c", S.pop() ); //逆序输出
}
```

04-D

栈与队列

括号匹配

邓俊辉

deng@tsinghua.edu.cn

实例

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

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

❖ 观察：除了各种括号，其余符号均可暂时忽略

`( [ ] [ ] ) [ ] [ ] )` //失配

`( [ ] [ ] [ ] [ ] )` //匹配

❖ 从简单入手，先来考查只有一种括号的情况...

尝试：由外而内

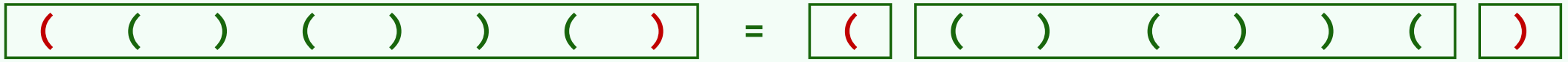
0）平凡：　无括号的表达式是匹配的

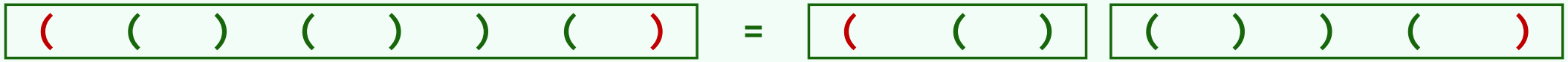
1）减治？　E 匹配，仅当 (E) 匹配

2）分治？　E 和 F 均匹配，仅当 E F 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，1）和2）均为**充分性**，比如反例：

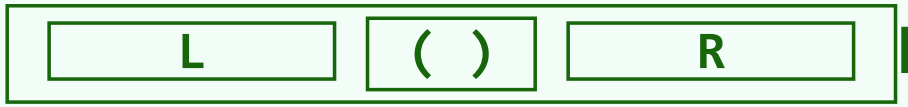




❖ 而为使问题有效简化，必须发现并借助**必要性**

构思：由内而外

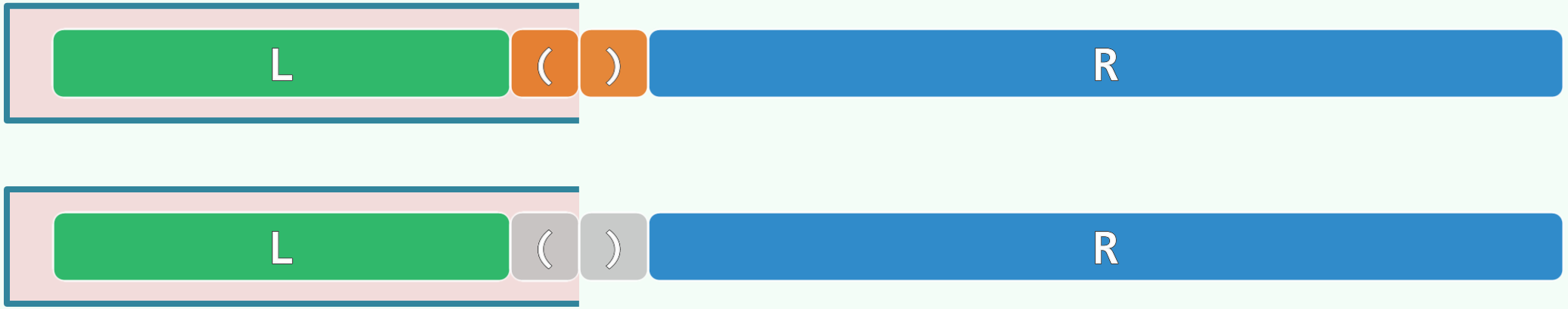
❖ 颠倒以上思路：消去一对紧邻的左右括号，不影响全局的匹配判断

亦即： 

匹配，仅当 

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　　//实际上只需记录左括号

反复迭代：凡遇"("，则进栈；凡遇")"，则出栈

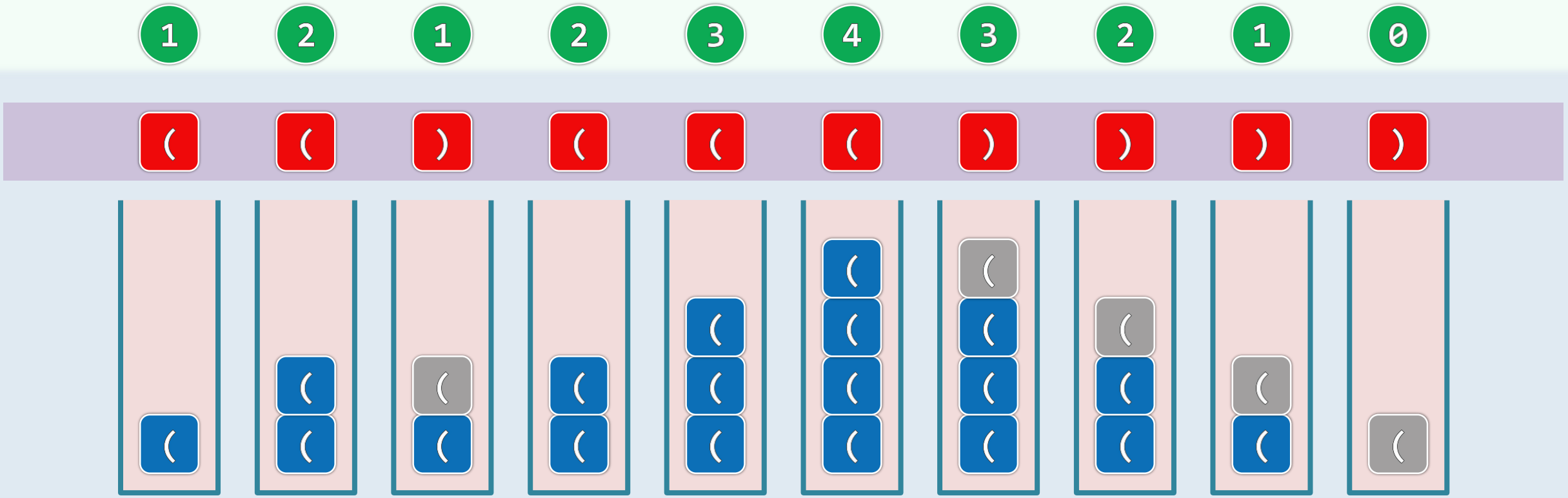
实现

