

数据结构与算法 复习笔记

Anzreww

癸卯隆冬于清华园

目录

1 绪论	1
1.1 算法	1
1.2 计算模型	1
1.3 渐进复杂度 Big-O Notation	3
1.4 复杂度分析：级数、递归与主定理 Master Theorem	5
1.5 迭代与递归	6
1.6 动态规划 Dynamic Programming	11
2 向量 Vector	16
2.1 可扩充向量	17
2.2 无序向量	18
2.3 有序向量：二分搜索	20
2.4 起泡排序 Bubble Sort	25
2.5 归并排序 Merge Sort	27
2.6 位图 Bitmap	29
3 列表 List	33
3.1 无序列表	34
3.2 有序列表	37
3.3 选择排序 Selection Sort	37
3.4 插入排序 Insertion Sort	38
3.5 归并排序 Merge Sort	39
3.6 游标实现	40
4 栈与队列 Stack & Queue	43
4.1 栈 Stack	43
4.2 队列 Queue	53
4.3 Steap + Queap	56
4.4 双栈当队	58
5 二叉树 Binary Tree	60
5.1 二叉树 Binary Tree	60
5.2 二叉树的遍历	63
5.3 二叉树的重构	70
5.4 Huffman 编码树	71
6 二叉搜索树 BST(Binary Search Tree)	76
6.1 顺序性——BST 的中序遍历	76

6.2 BST 的基本算法与实现	77
6.3 平衡二叉搜索树 BBST	79
6.4 AVL 树	80
7 更多 BST	86
7.1 区间树 Interval Tree	86
7.2 线段树 Segment Tree	87
7.3 高阶搜索树 Multi-Level Search Tree	90
7.4 kD 树 k Dimentional Tree	95
8 高级 BST	100
8.1 伸展树 Splay Tree	100
8.2 B 树	107
8.3 红黑树 Red-Black Tree	119
9 词典 Dictionary	136
9.1 散列 Hashtable	136
9.2 散列函数 Hash Function	136
9.3 冲突解决：开放散列	137
9.4 冲突解决：封闭散列	138
9.5 桶排序 Bucket sort	140
9.6 基数排序 Radix Sort	140
9.7 计数排序 Counting Sort	140
9.8 跳转表 Skiplist	141
10 优先级队列 Priority Queue	145
10.1 完全二叉堆 Heap	145
10.2 堆排序 Heap Sort	148
10.3 锦标赛树 Tournament Tree	148
11 串 string/char[]	151
11.1 蛮力算法 BF	151
11.2 KMP 算法	151
11.3 优化的 KMP 算法	154
11.4 BM 算法	154
11.5 Karp-Rabin 算法	156
12 排序	157
12.1 快速排序 Quick Sort	157
12.2 k-selection	162
12.3 Shell Sort	165

— 绪论

1.1 算法

计算机、程序、算法

- 计算 = 信息处理 = 借助某种工具，遵照一定规则，以明确而机械的形式进行
- 计算模型 = 计算机 = 信息处理工具
- 所谓算法，即特定计算模型下，旨在解决特定问题的指令序列

算法的特性：

- 输入 待处理的信息（问题）
- 输出 经处理的信息（答案）
- 正确性 确实可以解决指定的问题
- 确定性 可描述为一个由基本操作组成的序列
- 可行性 每一基本操作都可实现，且在常数时间内完成
- 有穷性 对于任何输入，经有穷次基本操作，都可以得到输出

不能确定有穷的程序不能称之为一个算法。可放宽为期望上有穷，例如几何级数。

1.2 计算模型

$T_A(P)$ = 算法 A 求解问题实例 P 的计算成本

$T_A(n) = \max\{T_A(P) \mid |P| = n\}$ 在规模同为 n 的所有实例中，只关注最坏（成本最高）者，有时候我们也会关注期望成分。

为给出客观的评判，需要抽象出一个理想的平台或模型

- 不再依赖于程序员、编译器、计算机、编程语言等具体因素
- 从而直接而准确地描述、测量并评价算法

1.2.1 图灵机 Turing Machine(TM)



图 1 图灵机

- 无限长的纸带(Tape), 纸带上有无限多个格子, 每个格子上有一个字符
- 读写头(Head), 每次只能读写一个格子, 经过一个节拍可以移动一格
- 字符集(Alphabet), 纸带上的字符来自于字符集
- 状态集(State), 有限个状态, 每个状态下有一个动作表, 约定 h 停机

转换函数

```
Transistion(q,c; d, L/R, p)
```

- q 当前状态
- c 当前字符
- d 写入字符
- L/R 移动方向
- p 下一状态

特别地, 一旦转入约定的状态 h , 则停机。从启动至停机, 所经历的节拍数目, 即可用以度量计算的成本; 亦等于 Head 累计的移动次数。

下面的例子就是 Increase 算法的 TM 实现



图 2 Increase 算法的 TM 实现

1.2.2 随机存取机 Random Access Machine(RAM)



图 3 RAM

- 无限长的存储器(Memory), 存储器被划分为若干个存储单元, 每个存储单元存储一个字
- call-by-rank, 每次可以直接访问任意存储单元

与 TM 模型一样, RAM 模型也是一般计算工具的简化与抽象, 使我们可以独立于具体的平台, 对算法的效率做出可信的比较与评判。

在这些模型中

- 算法的运行时间 \propto 算法需要执行的基本操作次数
- $T(n)$ = 算法为求解规模为 n 的问题, 所需执行的基本操作次数

下面的例子就是 Ceiling Division 算法的 RAM 实现



图 4 Ceiling Division 算法的 RAM 实现

1.3 渐进复杂度 Big-O Notation

渐近分析: 更关心问题规模足够大之后, 计算成本的增长趋势。

借用渐进分析中的 O 、 Ω 和 Θ 符号来描述算法的渐进复杂度。

1.3.1 多项式复杂度

1.3.1.1 $O(1)$:constant

这类算法的效率最高。

可能含循环、分支、递归等语句，但其执行次数与问题规模无关。

1.3.1.2 $O(\log^c n)$:poly-logarithmic

这类算法非常有效，复杂度无限接近于常数。

$$\forall c > 0, \log n = O(n^c)$$

1.3.1.3 $O(n^c)$:polynomial

线性 (linear function): $O(n)$

从 $O(1)$ 到 $O(n^2)$ ，一般来说都是能接受的。 $O(n^2)$ 有时候也过高。但是更大次幂，是非常低效的。

1.3.2 $O(c^n)$:exponential

这类算法的计算成本增长极快，通常被认为不可忍受。

从 $O(n^c)$ 到 $O(2^n)$ ，是从有效算法到无效算法的分水岭。

有些问题的最优算法的复杂度就是指数级的，例如 NPC 问题。

NPC 问题：就目前的计算模型而言，不存在可在多项式时间内解决此问题的算法。并且这类问题的解法，可以在多项式时间内验证。

一个典型的问题是 Subset Sum 问题。

给定一个集合 S ，以及一个目标值 T ，判断 S 中是否存在一个子集，其元素之和为 T 。

最优的解法是穷举法，复杂度为 $O(2^n)$ 。

常数	$O(1)$	再好不过，但难得如此幸运	对数据结构的基本操作
	$O(\log^* n)$	在这个宇宙中，几乎就是常数	逆Ackermann函数
对数	$O(\log n)$	与常数无限接近，且不难遇到	有序向量的二分查找；堆、词典的查询、插入与删除
线性	$O(n)$	努力目标，经常遇到	树、图的遍历
	$O(n \log^* n)$	几乎几乎几乎...就是线性	某些MST算法
	$O(n \log \log n)$	非常非常非常...接近线性	某些三角剖分算法
	$O(n \log n)$	最常出现，但不见得最优	排序、EU、Huffman编码
平方	$O(n^2)$	所有输入对象两两组合	Dijkstra算法
立方	$O(n^3)$	不常见	矩阵乘法
多项式	$O(n^c)$	P问题 = 存在多项式算法的问题	
指数	$O(2^n)$	很多问题的平凡算法，再尽可能优化	
...		绝大多数问题，并不存在算法	

图 5 渐近复杂度的层次级别

1.4 复杂度分析：级数、递归与主定理 Master Theorem

1.4.1 算法分析

两个主要任务 = 正确性 (不变性 \times 单调性) + 复杂度

为确定后者，不必将算法描述为 RAM 的基本指令，再累计各条代码的执行次数。C++ 等高级语言的基本指令，均等效于常数条 RAM 的基本指令；在渐近意义下，二者大体相当

- 分支转向： `goto` [算法的灵魂；为结构化而被隐藏]
- 迭代循环： `for()`、`while()`、... [本质上就是“`if + goto`”]
- 调用 + 递归（自我调用）[本质上也是 `goto`]

主要方法：迭代（级数求和）、递归（递归跟踪 + 递推方程）、实用（猜测 + 验证）

1.4.2 级数

一些常见的级数：

- 算术级数（与末项平方同阶）： $1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2} = O(n^2)$
- 幂方级数（比幂次高出一阶）： $\sum_{i=0}^n i^k = O(n^{k+1})$
- 几何级数（与末项同阶）： $\sum_{i=0}^n q^i = \frac{1-q^{n+1}}{1-q} = O(q^n), q > 1$
- 收敛级数，例如倒数平方和： $O(1)$
- 几何分布： $(1 - \lambda)(1 + 2\lambda + 3\lambda^2 + \dots + n\lambda^{n-1}) = O(1)$
- 调和级数： $\sum_{i=1}^n \frac{1}{i} = \Theta(\log n)$
- 对数级数： $\sum_{i=1}^n \log i = \Theta(n \log n)$
- 对数+线性+指数： $\sum_{i=1}^n i \log i = O(n^2 \log n)$, $\sum_{i=1}^n i 2^i = O(n 2^n)$

可以通过积分等方法求得。

1.4.3 迭代

可以画图进行分析

- 迭代+算术级数 $O(n^2)$

```
for( int i = 0; i < n; i++ )
    for( int j = 0; j < n; j++ )
        Oloop(const i, const j);

for( int i = 0; i < n; i++ )
    for( int j = 0; j < i; j++ )
        Oloop(const i, const j);
```

- 迭代+级数 $O(n)$

```
for( int i = 1; i < n; i <= 1 )
    for( int j = 0; j < i; j++ )
        Oloop( const i, const j );
```

- 迭代+复杂级数 如 $O(n \log n)$

```

for( int i = 0; i <= n; i++ )
for( int j = 1; j < i; j += j )
    Oloop( const i, const j );

```

1.4.4 封底估算

- 地球（赤道）周长 $\approx 787 \times \frac{360}{7.2} = 787 \times 50 = 39,350$ km
- 1 天 $= 24\text{hr} \times 60\text{min} \times 60\text{sec} \approx 25 \times 4000 = 10^5$ sec
- 1 生 ≈ 1 世纪 $= 100\text{yr} \times 365 = 3 \times 10^4$ day $= 3 \times 10^9$ sec
- “为祖国健康工作五十年” $\approx 1.6 \times 10^9$ sec
- “三生三世” ≈ 300 yr $= 10^{10} = (1 \text{ googel})^{\frac{1}{10}}$ sec
- 宇宙大爆炸至今 $= 4 \times 10^{17} > 10^8 \times \text{一生}$

1.5 迭代与递归

1.5.1 减而治之 Decrease-and-conquer

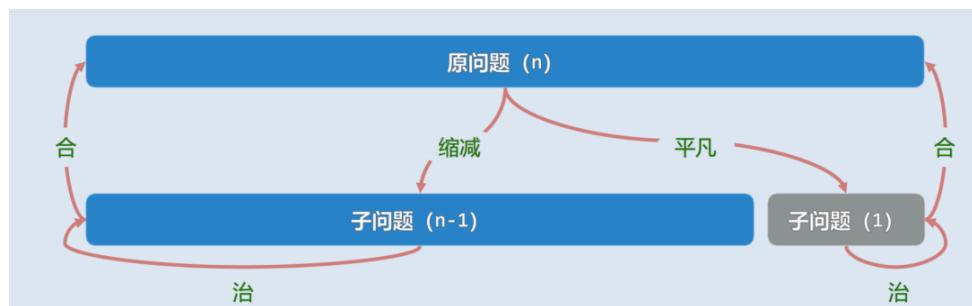


图 6 减而治之

为求解一个大规模的问题，可以

- 将其划分为两个子问题：其一平凡，另一规模缩减
- 分别求解子问题；再由子问题的解，得到原问题的解

例如

```

int SumI( int A[], int n ) {
    int sum = 0; //O(1)
    for ( int i = 0; i < n; i++ ) //O(n)
        sum += A[i]; //O(1)
    return sum; //O(1)
}
/* Decrease-and-conquer:Linear Recursion */
sum( int A[], int n )
{ return n < 1 ? 0 : sum(A, n - 1) + A[n - 1]; }

```

递归跟踪：绘出计算过程中出现过的所有递归实例（及其调用关系）

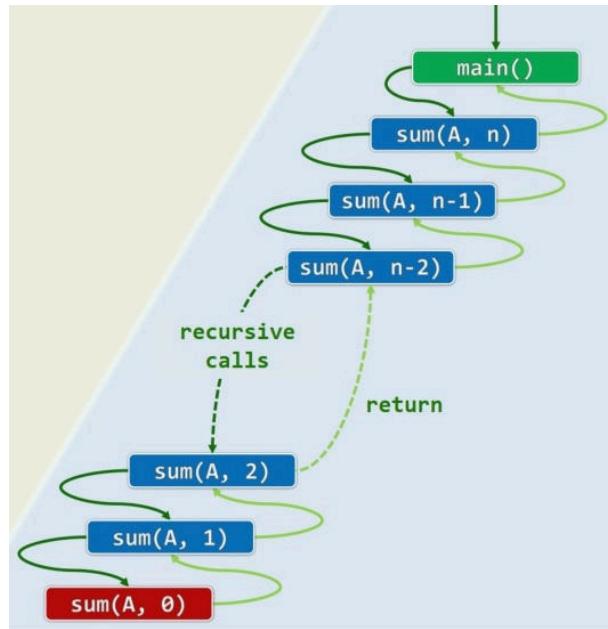


图 7 递归跟踪

本例中，共计 $n + 1$ 个递归实例，各自只需 $O(1)$ 时间，故总时间为 $O(n)$ 。开出 $n + 1$ 个栈帧，故空间为 $O(n)$ 。

递推方程：递归实例的计算成本 $T(n)$ 与其规模 n 的关系

$$T(n) = T(n - 1) + O(1) = O(n)$$

本例的复杂度是 $O(n)$ 。

尾递归：递归调用出现在函数体的最后一条语句中。尾递归可以被转化为迭代，从而节省空间。

例如，将数组中的区间 $A[lo, hi]$ 前后颠倒 `void reverse(int * A, int lo, int hi);`
 减治： $\text{Rev}(lo, hi) = [hi] + \text{Rev}(lo + 1, hi - 1) + [lo]$

```

if (lo < hi) { //递归版
    swap( A[lo], A[hi] );
    reverse( A, lo + 1, hi - 1 );
} //线性递归（尾递归），O(n)

while (lo < hi) //迭代版
    swap( A[lo++], A[hi--] ); //亦是 O(n)
    
```

1.5.2 分而治之 Divide-and-conquer

为求解一个大规模的问题，可以将其划分为若干子问题（通常两个，且规模大体相当）。分别求解子问题，由子问题的解合并得到原问题的解。

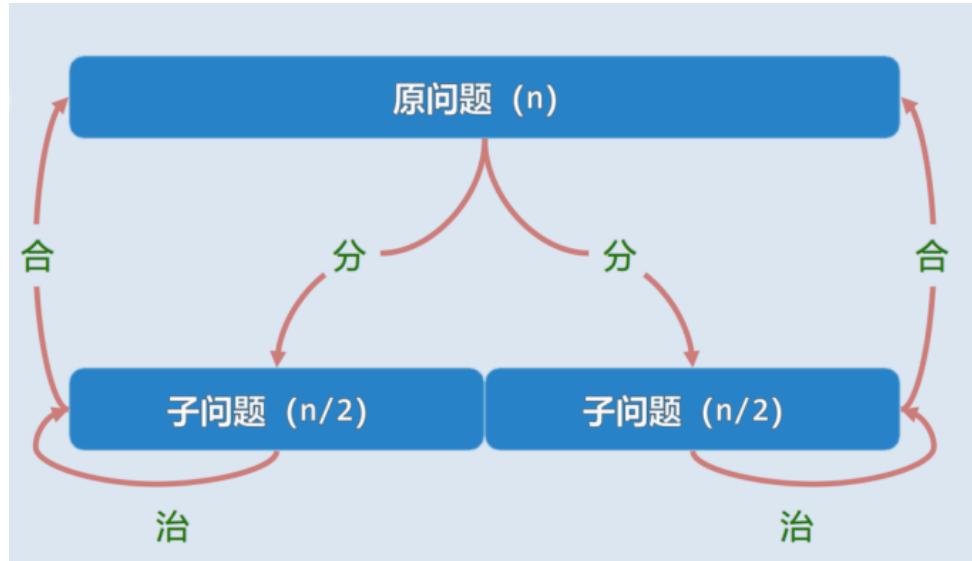


图 8 分而治之

例如前面的 SumI 算法，可以改写为

```
/* Divide-and-conquer:Binary Recursion */
sum( int A[], int lo, int hi ) { //区间范围 A[lo, hi]
    if ( hi - lo < 2 ) return A[lo];
    int mi = (lo + hi) >> 1; return sum( A, lo, mi ) + sum( A, mi, hi );
} //入口形式为 sum( A, 0, n )
```

可以列出递推方程

$$T(n) = 2T\left(\frac{n}{2}\right) + O(1) = O(n)$$

而对于一般的这种方程，可以用主定理求解

主定理 Master Theorem

$$T(n) = aT\left(\frac{n}{b}\right) + O(f(n))$$

- 若 $f(n) = O(n^{\log_b a} - \varepsilon)$ ，则 $T(n) = \Theta(n^{\log_b a})$
- 若 $f(n) = \Theta(n^{\log_b a} \cdot \log^k n)$ ，则 $T(n) = \Theta(n^{\log_b a} \cdot \log^{k+1} n)$
- 若 $f(n) = \Omega(n^{\log_b a} + \varepsilon)$ ，则 $T(n) = \Theta(f(n))$

即比较 $f(n)$ 与 $n^{\log_b a}$ 的大小关系，若 $f(n)$ 更大，则复杂度为 $f(n)$ ；若相当，则复杂度为 $n^{\log_b a} \cdot \log^k n$ ；若更小，则复杂度为 $n^{\log_b a}$ 。

几个后面会出现的例子：

- kd-search:** $T(n) = 2T\left(\frac{n}{4}\right) + O(1) = O(\sqrt{n})$
- binary search:** $T(n) = T\left(\frac{n}{2}\right) + O(1) = O(\log n)$
- merge sort:** $T(n) = 2T\left(\frac{n}{2}\right) + O(n) = O(n \log n)$

数据结构与算法 复习笔记

- STL merge sort: $T(n) = 2T\left(\frac{n}{2}\right) + O(n \log n) = O(n \log^2 n)$
- quickSelect: $T(n) = T\left(\frac{n}{2}\right) + O(n) = O(n)$

例如大数乘法:

1. Naive + DAC

```
AB  
CD  
x_____  
AC  
BD  
AD  
BC
```

$$T(n) = 4T\left(\frac{n}{2}\right) + O(n) = O(n^2)$$

2. Optimal

```
AB  
CD  
x_____  
AC  
BD  
AC  
BD  
(A-B)(D-C)
```

$$\text{这样只用计算三个乘法 } T(n) = 3T\left(\frac{n}{2}\right) + O(n) = O(n^{\log_2 3})$$

1.5.3 例：总和最大区段问题

给定一个整数序列 $A[0, n]$, 求其总和最大的区段 $A[i, j]$, 其中 $0 \leq i < j < n$, (有多个时, 短者、靠后者优先)。

1.5.3.1 蛮力算法 BF

枚举所有区段, 计算其总和, 选出最大者。

```
int gs_BF( int A[], int n ){ //蛮力策略: O(n^3)
    int gs = A[0]; //当前已知的最大和
    for ( int i = 0; i < n; i++ )
        for ( int j = i; j < n; j++ ) //枚举所有的 O(n^2) 个区段!
            int s = 0;
            for ( int k = i; k <= j; k++ ) s += A[k]; //用 O(n) 时间求和
            if ( gs < s ) gs = s; //择优、更新
    return gs;
}
```

1.5.3.2 递增策略

求和时记忆, 故相同开头的区段, 只需在前者的基础上加上一个元素即可。

```

int gs_IC( int A[], int n ){ //递增策略: O(n^2)
    int gs = A[0]; //当前已知的最大和
    for ( int i = 0; i < n; i++ ) //枚举所有起始于 i
        int s = 0;
        for ( int j = i; j < n; j++ ) //终止于 j 的区间
            s += A[j]; //递增地得到其总和: O(1)
            if ( gs < s ) gs = s; //择优、更新
    return gs;
}

```

1.5.4 分治策略：前缀 + 后缀

$$A[\text{lo}, \text{hi}] = A[\text{lo}, \text{mi}] \cup A[\text{mi}, \text{hi}] = P \cup S$$

借助递归，便可求得 P, S 内部的 GS；而剩余的实质任务是考察那些跨越切分线的区段。

所以每段返回两个值，一个是区段内的 GS，一个是含端点的 GS。

二者可以独立计算，累计用时为 $O(n)$ ，故总时间为 $O(n \log n)$ 。

```

int gs_DC( int A[], int lo, int hi ) { //Divide-And-Conquer: O(n*logn)
    if ( hi - lo < 2 ) return A[lo]; //递归基
    int mi = (lo + hi) / 2; //在中点切分
    int gsL = A[mi-1], sL = 0, i = mi; //枚举
    while ( lo < i-- ) //所有[i, mi]类区段
        if ( gsL < (sL += A[i]) ) gsL = sL; //更新
    int gsR = A[mi], sR = 0, j = mi-1; //枚举
    while ( ++j < hi ) //所有[mi, j]类区段
        if ( gsR < (sR += A[j]) ) gsR = sR; //更新
    return max( gsL + gsR, max( gs_DC(A, lo, mi), gs_DC(A, mi, hi) ) ); //
递归
}

```

1.5.5 分治策略：最短的总和非正的后缀 ~ 总和最大区段

考虑后缀 S ，若其总和非正，则可将其舍弃，因为其不可能是最大区段的一部分。

所以只需在后缀 S 中，找出总和最大的区段即可。通过一次线性扫描实现，不断剪除负和后缀。时间复杂度为 $O(n)$ 。

```

int gs_LS( int A[], int n ) { //Linear Scan: O(n)
    int gs = A[0], s = 0, i = n;
    while ( 0 < i-- ) { //在当前区间内
        s += A[i]; //递增地累计总和
        if ( gs < s ) gs = s; //并择优、更新
        if ( s <= 0 ) s = 0; //剪除负和后缀
    }
    return gs;
}

```

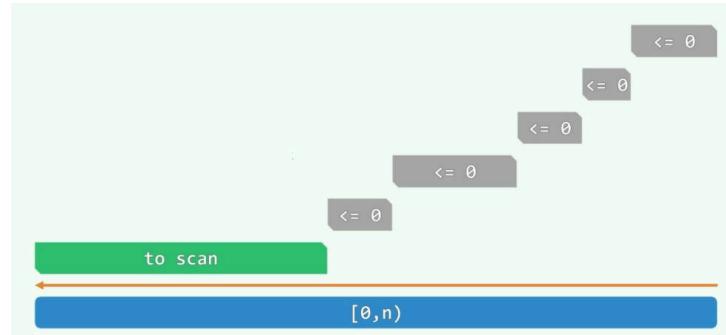


图 9 分治策略：最短的总和非正的后缀 \sim 总和最大区段

1.6 动态规划 Dynamic Programming

1.6.1 记忆法 Memoization

例如 `fib()` 的递归算法：

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

$$T(n) = T(n-1) + T(n-2) + O(1)$$

这个算法的时间复杂度是 $O(\varphi^n)$ 。



图 10 递归算法的递归跟踪

可以看到，有很多重复的计算，例如 `fib(3)` 被计算了两次，`fib(2)` 被计算了三次，`fib(1)` 被计算了五次。

所以可以用一个数组来存储已经计算过的值，这样就可以避免重复计算，这就是记忆法。

```
def f(n)
    if ( n < 1 ) return trivial( n );
    return f(n-X) + f(n-Y)*f(n-Z);
```

```
/* Memoization: Top-down Dynamic Programming */
T M[ N ]; //init. with UNDEFINED
def f(n)
    if ( n < 1 ) return trivial( n );
    // recur only when necessary & always write down the result
    if ( M[n] == UNDEFINED )
        M[n] = f(n-X) + f(n-Y)*f(n-Z);
    return M[n];
```

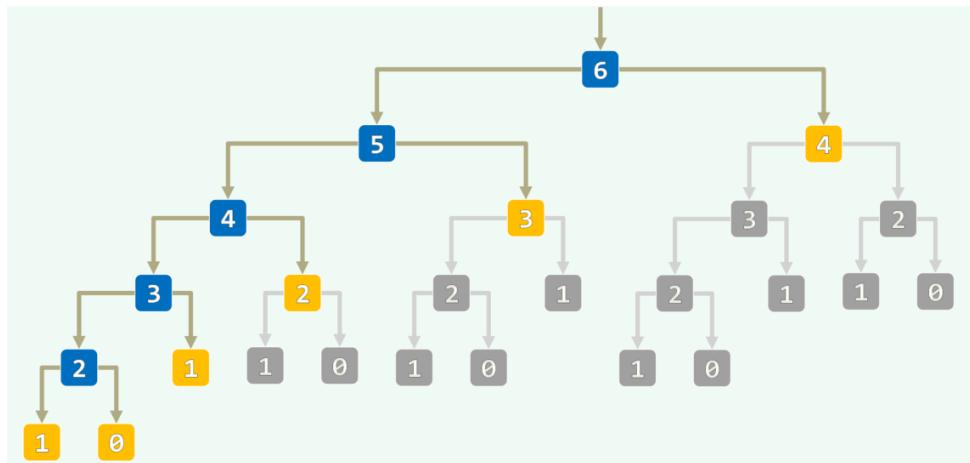


图 11 记忆法的递归跟踪

Dynamic programming, 颠倒计算方向: 由自顶而下递归, 改为自底而上迭代。

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

这样也节省了空间。

1.6.2 例: 最长公共子序列 LCS

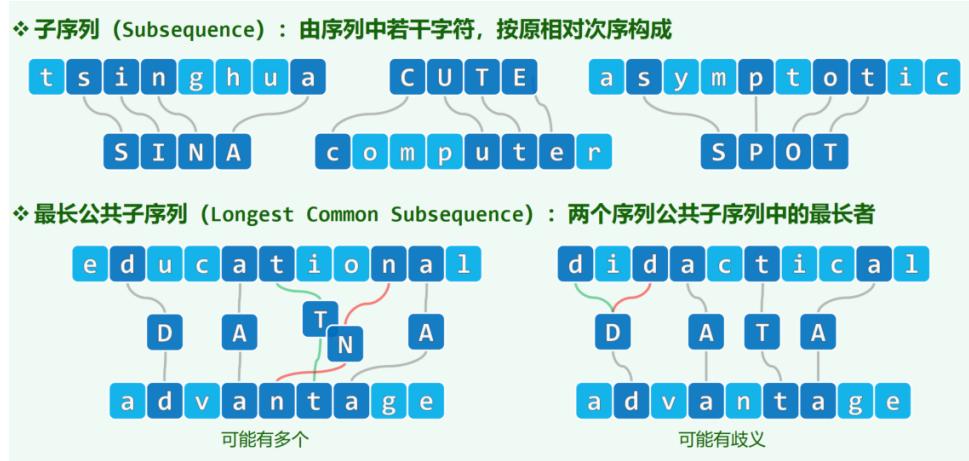


图 12 最长公共子序列 LCS

对于序列 $A[0, n)$ 和 $B[0, m)$, LCS(n, m) 有三种情况:

- 若 $n = 0$ 或 $m = 0$, 则取作空序列 (长度为零), 这是递归基: 必然总能抵达
- 若 $A[n-1] = B[m-1]$, 则取作: $\text{LCS}(n-1, m-1) + 'X'$ [减治策略]
- 若 $A[n-1] \neq B[m-1]$, 则在 $\text{LCS}(n, m-1)$ 与 $\text{LCS}(n-1, m)$ 中取更长者 [分治策略]

```

Input: two strings A and B of length n and m resp.,
Output: (the length of) the longest common subsequence of A and B
lcs( A[], n, B[], m )
    Compare the last characters of A and B, i.e., A[n-1] and B[m-1]
    If A[n-1] = B[m-1]
        Compute x = lcs(A, n-1, B, m-1) recursively and return 1 + x
    Else
        Compute x = lcs(A, n-1, B, m) & y = lcs(A, n, B, m-1) and return
        max(x, y)
    As the recursion base, return 0 when either n or m is 0
  
```

基本实现是

```

unsigned int lcs( char const * A, int n, char const * B, int m ) {
    if (n < 1 || m < 1) //trivial cases
        return 0;
    else if ( A[n-1] == B[m-1] ) //decrease & conquer
        return 1 + lcs(A, n-1, B, m-1);
    else //divide & conquer
        return max( lcs(A, n-1, B, m), lcs(A, n, B, m-1) );
}
  
```

如果用这种算法, $\text{LCS}(A[a], B[b])$ 被调用的次数是 $\binom{n+m-a-b}{n-a}$, 单 $\text{LCS}(A[0], B[0])$ 就会被调用 $\Omega(2^n)$ 次。

下面我们可以用记忆化进行优化:



图 13 最长公共子序列 LCS—记忆法

```

unsigned int lcsMemo(char const* A, int n, char const* B, int m) {
    unsigned int * lcs = new unsigned int[n*m]; //lookup-table of sub-
solutions
    memset(lcs, 0xFF, sizeof(unsigned int)*n*m); //initialized with n*m
UINT_MAX's
    unsigned int solu = lcsM(A, n, B, m, lcs, m);
    delete[] lcs;
    return solu;
}

unsigned int lcsM( char const * A, int n, char const * B, int m,
unsigned int * const lcs, int const M ) {
    if (n < 1 || m < 1) return 0; //trivial cases
    if (UINT_MAX != lcs[(n-1)*M + m-1]) return lcs[(n-1)*M + m-1]; //recursion
stops
    else return lcs[(n-1)*M + m-1] =
        (A[n-1] == B[m-1]) ?
            1 + lcsM(A, n-1, B, m-1, lcs, M)
            max( lcsM(A, n-1, B, m, lcs, M), lcsM(A, n, B, m-1, lcs, M) );
}

```

采用动态规划的策略，只需 $O(mn)$ 时间即可计算出所有子问题。

		d	i	d	a	c	t	i	c	a	l
	0	0	0	0	0	0	0	0	0	0	0
a	0	0	0	0	1	1	1	1	1	1	1
d	0	1	1	1	1	1	1	1	1	1	1
v	0	1	1	1	1	1	1	1	1	1	1
a	0	1	1	1	2	2	2	2	2	2	2
n	0	1	1	1	2	2	2	2	2	2	2
t	0	1	1	1	2	2	3	3	3	3	3
a	0	1	1	1	2	2	3	3	3	4	4
g	0	1	1	1	2	2	3	3	3	4	4
e	0	1	1	1	2	2	3	3	3	4	4

图 14 最长公共子序列 LCS—动态规划

```

unsigned int lcs(char const * A, int n, char const * B, int m) {
    if (n < m) { swap(A, B); swap(n, m); } //make sure m <= n
    unsigned int* lcs1 = new unsigned int[m+1]; //the current two rows are
    unsigned int* lcs2 = new unsigned int[m+1]; //buffered alternatively
    memset(lcs1, 0x00, sizeof(unsigned int) * (m+1)); lcs2[0] = 0;
    for (int i = 0; i < n; swap( lcs1, lcs2 ), i++)
        for (int j = 0; j < m; j++)
            lcs2[j+1] = ( A[i] == B[j] ) ? 1 + lcs1[j] : max( lcs2[j], lcs1[j+1] );
    unsigned int solu = lcs1[m]; delete[] lcs1; delete[] lcs2; return solu;
}

```

二 向量 Vector

和数组一样，寻秩访问(Call by Rank): 元素各由编号唯一指代，并可直接访问。为了使数组可以动态操作，引入 `ADT::Vector`。

向量是数组的抽象与泛化，由一组元素按线性次序封装而成。

各元素与 $[0, n)$ 内的秩 (rank) 一一对应：

```
using Rank = unsigned int; //call-by-rank
```

提供接口：

```
size() / empty() // 报告元素总数 / 判定是否为空 向量
get(r) / put(r, e) // 获取秩为 r 的元素 / 用 e 替换秩为 r 元素的数值 向量
insert(r, e) / insert(e) // 将 e 作为秩为 r 的 / 最后一个元素插入 向量
remove(lo, hi) / remove(r) // 删除秩为 r / 区间内的元素 向量
disordered() / sort(lo, hi) / unsort(lo, hi) // 检测是否整体有序 / 整体排序 / 整体置乱 向量
find(e, lo, hi) / search(e, lo, hi) // 在指定区间内查找目标 e 向量 / 有序向量
dedup() / uniquify() // 剔除重复元素 向量 / 有序向量
traverse( visit() ) // 遍历向量，统一按 visit() 处理所有元素 向量
```

模板类：

```
template <typename T> class Vector { //向量模板类
private: Rank _size; Rank _capacity; T* _elem; //规模、容量、数据区
protected:
/* ... 内部函数 */
public:
/* ... 构造函数 */
/* ... 析构函数 */
/* ... 只读接口 */
/* ... 可写接口 */
/* ... 遍历接口 */
/* ... 遍历接口 */
};
```

构造 + 析构：重载

```
#define DEFAULT_CAPACITY 3 //默认初始容量（实际应用中可设置为更大）
Vector( int c = DEFAULT_CAPACITY )
{ _elem = new T[ _capacity = c ]; _size = 0; } //默认构造
Vector( T const * A, Rank lo, Rank hi ) //数组区间复制
{ copyFrom( A, lo, hi ); }
Vector( Vector<T> const & V, Rank lo, Rank hi ) //向量区间复制
{ copyFrom( V._elem, lo, hi ); }
Vector( Vector<T> const & V ) //向量整体复制
```

```
{ copyFrom( V._elem, 0, V._size ); }
~Vector() { delete [] _elem; } //释放内部空间
```

基于复制的构造

```
template <typename T> //T 为基本类型, 或已重载赋值操作符'='
void Vector<T>::copyFrom( T const * A, Rank lo, Rank hi ) { //A 中元素不致被篡改
    _elem = new T[ _capacity = max( DEFAULT_CAPACITY, 2*(hi - lo) ) ]; //分配空间
    for ( _size = 0; lo < hi; _size++, lo++ ) //A[lo, hi) 内的元素, 逐一
        _elem[ _size ] = A[ lo ]; //复制至 _elem[0, hi-lo)
} //O(hi - lo) = O(n)
```

2.1 可扩充向量

开辟内部数组 `_elem[]` 并使用一段地址连续的物理空间, `_capacity`: 总容量, `_size`: 当前的实际规模 n 。

定义装填因子(load factor): $\lambda = \frac{\text{_size}}{\text{_capacity}}$, 分为:

- 上溢(overflow): `_elem[]` 不足以存放所有元素, 尽管此时系统往往仍有足够的空间
- 下溢(underflow): `_elem[]` 中的元素寥寥无几

这种时候就要进行动态空间管理, 扩容或缩容。

扩容:

```
template <typename T> void Vector<T>::expand() { //向量空间不足时扩容
    if ( _size < _capacity ) return; //尚未满员时, 不必扩容
    _capacity = max( _capacity, DEFAULT_CAPACITY ); //不低于最小容量
    T* oldElem = _elem; _elem = new T[ _capacity <<= 1 ]; //容量加倍
    for ( Rank i = 0; i < _size; i++ ) //复制原向量内容
        _elem[i] = oldElem[i]; //T 为基本类型, 或已重载赋值操作符'='
    delete [] oldElem; //释放原空间
} //得益于向量的封装, 尽管扩容之后数据区的物理地址有所改变, 却不致出现野指针
```