```
bool paren( const char exp[], Rank lo, Rank hi ) { //exp[lo, hi)
   Stack<char> S; //使用栈记录已发现但尚未匹配的左括号
   for ( Rank i = lo; i < hi; i++ ) //逐一检查当前字符
      if ( '(' == exp[i] ) S.push( exp[i] ); //遇左括号：则进栈
      else if ( ! S.empty() ) S.pop(); //遇右括号：若栈非空，则弹出对应的左括号
      else return false; //否则（遇右括号时栈已空），必不匹配
   return S.empty(); //最终栈空，当且仅当匹配
}
```

实例：一种括号

❖ 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）

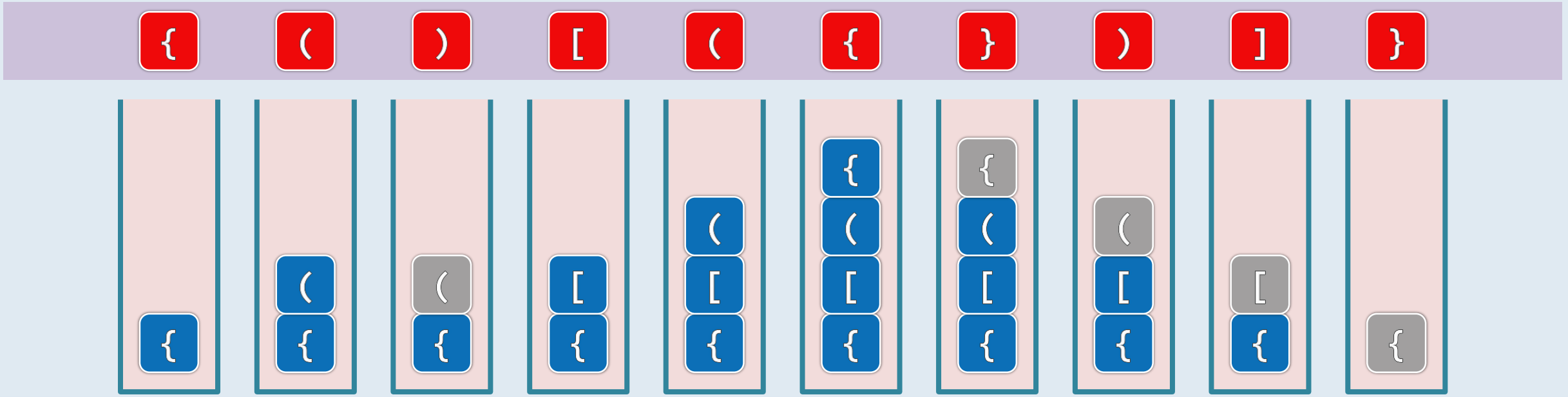


拓展：多类括号

❖ 以上思路及算法，可否推广至多种括号并存的情况？反例：[( ])

❖ 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>，<h1>|</h1>，|， | ， |，...



04-E

栈与队列

栈混洗

邓俊辉

deng@tsinghua.edu.cn

Stack Permutation

❖ 考查栈 $\mathcal{A} = \langle\, a_1,\ a_2,\ a_3,\ \ldots,\ a_n \,]$

$\mathcal{B} = \mathcal{S} = \varnothing$

❖ 只允许

- 将A的顶元素弹出并压入S，或
- 将S的顶元素弹出并压入B

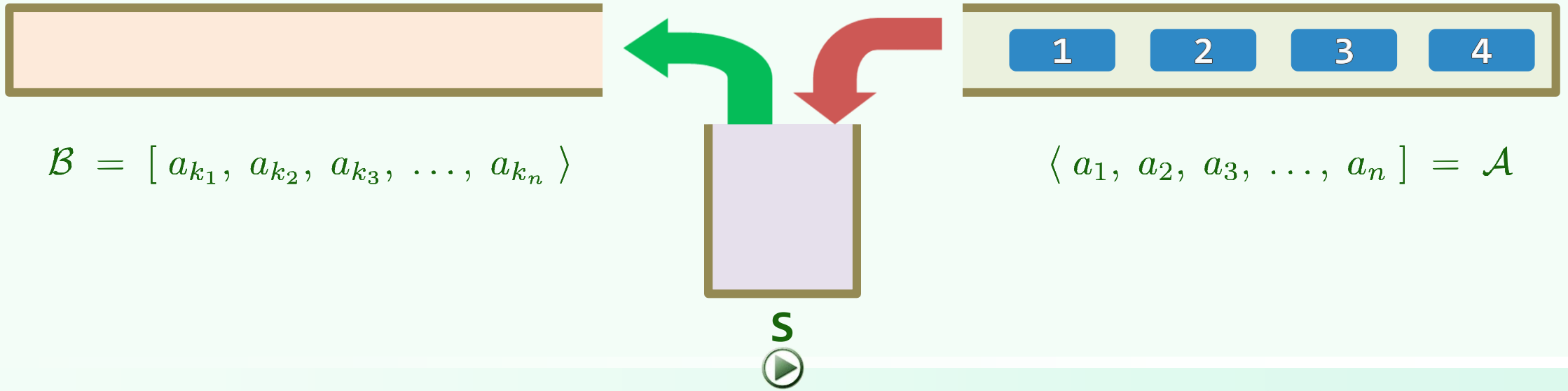
❖ 亦即 $\mathcal{S}.\text{push}(\mathcal{A}.\text{pop}())$

$\mathcal{B}.\text{push}(\mathcal{S}.\text{pop}())$

❖ 若经一系列以上操作后，A中元素全部转入B中

$\mathcal{B} = [\, a_{k_1},\ a_{k_2},\ a_{k_3},\ \ldots,\ a_{k_n} \,\rangle$

则称为A的一个**栈混洗**



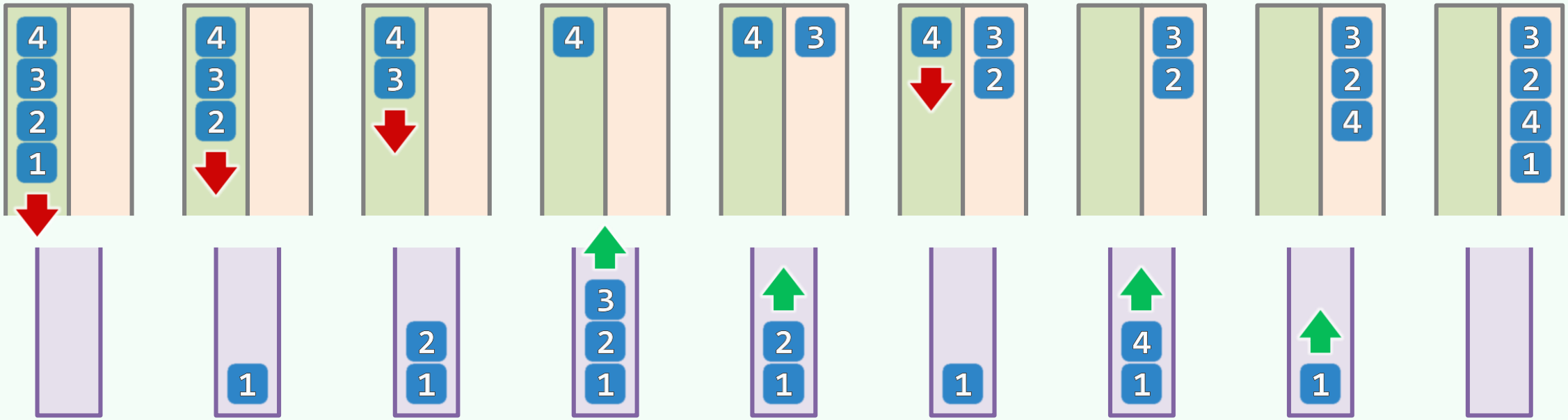
$\mathcal{B} = [\, a_{k_1},\ a_{k_2},\ a_{k_3},\ \ldots,\ a_{k_n} \,\rangle$

$\langle\, a_1,\ a_2,\ a_3,\ \ldots,\ a_n \,] = \mathcal{A}$

计数：SP(n)

❖ 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 > ...

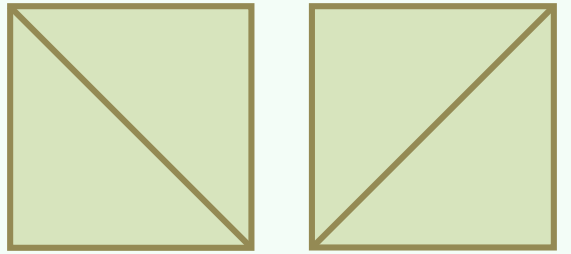
❖ 一般地，对于长度为n的序列，混洗总数SP(n) = ？



❖ 显然，SP(n) <= n!；更准确地呢?

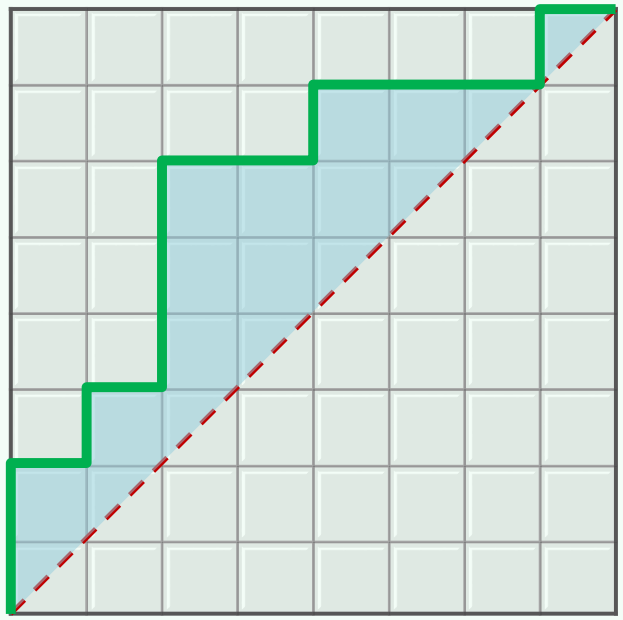
计数：catalan(n)

- $SP(1) = 1$
- **考查S再度变空（A首元素从S中弹出）的时刻，无非n种情况：**



$$SP(n) = \sum_{k=1}^{n} SP(k-1) \cdot SP(n-k)$$

$$= catalan(n) = \frac{(2n)!}{(n+1)! \cdot n!}$$

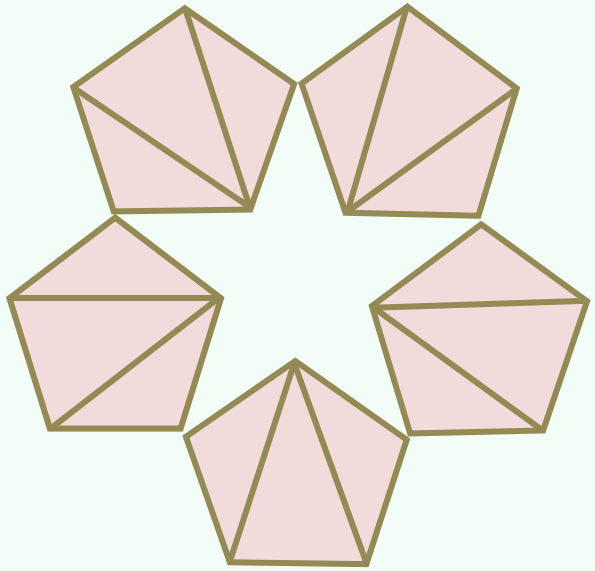


$SP(2) = 4!/3!/2! = 2$

$SP(3) = 6!/4!/3! = 5$

... ...

$SP(6) = 12!/7!/6! = 132$



计数：catalan(n)

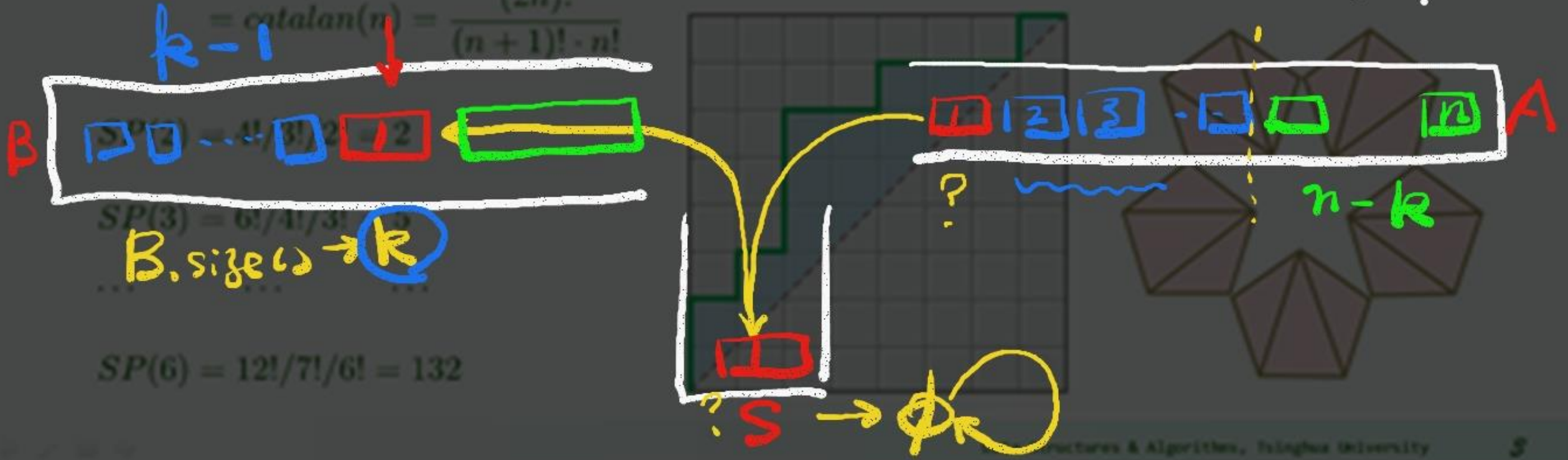
$$SP(n) = \sum_{k=1}^{n} SP(k-1) \cdot SP(n-k)$$

- $SP(1) = 1$
- 考查S再度变空（A首元素从S中弹出）的时刻，无非n种情况：

$$SP(n) = \sum_{k=1}^{n} SP(k-1) \cdot SP(n-k) = catalan(n) = \frac{(2n)!}{(n+1)! \cdot n!}$$

$$SP(3) = 6!/4!/3! = 30$$

$$SP(6) = 12!/7!/6! = 132$$



甄别：检测禁形

- 输入序列< 1, 2, 3, ..., n >的任一排列[p_1, p_2, p_3, ..., p_n >是否为栈混洗?
- 先考查简单情况：n = 3, A = < 1, 2, 3 >
  - 栈混洗共 6! / 4! / 3! = 5 种；全排列共 3! = 6 种 //少了一种...
- [3, 1, 2 > //为什么是它?
- 观察：任意三个元素能否按某相对次序出现于混洗中，与其它元素无关 //故可推而广之...
- 禁形：对任何1 ≤ i < j < k ≤ n, [..., k, ..., i, ..., j, ... > 必非栈混洗
- 反过来，不存在“312”模式的序列，一定是栈混洗吗?

甄别：直接模拟

❖ 充要性：　　A permutation is a stack permutation iff

(Knuth, 1968)　　it does NOT involve the permutation 312　　//习题[4-3]

❖ 如此，可得一个$O(n^3)$的甄别算法　　//进一步地...

❖ [p_1, p_2, p_3, ..., p_n]是[1, 2, 3, ..., n]的栈混洗，当且仅当

对于任意i < j，不含模式[..., j+1, ..., i, ..., j, ...]

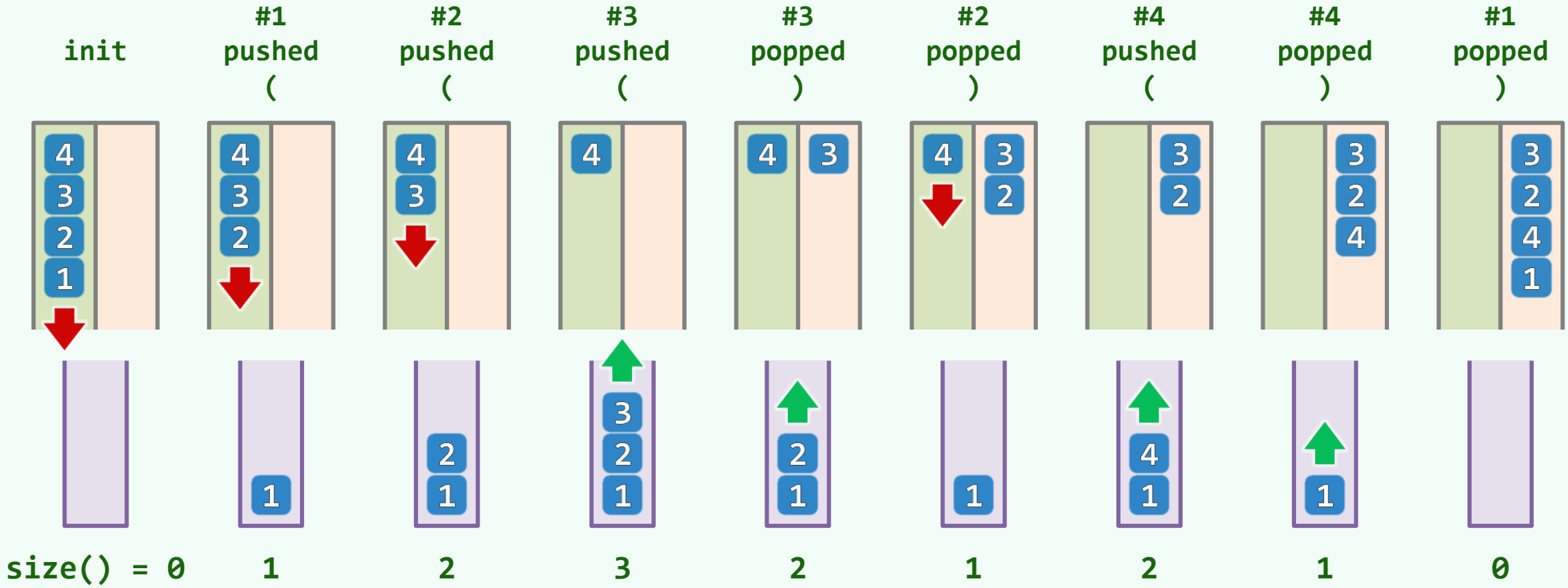
❖ 如此，可得一个$O(n^2)$的甄别算法　　//再进一步地...

❖ $O(n)$算法：　直接借助栈A、B和S，模拟混洗过程　　//为何可行?

每次S.pop()之前，检测S是否已空；或需弹出的元素在S中，却非顶元素

括号匹配

❖ 观察：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

04-F1

栈与队列

中缀表达式求值：问题与构思

邓俊辉

deng@tsinghua.edu.cn

知实而不知名，知名而不知实，皆不知也

应用

❖ 给定语法正确的算术表达式S，计算与之对应的数值

❖ UNIX: $ echo $((0 + (1 + 23) / 4 * 5 * 67 - 8 + 9))

❖ DOS: \> set /a (!0 ^<^< (1 - 2 + 3 * 4)) - 5 * (6 ^| 7) / (8 ^^ 9)

❖ PS: GS> 0 1 23 add 4 div 5 mul 67 mul add 8 sub 9 add =

❖ Excel: = COS(0) + 1 - (2 - POWER((FACT(3) - 4), 5)) * 67 - 8 + 9

❖ Word: = NOT(0) + 12 + 34 * 56 + 7 + 89

❖ calc: 0 ! + 12 + 34 * 56 + 7 + 89 =

❖ calc: 0 ! + 1 - (2 - (3 ! - 4) y 5) * 67 - 8 + 9 =

减而治之

❖ 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

❖ 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
- $val(S_0) = v_0$

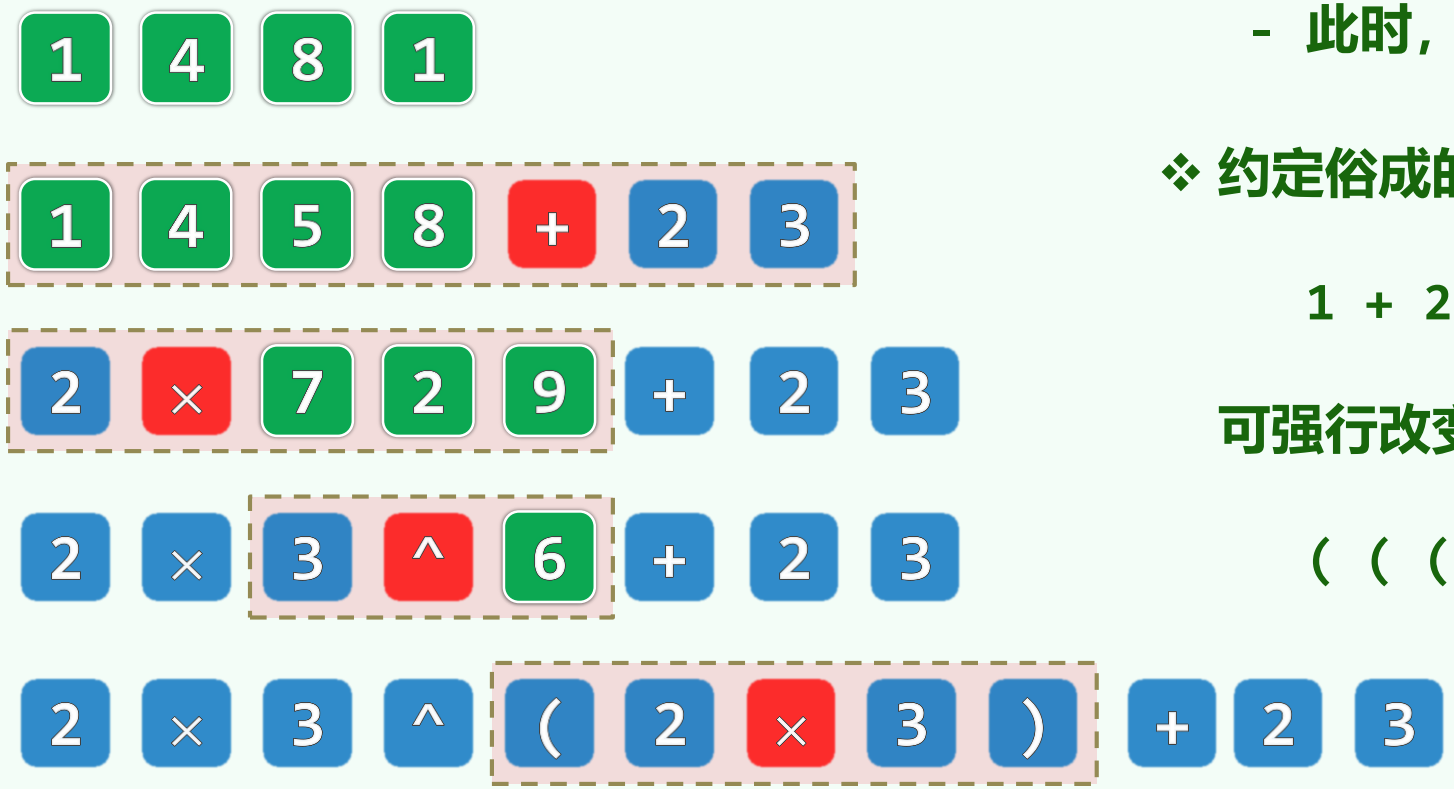
❖ 则有递推化简关系

$$val(S) = val(S_L + str(v_0) + S_R)$$

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) !

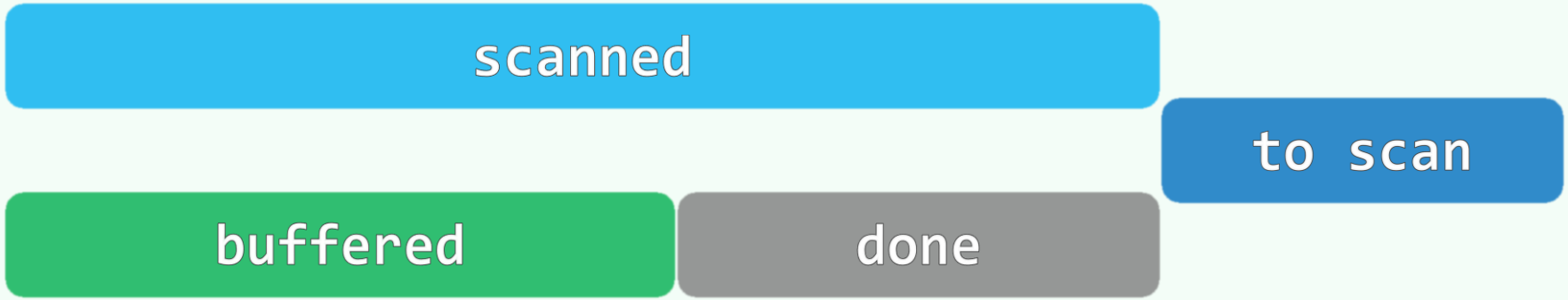
延迟缓冲

❖ 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...

求值算法 = 栈 + 线性扫描

- 自左向右扫描表达式

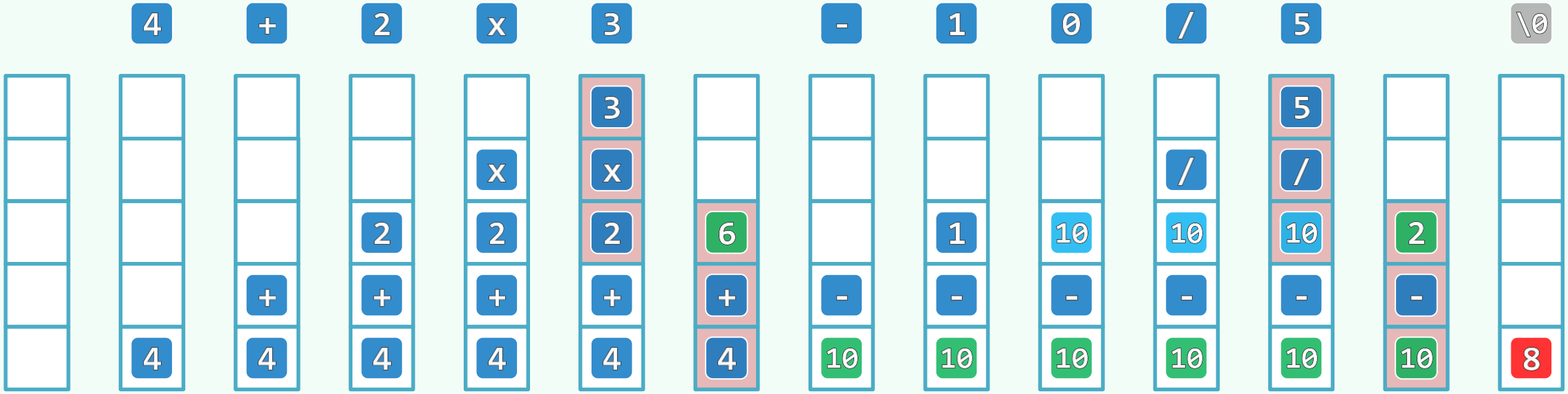
  用栈记录已扫描的部分，以及中间结果

- 栈内最终所剩的那个元素，即表达式之值

- 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



栈与队列

中缀表达式求值：算法

04-F2

邓俊辉

deng@tsinghua.edu.cn

主算法

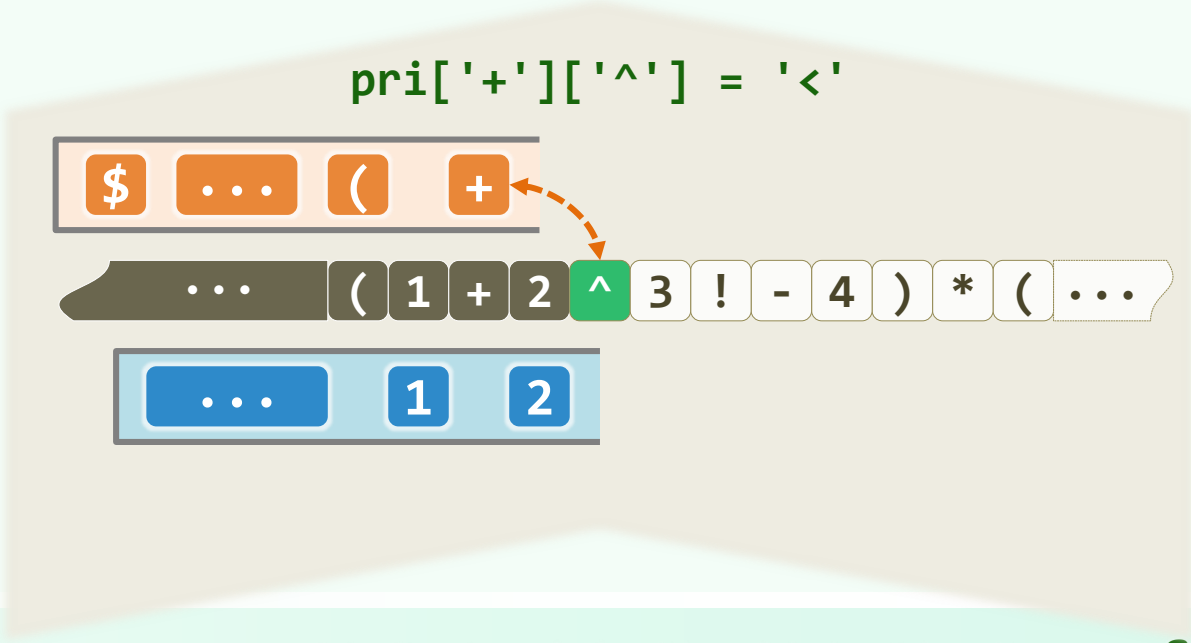
```
double evaluate( char* S, char* RPN ) { //S保证语法正确
   Stack<double> opnd; Stack<char> optr; //运算数栈、运算符栈
   optr.push('\0'); //哨兵
   while ( ! optr.empty() ) { //逐个处理各字符，直至运算符栈空
      if ( isdigit( *S ) ) //若为操作数（可能多位、小数），则
         readNumber( S, opnd ); //读入
      else //若为运算符，则视其与栈顶运算符之间优先级的高低
         switch( priority( optr.top(), *S ) ) { /* 分别处理 */ }
   } //while
   return opnd.pop(); //弹出并返回最后的计算结果
}
```

优先级表

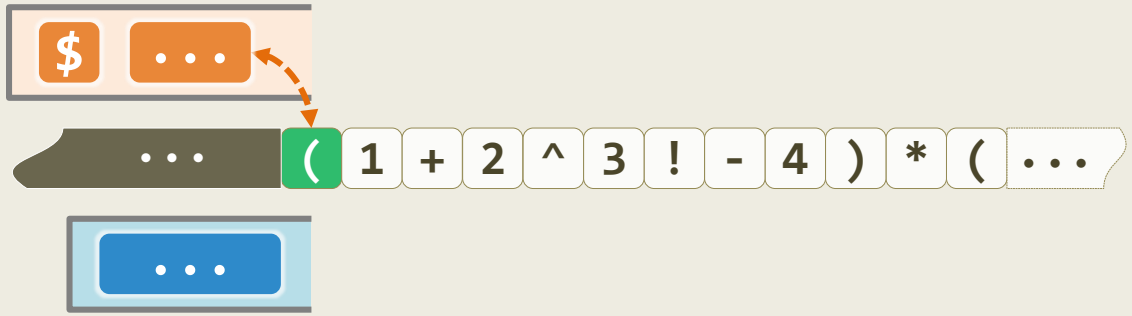
```
const char pri[N_OPTR][N_OPTR] = { //运算符优先等级 [栈顶][当前]
   /* --  + */  '>', '>', '<', '<', '<', '<', '<', '>', '>',
   /*  |  - */  '>', '>', '<', '<', '<', '<', '<', '>', '>',
   /* 栈  * */  '>', '>', '>', '>', '<', '<', '<', '>', '>',
   /* 顶  / */  '>', '>', '>', '>', '<', '<', '<', '>', '>',
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

'<': 静待时机: 算法

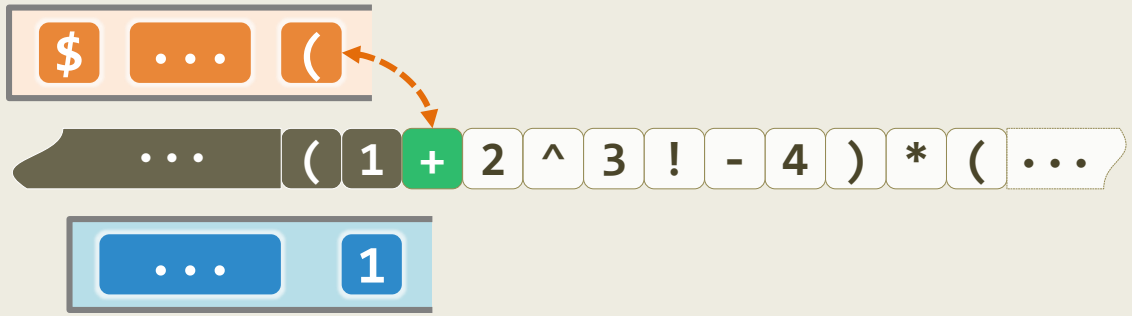
```
switch( priority( optr.top(), *S ) ) {
   case '<': //栈顶运算符优先级更低
      optr.push( *S ); S++; break; //计算推迟，当前运算符进栈
   case '=':
      /* ..... */
   case '>': {
      /* ..... */
      break;
   } //case '>'
} //switch
```



'<'：静待时机：实例

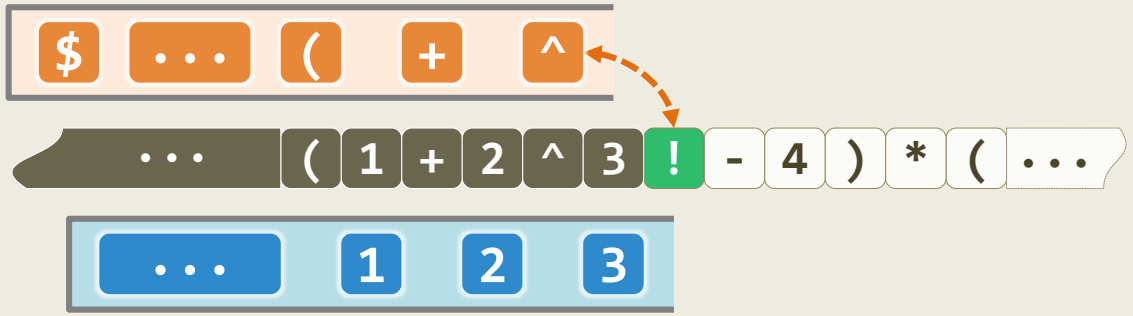


pri[ ]['('] = '<'

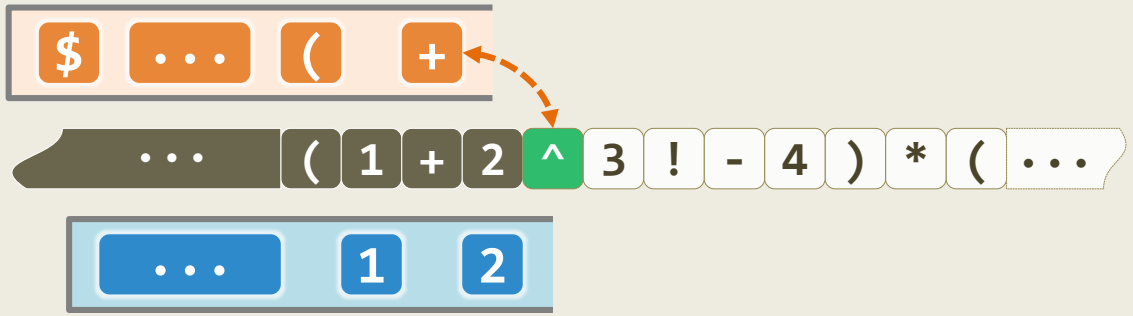


pri['('][+'] = '<'

pri['^']['!'] = '<'

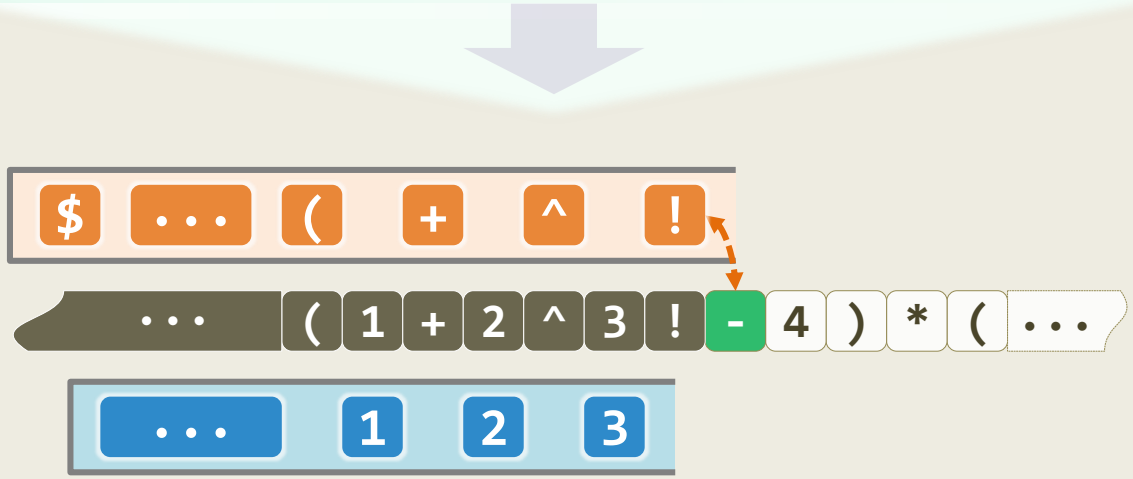


pri['+']['^'] = '<'



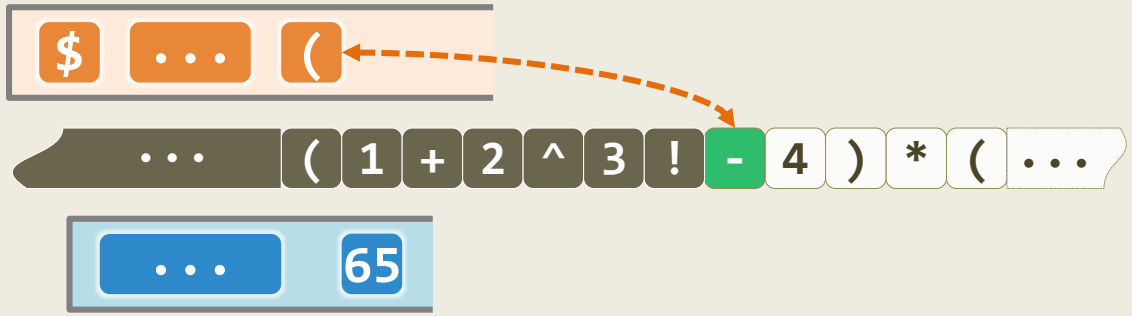
'>': 时机已到：算法

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

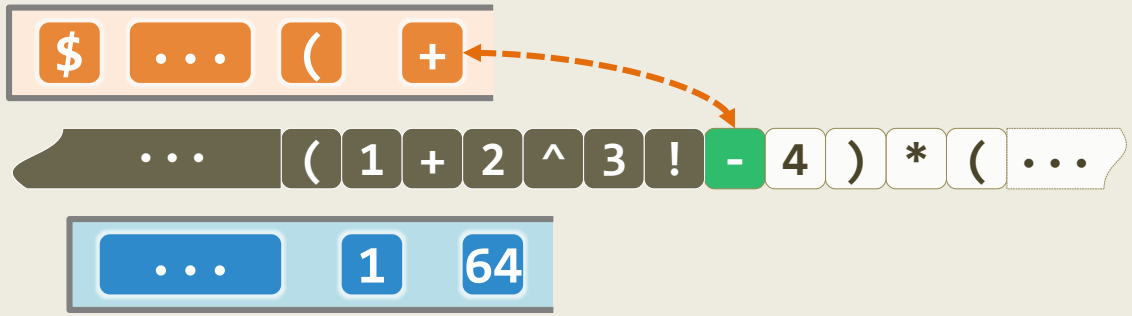


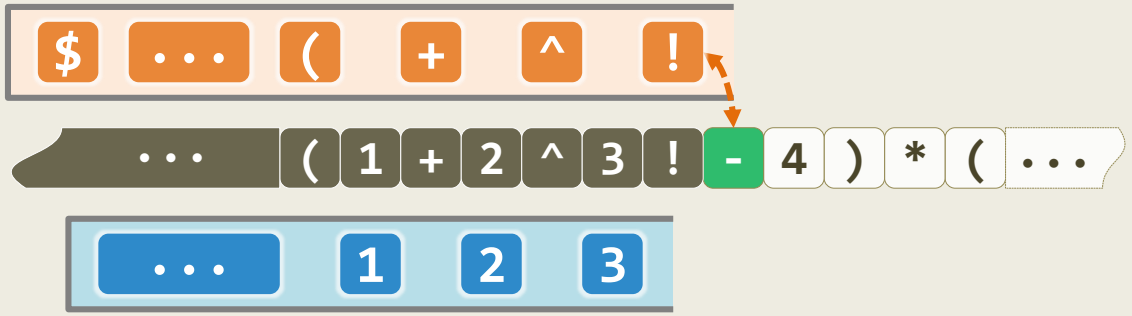
'>': 时机已到：实例

pri['('['-'] = '<'