分析扩容的复杂度需要用到分摊想法, 因为扩容本身不是每次都发生的, 而是在一定条件下才发生的, 所以要分析一系列操作的总体复杂度, 而不是单纯的扩容操作。

考虑连续插入 n 次, 扩容的次数不超过 $\log_2 n$ 次, 每次扩容的复杂度为 $O(n)$, 所以分摊复杂度为 $O(1)$ 。采取装填因子比一半大就扩容比每次都扩容好。

平均 (*average complexity*): 根据各种操作出现概率的分布, 将对应的成本加权平均

- 各种可能的操作, 作为独立事件分别考查
- 割裂了操作之间的相关性和连贯性
- 往往不能准确地评判数据结构和算法的真实性能

分摊 (*amortized complexity*): 连续实施的足够多次操作, 所需总体成本摊还至单次操作

- 从实际可行的角度, 对一系列操作做整体的考量

- 更加忠实地刻画了可能出现的操作序列
- 更为精准地评判数据结构和算法的真实性能

2.2 无序向量

2.2.1 基本操作

2.2.1.1 元素访问

```
template <typename T> //可作为左值: V[r] = (T) (2*x + 3)
    T & Vector<T>::operator[]( Rank r ) { return _elem[ r ]; }
template <typename T> //仅限于右值: T x = V[r] + U[s] * W[t]
    const T & Vector<T>::operator[]( Rank r ) const { return _elem[ r ]; }
// 这里采用了简易的方式处理意外和错误(比如, 入口参数约定: 0 <= r < _size)
```

抛弃 `V.get(r)V.put(r,e)` 接口, 重载`[]`操作符。

2.2.1.2 插入

在装填因子大于一半时扩容:

```
template <typename T> Rank Vector<T>::insert( Rank r, T const & e ) { //
0<=r<=_size
    expand(); //如必要, 先扩容
    for ( Rank i = _size; r < i; i-- ) //0(n-r): 自后向前
        _elem[i] = _elem[i - 1]; //后继元素顺次后移一个单元
    _elem[r] = e; _size++; return r; //置入新元素, 更新容量, 返回秩
}
```

插入操作的复杂度为 $O(n)$ 。

2.2.1.3 区间删除

在装填因子小于 $\frac{1}{4}$ 时缩容:

```
template <typename T> Rank Vector<T>::remove( Rank lo, Rank hi ) { //
0<=lo<=hi<=n
    if ( lo == hi ) return 0; //出于效率考虑, 单独处理退化情况
    while ( hi < _size ) _elem[ lo++ ] = _elem[ hi++ ]; //后缀[hi,n)前移
    _size = lo; shrink(); //更新规模, lo = _size 之后的内容无需清零; 如必要, 则缩
    容
    return hi - lo; //返回被删除元素的数目
}
```

区间删除操作的复杂度为 $O(n)$ 。选取 $\frac{1}{4}$ 的界而不是 $\frac{1}{2}$ 的界, 是为了避免在连续删除操作中频繁地扩容和缩容。

2.2.1.4 单元素删除

```
template <typename T>
T Vector<T>::remove( Rank r ) {
```

```

T e = _elem[r]; //备份
remove( r, r+1 ); //“区间”删除
return e; //返回被删除元素
} //O(n-r)

```

要先定义 `remove(Rank lo, Rank hi)`, 再定义 `remove(Rank r)`, 否则如果用后者实现前者, 会导致前者的复杂度为 $O(n^2)$ 。

2.2.1.5 查找

对于词条定义判等器和比较器, 在无需向量中、判等器即可, 有序向量中、可以定义比较器。

```

template <typename K, typename V> struct Entry { //词条模板类
    K key; V value; //关键码、数值
    Entry ( K k = K(), V v = V() ) : key ( k ), value ( v ) {}; //默认构造函数
    Entry ( Entry<K, V> const& e ) : key ( e.key ), value ( e.value ) {}; //克隆
    bool operator== ( Entry<K, V> const& e ) { return key == e.key; } //等于
    bool operator!= ( Entry<K, V> const& e ) { return key != e.key; } //不等于
    bool operator< ( Entry<K, V> const& e ) { return key < e.key; } //小于
    bool operator> ( Entry<K, V> const& e ) { return key > e.key; } //大于
}; //得益于比较器和判等器, 从此往后, 不必严格区分词条及其对应的关键码

```

查找在无序向量中只能采用顺序查找:

```

template <typename T> Rank Vector<T>:: //O(hi - lo) = O(n)
    find( T const & e, Rank lo, Rank hi ) const { //0 <= lo < hi <= _size
        while ( (lo < hi--) && (e != _elem[hi]) );
        return hi; //返回值小于 lo 即意味着失败; 否则即命中者的秩(有多个时, 返回最大者)
    }

```

复杂度为 $O(n)$ 。

2.2.1.6 去重

```

template <typename T> Rank Vector<T>::dedup() {
    Rank oldSize = _size;
    for ( Rank i = 1; i < _size; )
        if ( -1 == find( _elem[i], 0, i ) ) //O(i)
            i++;
        else
            remove(i); //O(_size - i)
    return oldSize - _size;
} //O(n^2): 对于每一个 e, 只要 find() 不是最坏情况(查找成功), 则 remove() 必执行

```

只要既运行 `find()` 又运行 `remove()`, 这一次严格 n 此操作, 从而复杂度为 $O(n^2)$ 。

2.2.1.7 遍历

对向量中的每一元素，统一实施 `visit()` 操作，利用函数指针或者函数对象

```
template <typename T> //函数指针，只读或局部性修改
void Vector<T>::traverse( void ( * visit )( T & ) )
{ for ( Rank i = 0; i < _size; i++ ) visit( _elem[i] ); }

template <typename T> template <typename VST> //函数对象，全局性修改更便捷
void Vector<T>::traverse( VST & visit )
{ for ( Rank i = 0; i < _size; i++ ) visit( _elem[i] ); }
```

例如遍历加一：

```
/* 先实现一个可使单个 T 类型元素加一的类（结构） */
template <typename T> //假设 T 可直接递增或已重载操作符“++”
struct Increase //函数对象：通过重载操作符“()”实现
{ virtual void operator()( T & e ) { e++; } }; //加一
/* 再将其作为参数传递给遍历算法 */
template <typename T> void increase( Vector<T> & V )
{ V.traverse( Increase<T>() ); } //即可以之作为基本操作，遍历向量
```

2.3 有序向量：二分搜索

2.3.1 有序性

通过计算逆序对的数量，可以统计向量的有序性：

```
template <typename T> void checkOrder ( Vector<T> & V ) { //通过遍历
    int unsorted = 0; V.traverse( CheckOrder<T>(unsorted, V[0]) ); //统计紧邻逆
    序对
    if ( 0 < unsorted )
        printf ( "Unsorted with %d adjacent inversion(s)\n", unsorted );
    else
        printf ( "Sorted\n" );
}
```

2.3.2 基本操作

2.3.2.1 唯一化

```
/* 勤奋的低效算法 */
template <typename T> int Vector<T>::uniquify() {
    int oldSize = _size; int i = 1;
    while ( i < _size )
        _elem[i-1] == _elem[i] ? remove( i ) : i++;
    return oldSize - _size;
}
/* 懒惰的高效算法： Two-Pointer Technique */
template <typename T> int Vector<T>::uniquify() {
    Rank i = 0, j = 0;
    while ( ++j < _size )
```

```

        if ( _elem[ i ] != _elem[ j ] )
            _elem[ ++i ] = _elem[ j ]; //可能徒劳无益
    _size = ++i;
    shrink();
    return j - i;
}

```

直接就地改写，而不是先统计再删除，这样可以避免多余的删除操作。

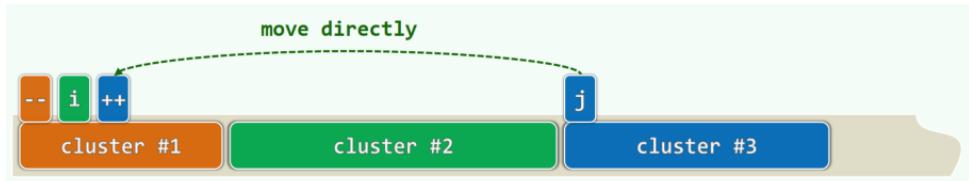


图 15 唯一化算法

2.3.2.2 查找

```

template <typename T> //查找算法统一接口, 0 <= lo < hi <= _size
Rank Vector<T>::search( T const & e, Rank lo, Rank hi ) const {
    return ( rand() % 2 ) ? //等概率地随机选用
        binSearch( _elem, e, lo, hi ) //二分查找算法, 或
        : fibSearch( _elem, e, lo, hi ); //Fibonacci 查找算法
}

```

对于有序向量，利用序性，可以加速查找。最常见的方法就是二分查找。

二分查找的想法就是分治：将查找区间一分为二，然后判断目标元素在哪一部分，然后递归地在该部分中查找。二分查找的复杂度是 $O(\log n)$ 。

2.3.2.2.1 版本 A

第一种实现方式是分成三部分，如果命中，返回秩；如果小于，递归地在左侧查找；如果大于，递归地在右侧查找。

```

template <typename T> //在有序向量[lo, hi)区间内查找元素 e
static Rank binSearch( T * S, T const & e, Rank lo, Rank hi ) {
    while ( lo < hi ) { //每步迭代可能要做两次比较判断，有三个分支
        Rank mi = ( lo + hi ) >> 1; //以中点为轴点（区间宽度折半，其数值表示右移一位）
        if ( e < S[mi] ) hi = mi; //深入前半段[lo, mi)
        else if ( S[mi] < e ) lo = mi + 1; //深入后半段(mi, hi)
        else return mi; //命中
    }
    return -1; //失败
}

```

每个元素都是轴点，每步迭代可能要做两次比较判断，有三个分支。可以分析关键码的比较次数，即查找长度（search length）：

需分别针对成功与失败查找，从最好、最坏、平均等角度评估，这种实现方式的成功、失败时的平均查找长度均大致为 $O(1.5 \log n)$ 。

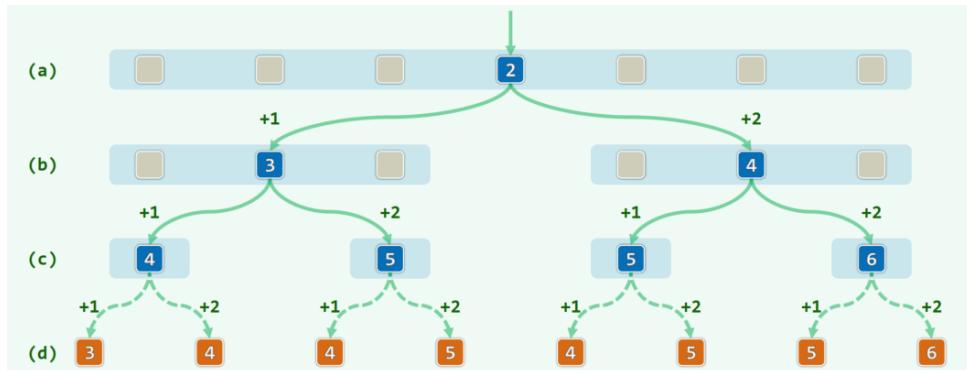


图 16 二分查找算法的查找长度

2.3.2.2.2 版本 B

二分查找中左、右分支转向代价不平衡的问题，也可直接解决，每次迭代仅做 1 次关键码比较；如此，所有分支只有 2 个方向，而不再是 3 个。

同样地，轴点 mi 取作中点，则查找每深入一层，问题规模依然会缩减一半

- $e < x$: 则深入左侧的 $[lo, mi]$
- $x <= e$: 则深入右侧的 $[mi, hi]$

直到 $hi - lo = 1$ ，才明确判断是否命中。

相对于版本 A，最好（坏）情况下更坏（好），整体性能更趋均衡。

```
template <typename T>
static Rank binSearch( T * S, T const & e, Rank lo, Rank hi ) {
    while ( 1 < hi - lo ) { //有效查找区间的宽度缩短至 1 时，算法才终止
        Rank mi = (lo + hi) >> 1; //以中点为轴点，经比较后确定深入[lo, mi]或[mi,
        hi)
        e < S[mi] ? hi = mi : lo = mi;
    } //出口时 hi = lo + 1
    return e == S[lo] ? lo : -1 ;
} // 返回命中处的秩， 或失败标志
```

返回值可能不统一：

- 目标元素不存在
- 目标元素同时存在多个，可能返回的不是最后一个，对于 $V.insert(1 + V.search(e), e)$ 就不是合法的插入位置

希望做到

- 即便失败，也应给出新元素可安置的位置（有序性）
- 若有重复元素，也需按其插入的次序排列（稳定性）

返回值的语义扩充： $m = \text{search}(e) = M-1$ ，其中 $-\infty < m = \max\{k | S[k] \leq e\}$ ，

$M = \min\{k | e < S[k]\} \leq +\infty$ ，则

- 若查找成功，返回最后一个目标元素的秩
- 若查找失败，返回比它小的最大元素的秩

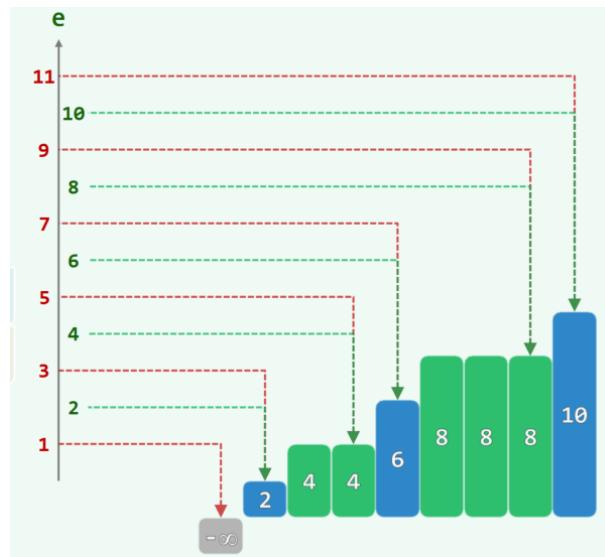


图 17 二分查找算法返回值的约定

2.3.2.2.3 版本 C

```
template <typename T>
static Rank binSearch( T * S, T const & e, Rank lo, Rank hi ) {
    while ( lo < hi ) { //不变性: A[0, lo) <= e < A[hi, n)
        Rank mi = (lo + hi) >> 1;
        e < S[mi] ? hi = mi : lo = mi + 1; // [lo, mi) 或 (mi, hi), A[mi] 或被遗漏?
    } //出口时，区间宽度缩短至 0，且必有 S[lo = hi] = M
    return lo - 1; //至此，[lo] 为大于 e 的最小者，故[lo-1] = m 即为不大于 e 的最大者
} //留意与版本 B 的差异
```

这样就遵守了返回值的语义约定。

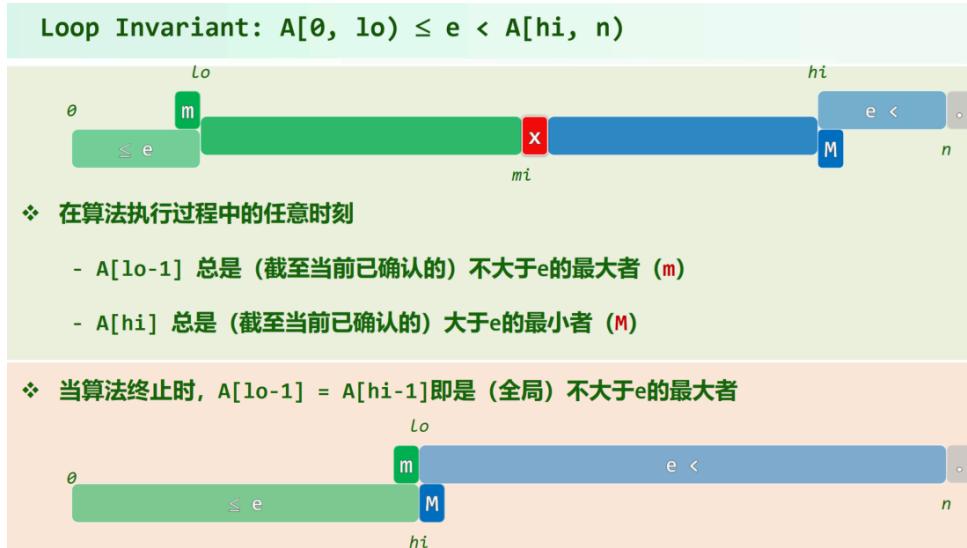


图 18 二分查找算法的正确性分析

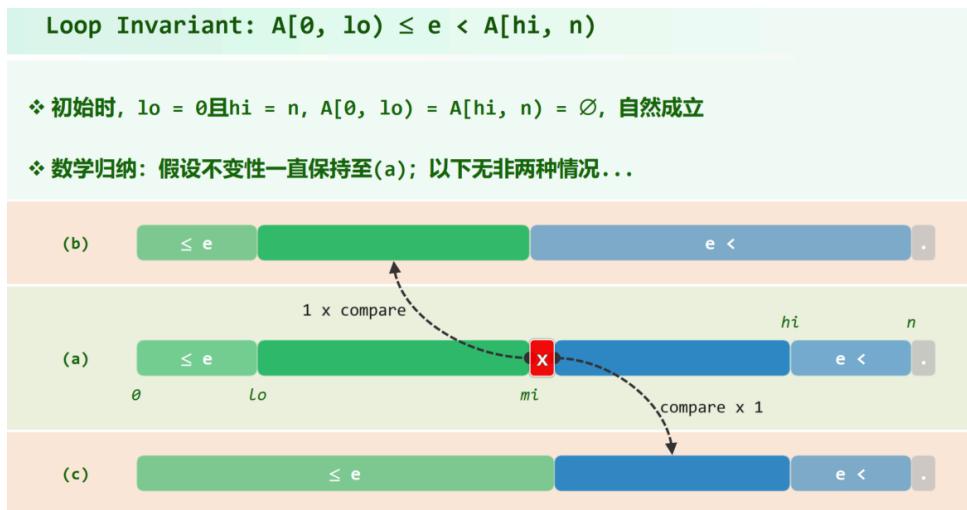


图 19 二分查找算法的正确性分析

2.3.2.2.4 插值查找

大数定律: 越长的序列, 元素的分布越有规律; 最为常见: 独立且均匀的随机分布。

于是 $[lo, hi]$ 内各元素应大致呈线性趋势增长, 因此通过猜测轴点 mi , 可以极大地提高收敛速度。

$$\frac{mi - lo}{hi - lo} = \frac{e - S[lo]}{S[hi] - S[lo]}$$

$$mi = lo + (hi - lo) \frac{e - S[lo]}{S[hi] - S[lo]}$$

最坏: $O(n)$, 平均: $O(\log \log n)$ 。

这是因为每次比较，待查找区间都会从宽度 n 缩短至 \sqrt{n} 。

每经一次比较，查找区间宽度的数值 n 开方，有效字长 $\log n$ 减半

- 插值查找 = 在字长意义上的折半查找 $\log(n^{\frac{1}{2}}) = 0.5 \log n$
- 二分查找 = 在字长意义上的顺序查找 $\log(\frac{n}{2}) = \log n - 1$

从 $O(\log n)$ 到 $O(\log \log n)$ ，优势并不明显

- 须引入乘法、除法运算
- 易受畸形分布的干扰和“蒙骗”
- 实际可行的方法：算法接力
 - 首先通过插值查找

迅速将查找范围缩小到一定的尺度

- 然后再改为二分查找
- 进一步缩小范围
 - 最后（当数据项只有 200~300 时）

改用顺序查找

2.3.3 排序

对于无序向量，可以排序成有序向量。

```
template <typename T> void Vector<T>::sort( Rank lo, Rank hi ) {
    switch ( rand() % 6 ) {
        case 1 : bubbleSort( lo, hi ); break; //起泡排序
        case 2 : selectionSort( lo, hi ); break; //选择排序
        case 3 : mergeSort( lo, hi ); break; //归并排序
        case 4 : heapSort( lo, hi ); break; //堆排序
        case 5 : quickSort( lo, hi ); break; //快速排序
        default : shellSort( lo, hi ); break; //希尔排序
    } //随机选择算法，以尽可能充分地测试。应用时可视具体问题的特点，灵活确定或扩充
}
```

2.4 起泡排序 Bubble Sort

反复地扫描交换：

- 观察：有序/无序序列中，任何/总有一对相邻元素顺序/逆序
- 扫描交换：依次比较每一对相邻元素；如有必要，交换之。直至某趟扫描后，确认相邻元素均已顺序。

其复杂度是 $O(n^2)$ ，但是可以通过优化来提高效率。

基本版：

```
template <typename T> void Vector<T>::bubbleSort( Rank lo, Rank hi ) {
    while ( lo < --hi ) //逐趟起泡扫描
        for ( Rank i = lo; i < hi; i++ ) //逐对检查相邻元素
```

```

        if ( _elem[i] > _elem[i + 1] ) //若逆序
            swap( _elem[i], _elem[i + 1] ); //则交换
    }

```

- Loop Invariant: 经 k 趟扫描交换后，最大的 k 个元素必然就位
- Convergence: 经 k 趟扫描交换后，问题规模缩减至 $n - k$
- Correctness: 经至多 n 趟扫描后，算法必然终止，且能给出正确解答

[hi]就位后，[lo, hi]可能已经有序（sorted）——此时，应该可以提前终止算法。

```

template <typename T> void Vector<T>::bubbleSort( Rank lo, Rank hi ) {
    for ( bool sorted = false; sorted = !sorted; hi-- )
        for ( Rank i = lo + 1; i < hi; i++ )
            if ( _elem[i-1] > _elem[i] )
                swap( _elem[i-1], _elem[i] ), sorted = false;
}

```

同样地，有可能某一后缀[last, hi]已然有序，继续优化：跳跃版

```

template <typename T> void Vector<T>::bubbleSort( Rank lo, Rank hi ) {
    for ( Rank last; lo < hi; hi = last )
        for ( Rank i = (last = lo) + 1; i < hi; i++ )
            if ( _elem[i-1] > _elem[i] )
                swap( _elem[i-1], _elem[i] ), last = i;
}

```

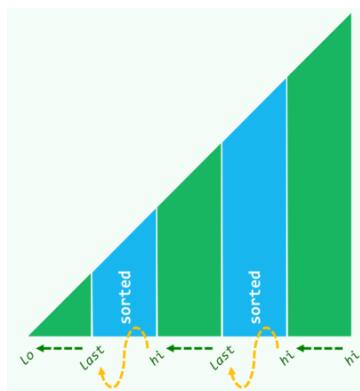


图 20 起泡排序算法——跳跃版

时间效率：最好 $O(n)$ ，最坏 $O(n^2)$ 输入含重复元素时，算法的稳定性（stability）是更为细致的要求。重复元素在输入、输出序列中的相对次序，需要保持不变。

- 输入： 6, 7a, 3, 2, 7b, 1, 5, 8, 7c, 4
- 输出：
 - 1, 2, 3, 4, 5, 6, 7a, 7b, 7c, 8 是 stable 的

- 1, 2, 3, 4, 5, 6, 7a, 7c, 7b, 8 是 unstable 的
- 起泡排序算法是稳定的，因为唯有相邻元素才可交换。

2.5 归并排序 Merge Sort

向量与列表通用的一种分而治之的排序方法。

- 序列一分为二
- 子序列递归排序
- 合并有序子序列

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

根据主定理，复杂度是 $O(n \log n)$ 。

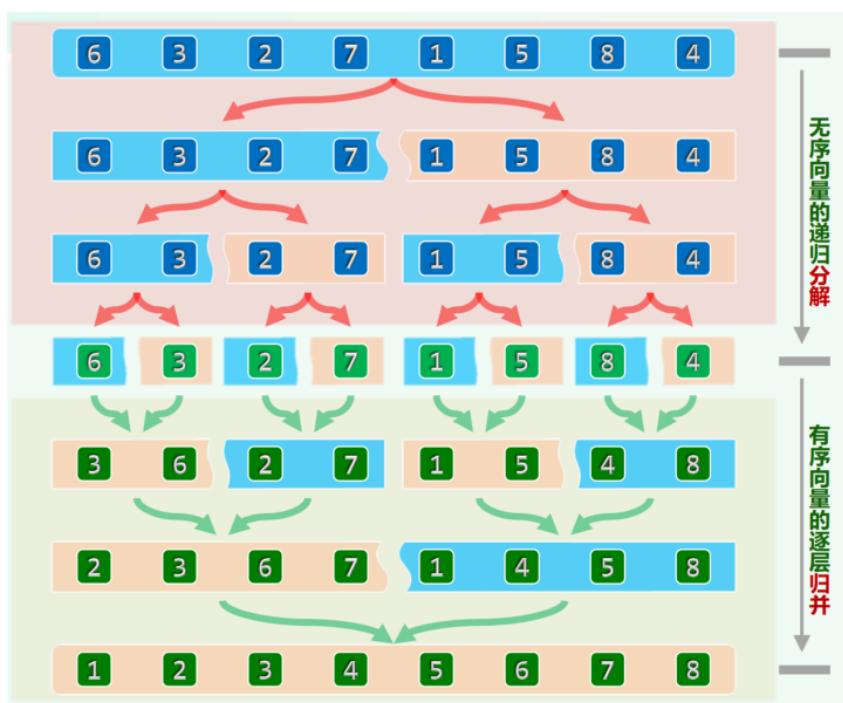


图 21 归并排序算法

```
template <typename T> void Vector<T>::mergeSort( Rank lo, Rank hi ) {
    if ( hi - lo < 2 ) return; // 单元素区间自然有序, 否则...
    Rank mi = (lo + hi) >> 1; // 以中点为界
    mergeSort( lo, mi ); // 对前半段排序
    mergeSort( mi, hi ); // 对后半段排序
    merge( lo, mi, hi ); // 归并
}
```

2.5.1 二路归并

2-way merge: 有序序列合二为一，保持有序

$$S[lo, mi) + S[mi, hi) \rightarrow S[lo, hi)$$

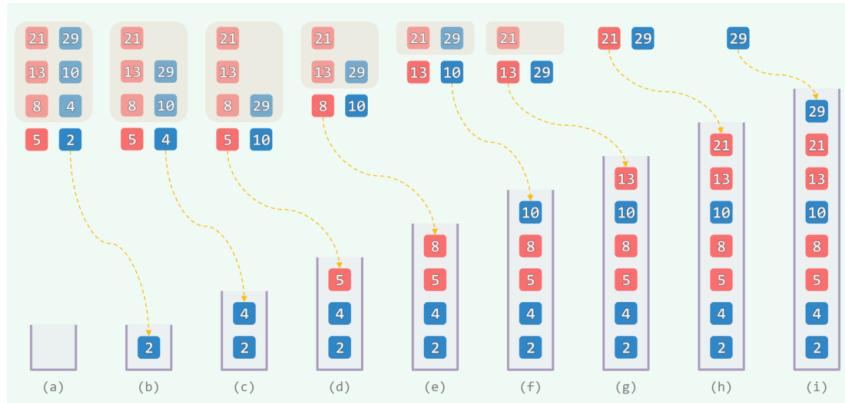


图 22 二路归并算法

```

template <typename T> // [lo, mi) 和 [mi, hi) 各自有序
void Vector<T>::merge( Rank lo, Rank mi, Rank hi ) { // lo < mi < hi
    Rank i = 0; T* A = _elem + lo; // A = _elem[lo, hi)
    Rank j = 0, lb = mi - lo; T* B = new T[lb]; // B[0, lb) <- _elem[lo, mi)
    for ( Rank i = 0; i < lb; i++ ) B[i] = A[i]; // 复制出 A 的前缀
    Rank k = 0, lc = hi - mi; T* C = _elem + mi;
    // 后缀 C[0, lc] = _elem[mi, hi), 就地
    while ( ( j < lb ) && ( k < lc ) ) // 反复地比较 B、C 的首元素
        A[i++] = ( B[j] <= C[k] ) ? B[j++] : C[k++];
    while ( j < lb ) // 若 C 先耗尽, 则
        A[i++] = B[j++];
    delete[] B; // new 和 delete 非常耗时, 如何减少?
}

```

只需要 $\frac{n}{2}$ 的辅助空间，这是因为保留后半段的，复制前半段的元素，每次选择元素后直接覆盖原数组，并不会覆盖到后半段未被复制的元素。需要 $O(n)$ 的时间复杂度。

2.5.2 复杂度

优点

- 实现最坏情况下最优 $O(n \log n)$ 性能的第一个排序算法
- 不需随机读写，完全顺序访问——尤其适用于列表之类的序列、磁带之类的设备
- 只要实现恰当，可保证稳定——出现雷同元素时，左侧子向量优先
- 可扩展性极佳，十分适宜于外部排序——海量网页搜索结果的归并
- 易于并行化

缺点

- 非就地，需要对等规模的辅助空间
- 即便输入已是完全（或接近）有序，仍需 $\Omega(n \log n)$ 时间

2.6 位图 Bitmap

对于有限整数集，直接用整数作为秩，可以将集合元素与秩一一对应，从而实现集合的快速查找。

```
class Bitmap {
private:
    unsigned char * M;
    Rank N, _sz;
public:
    Bitmap( Rank n = 8 )
        { M = new unsigned char[ N = (n+7)/8 ]; memset( M, 0, N ); _sz = 0; }
    ~Bitmap() { delete [] M; M = NULL; _sz = 0; }
    void set( int k );
    void clear( int k );
    bool test( int k );
};
```

可以精简到就用 1 位表示 true/false，这样一个 8 位的 `unsigned char` 就可以表示 8 个元素。

```
bool test( int k ) { expand( k ); return M[ k >> 3 ] & ( 0x80 >> (k & 0x07) );
}
void set( int k ) { expand( k ); _sz++; M[ k >> 3 ] |= ( 0x80 >> (k & 0x07) );
}
void clear( int k ) { expand( k ); _sz--; M[ k >> 3 ] &= ~( 0x80 >> (k & 0x07) );
}
```

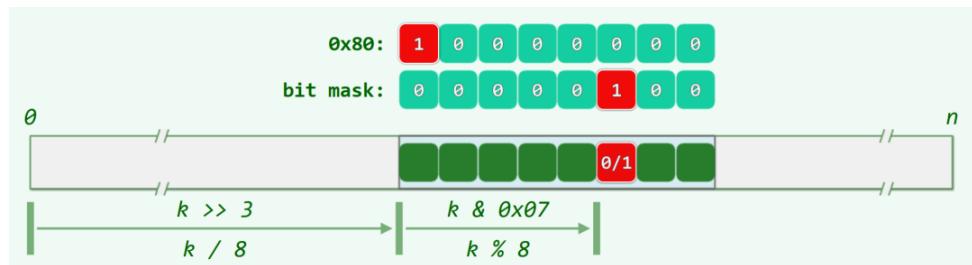


图 23 位图的实现

利用掩码 `0x80`，可以将 `M[k >> 3]` 的第 `k & 0x07`（是模余 8，找到掩码移位数）位取出来，然后与 `M[k >> 3]`（是寻找到对应的 8 位 `unsigned char`）进行与或非操作，从而实现对位的操作。

2.6.1 应用

小集合 + 大数据：`int A[n]` 的元素均取自 $[0, m)$ ，如何剔除其中的重复者？

仿照 `Vector::dedup()` 改进版，先排序，再扫描 $O(n \log n + n)$ 。

但对于大规模的数据，即便能够申请到这么多空间，频繁的 I/O 也将导致整体效率的低下。

利用位图可以有 $O(n + m)$ 的解决方法。

Eratosthenes 筛法：

从 2 开始，将所有 2 的倍数划去；再从 3 开始，将所有 3 的倍数划去；再从 5 开始，将所有 5 的倍数划去；……。

```
void Eratosthenes( Rank n, char * file ) {
    Bitmap B( n ); B.set( 0 ); B.set( 1 );
    for ( Rank i = 2; i < n; i++ )
        if ( ! B.test( i ) )
            for ( Rank j = 2*i; j < n; j += i )
                B.set( j );
    B.dump( file );
}
```

效率：不计内循环，外循环自身每次仅一次加法、两次判断，累计 $O(n)$ ；内循环每次 $O(\frac{n}{i})$ ，由素数定理，外循环至多 $\frac{n}{\log n}$ 次，累计是 $O(n \log n)$ 。

优化：内循环的起点 $2*i$ 可改作 $i*i$ ；外循环的终止条件 $i < n$ 可改作 $i*i < n$ 。这样内循环每次迭代 $O(\max(1, \frac{n}{i} - i))$ ，外循环至多 $\frac{\sqrt{n}}{\log \sqrt{n}}$ 次。

2.6.2 快速初始化

Bitmap 的构造函数中，通过 `memset(M, 0, N)` 统一清零，需要 $O(n)$ 时间。成为位图最大的瓶颈。

有时，对于大规模的散列表，初始化的效率直接影响到实际性能

例如：字符串中(后续章节讲到)bc[]表的构造算法，需要 $O(|\Sigma| + m) = O(s + m)$ 时间，若能省去 bc[]表各项的初始化，则可严格地保证是 $O(m)$ 。

这时候可以用校验环策略：

将 B[]拆分成一对等长向量：Rank F[m]，T[m]，top = 0；

构成校验环：T[F[k]] == k & F[T[i]] == i

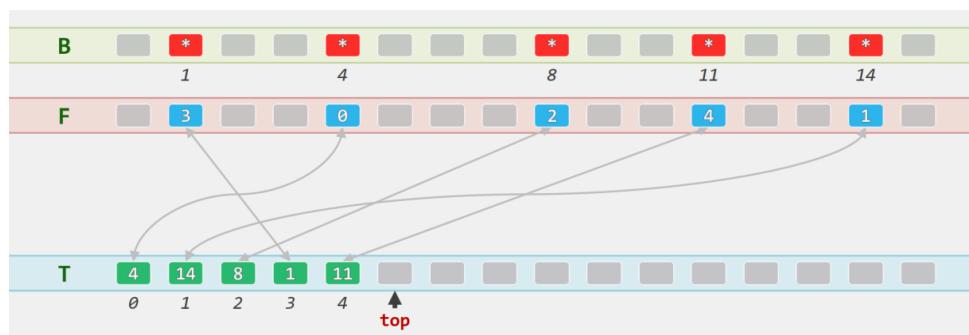


图 24 校验环

检验：

```
bool Bitmap::test( Rank k ) { return (0 <= F[k]) && (F[k] < top) && (k == T[F[k]]); }
```

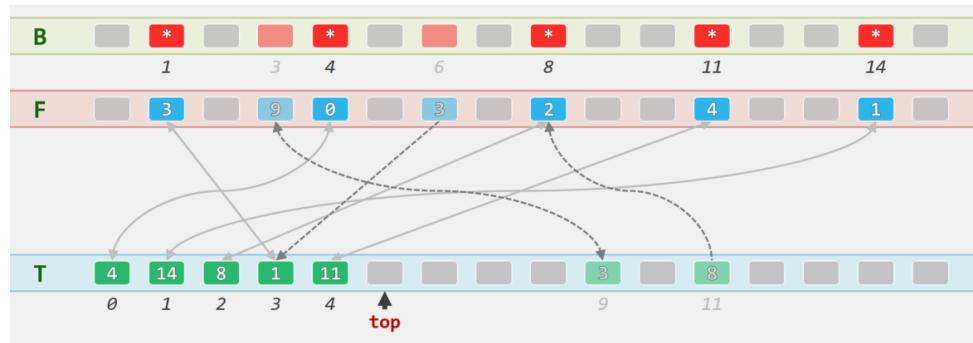


图 25 校验环——检验

复位: $O(1)$

```
void Bitmap::reset() { top = 0; }
```

插入: $O(1)$

```
void Bitmap::set( Rank k ) { if ( !test( k ) ) { T[top] = k; F[k] = top++; } }
```

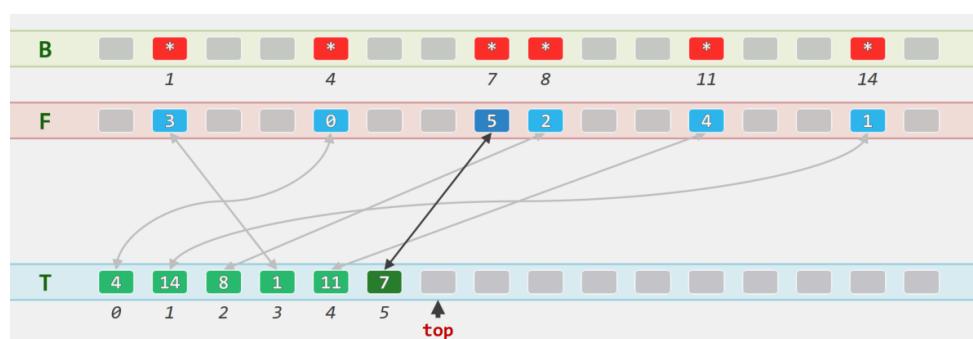


图 26 校验环——插入

删除: $O(1)$

```
void Bitmap::clear( Rank k )
{ if ( test( k ) && ( --top ) ) { F[T[top]] = F[k]; T[F[k]] = T[top]; } }
```

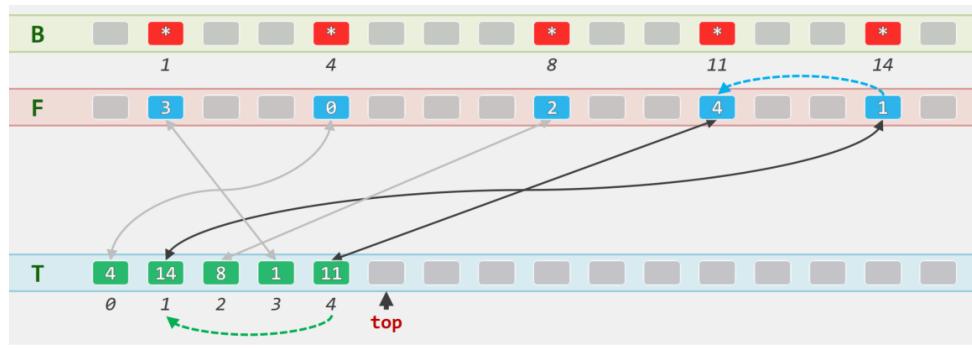


图 27 校验环——删除

删除的时候将最后一个元素放到被删除元素的位置，然后更新校验环。

三 列表 List

根据是否修改数据结构，所有操作大致分为两类方式

- 静态：仅读取，数据结构的内容及组成一般不变：`get`、`search`
- 动态：需写入，数据结构的局部或整体将改变：`put`、`insert`、`remove`

与操作方式相对应地，数据元素的存储与组织方式也分为两种

- 静态：数据空间整体创建或销毁

数据元素的物理次序与其逻辑次序严格一致；可支持高效的静态操作；比如向量，元素的物理地址与其逻辑次序线性对应

- 动态：为各数据元素动态地分配和回收的物理空间

相邻元素记录彼此的物理地址，在逻辑上形成一个整体；可支持高效的动态操作

列表（list）是采用动态储存策略的典型结构

- 其中的元素称作节点（node），通过指针或引用彼此联接
- 在逻辑上构成一个线性序列
- 相邻节点彼此互称前驱（predecessor）或后继（successor）
- 没有前驱/后继的节点称作首（first/front）/末（last/rear）节点
- 循位置访问(call by position)

ListNode 接口

```
pred() / succ() // 当前节点前驱/后继节点的位置
data() // 当前节点所存数据对象
insertAsPred() / insertAsSucc() // 插入前驱/后继节点，返回新节点位置
```

```
template <typename T> using ListNodePosi = ListNode<T>*; //列表节点位置 (C++.0x)
template <typename T> struct ListNode { //简洁起见，完全开放而不再严格封装
    T data; //数值
    ListNodePosi<T> pred; //前驱
    ListNodePosi<T> succ; //后继
    ListNode() {} //针对 header 和 trailer 的构造
    ListNode(T e, ListNodePosi<T> p = NULL, ListNodePosi<T> s = NULL)
        : data(e), pred(p), succ(s) {} //默认构造器
    ListNodePosi<T> insertAsPred( T const & e ); //前插入
    ListNodePosi<T> insertAsSucc( T const & e ); //后插入
};
```

List 接口

```
size() / empty() // 报告节点总数 / 判定是否为空 列表
first() / last() // 返回首 / 末节点的位置 列表
```

```
insertAsFirst(e) / insertAsLast(e) // 将 e 当作首 / 末节点插入 列表
insert(p, e), insert(e, p) // 将 e 当作节点 p 的直接后继、前驱插入 列表
remove(p) // 删除节点 p 列表
sort(p, n) / sort() // 区间 / 整体排序 列表
find(e, n, p) / search(e, n, p) // 在指定区间内查找目标 e 列表 / 有序列表
dedup() / uniquify() // 剔除重复节点 列表 / 有序列表
traverse( visit() ) // 遍历列表，统一按 visit() 处理所有节点 列表
```

```
template <typename T> class List { //列表模板类
private: Rank _size; ListNodePosi<T> header, trailer; //哨兵
//头、首、末、尾节点的秩，可分别理解为-1、0、n-1、n
protected: /* ... 内部函数 */
public: /* ... 构造函数、析构函数、只读接口、可写接口、遍历接口 */
};
```

构造

```
template <typename T> void List<T>::init() { //初始化，创建列表对象时统一调用
    header = new ListNode<T>;
    trailer = new ListNode<T>;
    header->succ = trailer; header->pred = NULL;
    trailer->pred = header; trailer->succ = NULL;
    _size = 0;
}
```

访问：重载下标操作符，可模仿向量的循秩访问方式

```
template <typename T> //O(r)效率，虽方便，勿多用
ListNodePosi<T> List<T>::operator[]( Rank r ) const { //0 <= r < size
    ListNodePosi<T> p = first(); //从首节点出发
    while ( 0 < r-- ) p = p->succ; //顺数第 r 个节点即是
    return p; //目标节点
} //秩 == 前驱的总数
```

时间复杂度为 $O(r)$ ，均匀分布时期望为 $O(n)$ 。

3.1 无序列表

3.1.1 插入与删除

实现是容易的，就是修改指针指向的问题，但是要注意的是，插入的时候，要先修改前驱的后继，再修改后继的前驱，删除的时候，先修改前驱的后继，再修改后继的前驱，最后删除节点。

通过重载性质，直接从函数声明上区分前插入、后插入

```
template <typename T> ListNodePosi<T> List<T>:: //e 当作 p 的前驱插入
insert(T const & e, ListNodePosi<T> p) { _size++; return p-
```

```
>insertAsPred( e ); }

template <typename T> //前插入算法（后插入算法完全对称）
ListNodePosi<T> ListNode<T>::insertAsPred( T const & e ) { //O(1)
    ListNodePosi<T> x = new ListNode( e, pred, this ); //创建
    pred->succ = x; pred = x; //次序不可颠倒
    return x; //建立链接，返回新节点的位置
} //得益于哨兵，即便 this 为首先节点亦不必特殊处理—此时等效于 insertAsFirst(e)
```

```
template <typename T> T List<T>::remove( ListNodePosi<T> p ) { //删除合法节点 p
    T e = p->data; //备份待删除节点存放的数值（设类型 T 可直接赋值）
    p->pred->succ = p->succ; p->succ->pred = p->pred; //短路联接
    delete p; _size--; return e; //返回备份的数值
} //O(1)
```

3.1.2 构造与析构

copyNodes() + 构造

```
template <typename T> void List<T>::copyNodes( ListNodePosi<T> p, Rank n )
{ //O(n)
    init(); //创建头、尾哨兵节点并做初始化
    while ( n-- ) { //将起自 p 的 n 项依次作为末节点
        insertAsLast( p->data ); //插入
        p = p->succ;
    }
}

List<T>::List( List<T> const & L ) { copyNodes( L.first(), L._size ); }
```

clear() + 析构

```
template <typename T> List<T>::~List() //列表析构
{ clear(); delete header; delete trailer; } //清空列表，释放头、尾哨兵节点

template <typename T> Rank List<T>::clear() { //清空列表
    Rank oldSize = _size;
    while ( 0 < _size ) //反复
        remove( header->succ ); //删除首先节点， O(n)
    return oldSize;
}
```

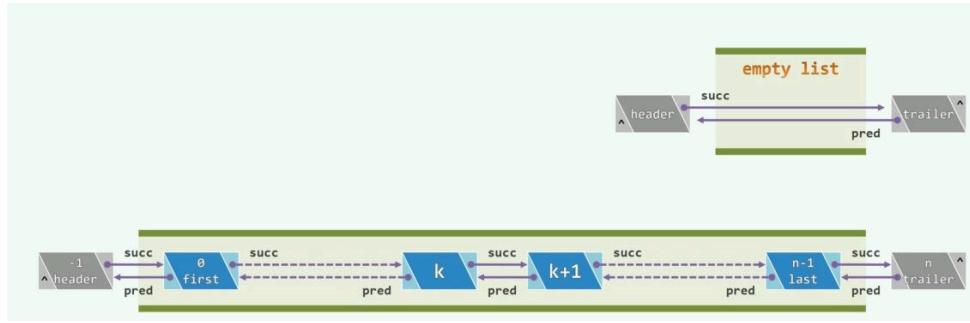


图 28 列表的构造与析构

3.1.3 查找与去重

无序的查找也只能 $O(n)$ 。

```
template <typename T> //0 <= n <= rank(p) < _size
ListNodePosi<T> List<T>::find( T const & e, Rank n, ListNodePosi<T> p ) const
{
    while ( 0 < n-- ) //自后向前
        if ( e == ( p = p->pred ) ->data ) //逐个比对 (假定类型 T 已重载“==”)
            return p; //在 p 的 n 个前驱中, 等于 e 的最靠后者
    return NULL; //失败
} //O(n)

template <typename T>
ListNodePosi<T> find( T const & e ) const { return find( e, _size,
trailer ); }
```

去重

```
template <typename T> Rank List<T>::dedup() {
    Rank oldSize = _size;
    ListNodePosi<T> p = first();
    for ( Rank r = 0; p != trailer; p = p->succ ) //O(n)
        if ( ListNodePosi<T> q = find( p->data, r, p ) ) //O(n)
            remove( q );
        else
            r++;
    return oldSize - _size; //删除元素总数
} //正确性及效率分析的方法与结论, 与 Vector::dedup() 相同
```

3.1.4 遍历

一样地, 利用函数对象或者函数指针。

```
template <typename T> void List<T>::traverse( void ( * visit )( T & ) )
{ for( NodePosi<T> p = header->succ; p != trailer; p = p->succ ) visit( p-
>data ); }
```

```
template <typename T> template <typename VST> void List<T>::traverse( VST &
visit )
{ for( NodePosi<T> p = header->succ; p != trailer; p = p->succ ) visit( p-
>data ); }
```

3.2 有序列表

3.2.1 唯一化

直接一趟线性扫描即可，时间复杂度为 $O(n)$ 。

```
template <typename T> Rank List<T>::uniquify() {
    if ( _size < 2 ) return 0; //平凡列表自然无重复
    Rank oldSize = _size; //记录原规模
    ListNodePosi<T> p = first(); ListNodePosi<T> q; //各区段起点及其直接后继
    while ( trailer != ( q = p->succ ) ) //反复考查紧邻的节点对(p,q)
        if ( p->data != q->data ) p = q; //若互异，则转向下一对
        else remove(q); //否则（雷同） 直接删除后者，不必如向量那样间接地完成删除
    return oldSize - _size; //规模变化量，即被删除元素总数
} //只需遍历整个列表一趟， O(n)
```

3.2.2 查找

`search`相较于`find`，在有序结构中，约定语义，希望能返回失败的位置，例如返回失败时左边界前驱。

```
template <typename T> //在有序列表内节点 p 的 n 个真前驱中，找到不大于 e 的最靠后者
ListNodePosi<T> List<T>::search( T const & e, Rank n, ListNodePosi<T> p )
const {
    do { //初始有： 0 <= n <= rank(p) < _size; 此后， n 总是等于 p 在查找区间内的秩
        p = p->pred; n--;
    } while ( ( -1 != n ) && ( e < p->data ) ); //逐个比较，直至越界或命中
    return p; //最终停止的位置；失败时为区间左边界前驱（可能就是 header）
} //调用者可据此判断查找是否成功
```

最好 $O(1)$ ，最坏 $O(n)$ ；等概率时平均 $O(n)$ ，正比于区间宽度。

3.3 选择排序 Selection Sort

对于起泡排序，每次的效果是将最大的元素放到最后。不必经历那么多循环，直接找到最大的元素，放到最后即可。

```
template <typename T> void List<T>::selectionSort( ListNodePosi<T> p, Rank n )
{
    ListNodePosi<T> head = p->pred, tail = p;
    for ( Rank i = 0; i < n; i++ ) tail = tail->succ; //待排序区间为(head,
tail)
```

```

    while ( 1 < n ) { //反复从（非平凡）待排序区间内找出最大者，并移至有序区间前端
        insert( remove( selectMax( head->succ, n ) ), tail ); //可能就在原地...
        tail = tail->pred; n--;
    }
}

```

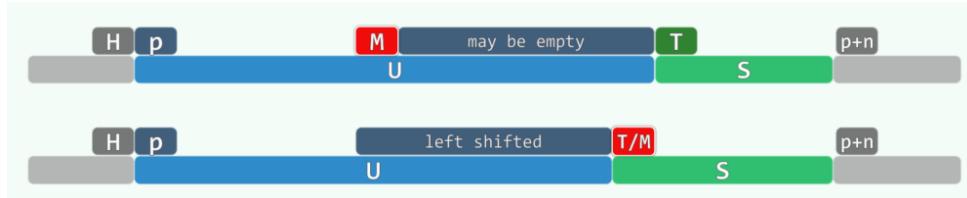


图 29 选择排序

只能线性搜索，向前试探

```

template <typename T> //从起始于位置 p 的 n 个元素中选出最大者, 1 < n
ListNodePosi<T> List<T>::selectMax( ListNodePosi<T> p, Rank n ) { //O(n)
    ListNodePosi<T> max = p; //最大者暂定为 p
    for ( ListNodePosi<T> cur = p; 1 < n; n-- ) //后续节点逐一与 max 比较
        if ( ! lt( (cur = cur->succ)->data, max->data ) ) //data ≦ max
            max = cur; //则更新最大元素位置记录
    return max; //返回最大节点位置
}

```

稳定性: 有多个元素同时命中时，约定返回其中特定的某一个（比如最靠后者）。若采用平移法，如此即可保证，重复元素在列表中的相对次序，与其插入次序一致。

复杂度是 $\Theta(n^2)$ ，但是比起起泡排序，减少了交换次数，因此效率更高。如果采用堆优化，可以将复杂度降低到 $O(n \log n)$ ，但是丧失了稳定性。

3.4 插入排序 Insertion Sort

始终将序列视作两部分：

- 前缀 $S[0, r)$: 有序
- 后缀 $U[r, n)$: 待排序

初始化： $|S| = r = 0$

反复地，针对 $e = A[r]$

- 在 S 中查找适当位置
- 插入 e
- $r++$

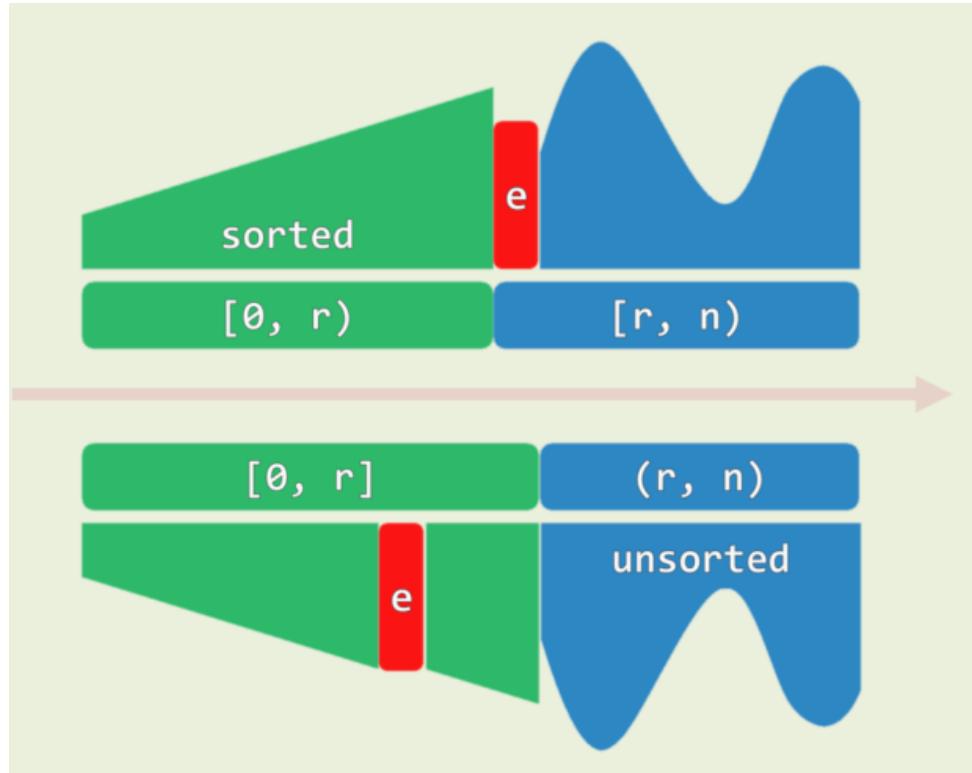


图 30 插入排序

核心想法是减而之治。

```
template <typename T> void List<T>::insertionSort( ListNodePosi<T> p, Rank n )
{
    for ( Rank r = 0; r < n; r++ ) { //逐一引入各节点, 由 Sr 得到 Sr+1
        insert( search( p->data, r, p ), p->data ); //查找 + 插入
        p = p->succ; remove( p->pred ); //转向下一节点
    } //n 次迭代, 每次 O(r + 1)
} //仅使用 O(1) 辅助空间, 属于就地算法
```

- 得益于此前约定的 `search()` 接口语义, 前缀的确总是保持有序, 而且稳定
- 复杂度, 最好 $O(n)$, 最坏 $O(n^2)$, 平均:

若前缀已经有序, 对于随机的下一个元素, 插入位置是均等的, 则有

$$1 + \sum_{k=0}^r \frac{k}{r+1} = 1 + \frac{1}{r+1}$$

从而总体复杂度是 $O(n^2)$ 。

- 可简明度量有序/乱序的程度与时间成本之成正比
- 输入敏感性/input-sensitivity
- 在线 online:** 在数据完全就绪之前, 即可开始计算

3.5 归并排序 Merge Sort

就像向量中的二路归并排序一样，将列表分为两部分，分别排序，然后合并。

```
template <typename T> void List<T>::mergeSort( ListNodePosi<T> & p, Rank n ) {
    if ( n < 2 ) return; //待排序范围足够小时直接返回，否则...
    ListNodePosi<T> q = p; Rank m = n >> 1; //以中点为界
    for ( Rank i = 0; i < m; i++ ) q = q->succ; //均分列表: O(m) = O(n)
    mergeSort( p, m ); mergeSort( q, n - m ); //子序列分别排序
    p = merge( p, m, *this, q, n - m ); //归并
} //若归并可在线性时间内完成，则总体运行时间亦为 O(nlogn)
```

```
template <typename T> ListNodePosi<T> //this.[p +n) & L.[q +m): 归并排序时, L
== this
List<T>::merge( ListNodePosi<T> p, Rank n, List<T>& L, ListNodePosi<T> q, Rank
m ) {
    ListNodePosi<T> pp = p->pred; //归并之后 p 或不再指向首节点，故需先记忆，以便返
    回前更新
    while ( ( 0 < m ) && ( q != p ) ) //小者优先归入
        if ( ( 0 < n ) && ( p->data <= q->data ) ) { p = p->succ; n--; } //p 直
        接后移
        else { insert( L.remove( ( q = q->succ )->pred ), p ); m--; } //q 转至
        p 之前
    return pp->succ; //更新的首节点
} //运行时间 O(n + m)，线性正比于节点总数
```

良好的搜索语义保证了稳定性。

3.6 游标实现

有些语言不支持指针类型，可以利用线性数组，以游标方式模拟列表

- elem[]: 对外可见的数据项
- link[]: 数据项之间的引用

维护逻辑上互补的列表 data 和 free

下图中 **data** 箭头所指的地方是列表的头，**free** 箭头所指的地方是空闲的头。其左侧 **elem[]** 对应的指针就是他的后继。

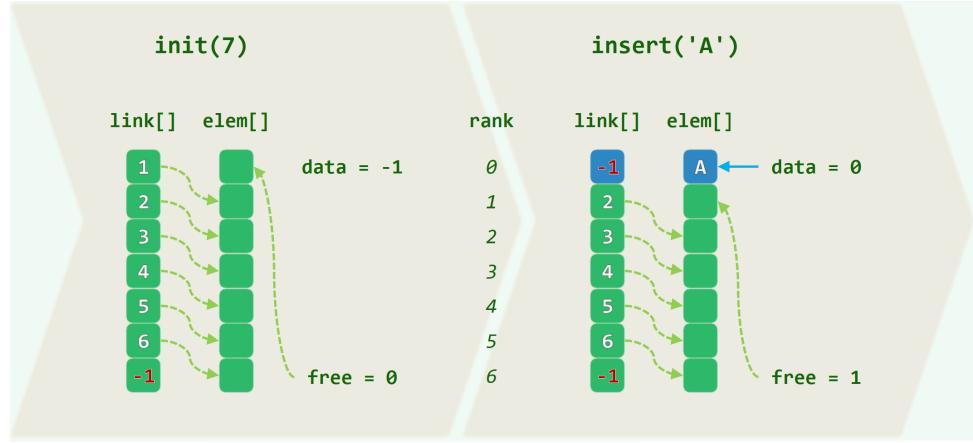


图 31 游标实现

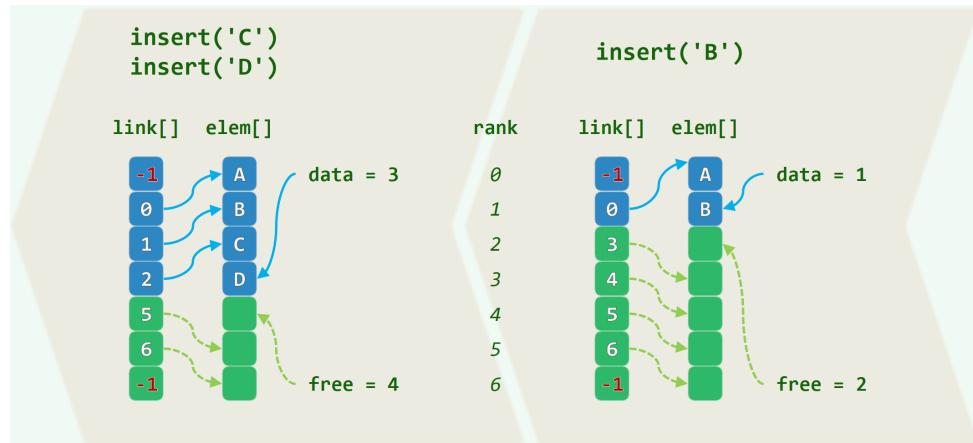


图 32 游标实现

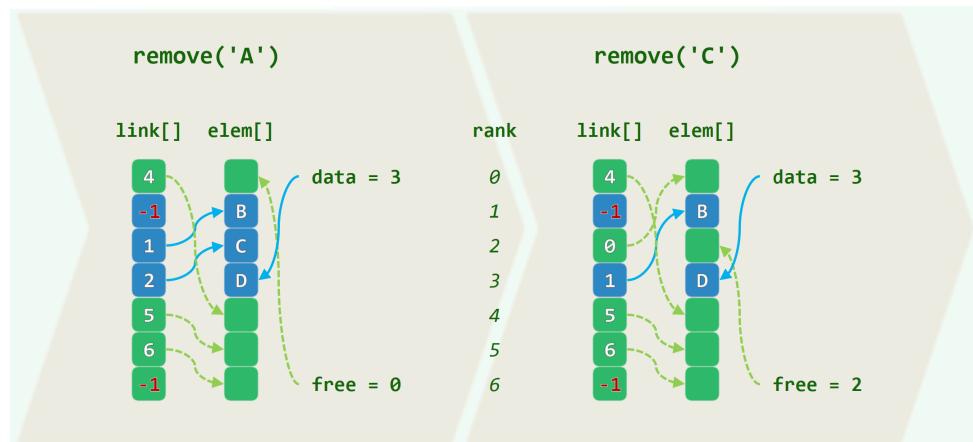


图 33 游标实现

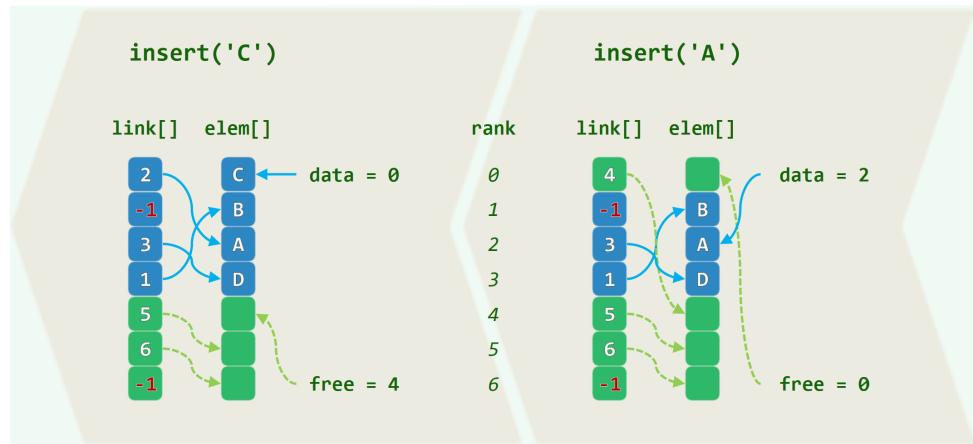


图 34 游标实现

四 栈与队列 Stack & Queue

4.1 栈 Stack

4.1.1 接口与实现

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)

基本接口是

- `size()` / `empty()`
- `push()` 入栈
- `pop()` 出栈
- `top()` 查顶
- 扩展接口: `getMax()`...

直接基于向量或列表派生

```
template <typename T> class Stack: public Vector<T> { //原有接口一概沿用
public:
    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?
```

如此实现的栈各接口, 均只需 $O(1)$ 时间

4.1.2 调用栈

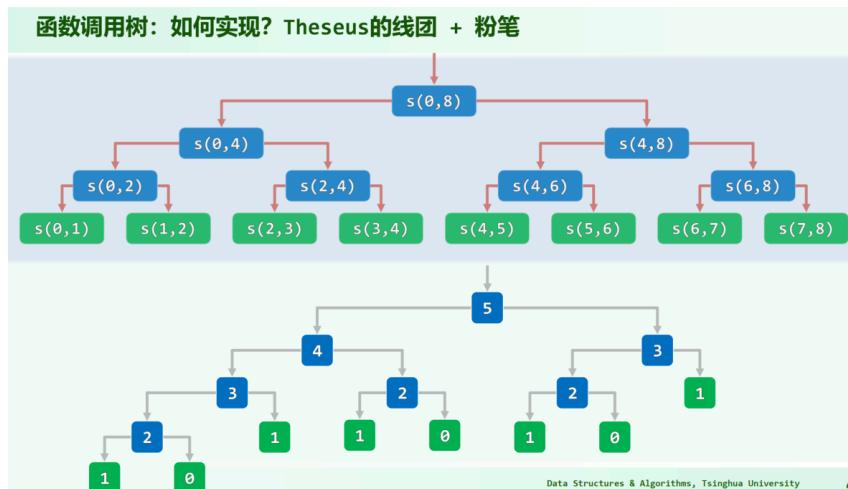


图 35 调用栈

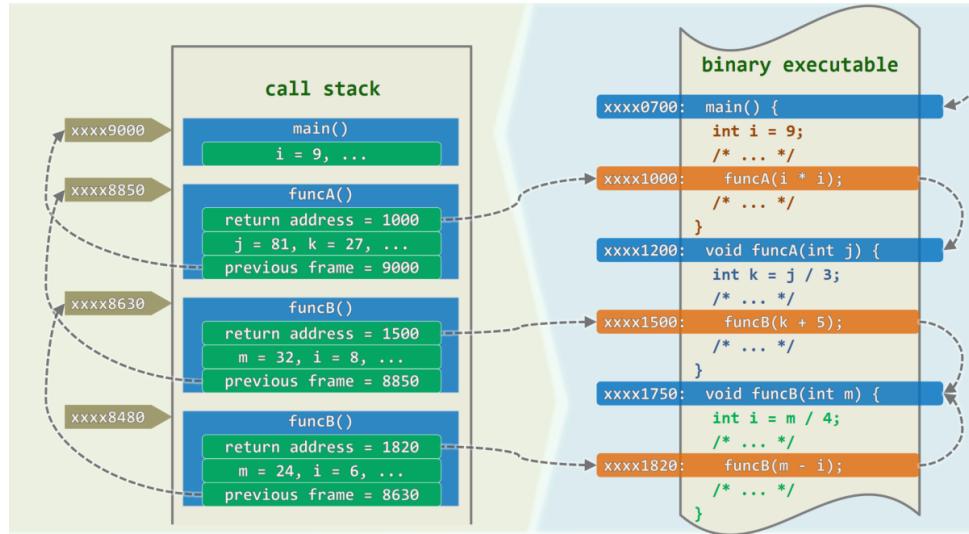


图 36 调用栈

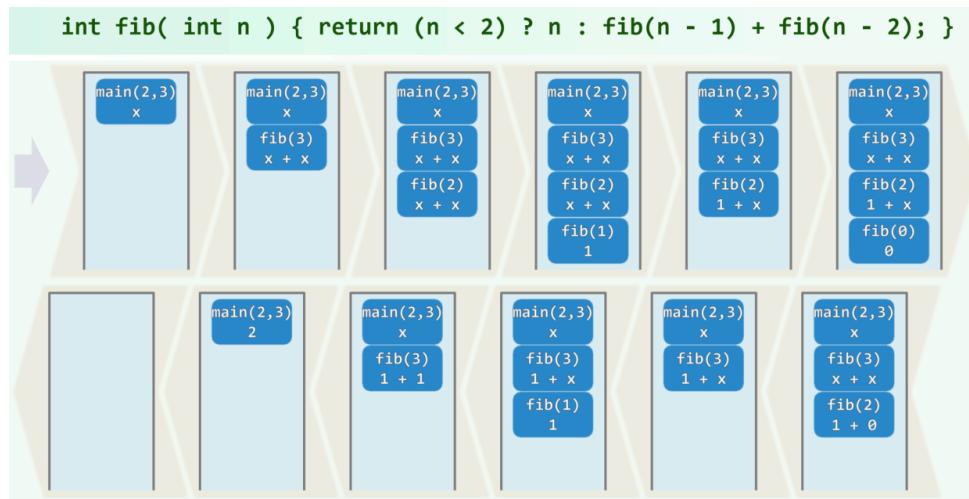


图 37 调用栈——递归深度

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。

4.1.2.1 消除递归

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

```

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

```

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。

4.1.2.2 尾递归

尾递归是在递归实例中，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
}

```

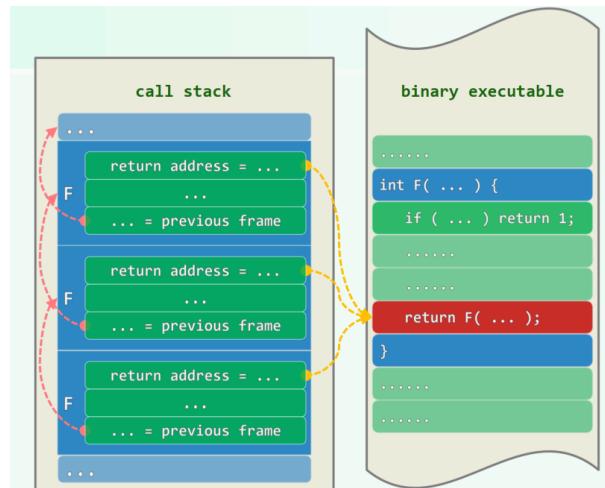


图 38 尾递归

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

尾递归优化：时间复杂度有常系数改进，空间复杂度或有渐近改进。

```

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

```

4.1.3 进制转换

常用短除法，进行进制转换。

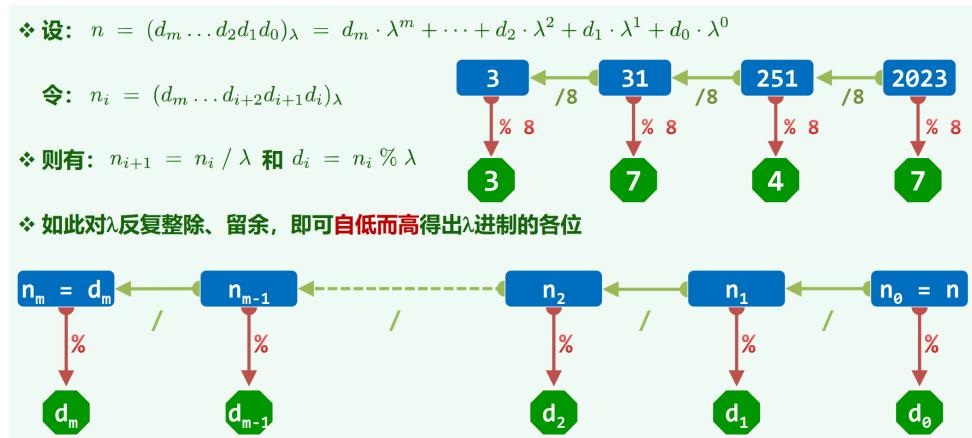


图 39 进制转换

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。

```

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
}

```

4.1.4 括号匹配

括号匹配的问题难以用减而之治和分而治之的思想由外而内解决。

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

$$L |()| R \Rightarrow L | R$$

顺序扫描表达式，用栈记录已扫描的部分的左括号。反复迭代：凡遇“(”，则进栈；凡遇”)”，则出栈。