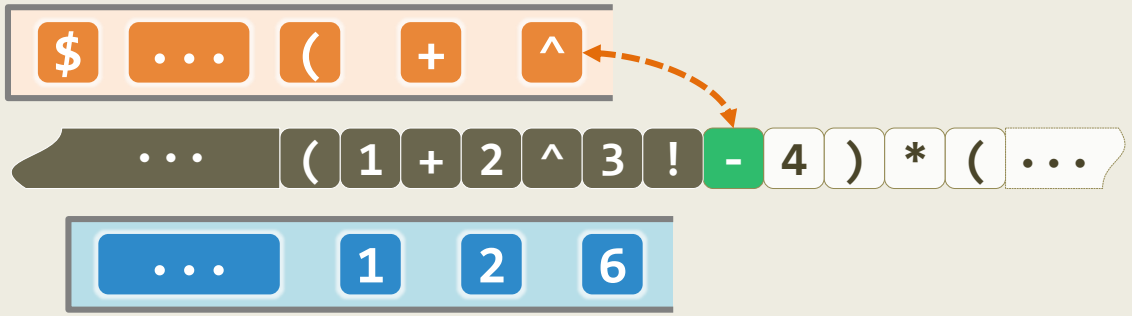


pri['+']['-'] = '>'





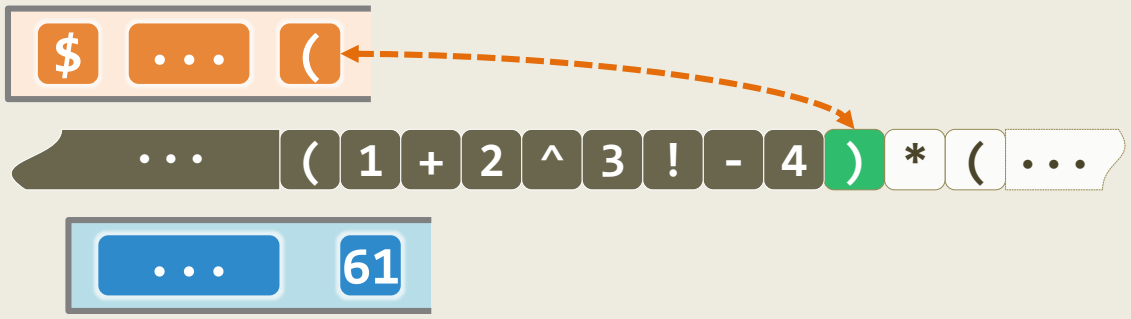
pri['!']['-'] = '>'



pri['^']['-'] = '>'

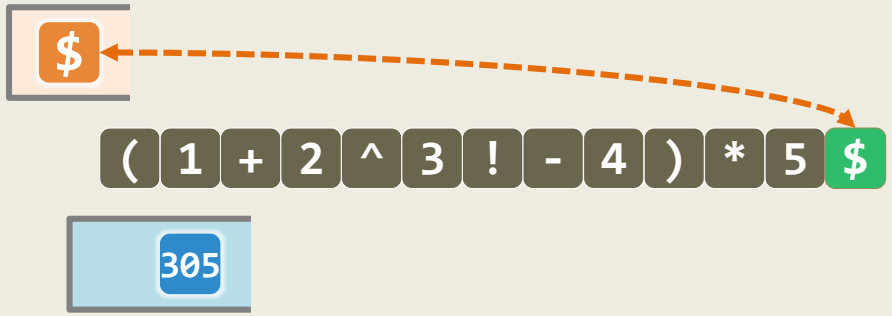
'=': 终须了断：算法

```
switch( priority( optr.top(), *S ) ) {
   case '<':
      /* ..... */
   case '=': //优先级相等（当前运算符为右括号，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
```

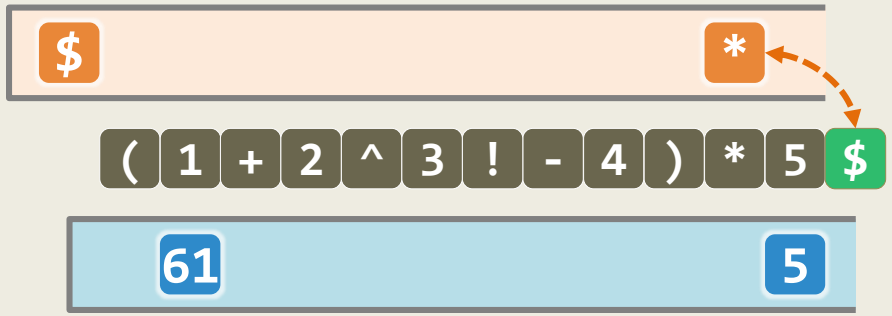


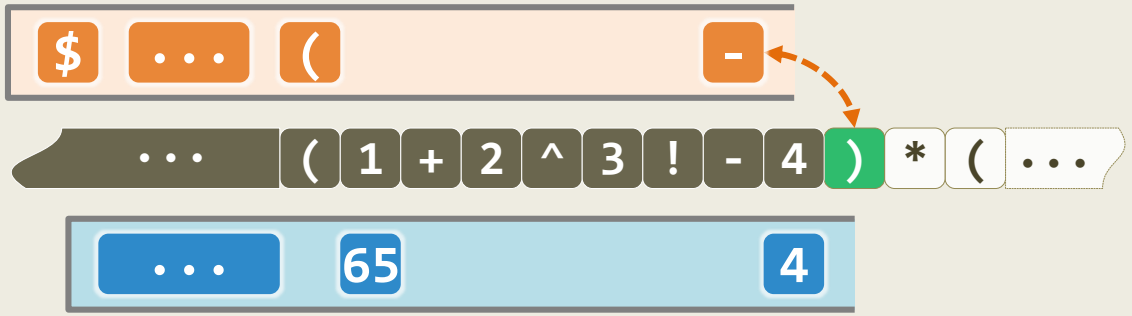
'=': 终须了断: 实例

pri['$']['$'] = '='

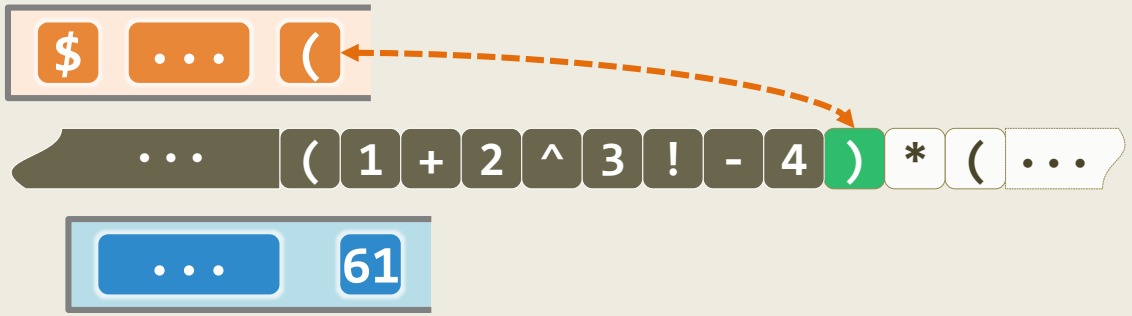


pri['*']['$'] = '>'





pri['-'][') '] = '>'



pri['('][')'] = '='

+ - * / !: 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 --------------|
};
```

04-F3

栈与队列

中缀表达式求值：实例

邓俊辉

deng@tsinghua.edu.cn

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

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

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

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

栈与队列

逆波兰表达式：定义与求值

04-G1

日月逝矣，岁不我与

碰见我爱人的面，要替我说到

说我在这儿，依旧在为她颠倒

邓俊辉

deng@tsinghua.edu.cn

Reverse Polish Notation

- 逆波兰表达式：J. Lukasiewicz (1878 ~ 1956)
- 在由运算符（operator）和操作数（operand）组成的表达式中

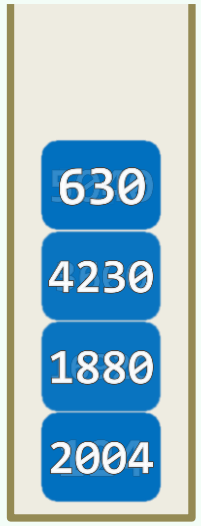
  不使用括号（parenthesis-free），即可表示带优先级的运算关系
- 例如：0 ! + 123 + 4 * (5 * 6 ! + 7 ! / 8) / 9

  0 ! 123 + 4 5 6 ! * 7 ! 8 / + * 9 / +
- 又如：(0 ! + 1) ^ (2 * 3 ! + 4 - 5) - 6 ! / (7 + 8 + 9)

  0 ! 1 + 2 3 ! * 4 + 5 - ^ 6 ! 7 8 + 9 + / -
- 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） //较之原表达式，未必更短

栈式求值

0	!	123	+	4	5	6	!	*	7	!	8	/	+	*	9	/	+



- 引入栈S //存放操作数

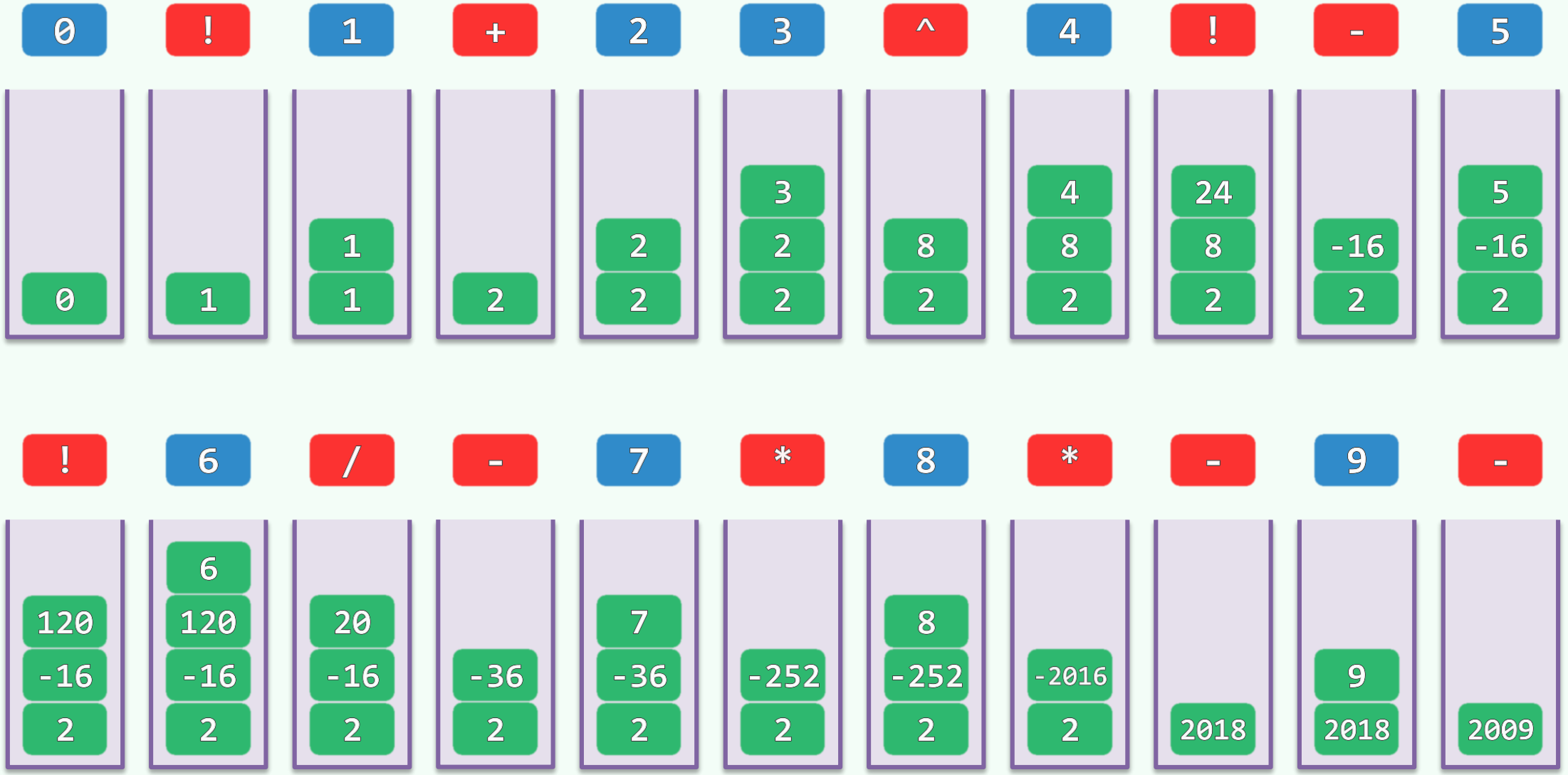
  逐个处理下一元素x