```
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
}
```

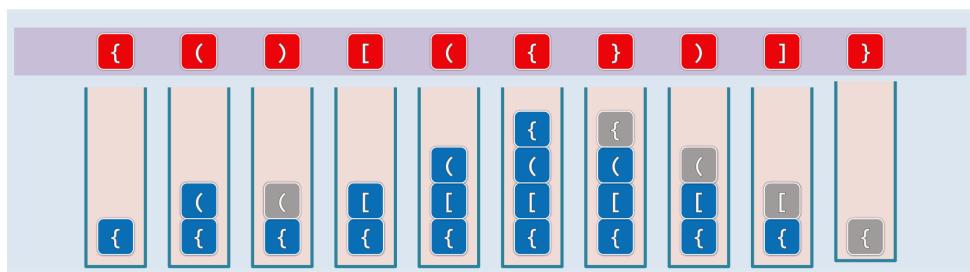


图 40 括号匹配

当括号为单一的一种的时候，甚至可以用一个计数器进行简化。

4.1.5 栈混洗 Stack Permutation

有栈 $A = (a_1, a_2, \dots, a_n)$, 栈 $B = S = \Phi$:

每次可以

- 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

最终所有元素都从 A 移动到 B ，且 B 为 A 的一个排列 $B = (a_{p_1}, a_{p_2}, \dots, a_{p_n})$ 。

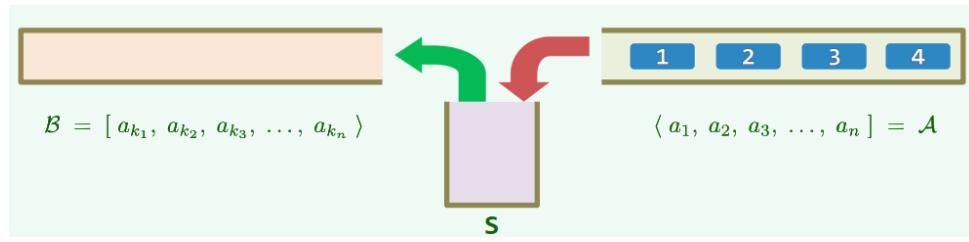


图 41 栈混洗

B 中的排列是有限制的，满足要求的排列有

$$\text{Catalan}(n) = \frac{1}{n+1} \binom{2n}{n}$$

个。这是因为，设 n 个元素的栈混洗排列有 $\text{SP}(n)$ 个，则有递推关系

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

栈混洗排列的充要条件：不存在 312 的模式。即在栈混洗排列中，任意三个连续元素 a_i, a_j, a_k ，满足 $i < j < k$ ，有 $a_i < a_k < a_j$ 。

这样可以得到一个 $O(n^3)$ 的检验算法。

可以简化成当且仅当对于任意 $i < j$ ，不含模式 $[..., j+1, ..., i, ..., j, ...]$ 。

这样可以得到一个 $O(n^2)$ 的检验算法。

事实上，可以直接用栈来模拟，这样可以得到一个 $O(n)$ 的检验算法。

括号匹配的合法排列就是栈混洗排列。

4.1.6 中缀表达式

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。 $\text{val}(S) = \text{val}(\text{SL} + \text{str}(\text{v}\theta) + \text{SR})$ 。

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。

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

- 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。

```

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
}

```

其中优先级的比较可以用一个表格来实现

```

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

```

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 '>'
} //

```

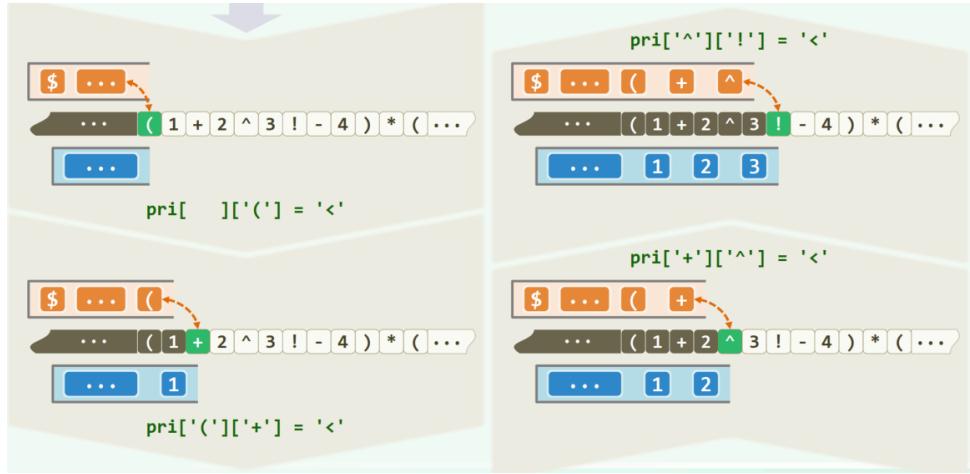


图 42 中缀表达式求值——'<'

'>': 时机已到

```
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
```

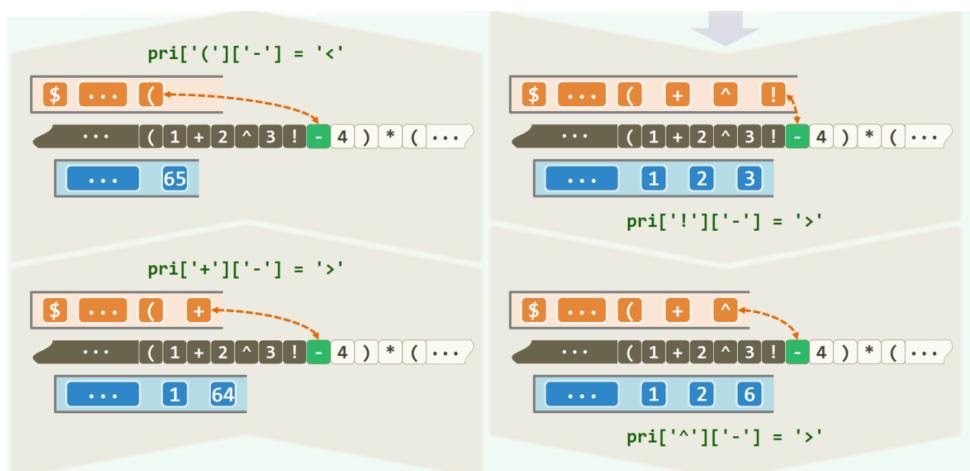


图 43 中缀表达式求值——'>'

'=': 终须了断

```

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

```

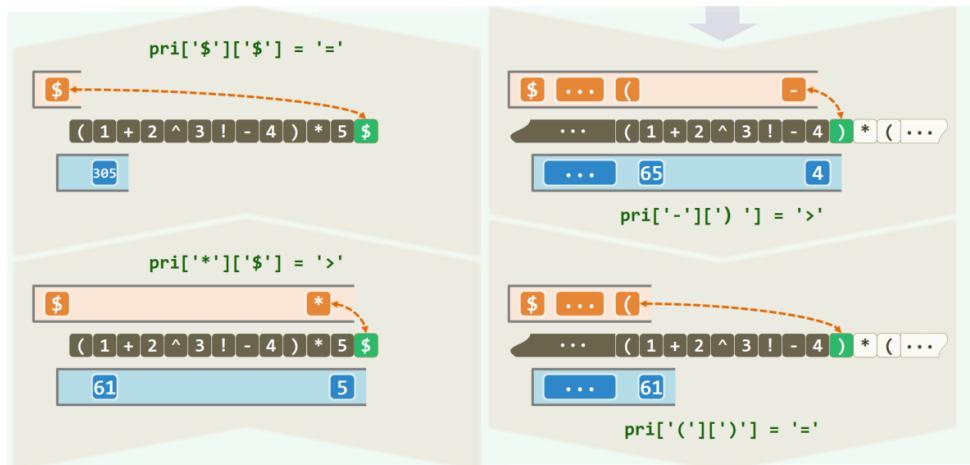


图 44 中缀表达式求值——'='

4.1.7 逆波兰表达式 Reverse Polish Notation(RPN)

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
123 + 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。

RPN 的求值很容易, 只需要一个栈即可。

```

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

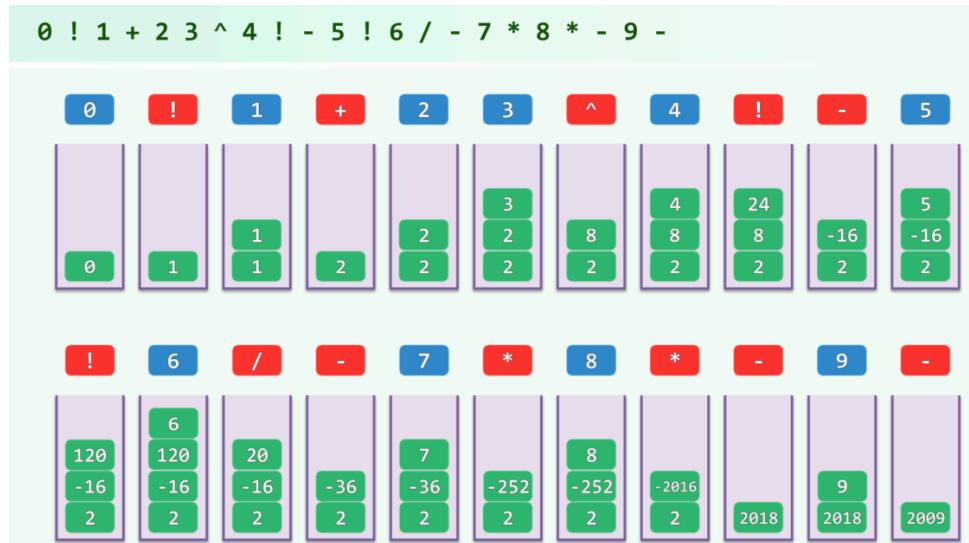


图 45 逆波兰表达式求值

4.1.7.1 中缀表达式转换为逆波兰表达式

如果要用程序自动转换，可以在刚才求值的基础上修改。

其实后缀表达式的运算就是在中缀表达式计算的时候，按照优先级的顺序排出来的序列。

```
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 '>'
                /* ..... */
            } //switch
        /* ..... */
    } //while
    /* ..... */
    return opnd.pop(); //弹出并返回最后的计算结果
}
```

手动转换的方法：

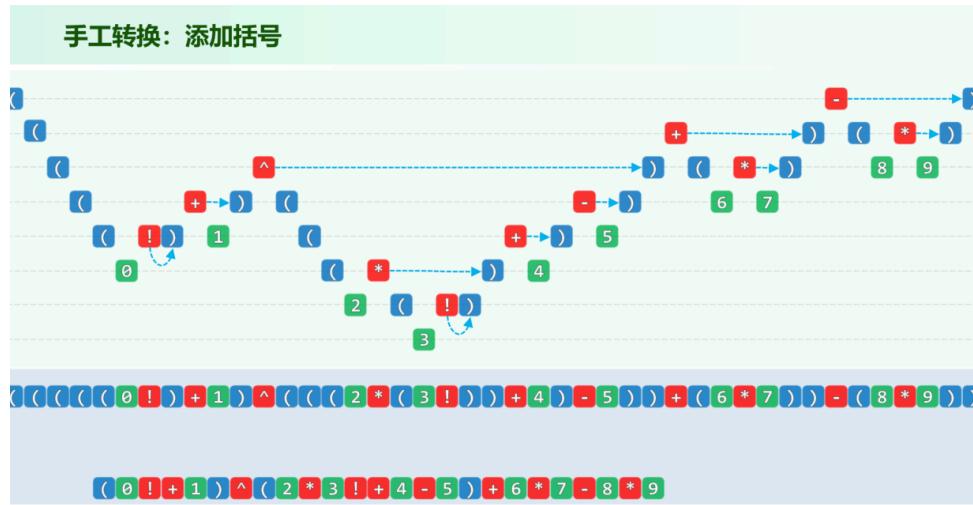


图 46 中缀表达式转换为逆波兰表达式



图 47 中缀表达式转换为逆波兰表达式

4.2 队列 Queue

4.2.1 接口与实现

队列 (queue) 也是受限的序列

- 先进先出 (FIFO)
- 后进后出 (LILO)

提供接口:

- 只能在队尾插入 (查询): enqueue() / rear()
- 只能在队头删除 (查询): dequeue() / front()
- 扩展接口: getMax()...

基于向量或列表派生

```
template <typename T> class Queue: public List<T> { //原有接口一概沿用
public:
    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?
```

如此实现的队列接口，均只需 $O(1)$ 时间。

4.2.2 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？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
```

4.2.3 直方图内最大矩形

设 $H[\theta, n]$ 是一个非负整数直方图

- 如何找到 $H[]$ 中最大的正交矩形？
- 为了消除可能存在的歧义，例如，我们可以选择最左侧的矩形

我们考虑由 $H[r]$ 支撑的最大矩形，满足：

$$\begin{aligned} \text{maxRect}[r] &= H[r] \times (t(r) - s(r)) \\ s(r) &= \max\{0 \leq k < r \mid H[k-1] < H[r]\} \\ t(r) &= \min\{t < k \leq n \mid H[r] < H[k]\} \end{aligned}$$

其中， $s(r)$ 是 r 左侧第一个小于 $H[r]$ 的位置， $t(r)$ 是 r 右侧第一个小于 $H[r]$ 的位置。

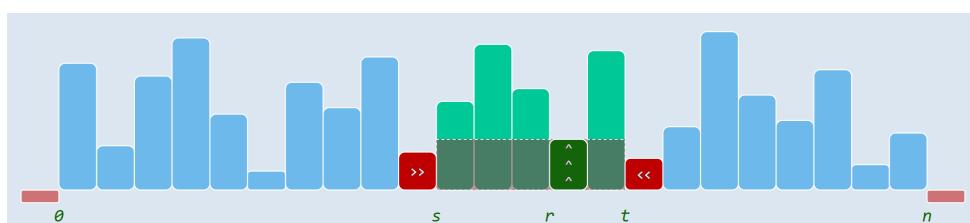


图 48 直方图内最大矩形

4.2.3.1 Brute-force 方法

按照刚才的分析，对每一个位置都计算最大支撑矩形，然后取最大值。这样的复杂度是 $O(n^2)$ 。

4.2.3.2 利用单调栈

我们希望可以对每个 r 快速找到 $s[r]$ 和 $t[r]$ 。

从前往后扫描，维护一个单调栈：

- 栈内元素单调递增
- 每次插入新元素的时候都会弹出栈内所有比它小的元素

```
Rank* s = new Rank[n]; Stack<Rank> S;
for ( Rank r = 0; r < n; r++ ) //using SENTINEL
    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();
```

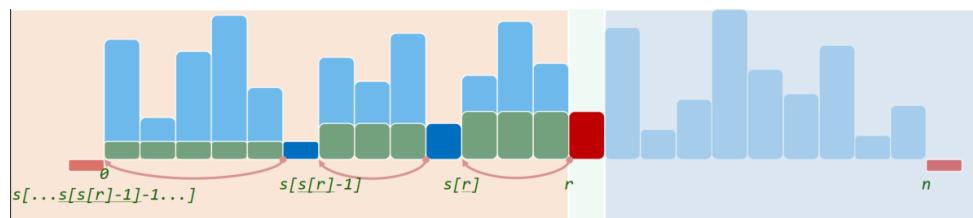


图 49 直方图内最大矩形——寻找` $s[r]$ `

- $s[r]$ 中记录的是左侧第一个 $H[k]$ 小于 $H[r]$ 的 k （有哨兵），这样就可以计算出 $H[r]$ 支撑的最大矩形的左边界。
- 对于单调栈：
 - 每个元素都会被压入栈一次，而每个元素也会在后序的某个时刻被弹出栈一次，所以总的时间复杂度是 $O(n)$ 。
 - 递增栈可以保证栈顶元素就是前一个最小的元素，即 $s[r]$ 。

按相反的顺序扫描一遍，可以计算出 $t[r]$ 。

4.2.3.3 一次扫描

如果数据是在线的，就很难做到正反两次扫描。对于一次扫描：

```
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;
```

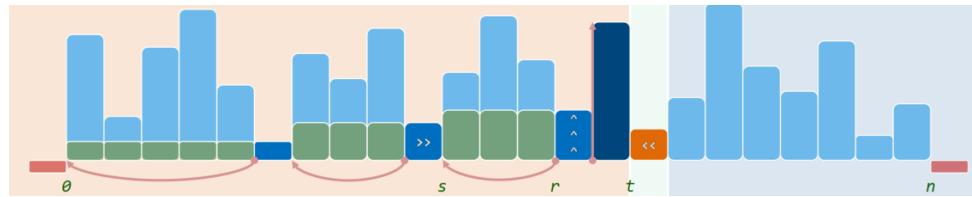


图 50 直方图内最大矩形——一次扫描

- SR 同样是一个递增栈，SR.top()是当前最小的元素，SR.2ndTop()是次小的元素。
 - 存先前最大的矩形，每次碰到有更小的数就到了右边界，然后计算面积，再与之前的最大值比较。

4.3 Steap + Queap

4.3.1 Steap = Stack + Heap = push + pop + getMax

希望每次可以在 $O(1)$ 时间内找到最大值，可以再开一个并列的堆来维护最大值。

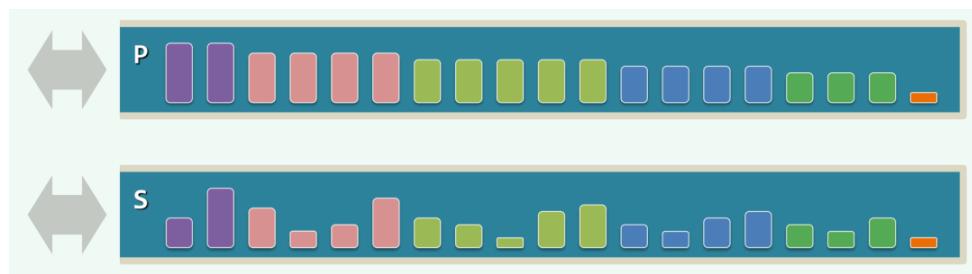


图 51 Steap

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)
```

通过看 P 的记录值，可以知道 S 中的最大值。

也可以用下面采用的指针+计数的方法： p' 中存入指针与计数器，每次操作修改计数，加入指针，或者计数器变为 0 删除指针

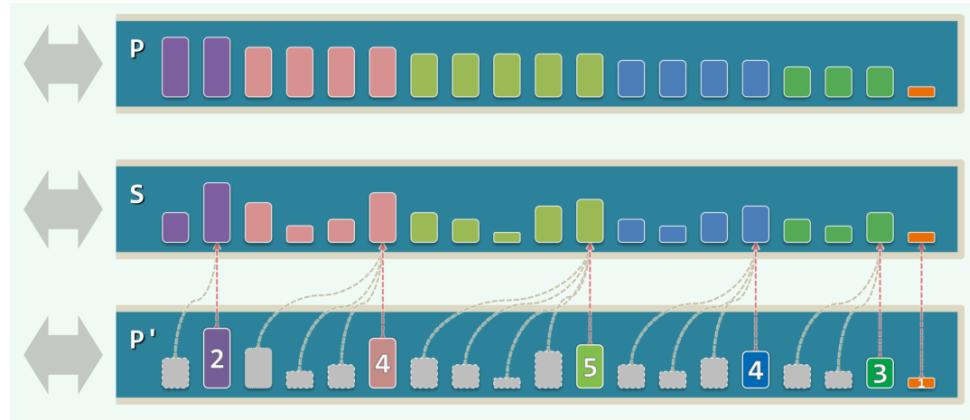


图 52 Steap

4.3.2 Queap = Queue + Heap = enqueue + dequeue + getMax

一样的方法，但是要注意出入口的操作：

```
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;
}
```

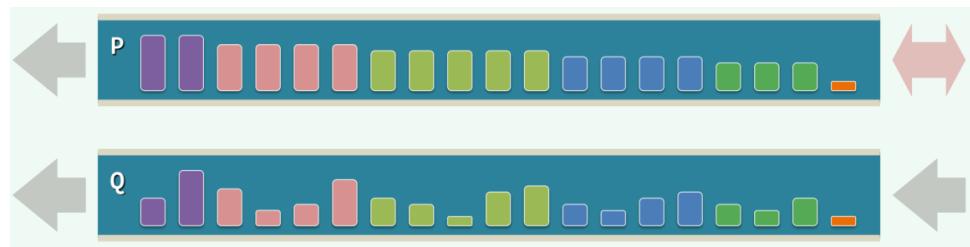


图 53 Queap

可以按照同样的方式化简成指针+计数的方法。

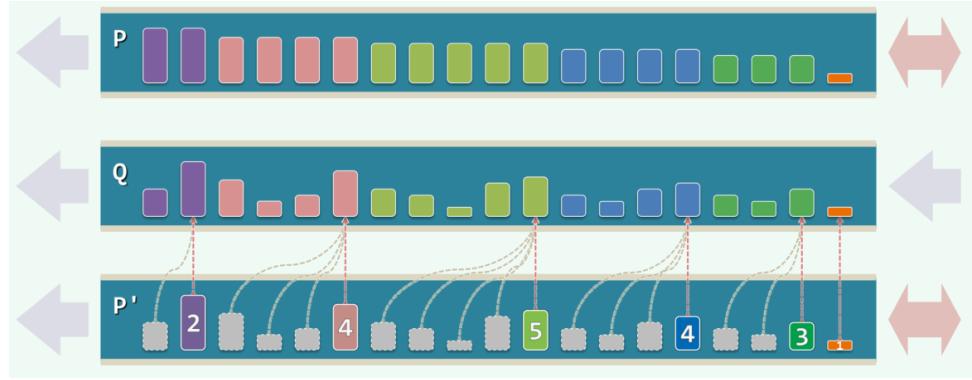


图 54 Queap

4.4 双栈当队

Queue = Stack x 2

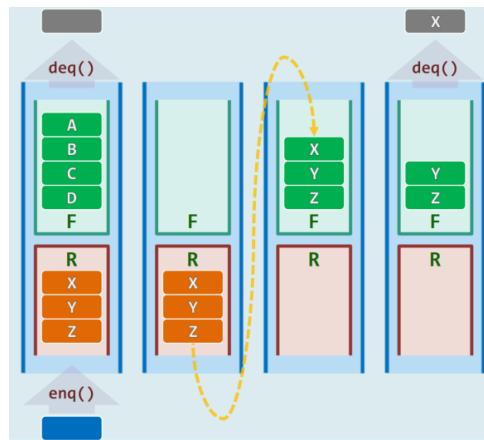


图 55 双栈当队

但是区别是，每次 `dequeue` 的时候，要把 `R` 中的元素全部倒入 `F` 中。

```

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();

```

这样单步可能出现 $O(n)$ 的情况。

现在来看分摊下来的复杂度：

4.4.1.1 Amortization By Accounting

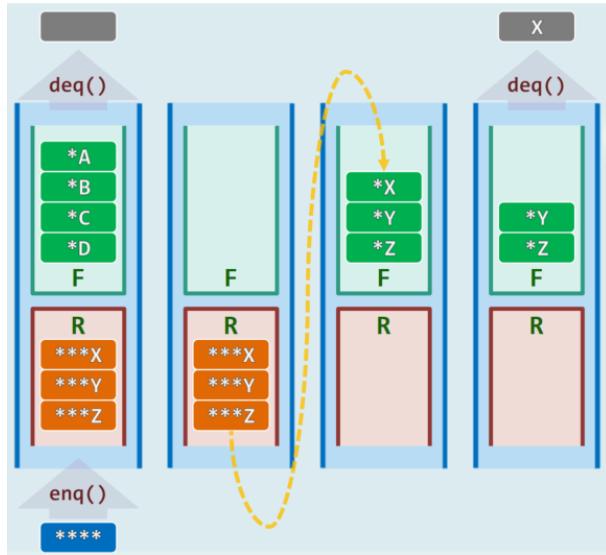


图 56 Amortization By Accounting

分析每个元素经历的操作次数。在整个过程中，每个元素至多经历 1 次 R 的 push，1 次 R 的 pop，1 次 F 的 push，1 次 F 的 pop，所以每个元素至多经历 4 次操作。这样下来，每个元素的分摊复杂度是 $O(1)$ 。

4.4.1.2 Amortization By Aggregate

考虑 d 次 `dequeue()` 和 e 次 `enqueue()` 已经做完了，一定有 $d \leq e$ 。所有的时间成本是 $4d + 3(e - d) = 3e + d$ 。

这样下来，每个元素的分摊复杂度是 $O(1)$ 。

4.4.1.3 Amortization By Potential

设第 k 次操作的势能

$$\Phi_k = |F_k| - |R_k|$$

又考虑每次操作的分摊成本 (Amortized Cost)

$$A_k = T_k + \Delta\Phi_k = T_k + \Phi_k - \Phi_{k-1} \equiv 2$$

其中 T_k 是实际成本 (Actual Cost)。从而可以得到：

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

$$T(n) < 3n = O(n)$$

所以每个元素的分摊复杂度是 $O(1)$ 。

五 二叉树 Binary Tree

- 树是有层次结构的表示:
 - 表达式
 - 文件系统
 - URL
- 树是综合性的数据结构
 - 兼具 Vector 和 List 的优点
 - 兼顾高效的查找、插入、删除
- 树是半线性的结构
 - 不再是简单的线性结构，但在确定某种次序之后，具有线性特征

5.1 二叉树 Binary Tree

5.1.1 图论基础

5.1.1.1 有根树 Rooted tree

树是极小连通图、极大无环图 $T(V, E)$ ，节点数 $n = |V|$ ，边数 $e = |E|$ 。

指定一个节点 $r \in V$ 作为根节点，其他节点到根节点有唯一路径，称为有根树。

若 T_1, T_2, \dots, T_d 为有根树，则 $T = ((\bigcup_i T_i) \cup \{r\}, ((\bigcup_i E_i)) \cup \{< r, r_i > | 1 \leq i \leq d\})$ 为有根树。相对于 T ， T_i 被称为以 r_i 为根的子树。

5.1.1.2 有序树 Ordered Tree

有根树中，节点的子树有次序，称为有序树。

可以证明树的结点和边满足 $n = e + 1$ 。

5.1.1.3 连通+无环

连通与无环意味着任意两个节点只有一条路径，即不存在环。

从而可以从根出发，对树进行深度的等价类划分。

5.1.1.4 深度+层次

节点 v 的深度定义为 $\text{depth}(v) = |\text{path}(v)|$ ，其中 $\text{path}(v)$ 为从根到 v 的路径。

根节点和叶子的深度为 0，其他节点的深度为其父节点的深度加 1，空树的深度为 -1。

所有叶子深度的最大者叫作树的高度，即 $\text{height}(T) = \max\{\text{depth}(v) | v \in V\}$ 。

5.1.2 树的表示

树提供接口：

<code>root()</code>	<code>parent()</code>	<code>firstChild()</code>	<code>nextSibling()</code>	<code>insert(i, e)</code>	<code>remove(i)</code>	<code>traverse()</code>
根节点	父节点	第一个孩子	下一个兄弟	插入	删除	遍历

树可以用线性结构储存：

1. 利用父节点的信息

除了根节点，每个节点都有一个父节点，可以用线性结构储存。在一个数组 `data` 存储树节点的值；另开一个数组 `parent` 存储每个节点的父节点的下标，根节点的父节点下标为其本身。

2. 利用孩子节点的信息

除了叶子节点，每个节点都有一个孩子节点，可以用线性结构储存。在一个数组 `data` 存储树节点的值；另开一个数组 `children` 存储每个节点的第一个孩子节点的下标，如果有多个孩子就用链表储存所有的孩子。

3. 父节点+孩子节点

前面的方法要么只能访问孩子节点，要么只能访问父节点。三个数组同时使用就能双向访问。

5.1.3 二叉树 Binary Tree

二叉树是所有节点的（出）度数都不超过 2 的树。是有根、有序的树。

5.1.3.1 二叉树和多叉树的等价变换

有根、有序的多叉树和二叉树可以相互转换。

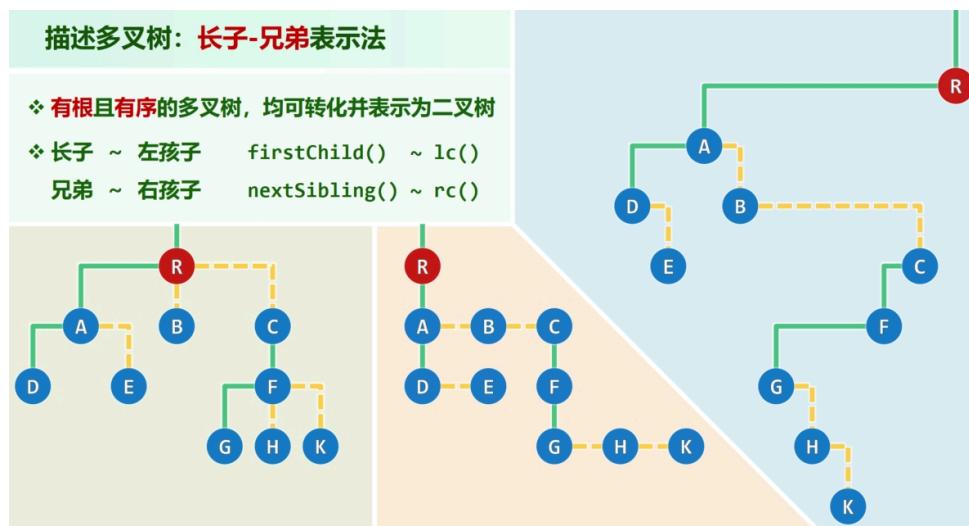


图 57 多叉树和二叉树的等价变换

只要是多叉树中，一个节点的长子，成为二叉树的左儿子，其他的兄弟顺延成为右儿子。也就是说在二叉树中每个节点的右儿子们都是原来的兄弟，左儿子是原来的长子。

5.1.3.2 满二叉树

5.1.3.3 真二叉树

引入 $n_1 + 2n_0$ 个外部节点 `null`，使得每个节点都有两个儿子，称为真二叉树。

对于红黑树之类的结构，真二叉树可以用来简化描述、理解、实现、分析。

5.1.4 二叉树的实现

用节点 `BinNode` 表示二叉树的节点，用 `BinTree` 表示二叉树。

```
template <typename T> using BinNodePosi = BinNode<T>*; //节点位置
template <typename T> struct BinNode {
    BinNodePosi<T> parent, lc, rc; //父亲、孩子
    T data; Rank height, npl; Rank size(); //高度、npl、子树规模
    BinNodePosi<T> insertAsLC( T const & ); //作为左孩子插入新节点
    BinNodePosi<T> insertAsRC( T const & ); //作为右孩子插入新节点
    BinNodePosi<T> succ(); // (中序遍历意义下) 当前节点的直接后继
    template <typename VST> void travLevel( VST & ); //层次遍历
    template <typename VST> void travPre( VST & ); //先序遍历
    template <typename VST> void travIn( VST & ); //中序遍历
    template <typename VST> void travPost( VST & ); //后序遍历
};
```

并且直接实现引入新节点

```
template <typename T>
BinNodePosi<T> BinNode<T>::insertAsLC( T const & e )
{
    return lc = new BinNode( e, this );
}
template <typename T>
BinNodePosi<T> BinNode<T>::insertAsRC( T const & e )
{
    return rc = new BinNode( e, this );
}
```

用 `BinNode` 实现 `BinTree`，并且实现 `BinTree` 的接口

```
template <typename T> class BinTree {
protected:
    Rank _size; //规模
    BinNodePosi<T> _root; //根节点
    virtual Rank updateHeight( BinNodePosi<T> x ); //更新节点x的高度
    void updateHeightAbove( BinNodePosi<T> x ); //更新x及祖先的高度
public:
    Rank size() const { return _size; } //规模
    bool empty() const { return !_root; } //判空
    BinNodePosi<T> root() const { return _root; } //树根
    /* ... 子树接入、删除和分离接口；遍历接口 ... */
}
```

引入新节点，用顺序直接重载函数，表示左右插入

```
BinNodePosi<T> BinTree<T>::insert( BinNodePosi<T> x, T const & e ); //作为右孩子
BinNodePosi<T> BinTree<T>::insert( T const & e, BinNodePosi<T> x ) { //作为左孩子
    _size++;
```

```
x->insertAsLC( e );
updateHeightAbove( x ); //及时维护高度
return x->lC;
}
```

接入子树

```
BinNodePosi<T> BinTree<T>::attach( BinTree<T>* &S, BinNodePosi<T> x ); //接入左
子树
BinNodePosi<T> BinTree<T>::attach( BinNodePosi<T> x, BinTree<T>* &S ) { //接入
右子树
    if ( x->rc = S->_root ) //去除插入空树的情况
        x->rc->parent = x;
    _size += S->_size;
    updateHeightAbove(x); //及时维护高度
    S->_root = NULL; S->_size = 0;
    release(S); S = NULL;
    return x;
}
```

用 `if` 中的赋值大大简化了代码量，其中实时更新高度的函数为

```
#define stature(p) ( (int) ( (p) ? (p)->height : -1 ) ) //空树高度-1, 以上递推
template <typename T> //勤奋策略：及时更新节点 x 高度，具体规则因树不同而异
Rank BinTree<T>::updateHeight( BinNodePosi<T> x ) //此处采用常规二叉树规则, O(1)
    { return x->height = 1 + max( stature( x->lC ), stature( x->rc ) ); }
template <typename T> //更新节点及其历代祖先的高度
void BinTree<T>::updateHeightAbove( BinNodePosi<T> x ) //O( n = depth(x) )
    { while (x) { updateHeight(x); x = x->parent; } } //可优化
```

分离子树

```
template <typename T> BinTree<T>* BinTree<T>::secede( BinNodePosi<T> x ) {
    FromParentTo( * x ) = NULL; updateHeightAbove( x->parent );
// 以上与 BinTree<T>::remove()一致；以下还需对分离出来的子树重新封装
    BinTree<T> * S = new BinTree<T>; //创建空树
    S->_root = x; x->parent = NULL; //新树以 x 为根
    S->_size = x->size(); _size -= S->_size; //更新规模
    return S; //返回封装后的子树
}
```

5.2 二叉树的遍历

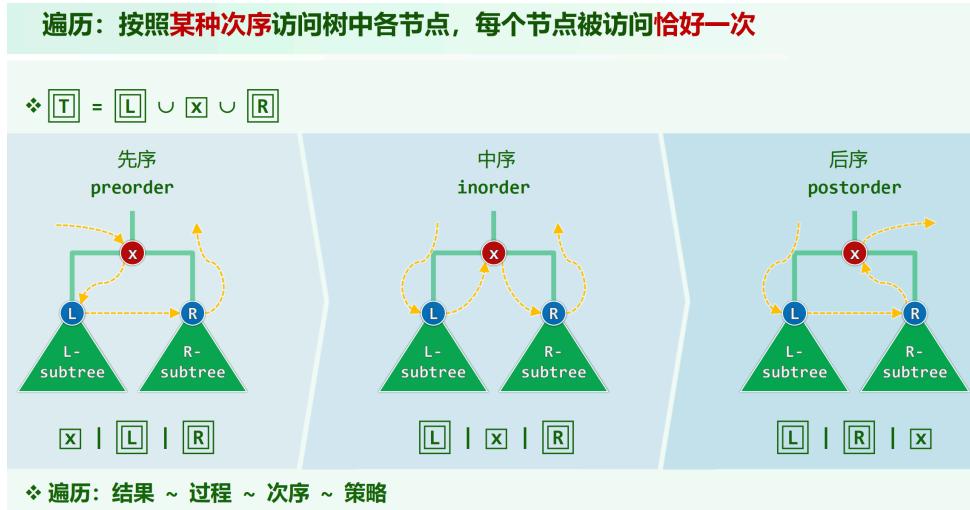


图 58 二叉树的遍历

树的遍历是按照一定顺序，访问树中的每个节点，且每个节点仅访问一次。

如果采用最直接的递归方式