  ```
  if ( x是操作数 ) 将x压入S
  else //运算符无需缓冲
      从S中弹出x所需数目的操作数
      执行相应的计算，结果压入S //无需顾及优先级！
  ```

  返回栈顶

- 只要输入的RPN语法正确，此时的栈顶亦是栈底，对应于最终的计算结果

0 ! 1 + 2 3 ^ 4 ! - 5 ! 6 / - 7 * 8 * - 9 -


0 ! 1 + 2 3 ^ 4 ! - 5
0 1 1 2 2 2 3 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

04-G2

栈与队列

逆波兰表达式：转换

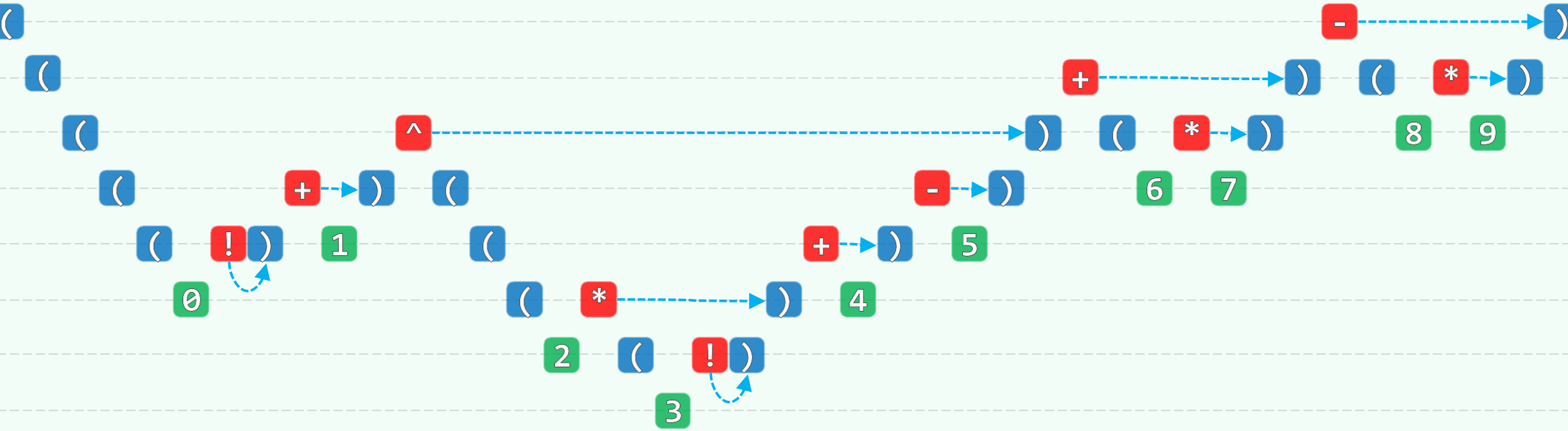
知与之为取，政之宝也

巨大的需求源于巨大的财富，而且要得到我们想要的，最好的办法往往是舍弃我们已拥有的。

邓俊辉

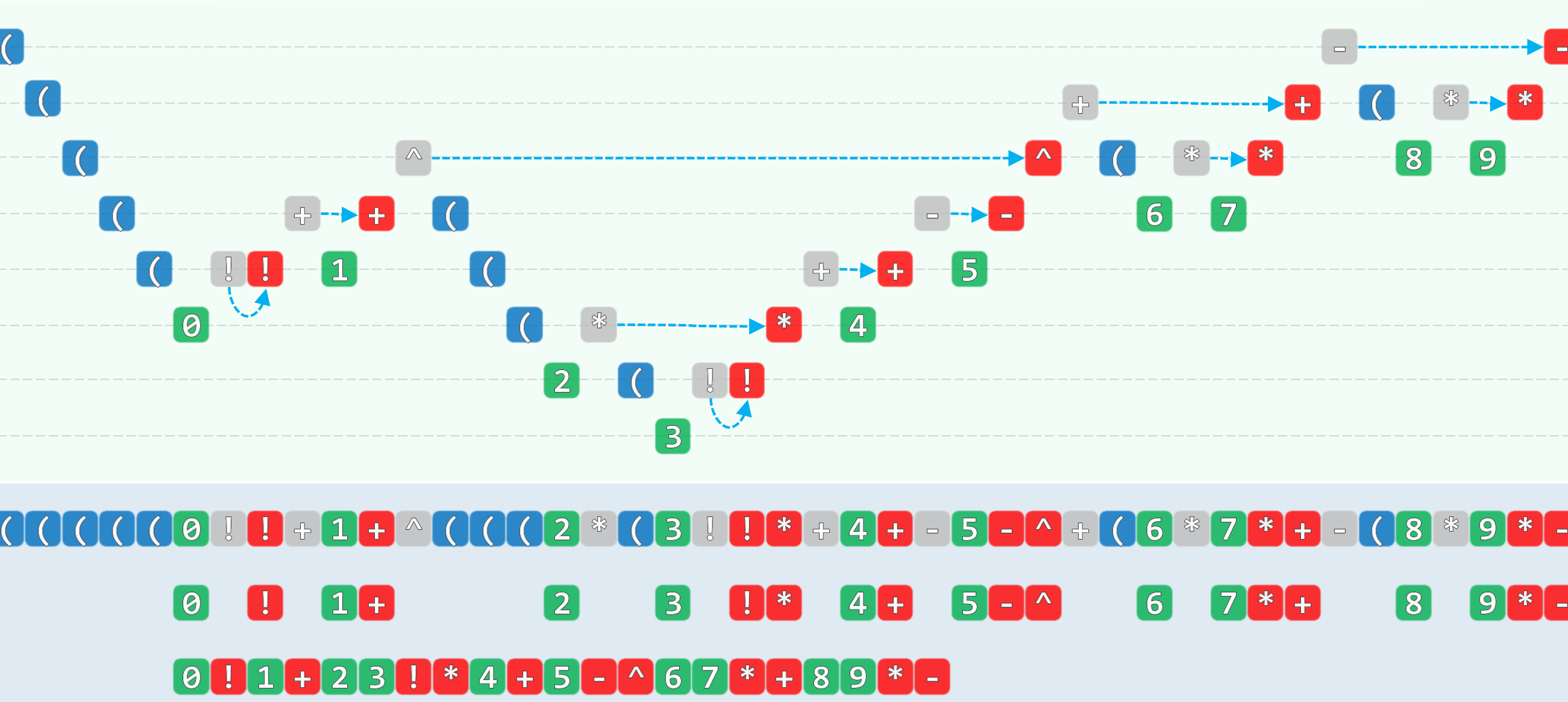
deng@tsinghua.edu.cn

手工转换：添加括号



(0 ! + 1) ^ (2 * 3 ! + 4 - 5) + 6 * 7 - 8 * 9

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() ) { //逐个处理各字符，直至运算符栈空
      if ( isdigit( * S ) ) //若当前字符为操作数，则直接
         { readNumber( S, opnd ); append( RPN, opnd.top() ); } //将其接入RPN
      else //若当前字符为运算符
         switch( priority( optr.top(), *S ) ) {
            /* ..................................... */
            case '>': { //且可立即执行，则在执行相应计算的同时
               char op = optr.pop(); append( RPN, op ); //将其接入RPN
               /* ..................................... */
            } //case '>'
         /* ..................................... */
```

04-G3

栈与队列

逆波兰表达式：PostScript

邓俊辉

deng@tsinghua.edu.cn

Stack x 5

❖ PostScript诞生于1985

支持设备独立的图形描述

❖ （ 1个解释器 + 5个栈 ）× RPN语法

❖ operand stack：存放操作数及运算结果

❖ 提供基础且强大的图形功能

支持数据类型、变量、函数/宏 ...

❖ 一旦遇到操作符，则

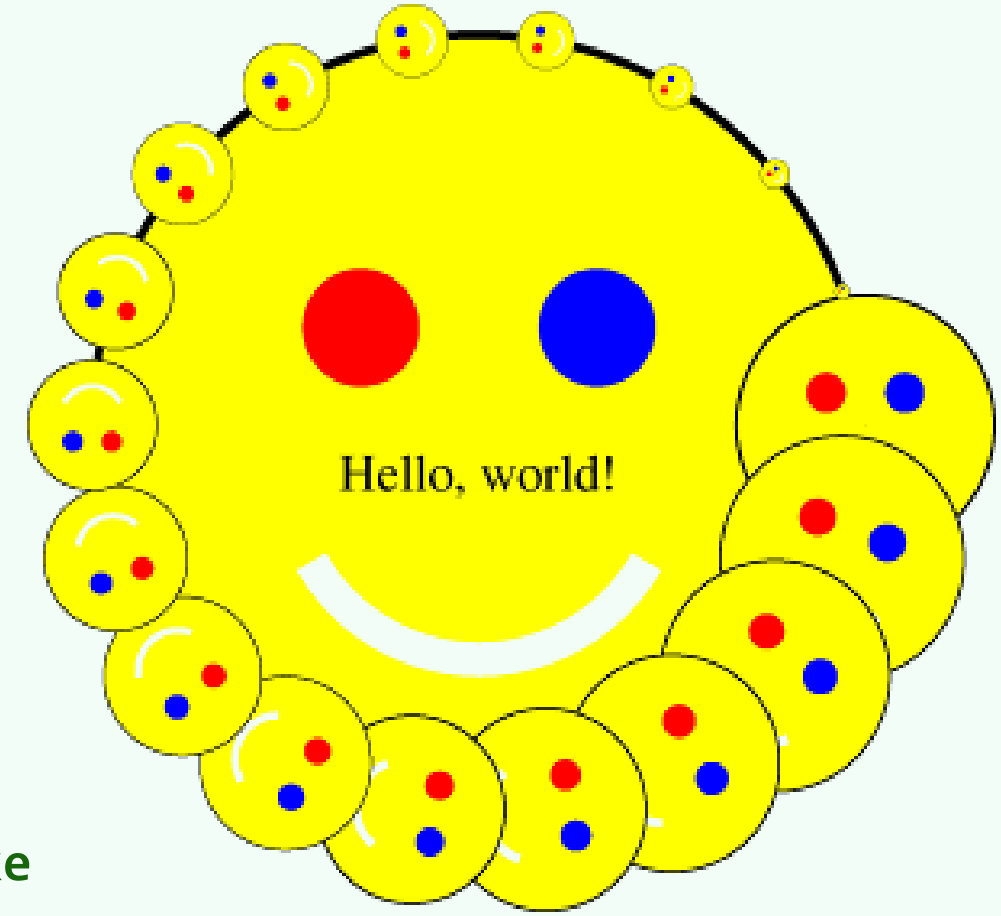
- 弹出相应数目的元素
- 实施计算，并
- 将（可能多个、一个或零个）结果入栈

❖ 实例

```
4 4 mul 5 5 mul add 7 mul 7 mul
```

实例

```
/smile {
    newpath
    gsave
    rotate
    0 translate
    180 div dup scale
    yellow   0 0 180 0 360 arc fill
    red      -55 45 27 0 360 arc fill
    blue     55 45 27 0 360 arc fill
    fatline  white 0 -18 90 210 330 arc stroke
    thinline black 0 0 180 0 360 arc stroke
    grestore
} def
```



04-H

栈与队列

队列接口与实现

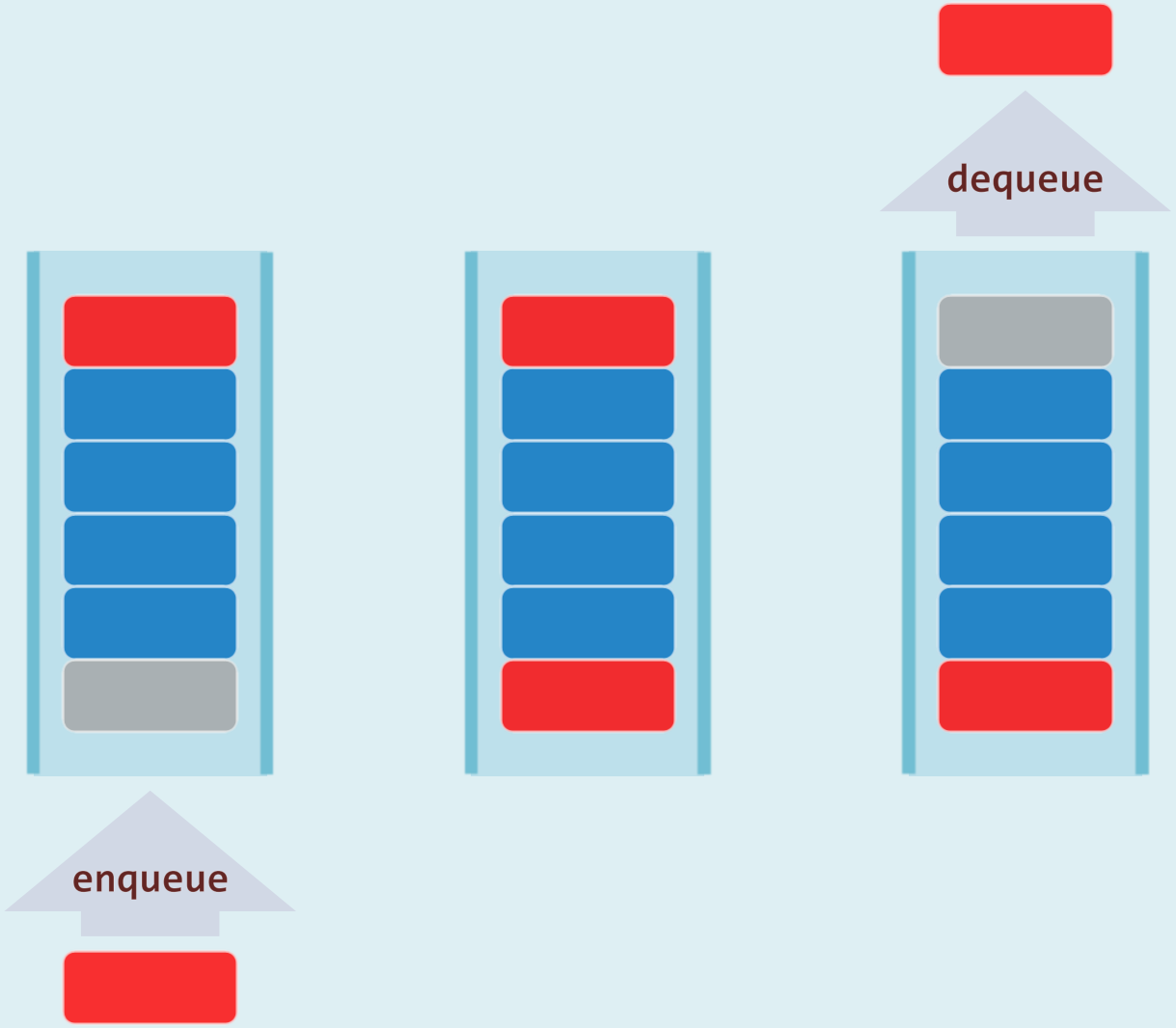
邓俊辉

deng@tsinghua.edu.cn

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）
- 扩展接口：getMax()...



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

❖ 队列既然属于序列的特例，故亦可直接基于向量或列表派生

❖
```
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
  }; //以列表首/末端为队列头/尾——颠倒过来呢?
```

❖ 确认：如此实现的队列接口，均只需$O(1)$时间

❖ 课后：基于向量派生实现队列模板类，并就其效率做一评估

04-1

栈与队列

队列应用

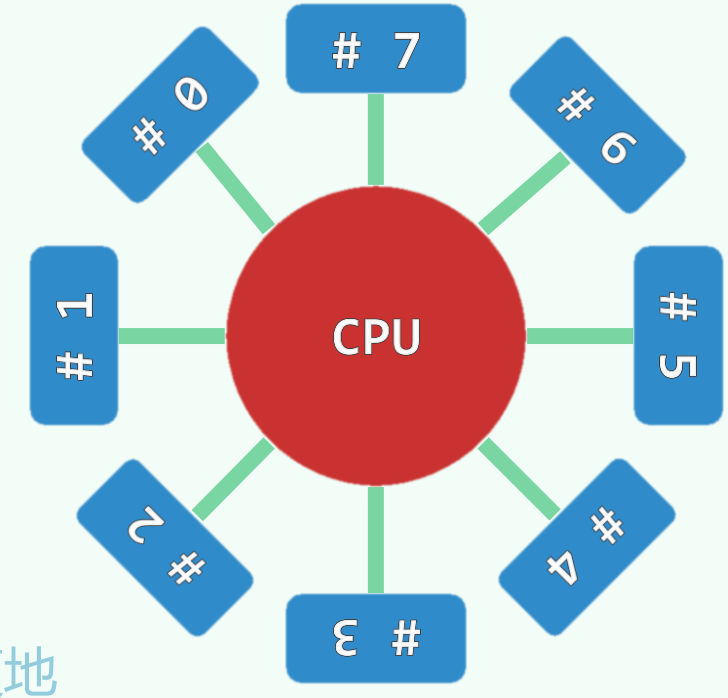
墙上一溜挂着五个烟斗。张大哥不等旧的已经不能再用才买新的，
而是使到半路就买个新的来；新旧替换着用，能多用些日子。

邓俊辉
deng@tsinghua.edu.cn

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？

比如，多个应用程序共享CPU，实验室成员共享打印机，...



❖ RoundRobin //循环分配器

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

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

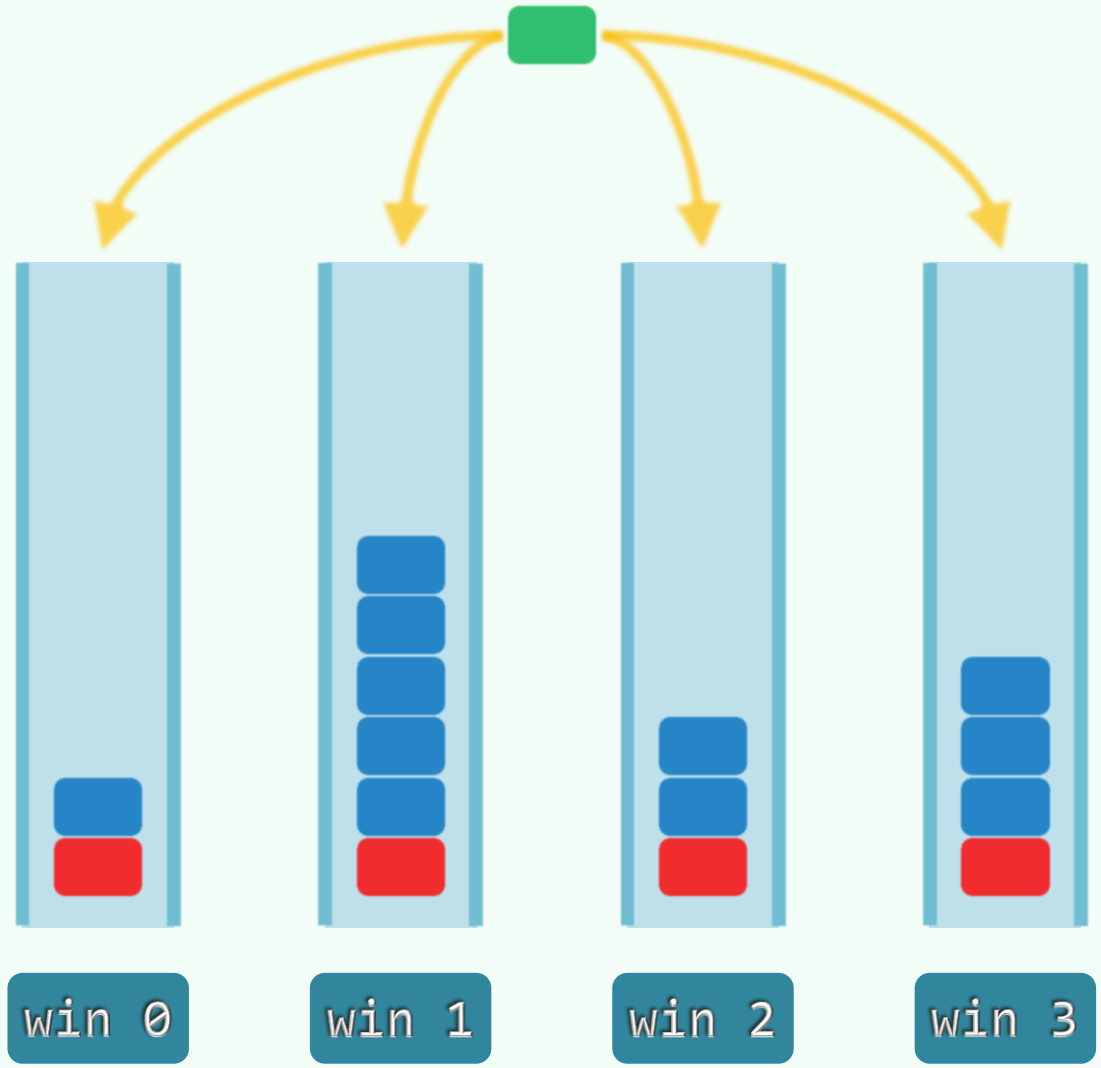
    选择和加入最短队列（的末尾）
- 参数： nWin //窗口（队列）数目

        servTime //营业时长
- 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++ ) { //在下班之前，每隔单位时间
      Customer c ; c.time = 1 + rand() % 50; //一位新顾客到达，其服务时长随机指定
      c.window = bestWindow( windows, nWin ); //找出最佳（最短）服务窗口
      windows[ c.window ].enqueue( c ); //新顾客加入对应的队列
      for ( int i = 0; i < nWin; i++ ) //分别检查
         if ( ! windows[ i ].empty() ) //各非空队列
            if ( -- windows[ i ].front().time <= 0 ) //队首顾客接受服务
               windows[ i ].dequeue(); //服务完毕则出列，由后继顾客接替
   } //for
   delete [] windows; //释放所有队列
}
```

04-J

栈与队列

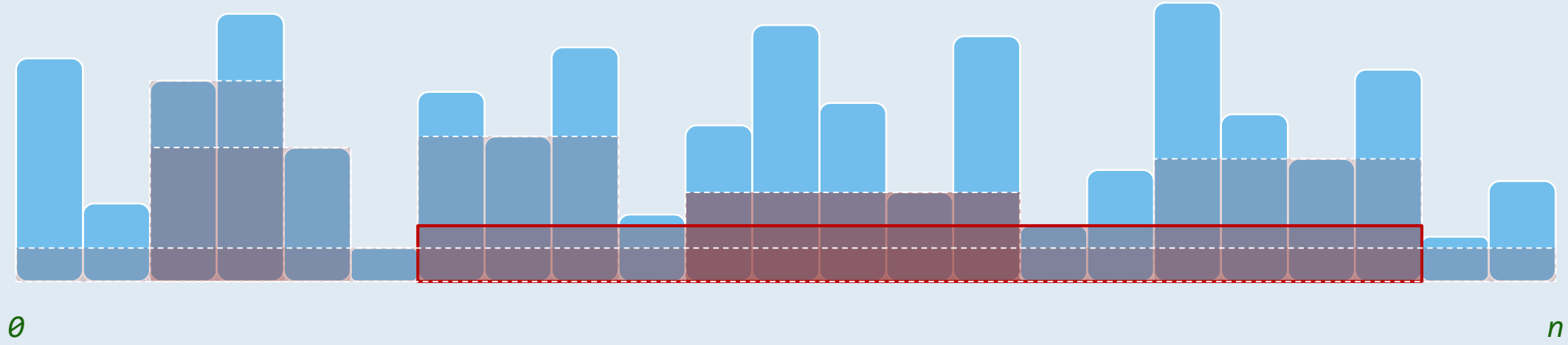
直方图内最大矩形

就这么着，我有了一所严丝密缝、涂抹灰泥的木板房子，七英尺宽，十五英尺长，立柱有八英尺高...

邓俊辉

deng@tsinghua.edu.cn

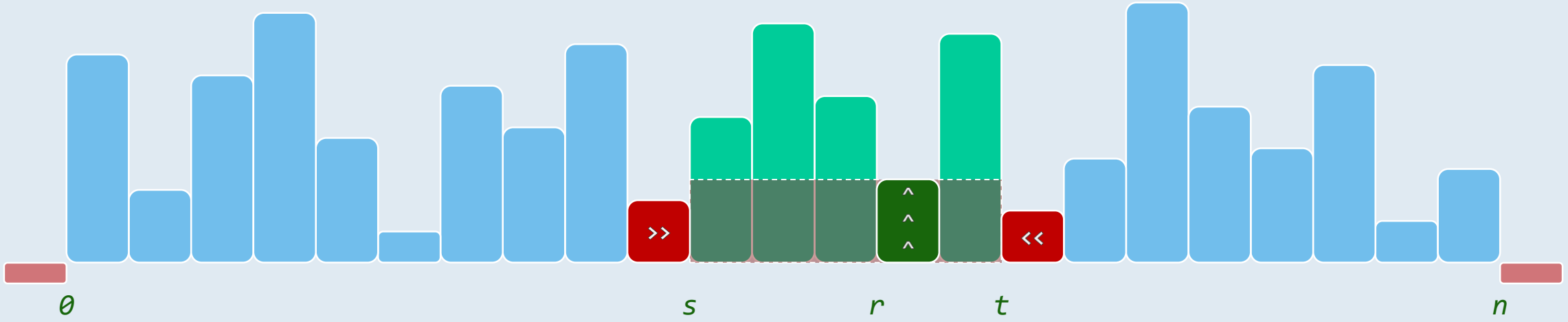
Maximum Rectangle



- Let H[0,n) be a histogram of non-negative integers
- How to find the largest orthogonal rectangle in H[]?
- To eliminate possible ambiguity

  we can, for example, choose the **LEFTMOST** one

Maximal Rectangles



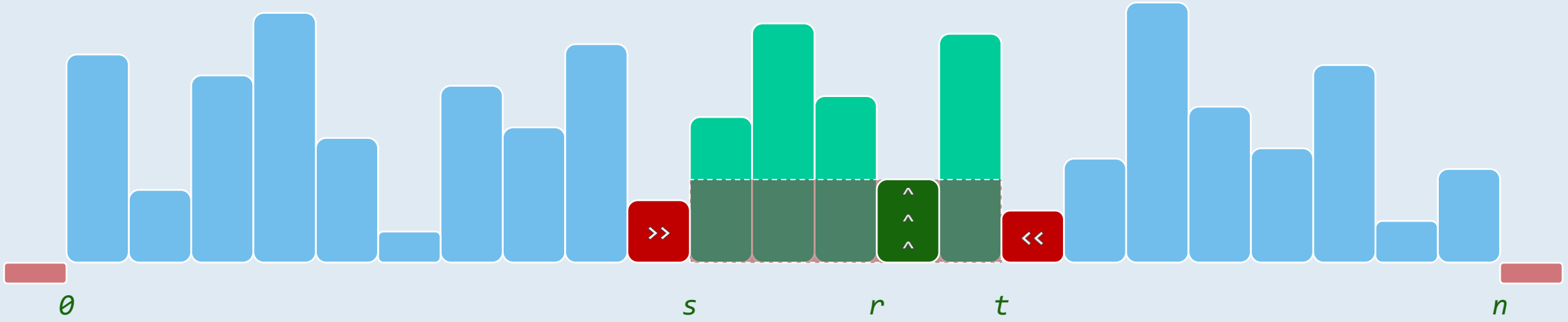
❖ **Maximal rectangle supported by H[r]:** $maxRect(r) = H[r] \cdot (t(r) - s(r))$

where

$$s(r) = \max\{\, k \mid 0 \le k \le r \text{ and } H[k-1] < H[r] \,\}$$

$$t(r) = \min\{\, k \mid r < k \le n \text{ and } H[r] > H[k] \,\}$$

Brute-force



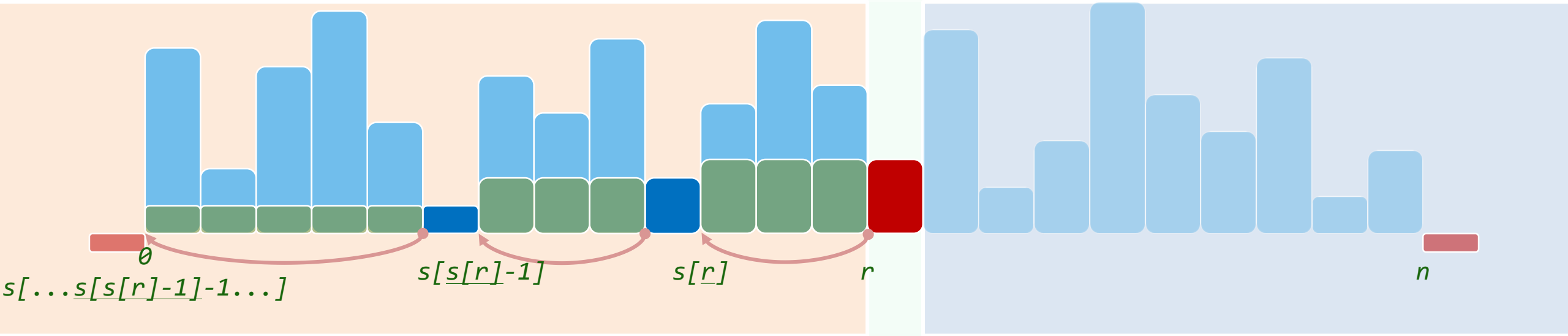
- Determining s(r) and t(r) for all r's requires $\mathcal{O}(n^2)$ time

$$s(r) = \max\{\ k \mid 0 \le k \le r \text{ and } H[k-1] < H[r]\ \}$$

$$t(r) = \min\{\ k \mid r < k \le n \text{ and } H[r] > H[k]\ \}$$

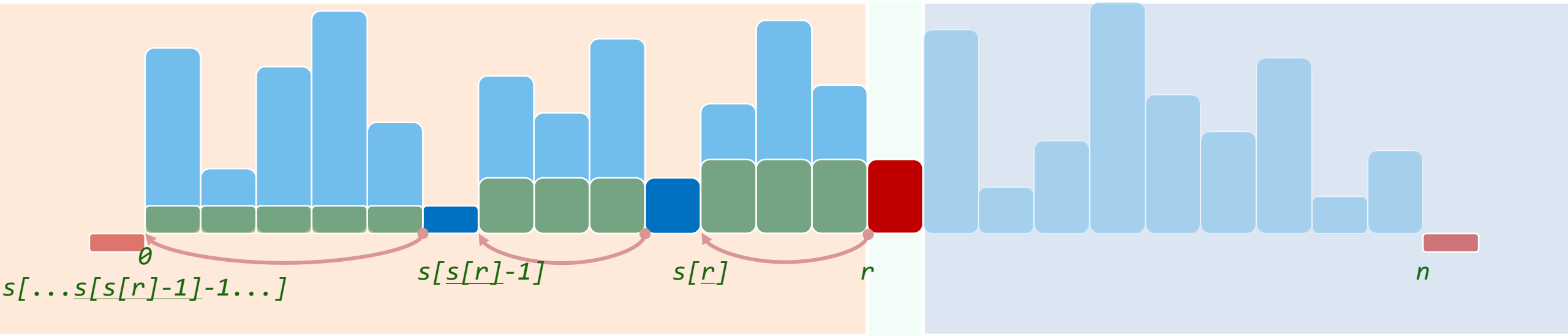
- Actually, all s(r)'s can be determined by a LINEAR scan of the histogram ...

Using Stack: Algorithm