```
/* 先序遍历 */
template <typename T, typename VST>
void traverse( BinNodePosi<T> x, VST & visit ) {
    if ( !x ) return;
    visit( x->data );
    traverse( x->lc, visit );
    traverse( x->rc, visit );
} //O(n)
/* 中序遍历 */
template <typename T, typename VST>
void traverse( BinNodePosi<T> x, VST & visit ) {
    if ( !x ) return;
    traverse( x->lc, visit );
    visit( x->data );
    traverse( x->rc, visit ); //tail
}
/* 后序遍历 */
template <typename T, typename VST>
void traverse( BinNodePosi<T> x, VST & visit ) {
    if ( !x ) return;
    traverse( x->lc, visit );
    traverse( x->rc, visit );
    visit( x->data );
}
```

则会导致栈溢出，因为递归深度与树的高度成正比。

5.2.1 先序遍历

按照 $x|L|R$ 的顺序进行遍历，如下图

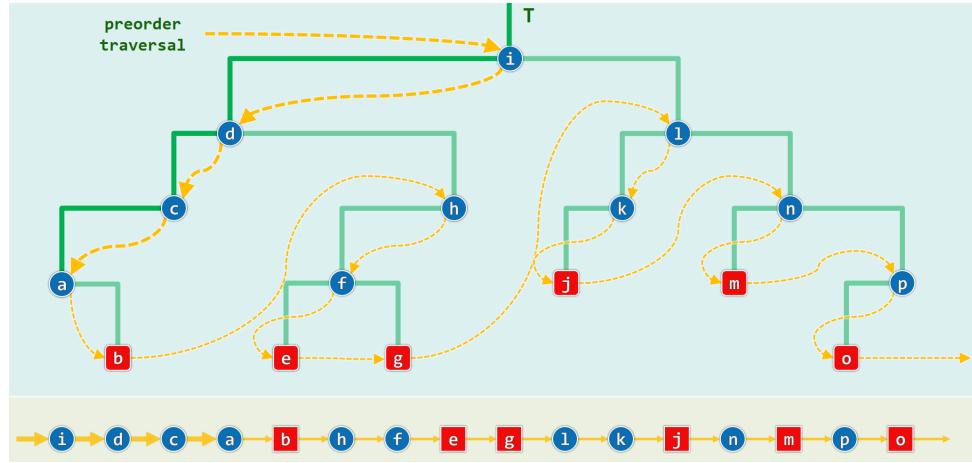


图 59 先序遍历

观察图，发现，可以理解为藤缠树。先顺着左儿子构成的藤爬下去，再顺着右儿子构成的藤爬上去；每个右儿子都是一棵子树，在子树中递归地调用先序遍历即可。

沿着左侧藤，整个遍历过程可分解为：

- 自上而下访问藤上节点，再
- 自下而上遍历各右子树

各右子树的遍历彼此独立自成一个子任务

爬藤而下：

```
template <typename T, typename VST> static void visitAlongVine
( BinNodePosi<T> x, VST & visit, Stack < BinNodePosi<T> > & S ) { //分摊 O(1)
    while ( x ) { //反复地
        visit( x->data ); //访问当前节点
        S.push( x->rc ); //右孩子（右子树）入栈（将来逆序出栈）
        x = x->lc; //沿藤下行
    } //只有右孩子、 NULL 可能入栈—增加判断以剔除后者，是否值得？
}
```

先序遍历：

```
template <typename T, typename VST>
void travPre_I2( BinNodePosi<T> x, VST & visit ) {
    Stack < BinNodePosi<T> > S; //辅助栈
    while ( true ) { //以右子树为单位，逐批访问节点
        visitAlongVine( x, visit, S ); //访问子树 x 的藤蔓，各右子树（根）入栈缓冲
        if ( S.empty() ) break; //栈空即退出
        x = S.pop(); //弹出下一右子树（根）
    } //#pop = #push = #visit = O(n) = 分摊 O(1)
}
```

用栈记录沿藤而下的节点，再向上去访问右子树。如果右子树还需要爬藤，就继续进栈出栈。

先序遍历是在向下爬藤时，就把左儿子遍历的。而下面的中序遍历在向下爬藤时，只进栈，等到向上爬藤时才遍历（，随即立刻遍历其右儿子）。

5.2.2 中序遍历

按照 $L|x|R$ 的顺序进行遍历，如下图

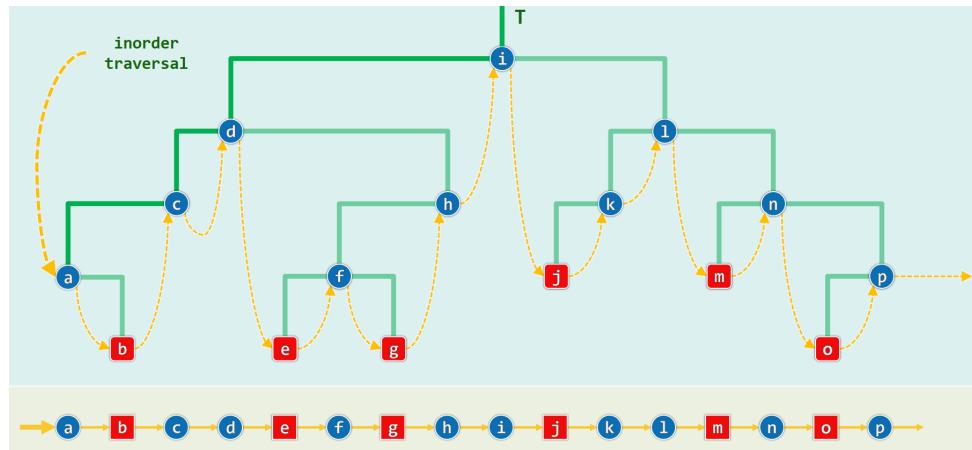


图 60 中序遍历

沿着左侧藤，遍历可自底而上分解为 $d + 1$ 步迭代：访问藤上节点，再遍历其右子树。各右子树的遍历彼此独立，自成一个子任务。

自藤底向上爬：

```
template <typename T>
static void goAlongVine(BinNodePosi<T> x, Stack < BinNodePosi<T> > & S){
    while ( x )
    { S.push( x ); x = x->lc; }
} //逐层深入，沿藤蔓各节点依次入栈
```

中序遍历：

```
template <typename T, typename V> void travIn_I1( BinNodePosi<T> x, V& visit )
{
    Stack < BinNodePosi<T> > S; //辅助栈
    while ( true ) { //反复地
        goAlongVine( x, S ); //从当前节点出发，逐批入栈
        if ( S.empty() ) break; //直至所有节点处理完毕
        x = S.pop(); //x的左子树或为空，或已遍历（等效于空），故可以
        visit( x->data ); //立即访问之
        x = x->rc; //再转向其右子树（可能为空，留意处理手法）
    }
}
```

和先序遍历的区别是，读取藤的时机。先序遍历向下爬藤时读取，中序遍历向上爬藤时读取。

其复杂度是 $O(n)$ 的，因为一共执行 $O(n)$ 次 `pop` 和 `push`，每次 `pop` 和 `push` 都是 $O(1)$ 的。

5.2.2.1 前驱和后继

考虑一个节点的后继：如果有右子树，直接后继是最靠左的右后代：相当于沿着该节点的右儿子爬藤而下到底。如果没有右子树，需要找最低的左祖先：相当于辅助栈中的下一个节点，寻找的方法是一直寻找父亲，直到访问的这条路成为了某一个父亲的左儿子。

```
//在中序遍历意义下的直接后继
//稍后将被 BST::remove 中的 removeAt() 调用
template <typename T>
BinNodePosi<T> BinNode<T>::succ() {
    BinNodePosi<T> s = this;
    if ( rc ) { //若有右孩子，则
        s = rc; //直接后继必是右子树中的
        while ( HasLChild( * s ) )
            s = s->lchild; //最小节点
    } else { //否则
        //后继应是“以当前节点为直接前驱者”
        while ( IsRChild( * s ) )
            s = s->parent; //不断朝左上移动
        //最后再朝右上移动一步
        s = s->parent; //可能是 NULL
    }
    return s; //两种情况下，运行时间分别为
} //当前节点的高度与深度，不过 O(h)
```

5.2.3 后序遍历

子树的删除和 `BinNode::size()` 和 `BinTree::updateHeight()` 维护，就是一个后序遍历的例子。

按照 $L|R|x$ 的顺序进行遍历，如下图

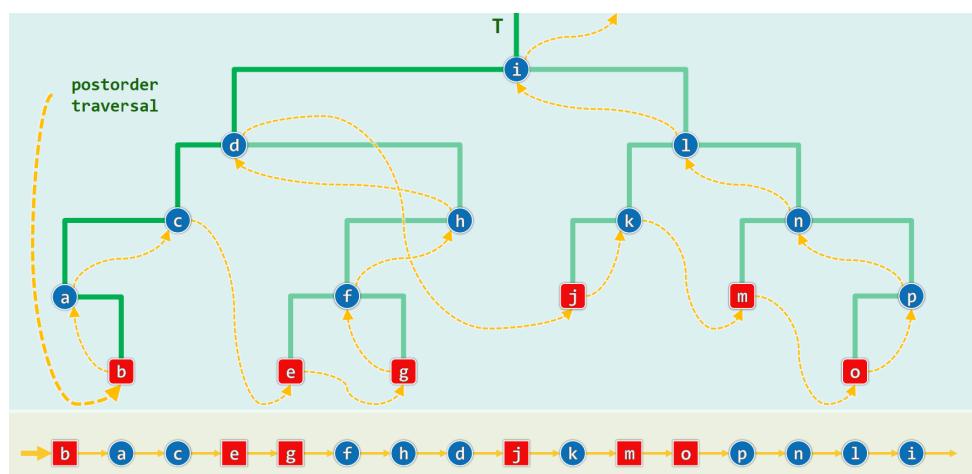


图 61 后序遍历

从根出发下行尽可能沿左分支，实不得已才沿右分支，这样找到 **leftmost leaf**。最后一个节点必是叶子，而且是按中序遍历次序最靠左者，也是递归版中 `visit()` 首次执行处。

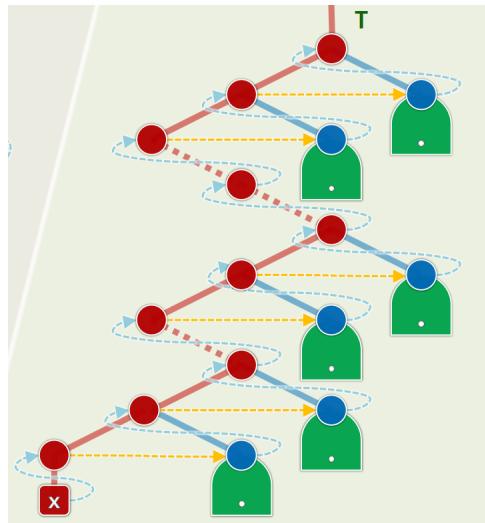


图 62 后序遍历

在沿着这条曲折的藤向上爬时，如果有右儿子，就对右子树进行递归，没有就向上走。这条藤事实上在右子树封装的情况下实现了后序遍历，只需要把封装的右子树展开。

寻找 **leftmost leaf**

```
template <typename T> static void gotoLeftmostLeaf( Stack <BinNodePosi<T>> & S ) {
    while ( BinNodePosi<T> x = S.top() ) //自顶而下反复检查栈顶节点
        if ( HasLChild( * x ) ) { //尽可能向左。在此之前
            if ( HasRChild( * x ) ) //若有右孩子，则
                S.push( x->rcc ); //优先入栈
                S.push( x->lcc ); //然后转向左孩子
            } else //实不得已
                S.push( x->rcc ); //才转向右孩子
        S.pop(); //返回之前，弹出栈顶的空节点
    }
```

后序遍历：

```
template <typename T, typename V> void travPost_I( BinNodePosi<T> x, V & visit ) {
    Stack < BinNodePosi<T> > S; //辅助栈
    if ( x ) S.push( x ); //根节点首先入栈
    while ( ! S.empty() ) { //x 始终为当前节点
        if ( S.top() != x->parent ) //若栈顶非 x 之父（而为右兄），则
            gotoLeftmostLeaf( S ); //在其右兄弟树中找到最靠左的叶子
        x = S.pop(); //弹出栈顶（即前一节点之后继）以更新 x
    }
```

```

        visit( x->data ); //并随即访问之
    }
}

```

也可以分析得到复杂度是 $O(n)$ 的。

5.2.3.1 表达式

一个运算表达式可以存储在一个树种，每个节点是一个运算符，每个叶子是一个操作数。括号的层级就是树的深度。

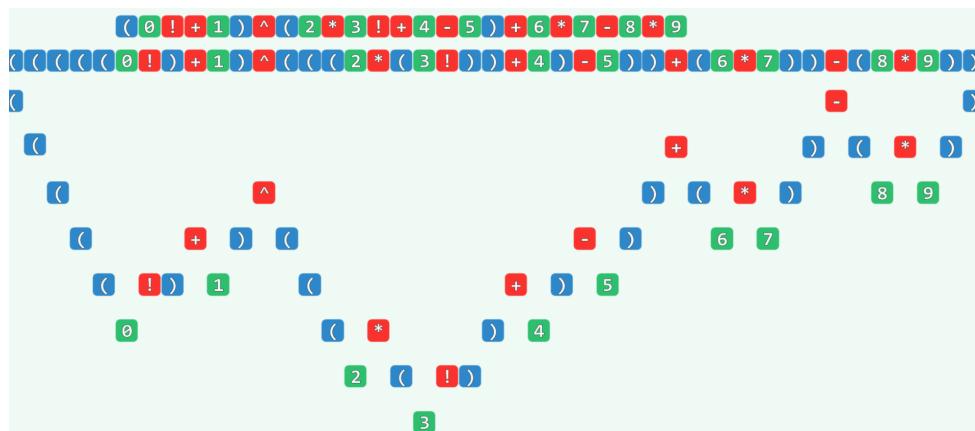


图 63 表达式

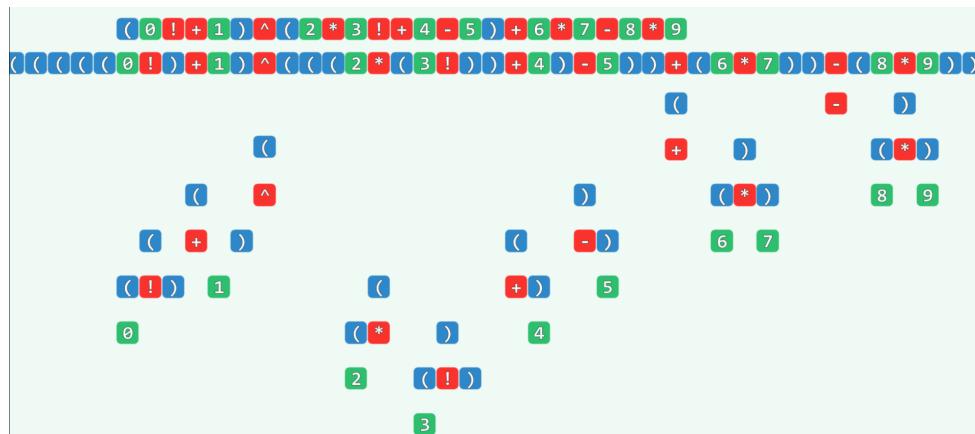


图 64 表达式

可以看出这些红色和绿色的节点构成一棵树，就是下面的表达式树。

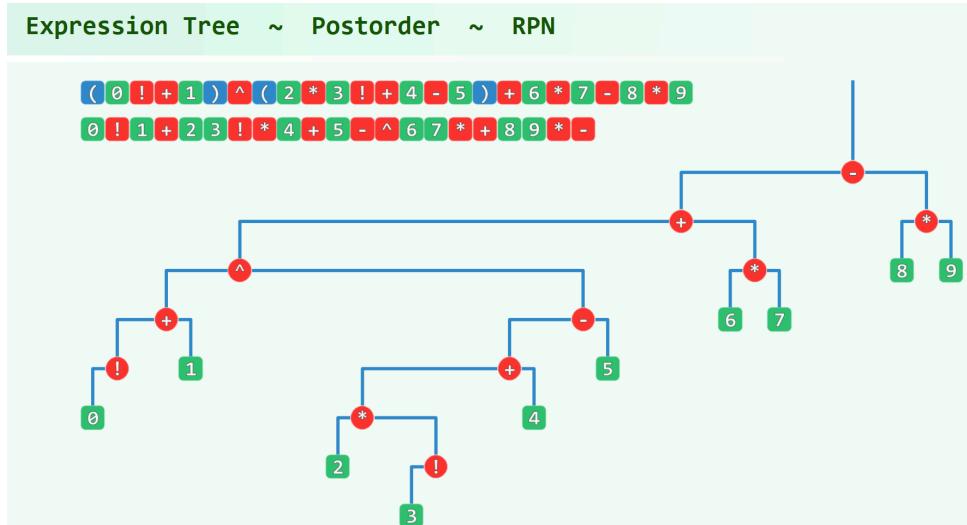


图 65 表达式树

而先序、中序、后序遍历就是表达式的前缀、中缀、后缀表示。

5.2.4 层次遍历

层次遍历是按照层次进行遍历。可以用一个队列实现：

在取出一个节点时，把它的左右儿子入队，这样达到的效果是按照顺序，将每一层的节点从左向右遍历。

```
template <typename T> template <typename VST>
void BinNode<T>::travLevel( VST & visit ) { //二叉树层次遍历
    Queue< BinNodePosi<T> > Q; Q.enqueue( this ); //引入辅助队列，根节点入队
    while ( ! Q.empty() ) { //在队列再次变空之前，反复迭代
        BinNodePosi<T> x = Q.dequeue(); visit( x->data ); //取出队首节点并随即访问
        if ( HasLChild( *x ) ) Q.enqueue( x->lch ); //左孩子入队
        if ( HasRChild( *x ) ) Q.enqueue( x->rch ); //右孩子入队
    }
}
```

5.3 二叉树的重构

二叉树的重构是指，已知二叉树的遍历序列，重构出二叉树。

5.3.1 [先序|后序]+中序

已知先序和中序，可以重构出二叉树。

先序序列： $x|L|R$

中序序列： $L|x|R$

先序序列的第一个节点是根节点，中序序列中根节点左边的是左子树，右边的是右子树。

后序同理。

5.3.2 先序+后序

不能重构，因为无法确定左右子树。

5.3.3 增强序列

在输出遍历序列时，将 `NULL` 也输出，这样就可以重构出二叉树。

可归纳证明：在增强的先序、后序遍历序列中

1. 任一子树依然对应于一个子序列，而且
2. 其中的 `NULL` 节点恰比非 `NULL` 节点多一个

5.4 Huffman 编码树

如何对各字符编码，使文件最小？

5.4.1 PFC 编码

将字符集 Σ 中的字符组织成一棵二叉树，以 0/1 表示左/右孩子，各字符 x 分别存放于对应的叶子 $v(x)$ 中。

字符 x 的编码串 $rps(v(x)) = rps(x)$ 由根到 $v(x)$ 的通路（root path）确定

字符编码不必等长，而且不同字符的编码互不为前缀，故不致歧义（Prefix-Free Code）。

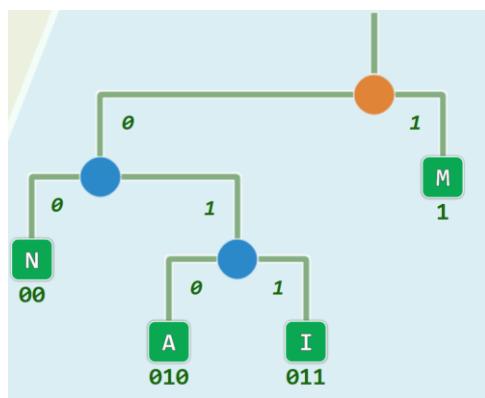


图 66 PFC 编码

按照这样的方法可以分析编码长度。

平均编码长度

$$\text{ald}(T) = \sum_{x \in \Sigma} \frac{\text{depth}(v(x))}{|\Sigma|}$$

定义对于特定的字符集 Σ ， $\text{ald}(T)$ 最小的二叉树 T_{Opt} 为 Σ 的最优编码树（Optimal Code Tree）。

5.4.2 最优编码树

最优编码树的性质：

$$\forall v \in T_{\text{Opt}} : \deg(v) = 0 \Leftrightarrow \text{depth}(v) \geq \text{depth}(T_{\text{Opt}}) - 1$$

亦即，叶子只能出现在倒数两层以内——否则，通过节点交换即可调整到更优的编码树。

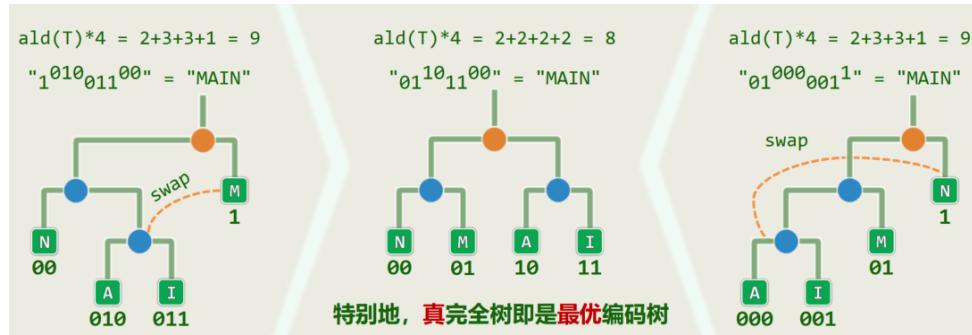


图 67 最优编码树

5.4.3 最优带权编码树

已知各字符的期望频率，此时的最优编码树称为最优带权编码树（Optimal Weighted Code Tree）。

文件长度 \propto 平均带权深度 $\text{wald}(T) = \sum_{x \in \Sigma} \text{rps}(x) \times w(x)$ 。此时，完全树未必就是最优编码树。

同样，频率高/低的（超）字符，应尽可能放在高/低处，通过适当交换，同样可以缩短 $\text{wald}(T)$ 。

5.4.4 Huffman 的贪心策略

频率低的字符优先引入，其位置亦更低。

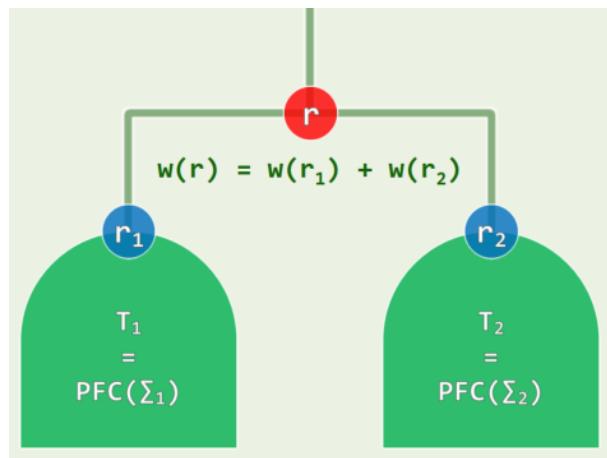


图 68 最优编码树

为每个字符创建一棵单节点的树，组成森林 F

按照出现频率，对所有树排序

while (F 中的树不止一棵)

取出频率最小的两棵树： T1 和 T2

将它们合并成一棵新树 T，并令：

lc(T) = T1 且 rc(T) = T2

w(root(T)) = w(root(T1)) + w(root(T2))

//尽管贪心策略未必总能得到最优解，但非常幸运，如上算法的确能够得到最优编码树之一

将树合成子树，每次比较根节点的权重，取出最小的两棵树，合成一棵新树，再放回去。可以用优先队列优化。

可以证明，Huffman 编码树是最优带权编码树。

下面是代码实现。先定义 Huffman (超) 字符

```
#define N_CHAR (0x80 - 0x20) //仅以可打印字符为例
struct HuffChar { //Huffman (超) 字符
    char ch; unsigned int weight; //字符、频率
    HuffChar ( char c = '^', unsigned int w = 0 ) : ch ( c ), weight ( w ) {};
    bool operator< ( HuffChar const& hc ) { return weight > hc.weight; } //比较器
    bool operator== ( HuffChar const& hc ) { return weight == hc.weight; } //判断器
};
```

再定义 Huffman 编码树

```
using HuffTree = BinTree< HuffChar >; //Huffman 编码树
using HuffForest = List< HuffTree* >; //Huffman 森林
/* 可以替换接口... */
using HuffForest = PQ_List< HuffTree* >; //基于列表的优先级队列
using HuffForest = PQ_ComplHeap< HuffTree* >; //完全二叉堆
using HuffForest = PQ_LeftHeap< HuffTree* >; //左式堆
```

构造编码树：反复合并二叉树

```
HuffTree* generateTree( HuffForest * forest ) { //Huffman 编码算法
    while ( 1 < forest->size() ) { //反复迭代，直至森林中仅含一棵树
        HuffTree *T1 = minHChar( forest ), *T2 = minHChar( forest );
        HuffTree *S = new HuffTree(); //创建新树，然后合并 T1 和 T2
        S->insert( HuffChar('^', T1->root()->data.weight + T2->root()->data.weight) );
        S->attach( T1, S->root() ); S->attach( S->root(), T2 );
        forest->insertAsLast( S ); //合并之后，重新插回森林
    } //assert: 森林中最终唯一的那棵树，即 Huffman 编码树
    return forest->first()->data; //故直接返回之
}
```

查找最小超字符：遍历 List/Vector

```

HuffTree* minHChar( HuffForest* forest ) { //此版本仅达到 O(n)，故整体为 O(n^2)
    ListNodePosi<HuffTree*> m = forest->first(); //从首节点出发，遍历所有节点
    for ( ListNodePosi<HuffTree*> p = m->succ; forest->valid( p ); p = p-
>succ )
        if( m->data->root()->data.weight > p->data->root()->data.weight ) //不
断更新
            m = p; //找到最小节点（所对应的 Huffman 子树）
    return forest->remove( m ); //从森林中取出该子树，并返回
} //Huffman 编码的整体效率，直接决定于 minHChar() 的效率

```

构造编码表：遍历二叉树

```

#include "Hashtable.h" //用HashTable 实现
using HuffTable = Hashtable< char, char* >; //Huffman 编码表
static void generateCT //通过遍历获取各字符的编码
( Bitmap* code, int length, HuffTable* table, BinNodePosi<HuffChar> v ) {
    if ( IsLeaf( * v ) ) //若是叶节点（还有多种方法可以判断）
        { table->put( v->data.ch, code->bits2string( length ) ); return; }
    if ( HasLChild( * v ) ) //Left = 0, 深入遍历
        { code->clear( length ); generateCT( code, length + 1, table, v-
>lch ); }
    if ( HasRChild( * v ) ) //Right = 1
        { code->set( length ); generateCT( code, length + 1, table, v->rch ); }
} //总体 O(n)

```

5.4.5 Huffman 树的改进

1. 基于向量或者列表

基于向量或数组的森林每次插入树都要寻找合适的位置以保证有序，这导致查找需要 $O(n)$ 的时间，从而使得整体的复杂度为 $O(n^2)$ 。

2. 基于堆

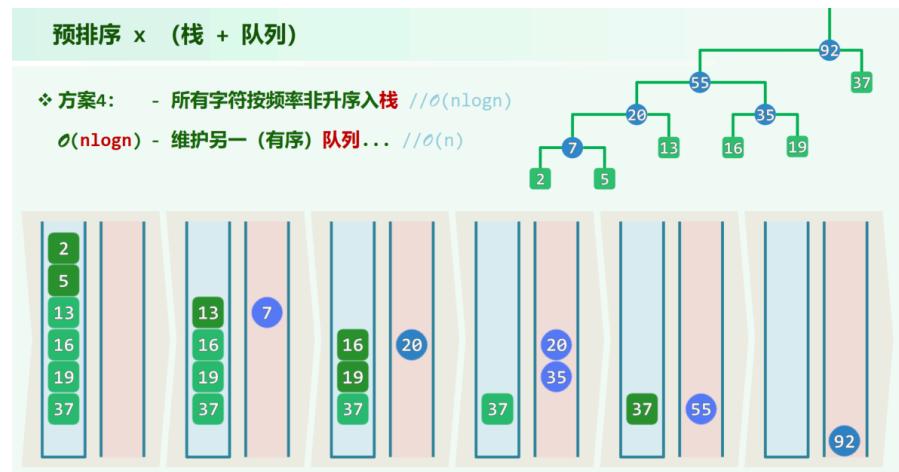
可以将所有树组成一个优先级队列，每次取出最小的两棵树，合并成一棵新树，再放回去。这样可以将复杂度降低到 $O(n \log n)$ 。

3. 基于栈+队列

还有一个小技巧是，先经过 $O(n \log n)$ 的预排序。再将

- 所有字符按频率非升序入栈 $O(n \log n)$
- 维护另一（有序）队列 $O(n)$

每次看栈顶两个元素与队首元素，取两个最小的，合并成一棵新树，入队。这样可以将复杂度降低到 $O(n \log n)$ 。



六 二叉搜索树 BST(Binary Search Tree)

各数据项依所持关键码而彼此区分，循关键码访问：call-by-KEY。关键码之间必须同时支持比较（大小）与比对（相等）。数据集中的数据项，统一地表示和实现为词条（entry）形式。

词条

```
template <typename K, typename V> struct Entry { //词条模板类
    K key; V value; //关键码、数值
    Entry( K k = K(), V v = V() ) : key(k), value(v) {}; //默认构造函数
    Entry( Entry<K, V> const & e ) : key(e.key), value(e.value) {}; //克隆
    // 比较器、判等器（从此，不必严格区分词条及其对应的关键码）
    bool operator< ( Entry<K, V> const & e ) { return key < e.key; } //小于
    bool operator> ( Entry<K, V> const & e ) { return key > e.key; } //大于
    bool operator==( Entry<K, V> const & e ) { return key == e.key; } //等于
    bool operator!=( Entry<K, V> const & e ) { return key != e.key; } //不等
};
```

6.1 顺序性—BST 的中序遍历

BST 的存储需要保证：

- 任一节点均不小于/大于其左/右后代

与 任一节点均不小于/不大于其左/右孩子并不等效

三位一体：节点 ~ 词条 ~ 关键码

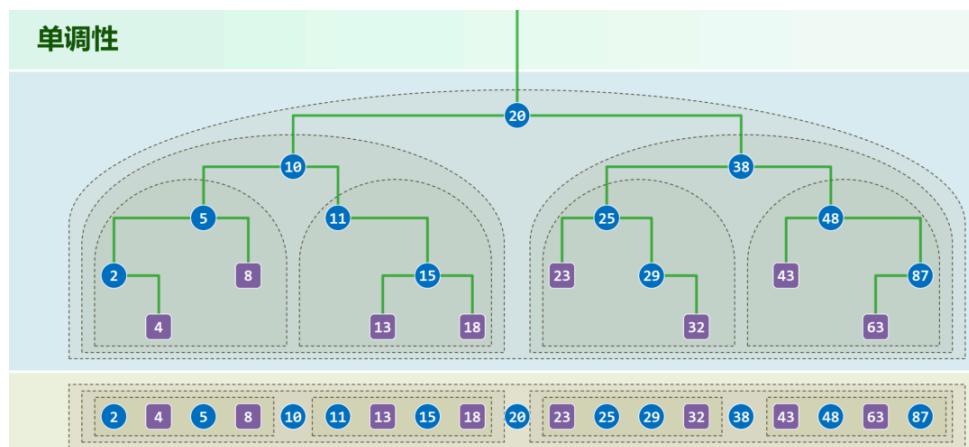


图 70 BST 的顺序性

顺序性虽只是对局部特征的刻画，却可导出 BST 的整体特征：**BST 的中序遍历序列，必然单调非降。**

BST 留出这样的接口：

```

template <typename T> class BST : public BinTree<T> {
public: //以 virtual 修饰，以便派生类重写
    virtual BinNodePosi<T> & search( const T & ); //查找
    virtual BinNodePosi<T> insert( const T & ); //插入
    virtual bool remove( const T & ); //删除
protected:
    BinNodePosi<T> _hot; //命中节点的父亲
    BinNodePosi<T> connect34( //3+4 重构
        BinNodePosi<T>, BinNodePosi<T>, BinNodePosi<T>,
        BinNodePosi<T>, BinNodePosi<T>, BinNodePosi<T>, BinNodePosi<T> );
    BinNodePosi<T> rotateAt( BinNodePosi<T> ); //旋转调整
};

```

6.2 BST 的基本算法与实现

6.2.1 查找 search()

从根节点出发，逐步地缩小查找范围，直到发现目标（成功），或抵达空树（失败）。本质上讲，就是有序向量的二分查找。

```

template <typename T> BinNodePosi<T> & BST<T>::search( const T & e ) {
    if ( !_root || e == _root->data ) //空树，或恰在树根命中
        { _hot = NULL; return _root; }
    for ( _hot = _root; ; ) { //否则，自顶而下
        BinNodePosi<T> & v = ( e < _hot->data ) ? _hot->lc : _hot->rc; //深入一层
        if ( !v || e == v->data ) return v; _hot = v; //一旦命中或抵达叶子，随即返回
    } //返回目标节点位置的引用，以便后续插入、删除操作
} //无论命中或失败，_hot 均指向 v 之父亲 (v 是根时，hot 为 NULL)

```

复杂度是 $O(h)$ ，其中 h 是 BST 的高度。若 BST 退化为链，则复杂度退化为 $O(n)$ 。

6.2.2 插入 insert()

先借助 `search(e)` 确定插入位置及方向。若 `e` 尚不存在，则再将新节点作为叶子插入

- `_hot` 为新节点的父亲
- `v = search(e)` 为 `_hot` 对新孩子的引用

令 `_hot` 通过 `v` 指向新节点

```

template <typename T> BinNodePosi<T> BST<T>::insert( const T & e ) {
    BinNodePosi<T> & x = search( e ); //查找目标（留意 _hot 的设置）
    if ( !x ) { //既禁止雷同元素，故仅在查找失败时才实施插入操作
        x = new BinNode<T>( e, _hot ); //在 x 处创建新节点，以 _hot 为父亲
        _size++; updateHeightAbove( x ); //更新全树规模，更新 x 及其历代祖先的高度
    }
    return x; //无论 e 是否存在于原树中，至此总有 x->data == e
} //验证：对于首个节点插入之类的边界情况，均可正确处置

```

时间主要消耗耗于 `search(e)` 和 `updateHeightAbove(x)`, 均为 $O(h)$, 其中 h 是 BST 的高度。

6.2.3 删除 `remove()`

```
template <typename T> bool BST<T>::remove( const T & e ) {
    BinNodePosi<T> & x = search( e ); //定位目标节点
    if ( !x ) return false; //确认目标存在 (此时 _hot 为 x 的父亲)
    removeAt( x, _hot ); _size--;
    _size--; updateHeightAbove( _hot ); //更新全树规模, 更新 _hot 及其历代祖先的高度
    return true;
} //删除成功与否, 由返回值指示
```

这样, 时间主要消耗于 `search(e)` 和 `updateHeightAbove(x)`, 后面证明 `removeAt(x, _hot)` 也为 $O(h)$, 其中 h 是 BST 的高度。

删除将分为两种情况:

1. 单分支

该节点只有一个孩子, 直接将其孩子接入其父亲即可。

```
template <typename T> static BinNodePosi<T>
removeAt( BinNodePosi<T> & x, BinNodePosi<T> & hot ) {
    BinNodePosi<T> w = x; //实际被摘除的节点, 初值同 x
    BinNodePosi<T> succ = NULL; //实际被删除节点的接替者
    if ( ! HasLChild( *x ) ) succ = x = x->rc; //左子树为空
    else if ( ! HasRChild( *x ) ) succ = x = x->lc; //右子树为空
    else { /* ...左、右子树并存的情况, 略微复杂些... */ }
    hot = w->parent; //记录实际被删除节点的父亲
    if ( succ ) succ->parent = hot; //将被删除节点的接替者与 hot 相联
    release( w->data ); release( w ); return succ; //释放被摘除节点, 返回接替者
} //此类情况仅需 O(1)时间
```

2. 双分支

该节点有两个孩子, 需要找到其直接后继 (或直接前驱) 节点, 将其值替换到该节点, 然后删除直接后继 (或直接前驱) 节点。由于直接后继一定没有左儿子, 从而转化为单分支情况。

```
template <typename T> static BinNodePosi<T>
removeAt( BinNodePosi<T> & x, BinNodePosi<T> & hot ) {
/* ..... */
else { //若 x 的左、右子树并存, 则
    w = w->succ(); swap( x->data, w->data ); //令*x 与其后继*w 互换数据
    BinNodePosi<T> u = w->parent; //原问题即转化为, 摘除非二度的节点 w
    ( u == x ? u->rc : u->lc ) = succ = w->rc; //兼顾特殊情况: u 可能就是 x
}
```