```
Rank* s = new Rank[n]; Stack<Rank> S;
for ( Rank r = 0; r < n; r++ ) //try using SENTINEL for simplicity by yourself
   while ( !S.empty() && ( H[S.top()] >= H[r] ) ) S.pop(); //until H[top] < H[r]
   s[r] = S.empty() ? 0 : 1 + S.top();   S.push(r); //S is always ASCENDING
while( !S.empty() ) S.pop();
```

Using Stack: Loop Invariant & Correctness



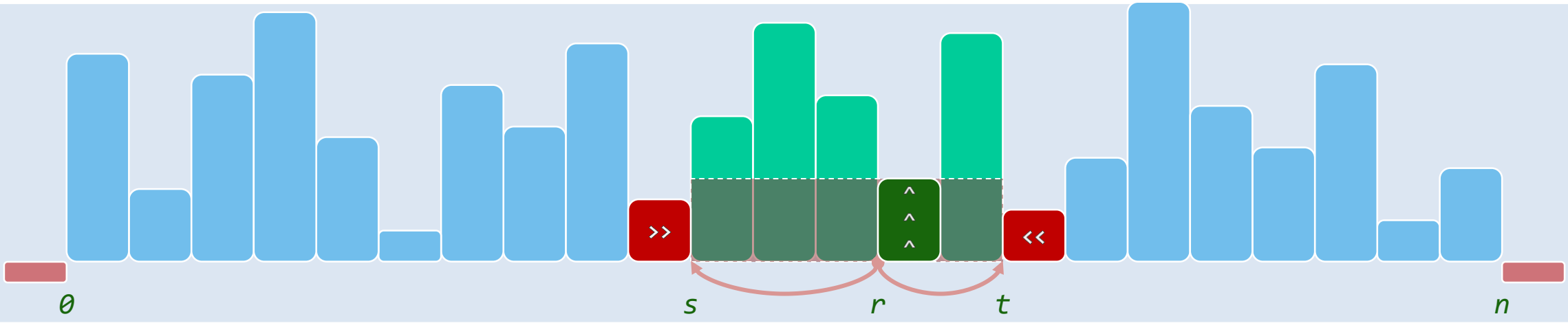
- All bars are scanned and pushed into S in turn
- After each iteration of the outer loop, S stores a chain in terms of s[]

$$S[S.size()-1] = S.top() = r \quad \textbf{and} \quad \forall\, 0 \le k < S.size(),\ \ S[k-1]+1 = s[S[k]]$$

- Every bar is popped when it will be of no use thereafter, i.e.

$$\forall\, 0 \le k < r \ \ but \ \ k \notin S, \quad \nexists\, r \le t \ \ s.t. \ \ k+1 = s(t)$$

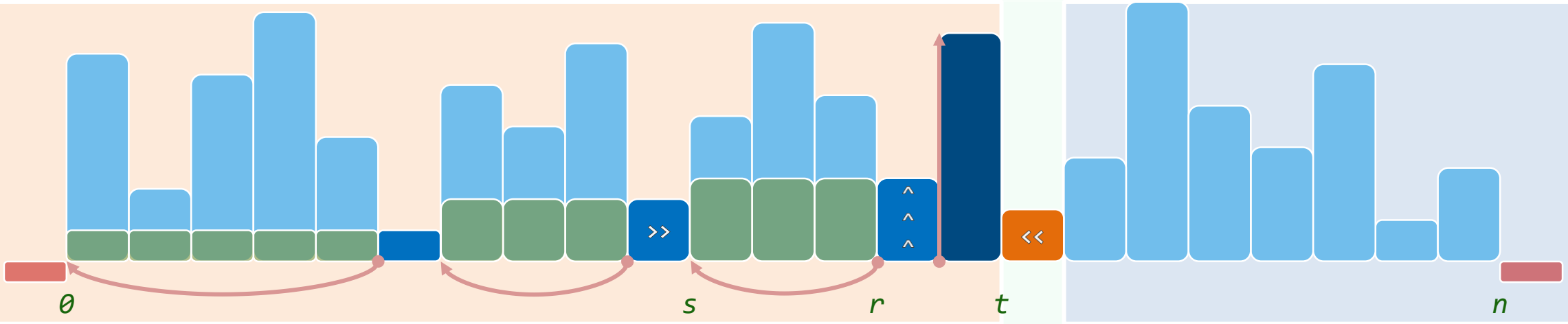
Using Stack: Complexity



- And t(r)'s can be determined by another scan in the REVERSED direction
- Hence all maximal rectangles can be computed in $O(n)$ time and using $O(n)$ space
- However, what if the histogram is given in an IN-PLACE and ON-LINE manner?

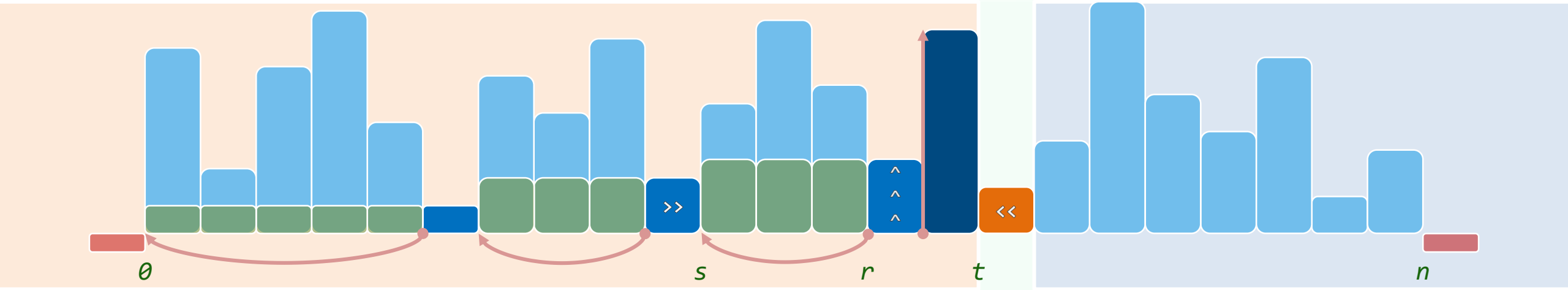
  Note that, the t(r)'s CAN'T be determined until the ENTIRE input is ready
- Is it possible to compute BOTH s(r)'s and t(r)'s by a SINGLE scan? //on-fly

One-Pass Scan: Algorithm




```
Stack<Rank> SR;  __int64 maxRect = 0; //SR.2ndTop() == s(r)-1 & SR.top() == r
for ( Rank t = 0; t <= n; t++ ) //amortized-O(n)
   while ( !SR.empty() && ( t == n || H[SR.top()] > H[t] ) )
      Rank r = SR.pop(), s = SR.empty() ? 0 : SR.top() + 1;
      maxRect = max( maxRect, H[r] * ( t - s ) );
   if ( t < n ) SR.push( t );
return maxRect;
```

One-Pass Scan: Loop Invariant & Correctness



❖ Again, at each iteration of the outer loop, we always have

$$\forall\, 0 \le k < SR.size(),\ \ SR[k-1] + 1 = s[SR[k]]$$

❖ For each bar r popped in the inner loop, we have

$$t(r) = t \quad \textbf{and} \quad s[r] = SR.top() + 1$$

04-K

栈与队列

Steap + Queap

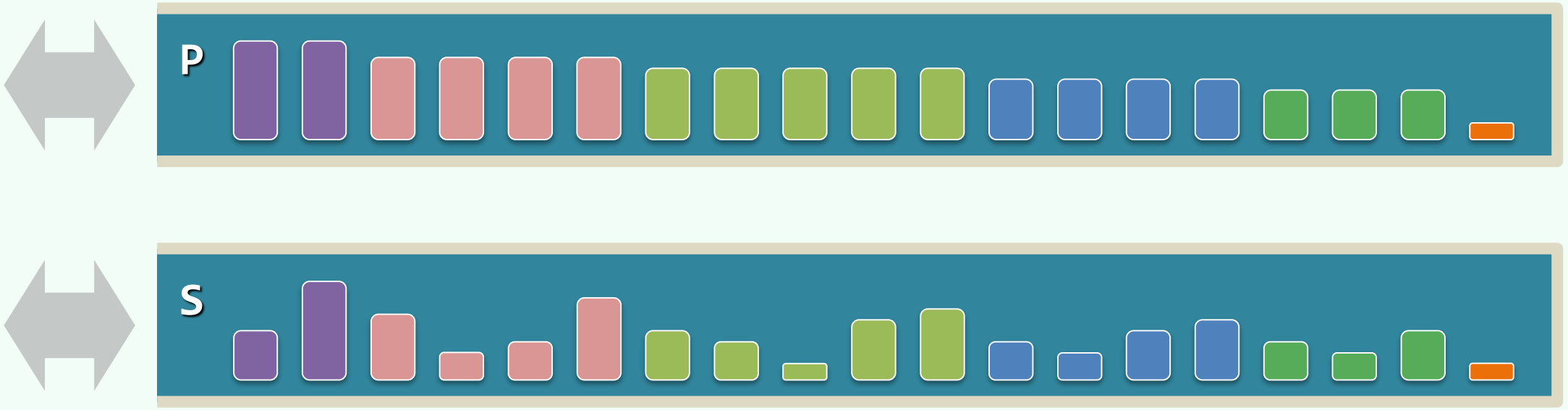
A hero is born among a hundred, a wise man is found among a thousand, but an accomplished one might not be found even among a hundred thousand men.

邓俊辉

deng@tsinghua.edu.cn

Steap = Stack + Heap = push + pop + getMax

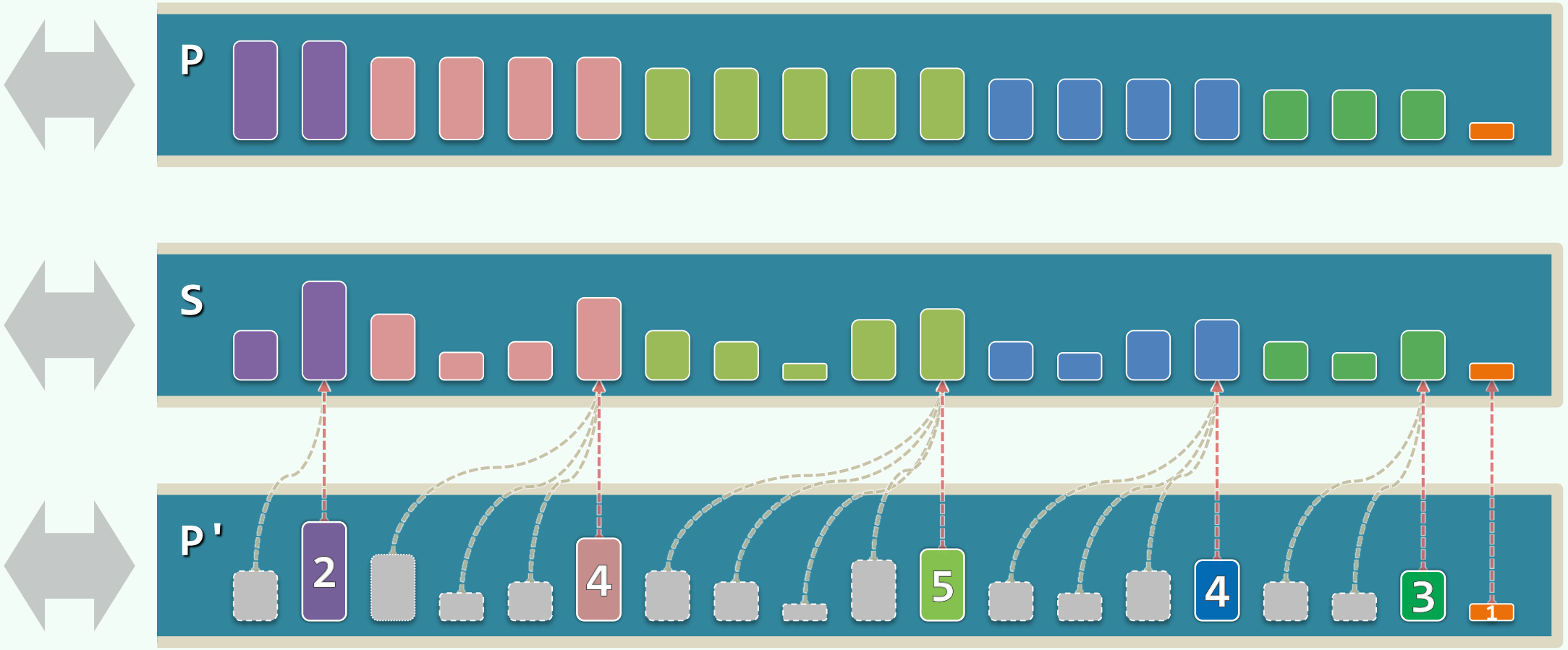
❖ P中每个元素，都是S中对应后缀里的最大者：Steap::getMax() { return P.top(); }



❖ Steap::pop() { P.pop(); return S.pop(); } //O(1)

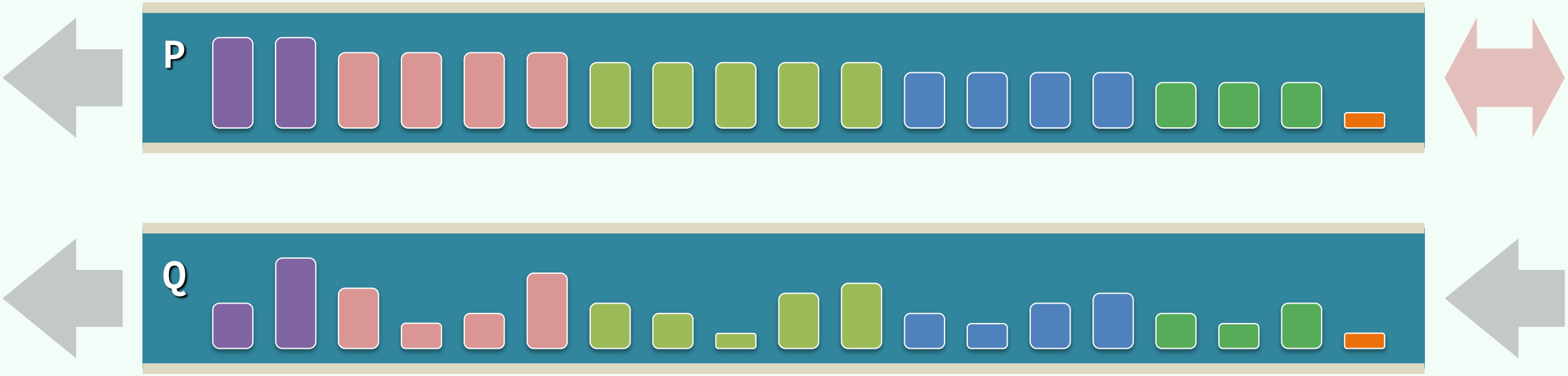
❖ Steap::push(e) { P.push(max(e, P.top())); S.push(e); } //O(1)

Steap = Stack + Heap = push + pop + getMax



Queap = Queue + Heap = enqueue + dequeue + getMax

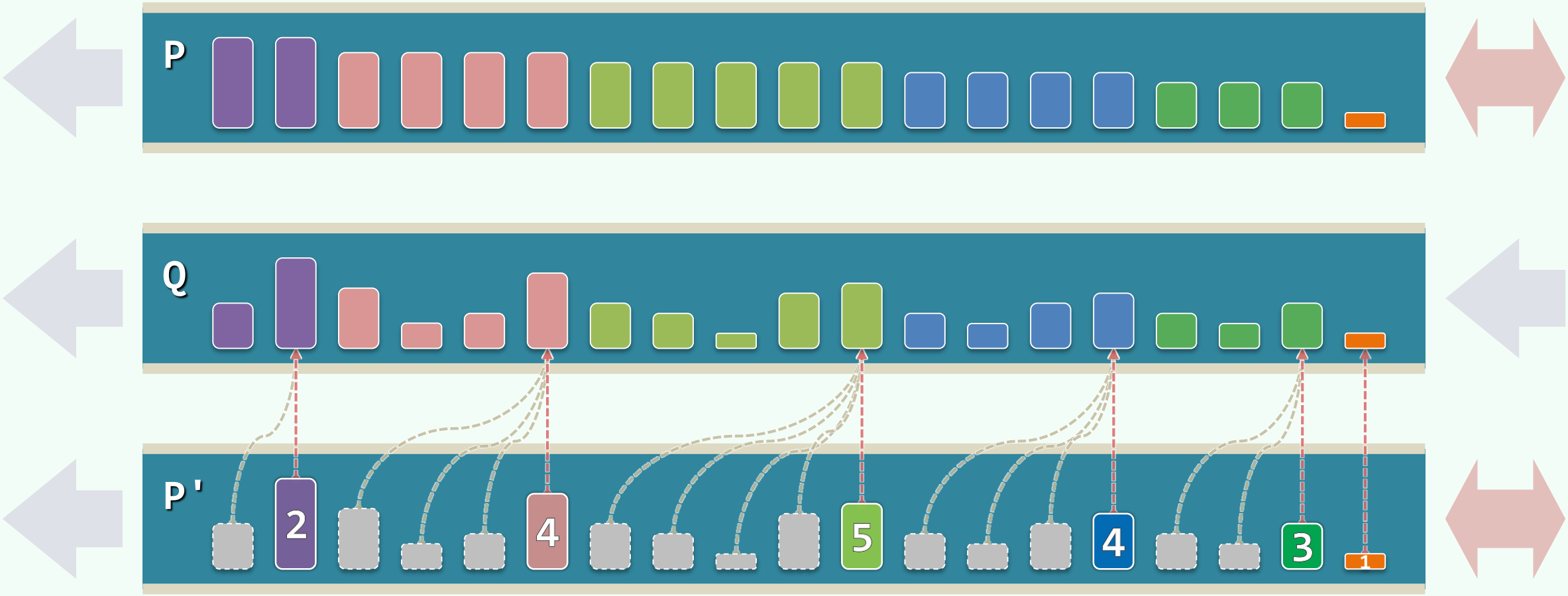
❖ Queap::dequeue() { P.dequeue(); return Q.dequeue(); } //O(1)



❖ Queap::enqueue(e) {

```
    Q.enqueue(e); P.enqueue(e);
    for ( x = P.rear(); x && (x->key <= e); x = x->pred ) //最坏情况O(n)
        x->key = e;
}
```

Queap = Queue + Heap = enqueue + dequeue + getMax



04-L

栈与队列

双栈当队

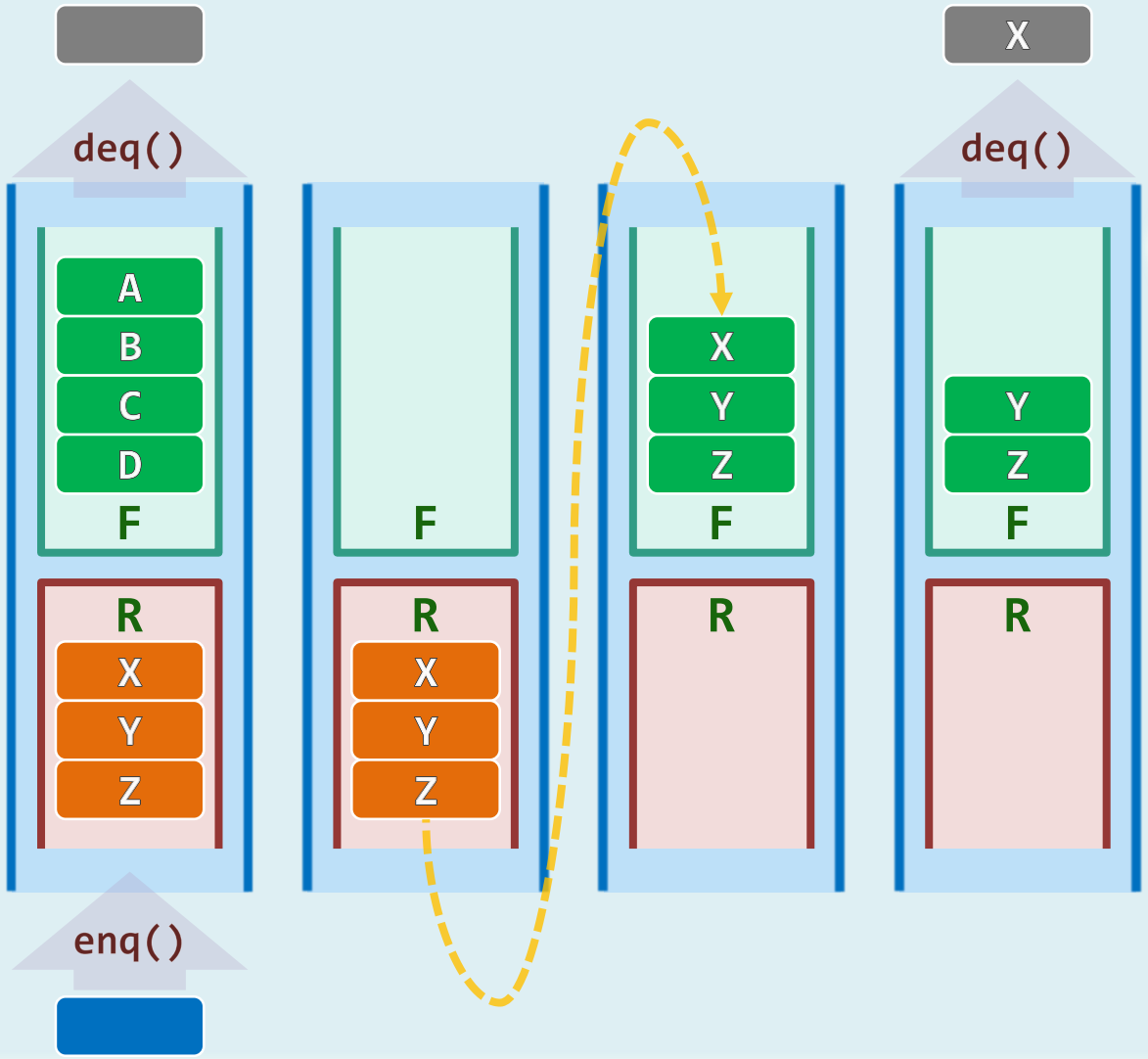
不恒其德，或承之羞，贞吝

我们也不用当场付款，要了什么东西都由店家记在一个小账本上，每两星期结一次账。

邓俊辉

deng@tsinghua.edu.cn

Queue = Stack x 2



- `def Q.enqueue(e)`

```
    R.push(e);
```

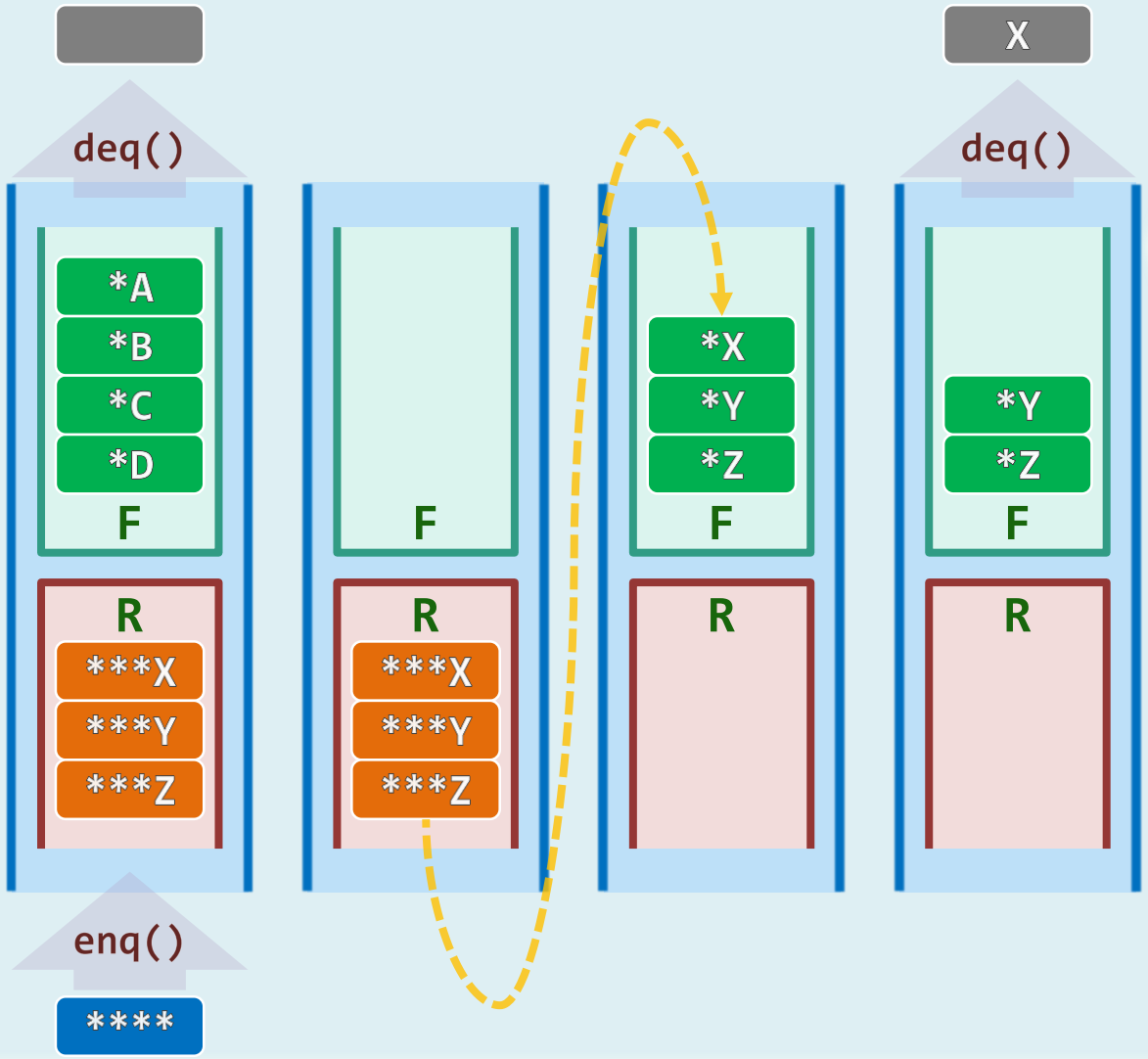
- `def Q.dequeue() # 0 < Q.size()`

```
    if ( F.empty() )
        while ( !R.empty() )
            F.push( R.pop() );
    return F.pop();
```

- Best/worst case: $O(1)/O(n)$

  Average? Amortization!

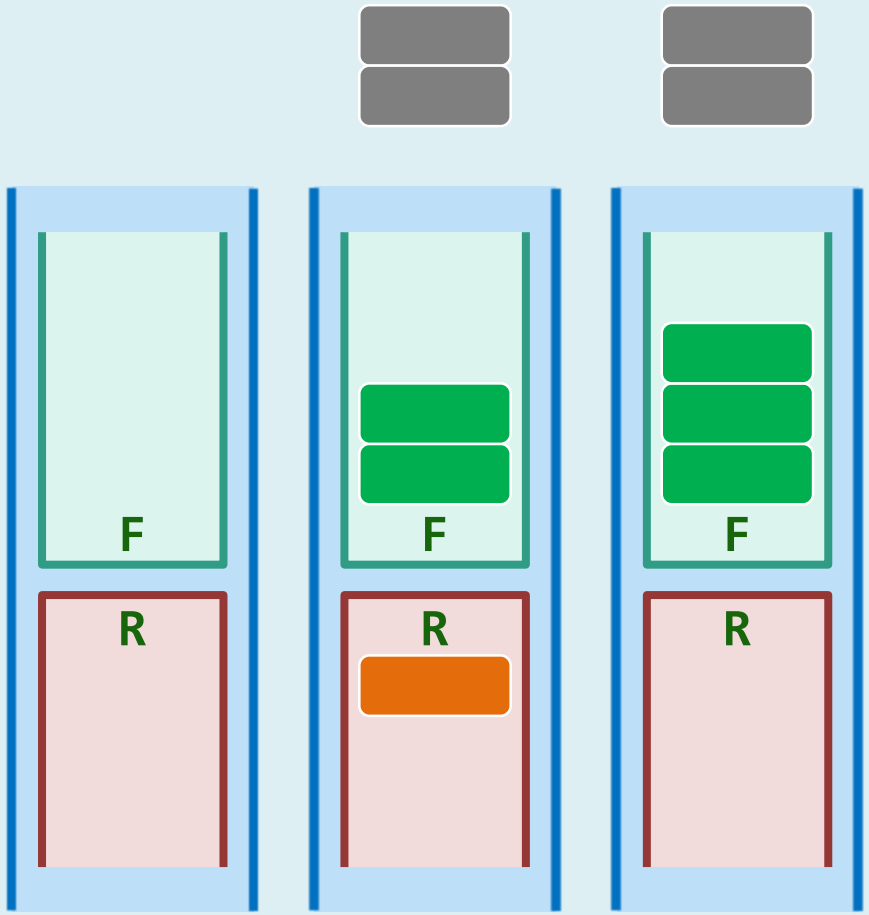
Amortization By Accounting



- Assign each new element with 4 coins //deposit
  - 1 for its enqueue()
  - 2 for transfer, and
  - the last 1 for dequeue()
- Hence every operation is pre-paid and ...
- The structure will never run out of credit
- Amortized cost of any operation sequence involving n ITEMs is

  $4n = O(n)$

Amortization By Aggregate



- Consider the moment when

  d dequeue()'s and **e** enqueue()'s

  have been done //d <= e

- The time cost

  for **ALL** the operations is

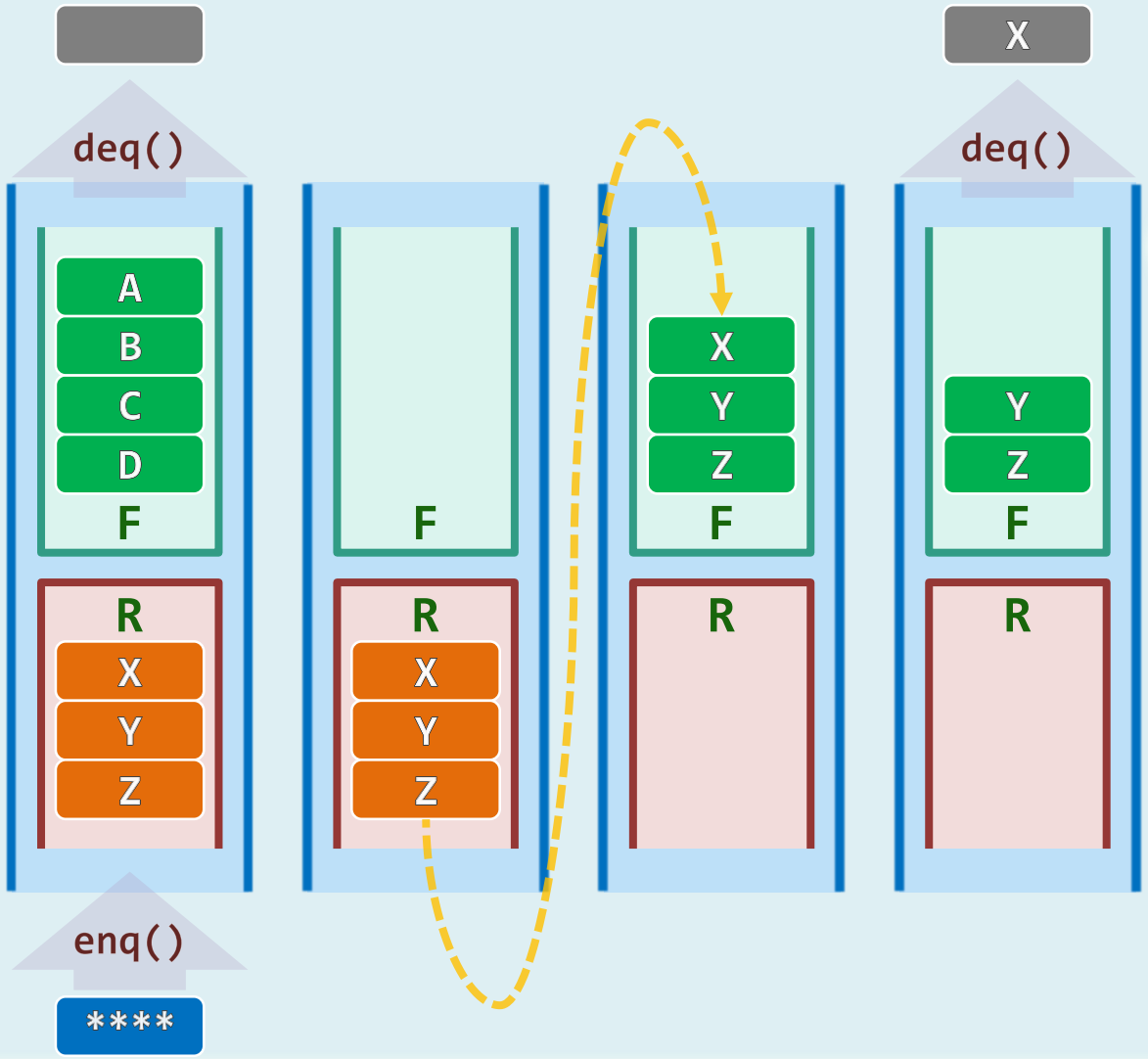
  $$\leq 4 \cdot d + 3 \cdot (e - d) = 3e + d$$

- The amortized cost

  for each OPERATION is

  $$\frac{3e + d}{e + d} < 3$$

Amortization By Potential



- Consider the k^{th} operation
- Define $\Phi_k = |R_k| - |F_k|$
- Then $A_k = T_k + \Phi_k - \Phi_{k-1} \equiv 2$
- Hence

$$2n \equiv \sum_{k=1}^{n} A_k = \sum_{k=1}^{n} T_k + \Phi_n - \Phi_0$$

$$2n = T(n) + \Phi_n - \Phi_0 > T(n) - n$$

$$T(n) < 3n = \mathcal{O}(n)$$