```

/* ..... */
} //时间主要消耗于 succ(), 正比于 x 的高度—更精确地, search()与 succ()总共不
过 O(h)

```

6.3 平衡二叉搜索树 BBST

6.3.1 平衡

若不能有效地控制树高, 就无法体现出 BST 相对于向量、列表等数据结构的明显优势, 比如在最(较)坏情况下, 二叉搜索树可能彻底地(接近地)退化为列表, 此时的性能不仅没有提高, 而且因为结构更为复杂, 反而会(在常系数意义上)下降。

用两种统计学口径分析平衡性:

1. 随机生成: 将 n 个词条 $\{e_i\}$ 随机插入一棵空树按随机排列 $\sigma = (i_1, i_2, \dots, i_n)$, 得到一棵随机生成的 BSTT, 其高度 h_T 是一个随机变量, 假定所有 BST 等概率地出现, 其期望值为 $E(h_T) = O(\log n)$
2. 随即组成: 将一样的拓扑结构视作一类, 随机生成的 BSTT 的高度 h_T 是一个随机变量, 假定所有 BST 等概率地出现, 其期望值为 $E(h_T) = O(\sqrt{n})$

n 个节点组成的 BST 的个数为 $S(n)$ 则

$$S(n) = \sum_{i=1}^n S(i-1)S(n-i) = \text{catalan}(n) = \frac{(2n)!}{n!(n+1)!}$$

理想随机在实际中绝难出现: 局部性、关联性、(分段)单调性、(近似)周期性、...较高甚至极高的 BST 频繁出现; 平衡化处理很有必要。

由 n 个节点组成的二叉树, 高度不致低于 $\lfloor \log_2(n+1) \rfloor$ 。达到这一下界时, 称作理想平衡。

而渐近平衡在渐近的意义下, 高度不致超过 $O(\log n)$ 。满足这样的 BST 称为平衡二叉树(Balanced Binary Search Tree, BBST)。

6.3.2 平衡等价变换

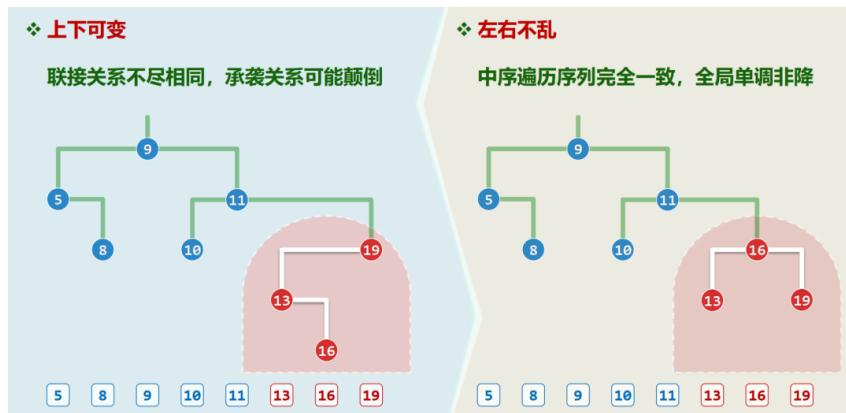


图 71 等价 BST

限制条件 + 局部性:

各种 BBST 都可视作 BST 的某一子集, 相应地满足精心设计的限制条件

- 单次动态修改操作后, 至多 $O(\log n)$ 处局部不再满足限制条件 (可能相继违反, 未必同时)
- 可在 $O(\log n)$ 时间内, 使这些局部 (以至全树) 重新满足

等价变换 + 旋转调整: 序齿不序爵

刚刚失衡的 BST, 必可速转换为一棵等价的 BBST。

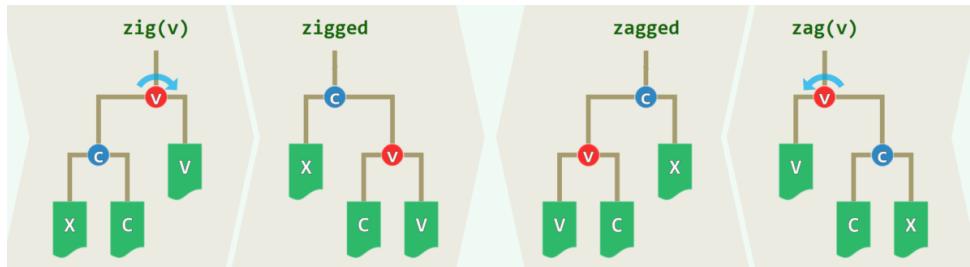


图 72 等价变换

zig 和 **zag**: 仅涉及常数个节点, 只需调整其间的联接关系; 均属于局部的基本操作。调整之后: **v/c** 深度加/减 1, 子(全)树高度的变化幅度, 上下差异不超过 1。

实际上, 经过不超过 $O(n)$ 次旋转, 等价的 BST 均可相互转化。

6.4 AVL 树

G. Adelson-Velsky & E. Landis (1962) 提出的平衡二叉搜索树, 以其发明者的名字命名。

6.4.1 AVL 树的定义

AVL 的核心是: 平衡因子 (Balance Factor, BF)。

$$BF(v) = \text{height}(v \rightarrow lc) - \text{height}(v \rightarrow rc)$$

AVL 在每次操作后要进行维护, 保证:

$$\forall v \in T, |BF(v)| \leq 1$$

AVL 树未必理想平衡, 但必然渐近平衡。

6.4.1.1 AVL 渐近平衡

对于固定高度 h 的 AVL 树, 其最少节点数 $S(h)$ 满足递推关系:

$$S(h) = S(h-1) + S(h-2) + 1$$

从而 $S(h) = \text{fib}(h+3) - 1$, 从而对于 n 个节点构成的 AVL 树, 其高度不会超过 $O(\log n)$ 。

6.4.1.2 Fibonacci Tree

高度为 h , 规模恰好为 $S(h)$ 的 AVL 树, 称为 **Fibonacci 树** (Fibonacci Tree)。

是最“瘦”的、临界的 AVL 树。

6.4.1.3 AVL 接口

```
#define Balanced(x) ( stature( (x).lc ) == stature( (x).rc ) ) //理想平衡
#define BalFac(x) (stature( (x).lc ) - stature( (x).rc ) ) //平衡因子
#define AvlBalanced(x) ( ( -2 < BalFac(x) ) && (BalFac(x) < 2 ) ) //AVL 平衡条件
template <typename T> class AVL : public BST<T> { //由 BST 派生
public: //BST::search()等接口, 可直接沿用
    BinNodePosi<T> insert( const T & ); //插入(重写)
    bool remove( const T & ); //删除(重写)
};
```

6.4.2 重平衡

AVL 树的插入和删除操作，都可能导致局部失衡，需要通过旋转调整来重平衡。

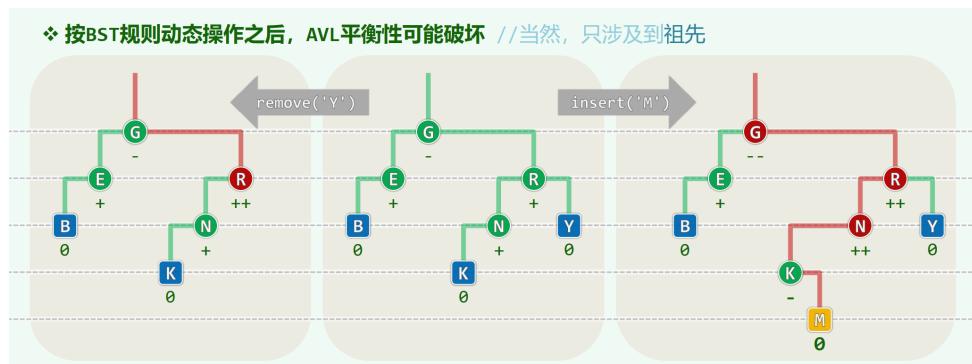


图 73 AVL 树的重平衡

- 插入：从祖父开始，每个祖先都有可能失衡，且可能同时失衡。
- 删除：从父亲开始，每个祖先都有可能失衡，但至多一个。

利用旋转变换进行重平衡：

- 局部性：所有的旋转都在局部进行，每次只需 $O(1)$ 时间
- 快速性：在每一深度只需检查并旋转至多一次，共 $O(\log n)$ 次

6.4.2.1 插入

插入分为两种情况：

单旋：黄色方块恰好存在其一

只需要经过一次 **zag** 或者 **zig**，并且旋转后的子树高度不变严格变回插入之前，即可恢复平衡，不需要再向上探；并且该子树的父亲的 BF 不变，不会导致更高层的失衡。



图 74 AVL 树的插入

双旋

需要经过两次 zig 或者 zag，并且旋转后的子树高度不变严格变回插入之前，即可恢复平衡，不需要再向上探；并且该子树的父亲的 BF 不变，不会导致更高层的失衡。

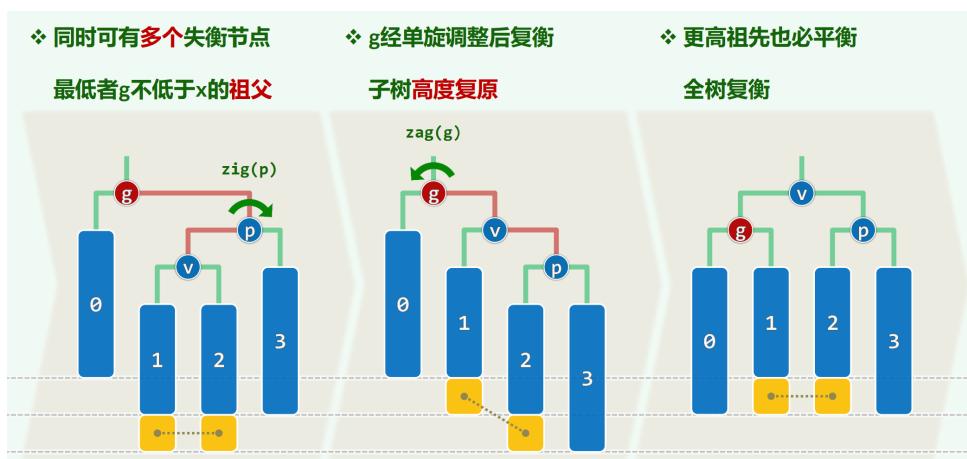


图 75 AVL 树的插入

注意：即便 g 未失衡，高度亦可能增加。

```
template <typename T> BinNodePosi<T> AVL<T>::insert( const T & e ) {
    BinNodePosi<T> & x = search( e ); if ( x ) return x; //若目标尚不存在
    BinNodePosi<T> xx = x = new BinNode<T>( e, _hot ); _size++;
    // 此时，若 x 的父亲_hot 增高，则祖父有可能失衡
    for ( BinNodePosi<T> g = _hot; g; g = g->parent ) //从_hot 起，逐层检查各代祖先 g
        if ( ! AvlBalanced( *g ) ) { //一旦发现 g 失衡，则通过调整恢复平衡
            FromParentTo(*g) = rotateAt( tallerChild( tallerChild( g ) ) );
            break; //局部子树复衡后，高度必然复原；其祖先亦必如此，故调整结束
        }
}
```

```

    } else //否则 (g 仍平衡)
        updateHeight( g ); //只需更新其高度 (注意: 即便 g 未失衡, 高度亦可能增
加)
    return xx; //返回新节点位置
}

```

插入的时间主要在 `search(e)` 上, 为 $O(\log n)$, 其余操作均为 $O(1)$, 故总体复杂度为 $O(\log n)$ 。

6.4.2.2 删除

删除分为两种情况:

单旋: 黄色方块至少存在其一; 红色方块可有可无

经过一次 `zag` 或者 `zig` 后, 可能失衡, 需要向上到根部, 进行调整。



图 76 AVL 树的删除

双旋

经过两次 `zag` 或者 `zig` 后, 可能失衡, 需要向上到根部, 进行调整。

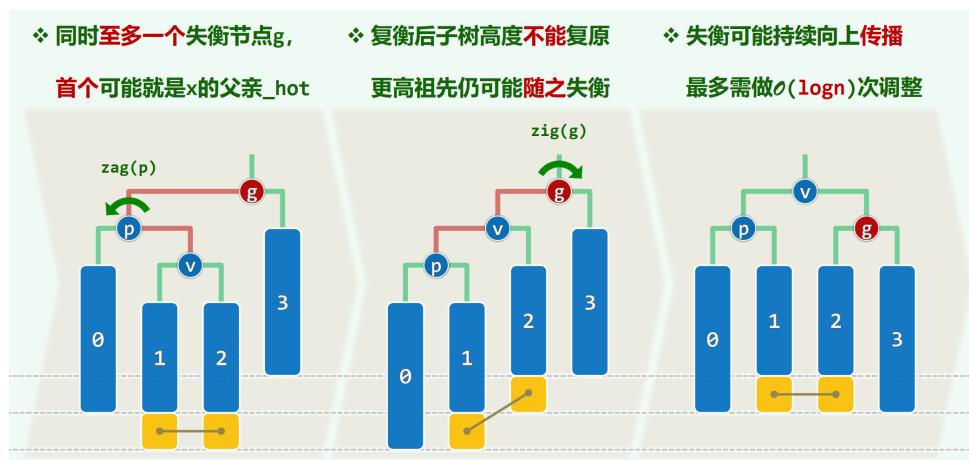


图 77 AVL 树的删除

```

template <typename T> bool AVL<T>::remove( const T & e ) {
    BinNodePosi<T> & x = search( e ); if ( !x ) return false; //若目标的确存在
    removeAt( x, _hot ); _size--; //则在按 BST 规则删除之后, _hot 及祖先均有可能失
衡
    // 以下, 从 _hot 出发逐层向上, 依次检查各代祖先 g
    for ( BinNodePosi<T> g = _hot; g; g = g->parent ) {
        if ( ! AvlBalanced( *g ) ) //一旦发现 g 失衡, 则通过调整恢复平衡
            g = FromParentTo( *g ) =
    rotateAt( tallerChild( tallerChild( g ) ) );
        updateHeight( g ); //更新高度 (注意: 即便 g 未曾失衡或已恢复平衡, 高度均可能
降低)
    } //可能需做过  $\Omega(\log n)$  次调整; 无论是否做过调整, 全树高度均可能下降
    return true; //删除成功
}

```

6.4.2.3 (3+4)-重构

zig 和 zag 的最终是通过(3+4)-重构来实现的。

设 g 为最低的失衡节点, 沿最长分支考察祖孙三代: $g \sim p \sim v$ 按中序遍历次序, 重命名为: $a < b < c$;

它们总共拥有四棵子树 (或为空), 按中序遍历次序, 重命名为: $T_0 < T_1 < T_2 < T_3$ 。

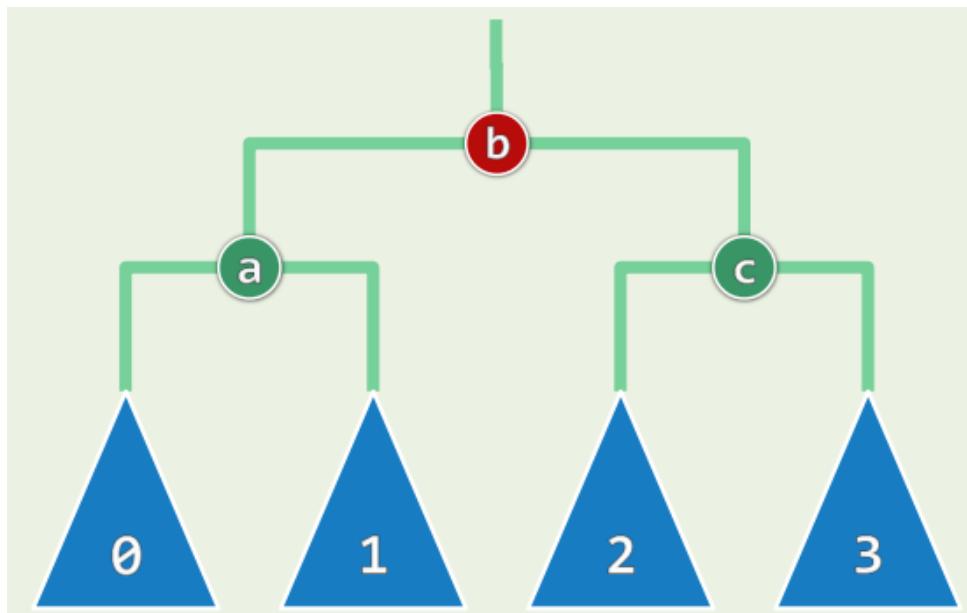


图 78 (3+4)-重构

```

template <typename T> BinNodePosi<T> BST<T>::connect34(
    BinNodePosi<T> a, BinNodePosi<T> b, BinNodePosi<T> c,
    BinNodePosi<T> T0, BinNodePosi<T> T1,
    BinNodePosi<T> T2, BinNodePosi<T> T3)

```

```
{
    a->lc = T0; if (T0) T0->parent = a;
    a->rc = T1; if (T1) T1->parent = a;
    c->lc = T2; if (T2) T2->parent = c;
    c->rc = T3; if (T3) T3->parent = c;
    b->lc = a; a->parent = b; b->rc = c; c->parent = b;
    updateHeight(a); updateHeight(c); updateHeight(b); return b;
}
```

利用 3+4 重构，实现 zag 和 zig：

```
template<typename T> BinNodePosi<T> BST<T>::rotateAt( BinNodePosi<T> v ) {
    BinNodePosi<T> p = v->parent, g = p->parent;
    if ( IsLChild( * p ) ) //zig
        if ( IsLChild( * v ) ) { //zig-zig
            p->parent = g->parent;
            return connect34( v, p, g, v->lc, v->rc, p->rc, g->rc );
        } else { //zig-zag
            v->parent = g->parent;
            return connect34( p, v, g, p->lc, v->lc, v->rc, g->rc );
        }
    else //zag
        if ( IsRChild( * v ) ) { //zag-zag
            p->parent = g->parent;
            return connect34( g, p, v, g->lc, p->lc, v->lc, v->rc );
        } else { //zag-zig
            v->parent = g->parent;
            return connect34( g, v, p, g->lc, v->lc, v->rc, p->rc );
        }
}
```

6.4.3 AVL 综合评价

优点：

- 无论查找、插入或删除，最坏情况下的复杂度均为 $O(\log n)$
- $O(n)$ 的存储空间

缺点：

- 借助高度或平衡因子，为此需改造元素结构，或额外封装；实测复杂度与理论值尚有差距
- 插入/删除后的旋转，成本不菲
- 删除操作后，最多需旋转 $\Omega(\log n)$ 次（Knuth：平均仅 0.21 次）
- 若需频繁进行插入/删除操作，未免得不偿失
- 单次动态调整后，全树拓扑结构的变化量可能高达 $\Omega(\log n)$

七 更多 BST

7.1 区间树 Interval Tree

看这样一个问题：

Stabbing Query: 给定集合

$$S = \{s_i = [x_i, x'_i] \mid 1 \leq i \leq n\}$$

以及一个待查询的点 q_x ， 目标是寻找所有的 s_i ，使得 q_x 在 s_i 的区间内，即

$$\{s_i \mid q_x \in s_i\}$$

为解决这个问题，我们引入区间树。

为了方便查询，我们需要进行预处理：

先找出区间端点构成的集合 $P = \partial S$ ，有 $|P| = 2n$ 。令 x_{mid} 为 P 中的中位数。

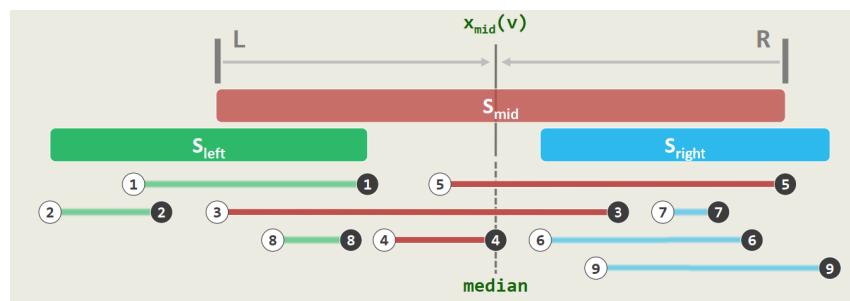


图 79 区间树——中位数

这些集合可分成三部分， S_{left} 是所有在 x_{mid} 左侧的区间， S_{right} 是所有在 x_{mid} 右侧的区间， S_{mid} 是所有包含 x_{mid} 的区间。

而 S_{left} 和 S_{right} 又可以分别继续递归地进行划分，直到只剩下一个区间，或者没有区间。

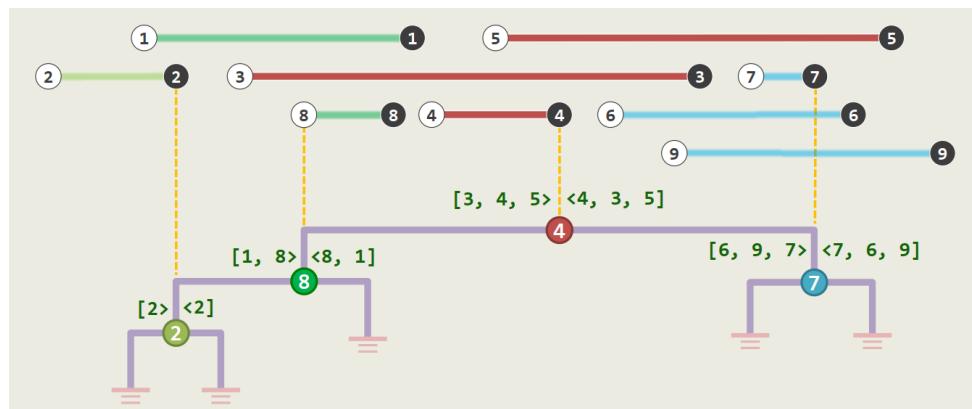


图 80 区间树——划分

这样，我们就得到了一棵二叉搜索树，称为区间树。

平衡性： 区间树的高度为 $O(\log n)$ 。

$$\max\{|S_{\text{left}}|, |S_{\text{right}}|\} \leq \frac{n}{2}$$

为了方便查询，我们还需要组织好区间树的结构。保证所有 S_{mid} 的区间都按照左/右排序。

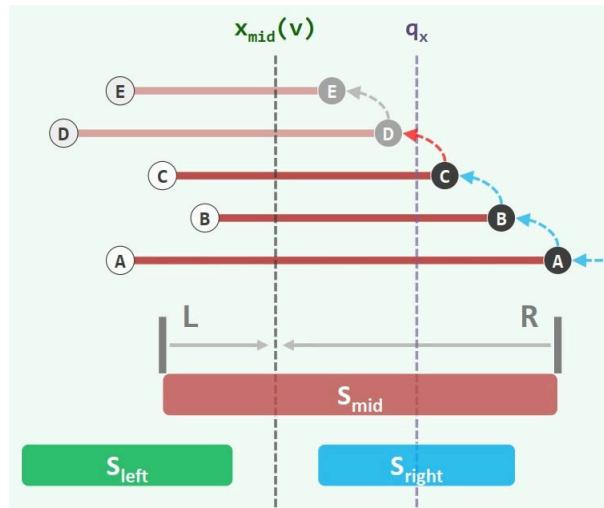


图 81 区间树——排序

空间大小： 是 $O(n)$ 的，因为每个区间只会在节点出现两次。

构造： 构造的时间是 $O(n \log n)$ 的，依次排序即可。

查询：

```
def queryIntervalTree( v, qx ):
    if ( ! v ) return; //base
    if ( qx < xmid(v) )
        report all segments of Smid(v) containing qx;
        queryIntervalTree( lc(v), qx );
    else if ( xmid(v) < qx )
        report all segments of Smid(v) containing qx;
        queryIntervalTree( rc(v), qx );
    else //with a probability ≈ 0
        report all segments of Smid( v ); //both rc(v) & lc(v) can be ignored
```

在查询时候，每次都从根节点开始，如果 q 在当前节点的区间内，则将当前节点加入结果集，然后递归地查询左右子树。总的查询时间是 $O(\log n + k)$ 的，其中 k 是结果集的大小。

7.2 线段树 Segment Tree

7.2.1 基本区间 Elementary Intervals

对于 n 个区间 $I = \{s_i = [x_i, x'_i] \mid 1 \leq i \leq n\}$ ，可以将区间端点排序 $\{p_1, p_2, \dots, p_{\{m\}}\}$ ，其中 $m \leq 2n$ 。将整个区间分成 $m + 1$ 个基本区间， $(-\infty, p_1], (p_1, p_2], \dots, (p_m, \infty)$ 。

对于给定的区间，我们就可以实现离散化（Discretization）。在每段基本区间上，他们有一样的性质。

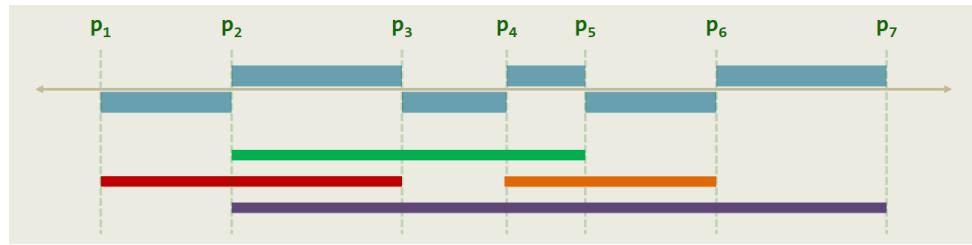


图 82 线段树——基本区间

可以用 $O(\log n)$ 二分查找目标区间，然后在 $O(k)$ 的进行输出，其中 k 是结果集的大小。

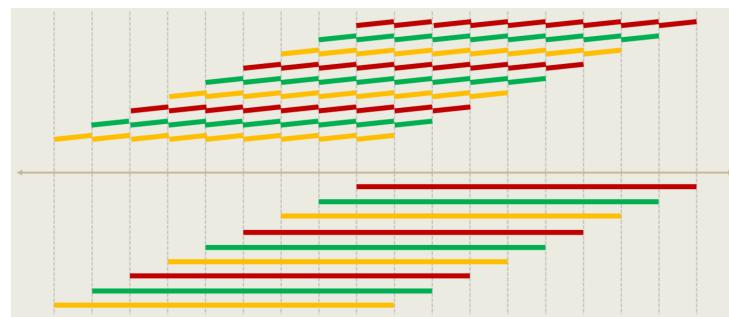


图 83 线段树——基本区间——最坏情况

但最坏情况需要占用 $O(n^2)$ 的空间。

7.2.2 线段树 Segment Tree

为了解决上述问题，我们引入线段树。线段树是个完全二叉树，最底层的每个节点都是一个基本区间。

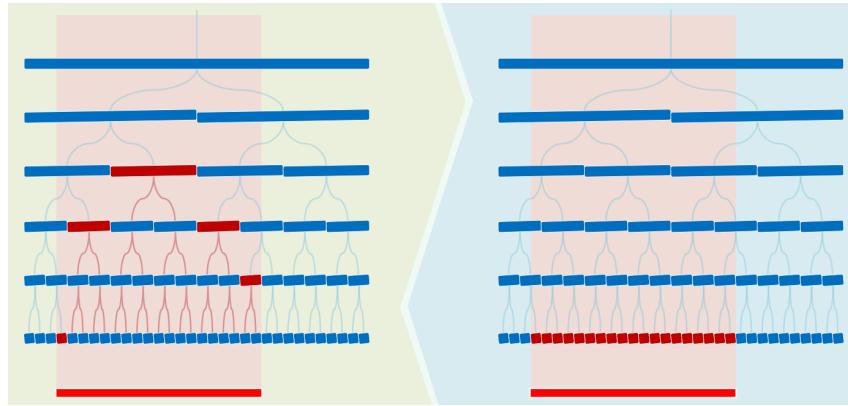


图 84 线段树

在存储的时候，先在最底层存满，对有共同祖先的区间向上贪婪合并(greedy merging)。

对于多个区间可以如下储存：



图 85 线段树——储存

这些合并的区间被称为标准子集 **Canonical Subsets**，占用的空间是 $O(n \log n)$ 的。

```

def BuildSegmentTree(I):
    Sort all endpoints in I before
        determining all the EI's //O(n log n)
    Create T a BBST on all the EI's //O(n)
        Determine R(v) for each node v
    //O(n) if done in a bottom-up manner
    For each s of I
        InsertSegment( T.root, s )
    
```

每次插入区间贪婪合并，但事实上实现是从顶层摔下去，保留包含在插入标区间内的节点区间。

```

def InsertSegment( v , s ):
    if ( R(v) is subset of s ) //greedy by top-down
        store s at v and return;
    if ( R( lc(v) ) n s != Empty ) //recurse
        InsertSegment( lc(v), s );
    if ( R( rc(v) ) n s != Empty ) //recurse
        InsertSegment( rc(v), s );

```

需要 $O(\log n)$ 的时间。

查询也很容易，只需要从根到叶子，逐层报告结果即可。

```

def Query( v , qx ):
    report all the intervals in Int(v)
    if ( v is a leaf )
        return
    if ( qx in R( lc(v) ) )
        Query( lc(v), qx )
    else //qx in R( rc(v) )
        Query( rc(v), qx )

```

查询的时间是 $O(\log n + k)$ 的，其中 k 是结果集的大小。因为每次在标准子集上的时间是 $k_i + 1$ ，而从根到叶子的时间是 $O(\log n)$ 。

7.3 高阶搜索树 Multi-Level Search Tree

7.3.1 Range Query

考虑 Range Query 问题

7.3.1.1 1D 情况

给定 $P = \{p_1, p_2, \dots, p_n\}$ 是在数轴上排列的 n 个点，给定一个查询区间 $I = [x, y]$ ，目标是找出所有在 I 内的点。

Brute-Force 的方法是 $O(n)$ 的。

但是我们可以用二分查找的方法：先补充 $P[0] = -\infty$ ，可以二分查找 I 的右端点，之后回退，直到找到左端点。这样可以将时间降低到 $O(\log n + k)$ ，其中 k 是结果集的大小。

```

For any interval I = (x1, x2]
    Find t = search(x2) = max{ i | p[i] <= x2 } //O(logn)
    Traverse the vector BACKWARD from p[t] and report each point //O(k)
    until escaping from I at point p[s]
    return k = t - s //output size

```

输出敏感度 (Output-Sensitivity)：如果 k 很小，那么算法的时间就很小。但是如果 k 很大，可能不如两次二分查找。

这个方法也无法拓展到 2D 情况。

7.3.1.2 2D 情况

给定 $P = \{p_1, p_2, \dots, p_n\}$ 是在平面上排列的 n 个点，给定一个查询区间 $I = [x_1, x_2] * [y_1, y_2]$ ，目标是找出所有在 I 内的点。

可以用类似动态规划的记忆法记住从左下角到该点的信息，再用容斥原理，可以得到一个小矩形内的信息。

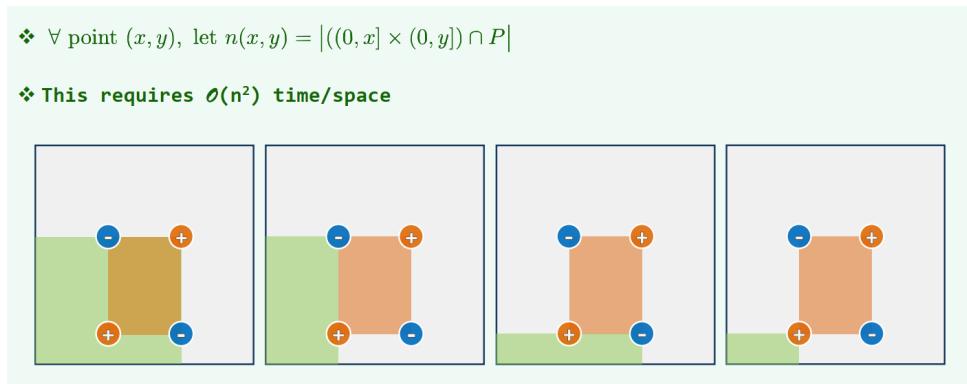


图 86 Range Query——2D——预处理

这样，我们就可以在 $O(\log n)$ （因为要用二分查找最近的给定点）的时间内得到一个小矩形内的信息。

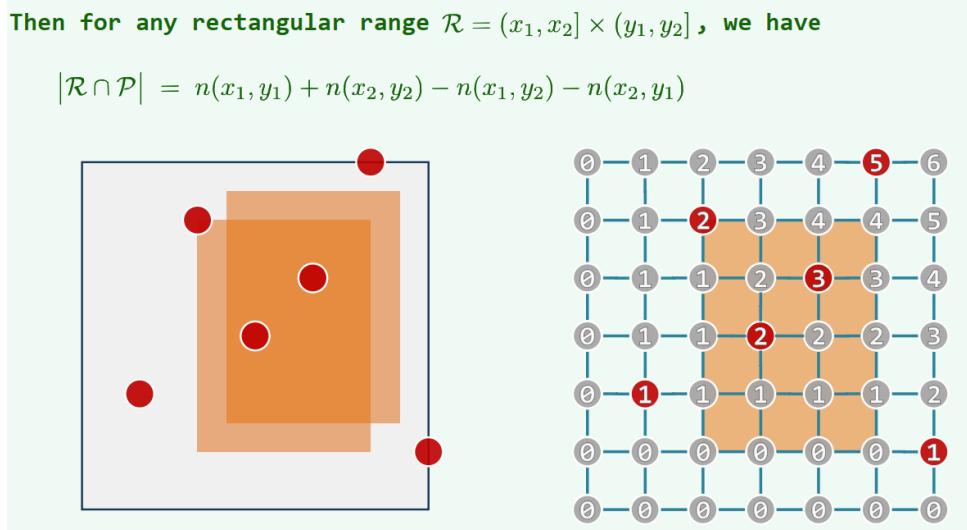


图 87 Range Query——2D——查询

但要占用 $O(n^2)$ 的空间。

7.3.2 Multi-Level Search Tree: 1D

结构是一个 Complete (Balanced) BST，重构为以下形式：

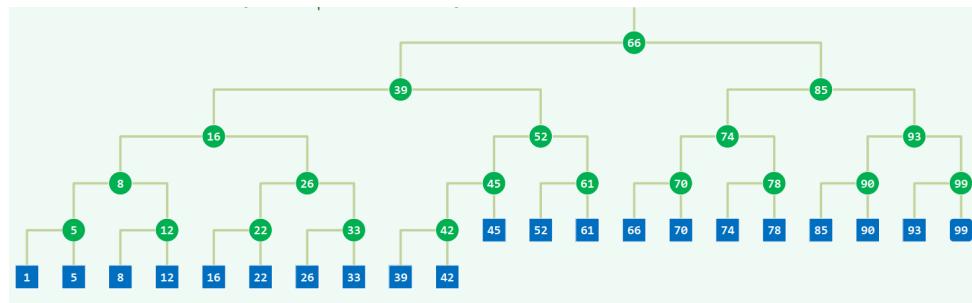


图 88 Multi-Level Search Tree—1D

$$\forall v, v.\text{key} = \min\{u.\text{key} \mid u \in v.\text{rTree}\} = v.\text{succ.key}$$

则有性质 $\forall u \in v.\text{lTree}, u.\text{key} < v.\text{key}$ 和 $\forall u \in v.\text{rTree}, u.\text{key} \geq v.\text{key}$ 。令 `search(x)` 返回最大的 u , 使得 $u.\text{key} \leq x$ 。

保证树是完全二叉的, 这样可以保证叶节点恰好存满所有给定的数据。这棵树可以在一个完全二叉树的基础上改进得到。原先二叉树最下层的每一个节点的左儿子 (如果没有的话) 存其前驱, 而右节点存自己本身, 就可以得到这棵树。

核心想法是寻找最低的公共祖先 (Lowest Common Ancestor)

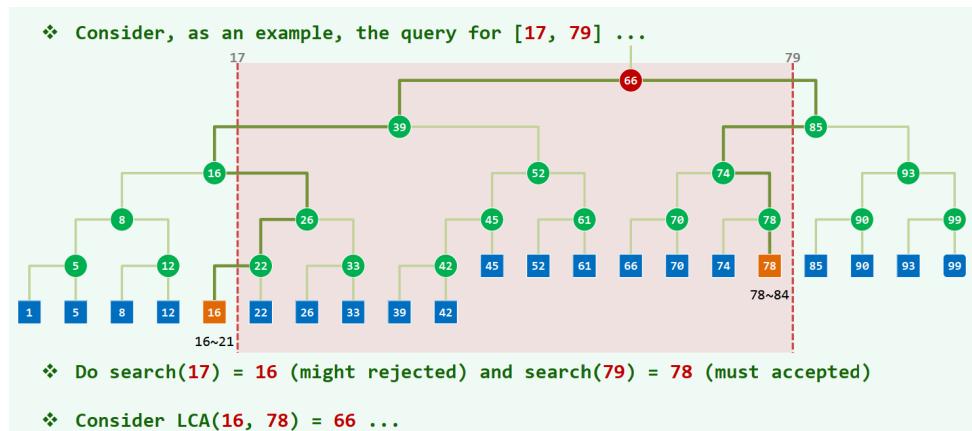


图 89 Multi-Level Search Tree—1D—查找

由公共祖先 LCA 出发, 取从左上来的路上节点的右子树, 和从右上来的路上节点的左子树, 就可以得到结果。

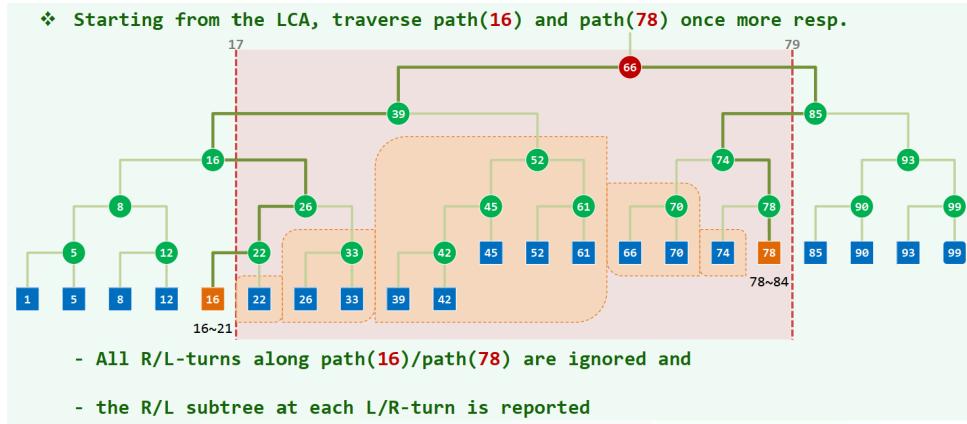


图 90 Multi-Level Search Tree—1D—查找

查询复杂度是 $O(\log n + k)$, 预处理复杂度是 $O(n \log n)$, 空间复杂度是 $O(n)$ 。

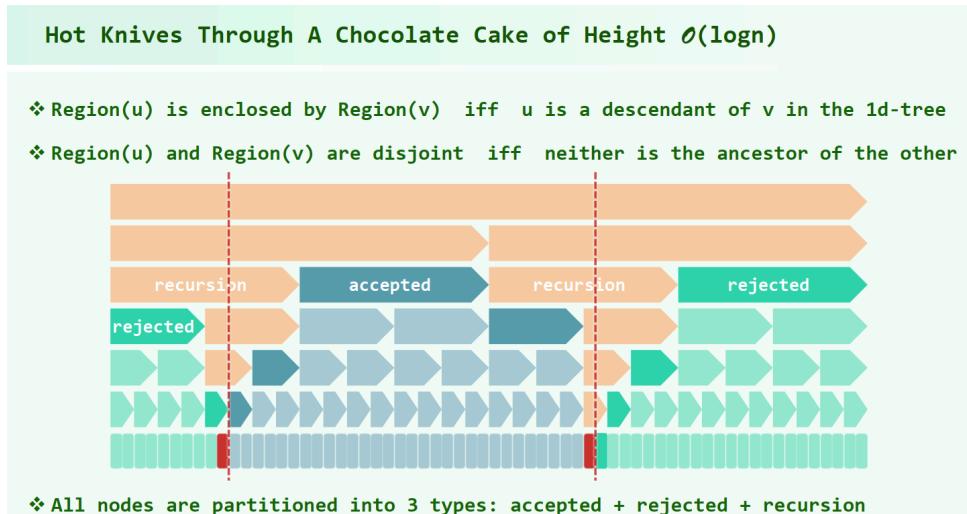


图 91 Multi-Level Search Tree—1D—总结

用线段树理解就是上图的样子。

7.3.3 Multi-Level Search Tree: 2D

7.3.3.1 2D Range Query = x-Query + y-Query

先对 x-Query, 再对剩余的候选者做 y-Query。

对于最坏的情况, 用 k-d tree 的方法可以做到 $O(1 + \sqrt{n})$, 但是用 Multi-Level Search Tree 的方法需要做到 $O(n)$ 。

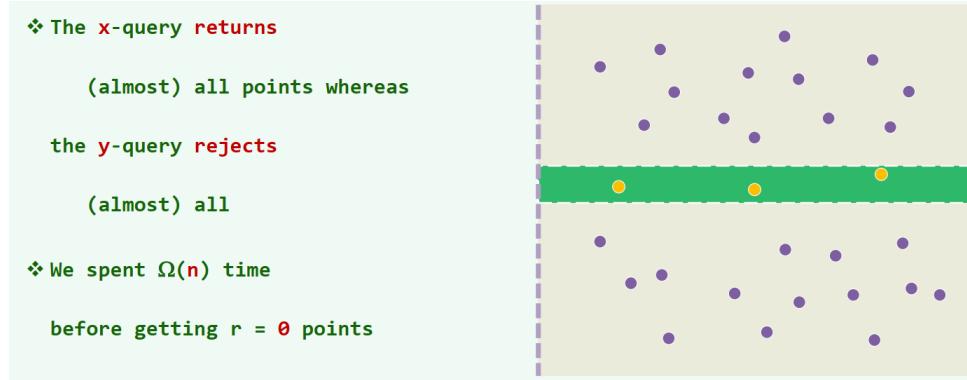


图 92 Multi-Level Search Tree——2D——最坏情况

7.3.3.2 2D Range Query = x-Query * y-Query

需要构造一棵树的树：

- 为第一个维度的 query 问题 (x-query) 构造一个一维的 BBST(x-tree)
- 而对于每个 x-range tree 的节点 v，建立一个 y 维度的 BBST(y-tree)，其中包含与 v 关联的标准子集(canonical subset)

也就是构造一个 x-tree 和数个 y-tree，称作 Multi-Level Search Tree。

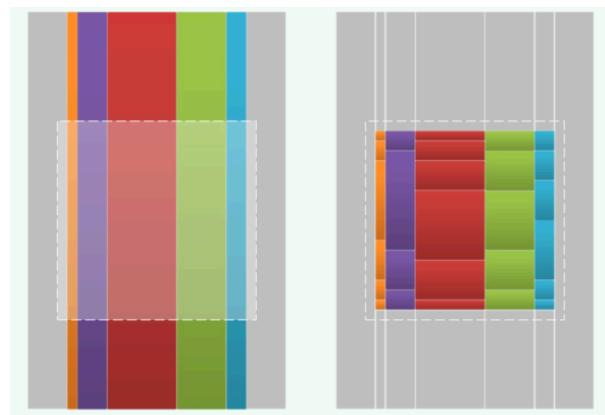


图 93 Multi-Level Search Tree——2D——查找

这样的复杂度是 $O(\log^2 n + k)$ ，其中 k 是结果集的大小。

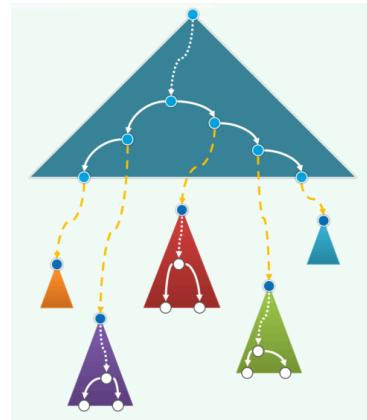


图 94 Multi-Level Search Tree——2D——构造

Query Algorithm:

```

1. Determine the canonical subsets of points that satisfy the first query
// there will be O(log n) such canonical sets,
// each of which is just represented as a node in the x-tree
1. Find out from each canonical subset which points lie within the y-range
// To do this,
// for each canonical subset,
// we access the y-tree for the corresponding node
// this will be again a 1D range search (on the y-range)

```

整体的复杂度是：

对于一个 2 阶搜索树，对于空间中的 n 个点，需要 $O(n \log n)$ 的时间构造， $O(n \log n)$ 的空间， $O(\log^2 n + k)$ 的时间查询。

7.3.4 Multi-Level Search Tree: dD

对于 d 维的情况，整体的复杂度是：

对于一个 d 阶搜索树，对于空间中的 n 个点，需要 $O(n \log^{d-1} n)$ 的时间构造， $O(n \log^{d-1} n)$ 的空间， $O(\log^d n + k)$ 的时间查询。

7.4 kD 树 k Dimensional Tree

我们想把 BBST 的搜索策略应用到几何范围搜索（Geometric Range Search, GRS）问题中。

从单一区域（整个平面）开始

- 在每个偶/奇数层上
- 垂直/水平划分区域递归划分子区域

为了使其正常工作

- 每个分区应尽可能均匀（中位数）
- 每个区域定义为开/封在左下方/右上方

大致划分过程如下：

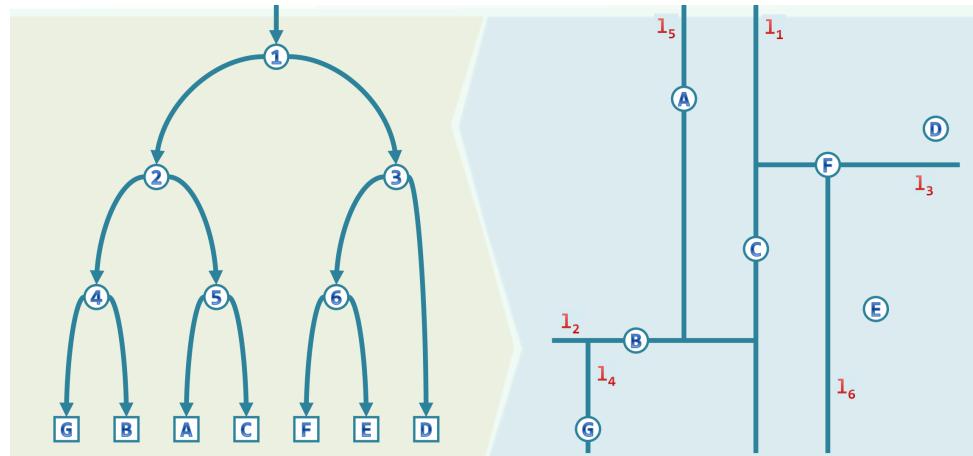


图 95 kD 树——划分

有时候对于二维平面，可以用 quadTree，四叉树，来进行划分。

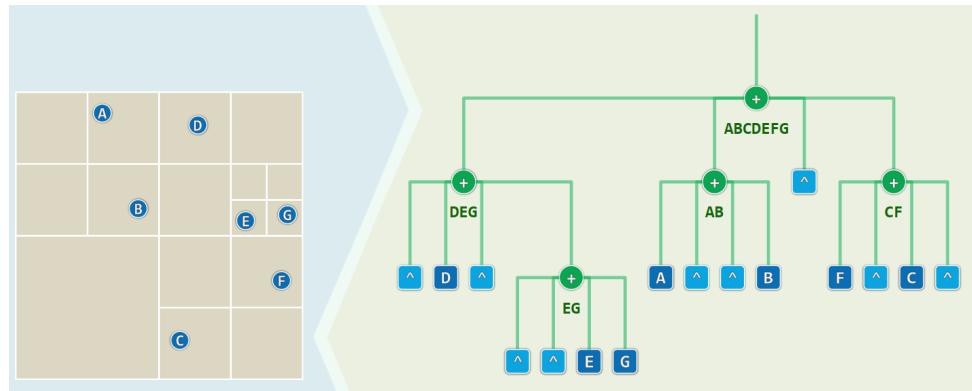


图 96 quadTree

7.4.1 kD 树的构造

```
buildKdTree(P,d) //construct a 2d-tree for point set P at depth d
    if ( P == {p} ) return createLeaf( p ) //base
    Root = createKdNode()
    Root->SplitDirection = even(d) ? VERTICAL : HORIZONTAL
    Root->SplitLine = findMedian( root->SplitDirection, P ) //O(n) !
    ( P1, P2 ) = divide( P, Root->SplitDirection, Root->SplitLine ) //DAC
    Root->LC = buildKdTree( P1, d + 1 ) //recurse
    Root->RC = buildKdTree( P2, d + 1 ) //recurse
    return( Root )
```

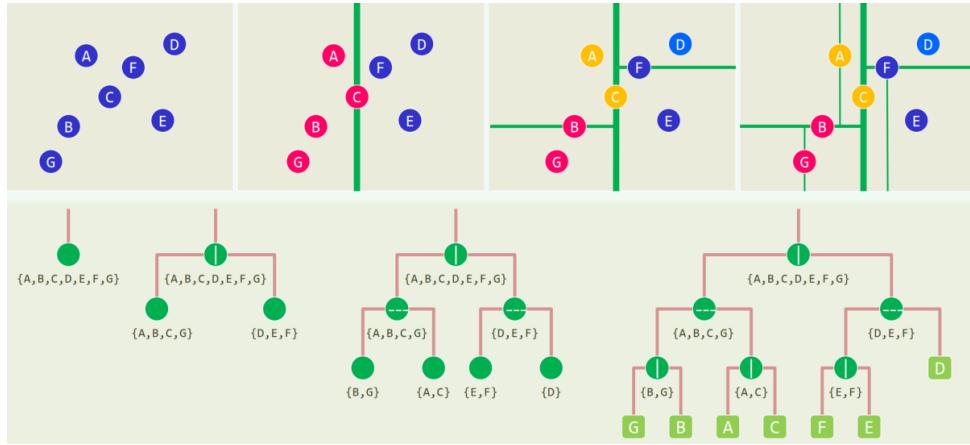


图 97 kD 树——构造

7.4.2 标准子集 Canonical Subset

每个节点对应

- 平面的一个矩形子区域，以及
- 子区域中包含的点的子集，每一个都被称为典型子集（Canonical Subset）

对于每个有子节点 L 和 R 的内部节点 X，有： $\text{region}(X) = \text{region}(L) \cup \text{region}(R)$

同一深度的节点子区域互不相交，且它们的并集覆盖整个平面。

每个二维范围查询都可以由多个 CS 的并集来回答。

7.4.3 kD 树的查询

```
def kdSearch(v, R): // 热刀来切千 (logn) 层巧克力
    if ( isLeaf( v ) )
        if ( inside( v, R ) ) report(v)
        return
    if ( region( v->lc ) ⊆ R )
        reportSubtree( v->lc )
    else if ( region( v->lc ) ∩ R != Empty )
        kdSearch( v->lc, R )
    if ( region( v->rc ) ⊆ R )
        reportSubtree( v->rc )
    else if ( region( v->rc ) ∩ R != Empty )
        kdSearch( v->rc, R )
```

和 BBST 的查询类似，不断向两个子树递归。

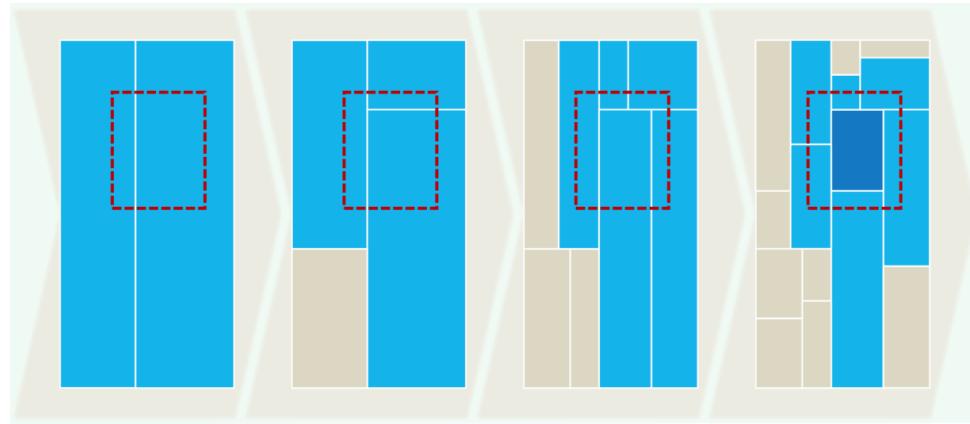


图 98 kD 树——查询

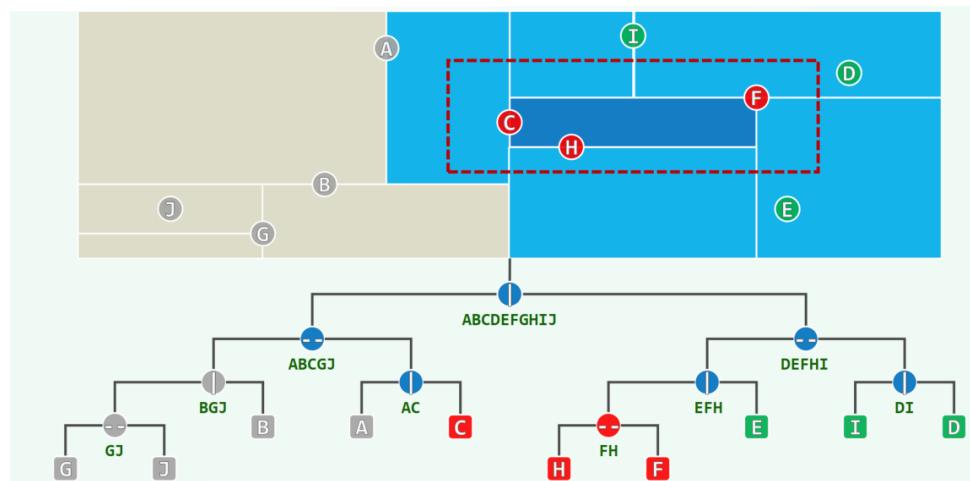


图 99 kD 树——查询

当然，对于最后得结果，我们会发现最终的目标区域还和一些周围的区域有交集。对于这种相交但不包含的，直接查询叶子是否在其中即可。

如果想避免这种情况，可以适当缩小 Bounding Box。

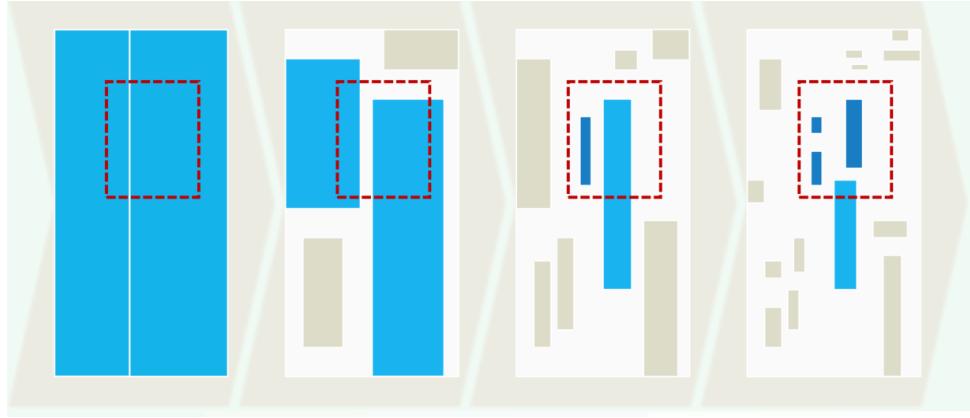


图 100 kD 树——查询

7.4.4 kD 树的性能

- **Preprocessing:** 将平面划分成 n 个区域，满足 $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$ ，所以 $T(n) = O(n \log n)$ 。
- **Storage:** 树的高度是 $O(\log n)$ ，所以空间是 $O(n)$ 。
- **Query Time:** $O(\sqrt{n} + k)$ ，其中 k 是结果集的大小。
 搜索时间取决于 $Q(n)$ ：
 - 递归调用次数，即
 - 与查询区域相交的子区域（各级），而在完全在查询区域内的子区域（各级）不会被递归调用。

可以证明：从被分成四块区域开始，每块分别对应一条边。下面的叙述是对于与一条边相交的情况。

每个被递归的节点至多有 2 个孙子(隔层比较)会被递归，即 $Q(n) = 2Q\left(\frac{n}{4}\right) + O(1)$ ，所以 $Q(n) = O(\sqrt{n})$ 。

更一般地，对于 d 维的情况，整体的复杂度是：

- constructed: $O(n \log n)$
- size: $O(n)$
- query time: $O\left(n^{\{1-\frac{1}{d}\}} + k\right)$

八 高级 BST

8.1 伸展树 Splay Tree

8.1.1 局部性/Locality

时间：刚被访问过的节点，极有可能很快地再次被访问

空间：下一将要访问的节点，极有可能就在刚被访问过节点的附近

AVL 连续的 m 次查找 ($m \gg n$)，共需 $O(m \log n)$ 时间，希望可以利用局部性加速。

- 自适应链表：节点一旦被访问，随即移动到最前端
- 模仿：希望 BST 的节点一旦被访问，随即调整到树根

如果节点被访问，就将其旋转到根节点，这样下次访问时，就可以直接访问到了。

8.1.2 逐层伸展

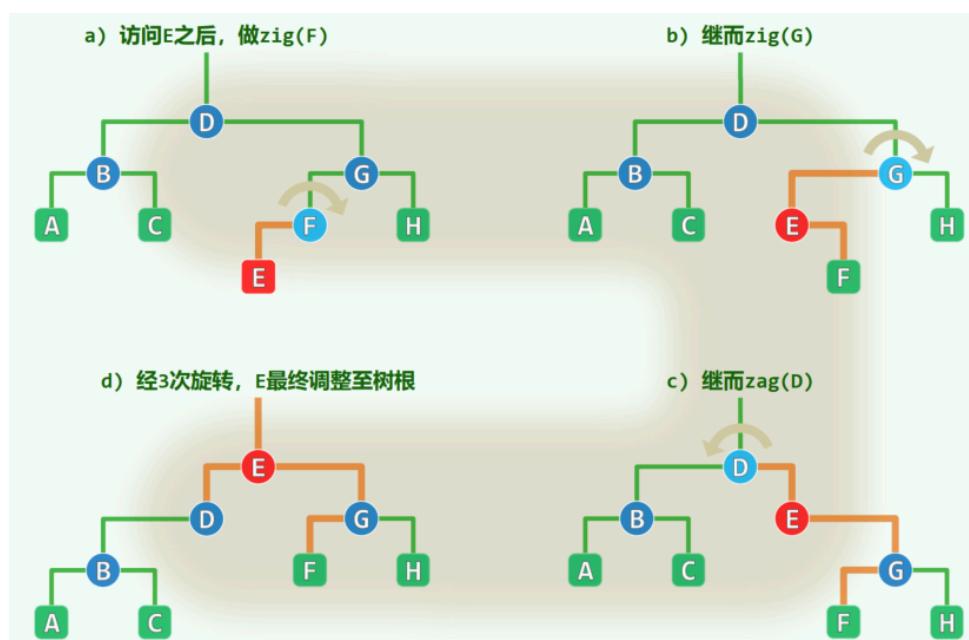


图 101 逐层伸展

但这样很有可能导致树的不平衡，比如下面的例子：

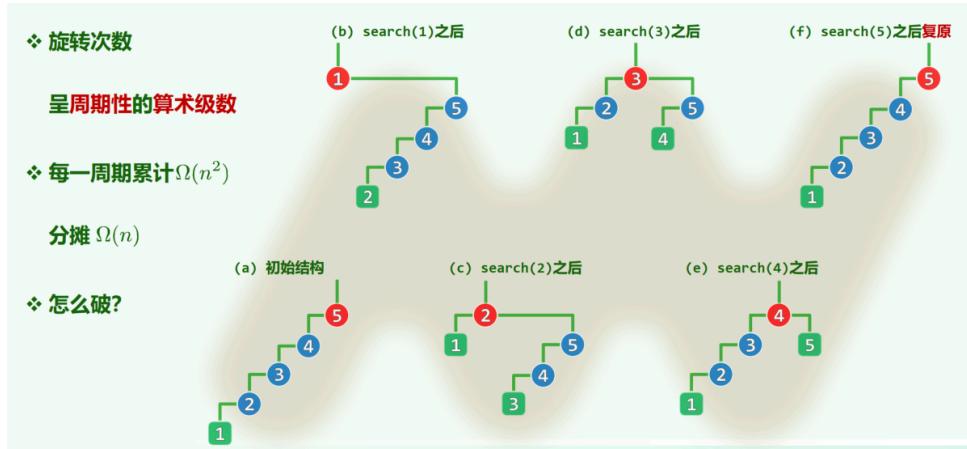


图 102 逐层伸展——最坏情况

8.1.3 双层伸展

向上追溯两层，而非一层。

反复考察祖孙三代： $g = \text{parent}(p)$, $p = \text{parent}(v)$, v 。根据它们的相对位置，经两次旋转，使 v 上升两层，成为（子）树根。

zig-zag/zag-zig

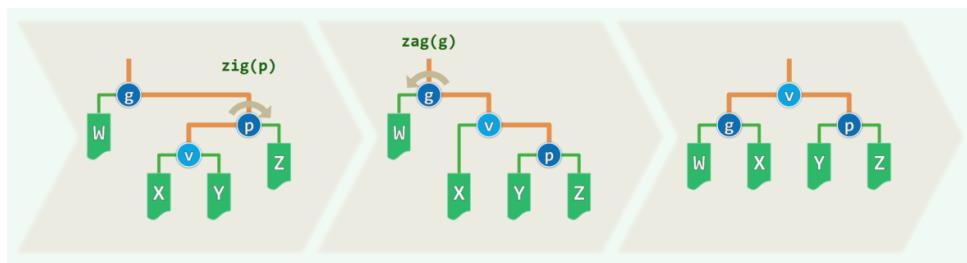


图 103 双层伸展——情形 1

对于 v 是 p 的左孩子， p 是 g 的右孩子的情况，先对 p 进行一次旋转，再对 v 进行一次旋转。

这样的效果事实上和逐层旋转是一样的。

zig-zig/zag-zag

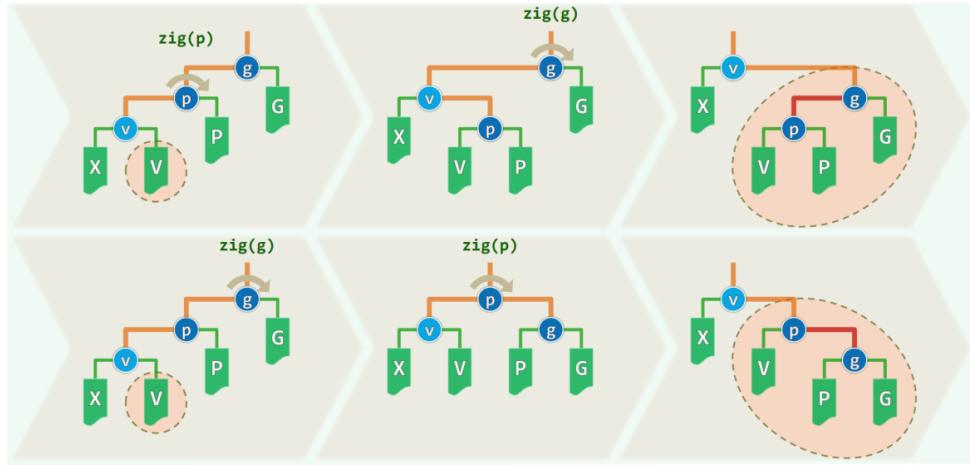


图 104 双层伸展——情形 2

但是如果是 v 是 p 的左孩子， p 是 g 的左孩子的情况，两种旋转方式就有区别。

连续两次旋转根节点（上图下面的旋转方法），可以使得 v 上升两层，成为（子）树根。这种情况下，节点访问之后，对应路径的长度随即折半。最坏情况不致持续发生。
伸展操作分摊下来，仍然是 $O(\log n)$ 的。

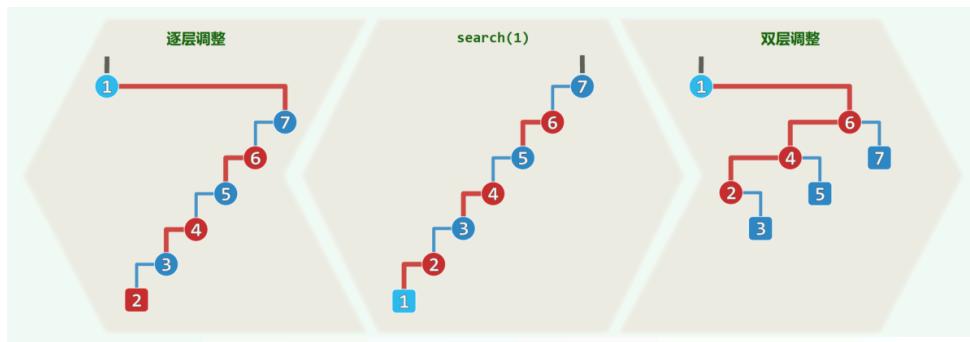


图 105 双层伸展——情形 2

zig/zag

如果 v 只有父亲，没有祖父，此时必有 $v.parent() == T.root()$ ，只做一次旋转即可。只会出现在最后一次。

8.1.4 算法实现

接口

```
template <typename T> class Splay : public BST<T> { //由 BST 派生
protected:
    BinNodePosi<T> splay( BinNodePosi<T> v ); //将 v 伸展至根
public: //伸展树的查找也会引起整树的结构调整，故 search() 也需重写
    BinNodePosi<T> & search( const T & e ); //查找（重写）
```

```

BinNodePosi<T> insert( const T & e ); //插入（重写）
bool remove( const T & e ); //删除（重写）
};

```

伸展算法

```

template <typename T> BinNodePosi<T> Splay<T>::splay( BinNodePosi<T> v ) {
    if ( ! v ) return NULL; BinNodePosi<T> p; BinNodePosi<T> g; //父亲、祖父
    while ( (p = v->parent) && (g = p->parent) ) {
        /* 自下而上， 反复地双层伸展 */
    }
    if ( p = v->parent ) { /* 若 p 果真是根， 只需再额外单旋一次 */ }
    v->parent = NULL; return v; //伸展完成， v 抵达树根
}

```

填充上面的空白，得到伸展算法的实现。

```

while ( (p = v->parent) && (g = p->parent) ) { //自下而上，反复双层伸展
    BinNodePosi<T> gg = g->parent; //每轮之后， v 都将以原曾祖父为父
    if ( IsLChild( * v ) )
        if ( IsLChild( * p ) ) { /* zig-zig */ } else { /* zig-zag */ }
    else
        if ( IsRChild( * p ) ) { /* zag-zag */ } else { /* zag-zig */ }
    if ( !gg ) v->parent = NULL; //无曾祖父 gg 的 v 即为树根；否则， gg 此后应以 v 为
    else ( g == gg->lC ) ? attachAsLC(v, gg) : attachAsRC(gg, v); //左或右孩子
    updateHeight( g ); updateHeight( p ); updateHeight( v );
}

```

对于 zig-zig 的情况，有：

```

if ( IsLChild( * v ) )
    if ( IsLChild( * p ) ) { //zIg-zIg
        attachAsLC( p->rC, g ); //Y
        attachAsLC( v->rC, p ); //X
        attachAsRC( p, g );
        attachAsRC( v, p );
    } else { /* zIg-zAg */ }
else
    if ( IsRChild( * p ) ) { /* zAg-zAg */ } else { /* zAg-zIg */ }

```

剩下情况类似，不再赘述。

查找算法。伸展树的查找，与常规 BST::search() 不同：很可能改变树的拓扑结构，不再属于静态操作：

```

template <typename T> BinNodePosi<T> & Splay<T>::search( const T & e ) {
// 调用标准 BST 的内部接口定位目标节点
    BinNodePosi<T> p = BST<T>::search( e );
}

```

```
// 无论成功与否，最后被访问的节点都将伸展至根
    _root = splay( p ? p : _hot ); //成功、失败
// 总是返回根节点
    return _root;
}
```

插入算法。Splay::search()已集成 splay()，查找失败之后，_hot 即是根，随即就在树根附近接入新节点。

```
template <typename T> BinNodePosi<T> Splay<T>::insert( const T & e ) {
    if ( !_root ) { _size = 1; return _root = new BinNode<T>( e ); } //原树为空
    BinNodePosi<T> t = search( e ); if ( e == t->data ) return t; //t若存在，伸展至根
    if ( t->data < e ) { //在右侧嫁接 (rc 或为空, lc == t 必非空)
        t->parent = _root = new BinNode<T>( e, NULL, t, t->rc );
        if ( t->rc ) { t->rc->parent = _root; t->rc = NULL; }
    } else { //e < t->data, 在左侧嫁接 (lc 或为空, rc == t 必非空)
        t->parent = _root = new BinNode<T>( e, NULL, t->lc, t );
        if ( t->lc ) { t->lc->parent = _root; t->lc = NULL; }
    }
    _size++; updateHeightAbove( t ); return _root; //更新规模及 t 与 _root 的高度,
    插入成功
} //无论如何，返回时总有 _root->data == e
```



图 106 伸展树——插入

删除算法。Splay::search()成功之后，目标节点即是树根，在树根附近完成目标节点的摘除。

```
template <typename T> bool Splay<T>::remove( const T & e ) {
    if ( !_root || ( e != search( e )->data ) ) return false; //若目标存在，则伸展至根
    BinNodePosi<T> L = _root->lc, R = _root->rc; release(_root); //记下子树后,
    释放之
```

```

if ( !R ) { //若 R 空
    if ( L ) L->parent = NULL; _root = L; //则 L 即是余树
} else { //否则
    _root = R; R->parent = NULL; search( e ); //在 R 中再找 e: 注定失败，但最
小节点必
    if ( L ) L->parent = _root; _root->lc = L; //伸展至根，故可令其以 L 作为左
子树
}
_size--; if ( _root ) updateHeight( _root ); //更新记录
return true; //删除成功
}

```



图 107 伸展树——删除

综合评价:

- 无需记录高度或平衡因子；编程实现简单——优于 AVL 树
- 分摊复杂度 $O(n \log n)$ ——与 AVL 树相当
- 局部性强、缓存命中率极高时（即 $k \ll n \ll m$ ）时候（ k 是被访问的节点数， m 是被访问次数），性能优于 AVL 树
 - 效率甚至可以更高——自适应的 $O(\log k)$
 - 任何连续的 m 次查找，仅需 $O(m \log k + n \log n)$ 时间
- 若反复地顺序访问任一子集，分摊成本仅为常数
- 不能杜绝单次最坏情况，不适用于对效率敏感的场合

8.1.5 分摊分析

利用势能的方法，对伸展树的分摊复杂度进行分析。

对于伸展树，势能函数定义为：

$$\Phi(S) = \log \left(\prod_{v \in S} \text{size}(v) \right) = \sum_{v \in S} \log(\text{size}(v)) = \sum_{v \in S} \text{rank}(v) = \sum_{v \in S} \log V$$

越平衡/倾侧的树，势能越小/大。单链是 $O(n \log n)$ ，满树是 $O(n)$ 。

考查对伸展树 S 的 $m \gg n$ 次连续访问（不妨仅考查 `search()`），记

$$A^k = T^k + \Delta\Phi^k$$

则有

$$A - O(n \log n) \leq T = A - \Delta\Phi \leq A + O(n \log n)$$

下面证明

$$A = O(m \log n)$$

则有

$$T = O(n \log n)$$

而 A^k 都不致超过节点 v 的势能变化量，即： $O(\text{rank}^k(v) - \text{rank}^{k-1}(v)) = O(\log n)$ 。

A^k 是 v 的若干次连续伸展操作（时间成本）的累积，这些操作无非三种情况。

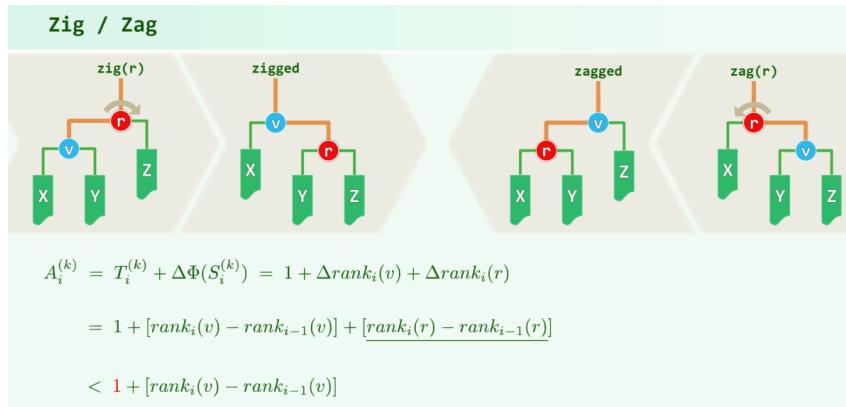


图 108 伸展树——分摊分析

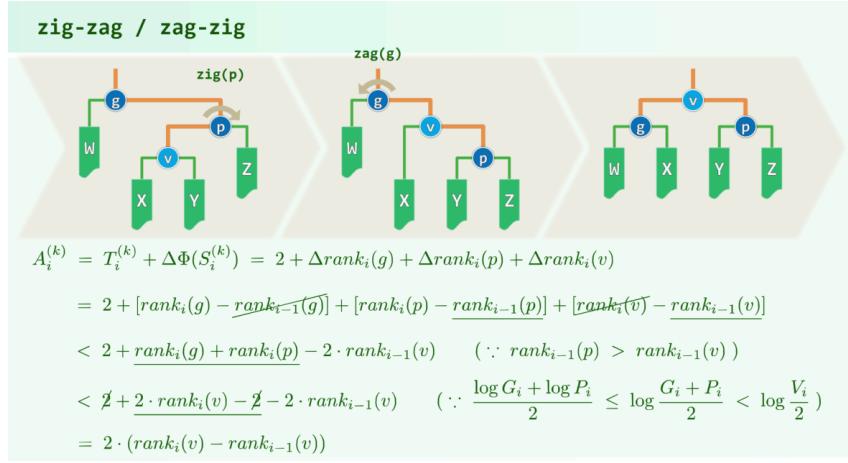


图 109 伸展树——分摊分析

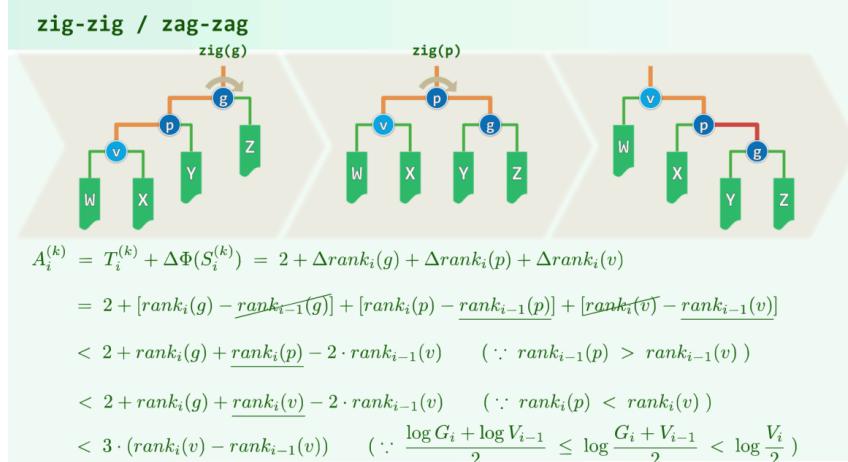


图 110 伸展树——分摊分析

8.2 B 树

8.2.1 缓存 Cache

先考虑这样一个问题：就地循环位移

仅用 $O(1)$ 辅助空间，将数组 $A[0, n]$ 中的元素向左循环移动 k 个单元 `void shift(int *A, int n, int k);`

蛮力解法：每次移动一个单元，共移动 k 次，时间复杂度 $O(kn)$ 。

```
void shift0( int * A, int n, int k ) //反复以 1 为间距循环左移
{ while ( k-- ) shift( A, n, 0, 1 ); } //共迭代 k 次， O(n*k)
```

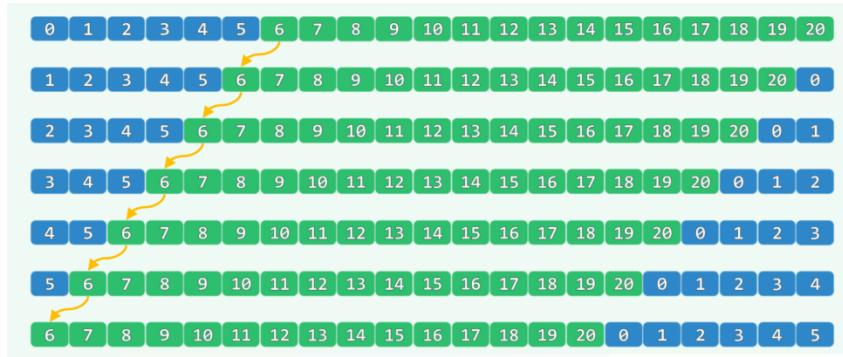


图 111 就地循环位移——蛮力解法

迭代版 Stride-k Reference Pattern : 分成 k 组，每组内部循环左移，共移动 n 次，时间复杂度 $O(n)$ 。

```

int shift( int * A, int n, int s, int k ) { // O( n / GCD(n, k) )
    int b = A[s]; int i = s, j = (s + k) % n; int mov = 0; //mov 记录移动次数
    while ( s != j ) //从 A[s]出发，以 k 为间隔，依次左移 k 位
        { A[i] = A[j]; i = j; j = (j + k) % n; mov++; }
    A[i] = b; return mov + 1; //最后，起始元素转入对应位置
} // [0, n) 由关于 k 的 g = GCD(n, k) 个同余类组成，shift(s, k) 能够且只能够使其中之一就位

void shift1(int* A, int n, int k) { //经多轮迭代，实现数组循环左移 k 位，累计 O(n+g)
    for (int s = 0, mov = 0; mov < n; s++) //O(g) = O(GCD(n, k))
        mov += shift(A, n, s, k);
}

```

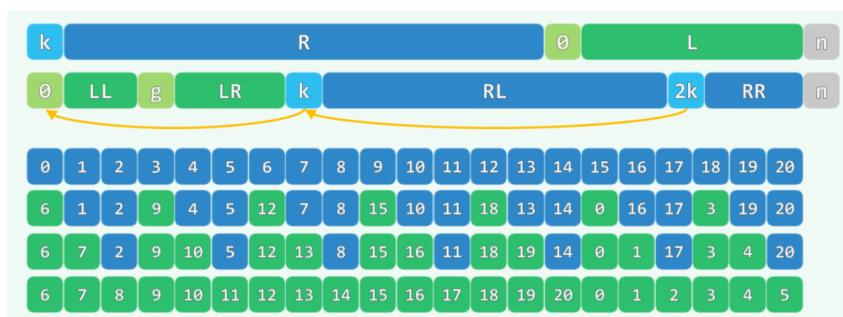


图 112 就地循环位移——Stride-k Reference Pattern

倒置版 Stride-1 Reference Pattern: 如下图，经过三次倒置即可。复杂度是 $O(3n)$ 。

```

void shift2( int * A, int n, int k ) {
    reverse( A, k ); //O(3k/2)
    reverse( A + k, n - k ); //O(3(n-k)/2)
}

```

```
    reverse( A, n ); //O(3n/2)
} //O(3n)
```

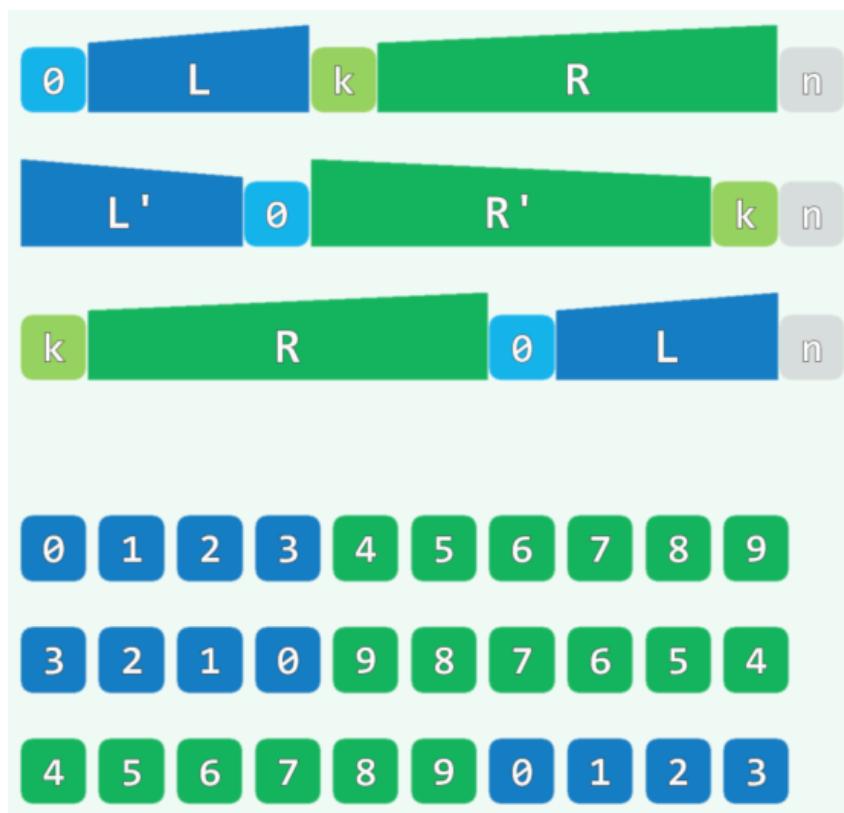


图 113 就地循环位移——Stride-1 Reference Pattern

可以看到这种方法虽然看上去常系数很大，但是实际上是最快的。这是因为利用了缓存。

- 实用的存储系统，由不同类型的存储器级联而成，以综合其各自的优势



图 114 存储系统

- 分级存储：利用数据访问的局部性

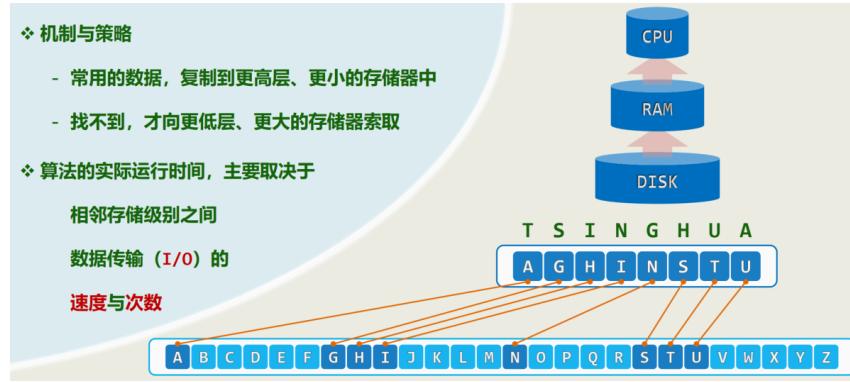


图 115 分级存储

- 这就导致：在外存读写 1B，与读写 1KB 几乎一样快
 - 以页（page）为单位，借助缓冲区批量访问，可大大缩短单位字节的平均访问时间

8.2.2 B 树的结构

出于缓存的考虑，B 树每 d 代合并为超级节点

- $m = 2^d$ 路
- $m - 1$ 个关键码

逻辑上与 BBST 完全等价。

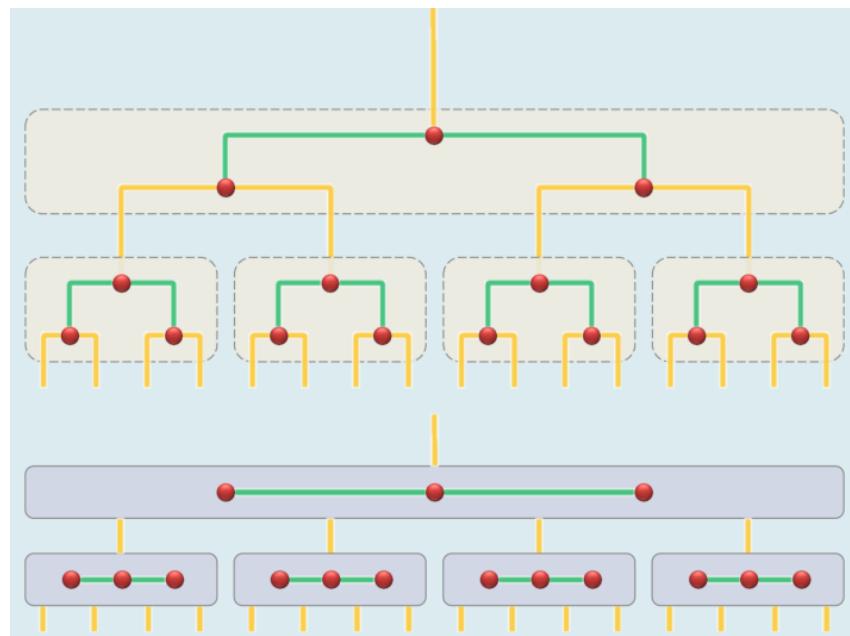


图 116 B 树

I/O 优化：多级存储系统中使用 B-树，可针对外部查找，大大减少 I/O 次数。

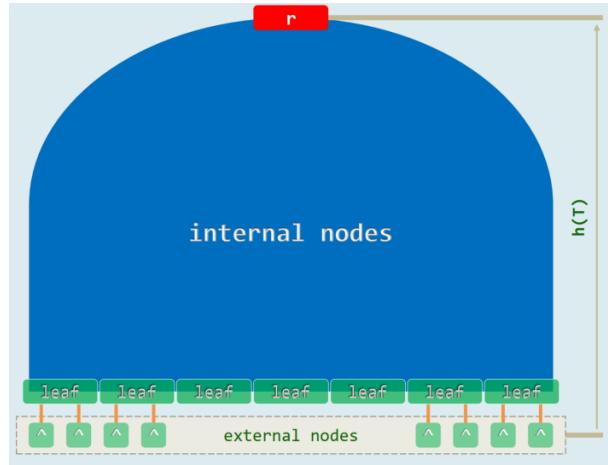


图 117 B 树

所谓 m 阶 B-树，即 m 路完全平衡搜索树 ($m \geq 3$)

- 外部节点的深度统一相等，约定以此深度作为树高 h
- 叶节点的深度统一相等 $h - 1$
- 内部节点
 - 各含 $n \leq m - 1$ 个关键码: $K_1 < K_2 < \dots < K_n$
 - 各有 $n + 1 \leq m$ 个分支: A_0, A_1, \dots, A_n
 - 反过来，分支数也不能太少
 - 树根: $2 \leq n + 1$
 - 其余: $\lceil \frac{m}{2} \rceil \leq n + 1$
 - 故也称作 $(\lceil \frac{m}{2} \rceil, m)$ -树

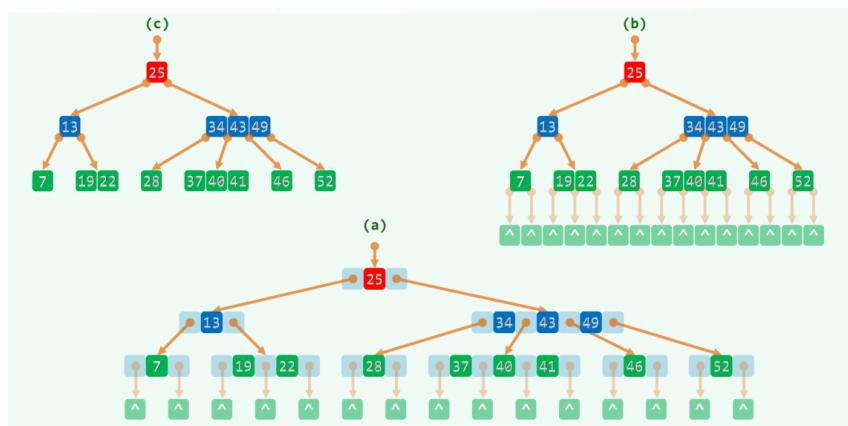


图 118 B 树——紧凑表示

BTNode: 用两个长度差 1 的向量存储关键码和孩子

```
template <typename T> struct BTNode { //B-树节点
    BTNodePosi<T> parent; //父
    Vector<T> key; //关键码(总比孩子少一个)
    Vector<BTNodePosi<T>> child; //孩子
    BTNode() { parent = NULL; child.insert( NULL ); }
    BTNode( T e, BTNodePosi<T> lc = NULL, BTNodePosi<T> rc = NULL ) {
        parent = NULL; //作为根节点
        key.insert( e ); //仅一个关键码, 以及
        child.insert( lc ); if ( lc ) lc->parent = this; //左孩子
        child.insert( rc ); if ( rc ) rc->parent = this; //右孩子
    }
};
```

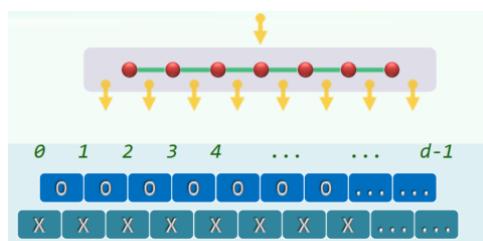


图 119 B 树——节点

BTree 模板类

```
template <typename T> using BTNodePosi = BTNode<T>*; //B-树节点位置
template <typename T> class BTree { //B-树
protected:
    Rank _size, _m; //关键码总数、阶次
    BTNodePosi<T> _root, _hot; //根、search()最后访问的非空节点
    void solveOverflow( BTNodePosi<T> ); //因插入而上溢后的分裂处理
    void solveUnderflow( BTNodePosi<T> ); //因删除而下溢后的合并处理
public:
    BTNodePosi<T> search( const T & e ); //查找
    bool insert( const T & e ); //插入
    bool remove( const T & e ); //删除
};
```

8.2.3 B 树的查找

和 BST 一样, B 树的查找也是从根节点开始, 逐层向下, 直到外部节点。

```
从(常驻 RAM 的)根节点开始
只要当前节点不是外部节点
在当前节点中顺序查找 //RAM 内部
若找到目标关键码, 则
    return 查找成功
```

```

否则 //止于某一向下的引用
沿引用找到孩子节点
将其读入内存 //I/O 耗时
return 查找失败

```

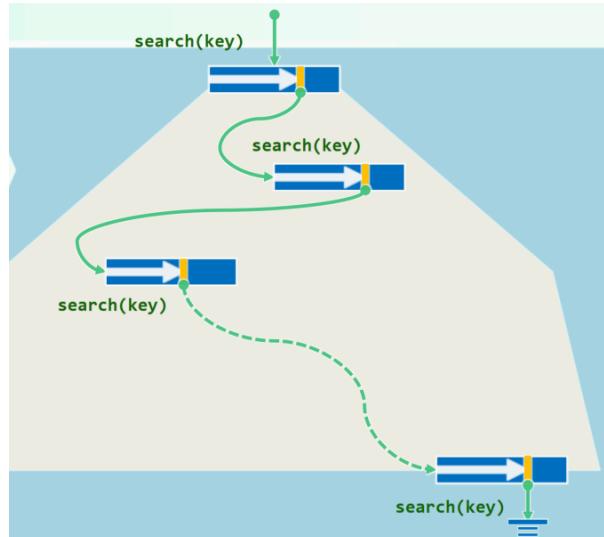


图 120 B 树——查找

```

template <typename T> BTNodePosi<T> BTree<T>::search( const T & e ) {
    BTNodePosi<T> v = _root; _hot = NULL; //从根节点出发
    while ( v ) { //逐层深入地
        Rank r = v->key.search( e ); //在当前节点对应的向量中顺序查找
        if ( 0 <= r && e == v->key[r] ) return v; //若成功，则返回；否则...
        _hot = v; v = v->child[ r + 1 ]; //沿引用转至对应的下层子树，并载入其根
        (I/O)
    } //若因!v 而退出，则意味着抵达外部节点
    return NULL; //失败
}

```

性能：忽略内存中的查找，运行时间主要取决于 I/O 次数，在每一深度至多一次 I/O，故 $O(h)$ 。

可以证明， $\log_m(N+1) \leq h \leq 1 + \left\lfloor \log_{\lceil \frac{m}{2} \rceil} \left(\frac{N+1}{2} \right) \right\rfloor$ ，其中 N 是关键码总数， h 是树高。

$h = O(\log_m N)$ 。

8.2.4 B 树的插入

插入算法的核心是 `solveOverflow()`，它的作用是：将上溢的节点分裂为两个节点，分别作为两个儿子，选出中位数推送到原来的父亲。

```

template <typename T> bool BTree<T>::insert( const T & e ) {
    BTNodePosi<T> v = search( e );
    if ( v ) return false; //确认 e 不存在

```

```

Rank r = _hot->key.search( e ); //在节点_hot 中确定插入位置
_hot->key.insert( r+1, e ); //将新关键码插至对应的位置
_hot->child.insert( r+2, NULL ); _size++; //创建一个空子树指针
solveOverflow( _hot ); //若上溢, 则分裂
return true; //插入成功
}

```

设上溢节点中的关键码依次为:

$$\{k_0, k_1, \dots, k_{m-1}\}$$

取中位数 $s = \lfloor \frac{m}{2} \rfloor$, 则有划分:

$$\{k_0, k_1, \dots, k_{s-1}\} \{k_s\} \{k_{s+1}, \dots, k_{m-1}\}$$

关键码 k_s 上升一层。

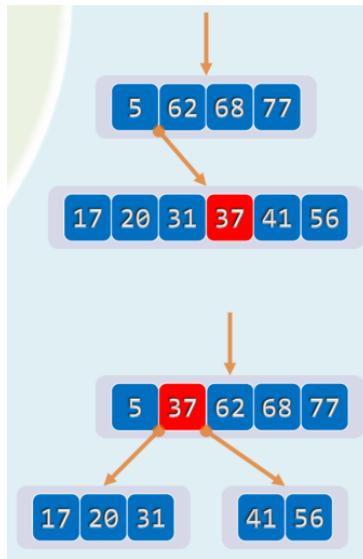


图 121 B 树——上溢

若上溢节点的父亲本已饱和, 则在接纳被提升的关键码之后, 也将上溢: 套用前法, 继续分裂。

上溢可能持续发生, 并逐层向上传播, 直至根节点。次数是 $O(h)$ 的。

此时, 如果根节点也上溢, 需创建新的根节点, 作为 B 树的新根。注意: 新生的树根仅有两个分支。

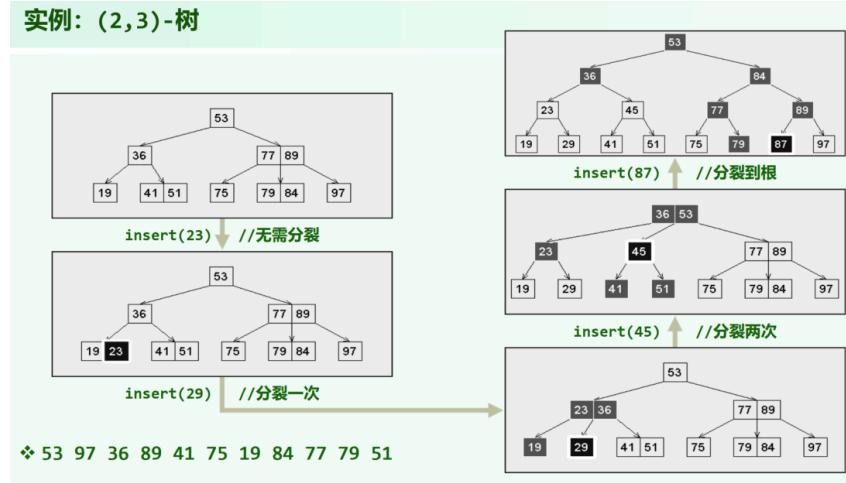


图 122 B 树——上溢——实例

```

template <typename T> void BTree<T>::solveOverflow( BTNodePosi<T> v ) {
    while ( _m <= v->key.size() ) { //除非当前节点不再上溢
        Rank s = _m / 2; //轴点 (此时_m = key.size() = child.size() - 1)
        BTNodePosi<T> u = new BTNode<T>(); //注意: 新节点已有一个空孩子
        for ( Rank j = 0; j < _m - s - 1; j++ ) { //分裂出右侧节点 u (效率低可改
            子
            u->child.insert( j, v->child.remove( s + 1 ) ); //v 右侧_m-s-1 个孩
            u->key.insert( j, v->key.remove( s + 1 ) ); //v 右侧_m-s-1 个关键码
        }
        u->child[ _m - s - 1 ] = v->child.remove( s + 1 ); //移动 v 最靠右的孩子
        if ( u->child[ 0 ] ) //若 u 的孩子们非空, 则统一令其以 u 为父节点
            for ( Rank j = 0; j < _m - s; j++ ) u->child[ j ]->parent = u;
        BTNodePosi<T> p = v->parent; //v 当前的父节点 p
        if ( ! p ) //若 p 为空, 则创建之 (全树长高一层, 新根节点恰好两度)
            { _root = p = new BTNode<T>(); p->child[ 0 ] = v; v->parent = p; }
        Rank r = 1 + p->key.search( v->key[ 0 ] ); //p 中指向 u 的指针的秩
        p->key.insert( r, v->key.remove( s ) ); //轴点关键码上升
        p->child.insert( r + 1, u ); u->parent = p; //新节点 u 与父节点 p 互联
        v = p; //上升一层, 如有必要则继续分裂—至多 O(logn) 层
    } //while
} //solveOverflow

```

8.2.5 B 树的删除

和 BST 一样, B 树的删除也是先寻找, 如果节点在叶子上, 直接删除; 如果节点在内部节点上, 找到其后继, 将后继的关键码替换到当前节点, 然后删除后继。

```

template <typename T>
bool BTree<T>::remove( const T & e ) {
    BTNodePosi<T> v = search( e );

```

```

if ( ! v ) return false; //确认 e 存在
Rank r = v->key.search(e); //e 在 v 中的秩
if ( v->child[0] ) { //若 v 非叶子, 则
    BTNodePosi<T> u = v->child[r + 1]; //在右子树中
    while ( u->child[0] ) u = u->child[0]; //一直向左, 即可找到 e 的后继 (必在
底层)
    v->key[r] = u->key[0]; v = u; r = 0; //交换
}
//assert: 至此, v 必位于最底层, 且其中第 r 个关键码就是待删除者
v->key.remove( r ); v->child.remove( r + 1 ); _size--;
solveUnderflow( v ); return true; //如有必要, 需做旋转或合并
}

```

处理下溢:

非根节点 v 下溢时, 必恰有 $\lceil \frac{m}{2} \rceil - 2$ 个关键码 $\lceil \frac{m}{2} \rceil - 1$ 个分支。视其左、右兄弟 L、R 的规模, 可分三种情况加以处理:

1. 若 L 存在, 且至少包含 $\lceil \frac{m}{2} \rceil$ 个关键码:
 - 将 P 中的分界关键码 y 移至 v 中 (作为最小关键码)
 - 将 L 中的最大关键码 x 移至 P 中 (取代原关键码 y)
 - 儿子也要转走

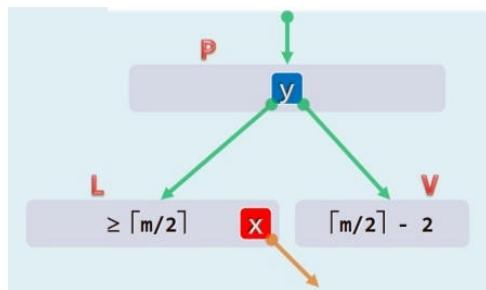


图 123 B 树——下溢——情形 1

如此旋转之后, 局部乃至全树都重新满足 B-树条件, 下溢修复完毕

2. 若 R 存在, 且至少包含 $\lceil \frac{m}{2} \rceil$ 个关键码
 - 也可旋转, 完全对称

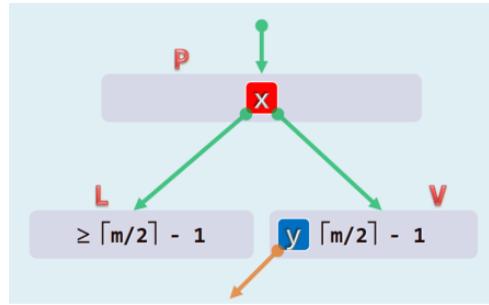


图 124 B 树——下溢——情形 2

3. L 和 R 或不存在，或均不足 $\lceil \frac{m}{2} \rceil$ 个关键码：

- L 和 R 仍必有其一（不妨以 L 为例），且
- 恰含 $\lceil \frac{m}{2} \rceil - 1$ 个关键码

从 P 中抽出介于 L 和 V 之间的分界关键码 y

- 通过 y 做粘接，将 L 和 V 合成一个节点
- 同时合并此前 y 的孩子引用

此处下溢得以修复，但可能继而导致 P 下溢，继续旋转或合并

- 下溢可能持续发生并向上传播；但至多不过 $O(h)$ 层

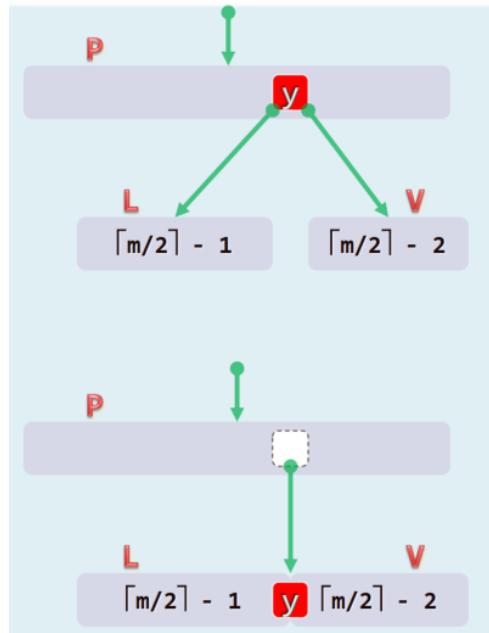


图 125 B 树——下溢——情形 3

下溢修复：

```

template <typename T> void BTTree<T>::solveUnderflow( BTNodePosi<T> v ) {
    while ( (_m + 1) / 2 > v->child.size() ) { //除非当前节点没有下溢
        BTNodePosi<T> p = v->parent; if ( !p ) { /* 已到根节点 */ }
        Rank r = 0; while ( p->child[r] != v ) r++; //确定 v 是 p 的第 r 个孩子
        if ( 0 < r ) { /* 情况 #1: 若 v 的左兄弟存在, 且... */ }
        if ( p->child.size() - 1 > r ) { /* 情况 #2: 若 v 的右兄弟存在, 且... */ }
        if ( 0 < r ) { /* 与左兄弟合并 */ } else { /* 与右兄弟合并 */ } //情况 #3
        v = p; //上升一层, 如有必要则继续旋转或合并—至多 O(logn) 层
    } //while
} //solveUnderflow

```

情况 1: 旋转 (向左兄弟借关键码)

```

if (0 < r) { //若 v 不是 p 的第一个孩子, 则
    BTNodePosi<T> ls = p->child[r - 1]; //左兄弟必存在
    if ( (_m + 1) / 2 < ls->child.size() ) { //若该兄弟足够“胖”, 则
        v->key.insert( 0, p->key[r-1] ); //p 借出一个关键码给 v (作为最小关键码)
        p->key[r - 1] = ls->key.remove( ls->key.size() - 1 ); //ls 的最大 key 转
入 p
        v->child.insert( 0, ls->child.remove( ls->child.size() - 1 ) ); //同时
ls 的最右侧孩子过继给 v (作为 v 的最左侧孩子)
        if ( v->child[0] ) v->child[0]->parent = v;
        return; //至此, 通过右旋已完成当前层 (以及所有层) 的下溢处理
    }
} //情况 #2 完全对称

```

情况 3: 合并

```

if (0 < r) { //与左兄弟合并
    BTNodePosi<T> ls = p->child[r-1]; //左兄弟必存在
    ls->key.insert( ls->key.size(), p->key.remove(r - 1) );
    p->child.remove( r ); //p 的第 r - 1 个关键码转入 ls, v 不再是 p 的第 r 个孩子
    ls->child.insert( ls->child.size(), v->child.remove( 0 ) );
    if ( ls->child[ ls->child.size() - 1 ] ) //v 的最左侧孩子过继给 ls 做最右侧孩子
        ls->child[ ls->child.size() - 1 ]->parent = ls;
    while ( !v->key.empty() ) { //v 剩余的关键码和孩子, 依次转入 ls
        ls->key.insert( ls->key.size(), v->key.remove( 0 ) );
        ls->child.insert( ls->child.size(), v->child.remove( 0 ) );
        if ( ls->child[ ls->child.size() - 1 ] )
            ls->child[ ls->child.size() - 1 ]->parent = ls;
    } //while
    release(v); //释放 v
} else
{ /* 与右兄弟合并, 完全对称 */ }

```

下面的图就给了下溢处理的例子:

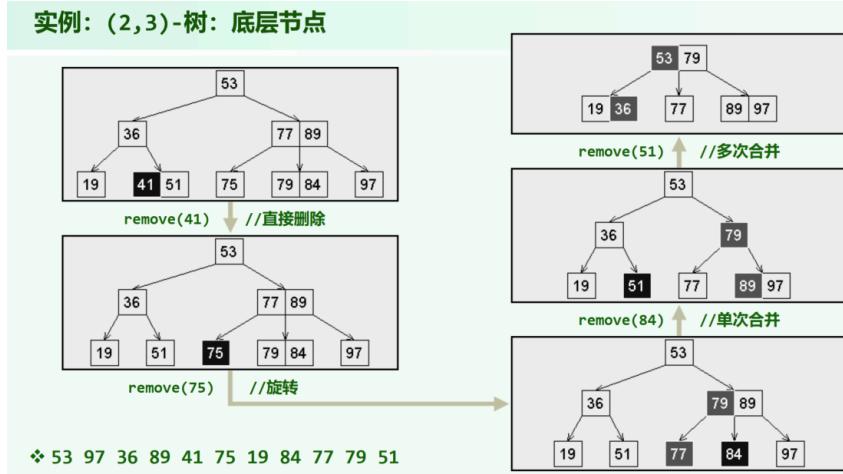


图 126 B 树——下溢——实例

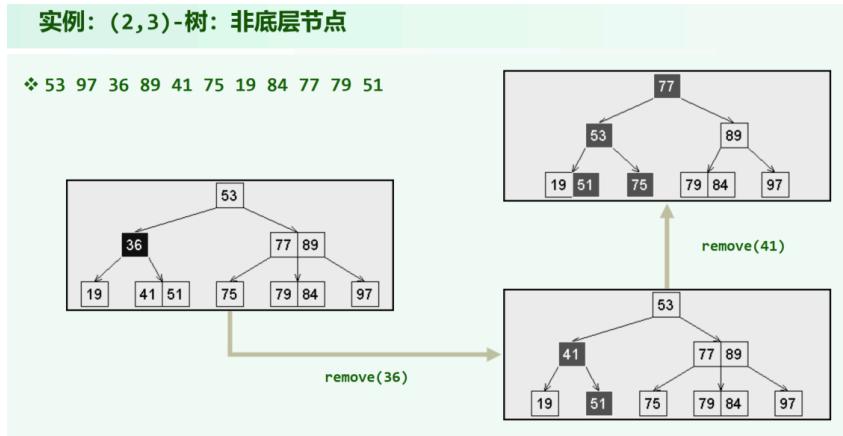


图 127 B 树——下溢——实例

8.3 红黑树 Red-Black Tree

8.3.1 动机

并发性: Concurrent Access To A Database

修改之前先加锁 (lock); 完成后解锁 (unlock), 访问延迟主要取决于“lock/unlock”周期

对于 BST 而言, 每次修改过程中, 唯结构有变 (reconstruction) 处才需加锁, 访问延迟主要取决于这类局部之数量...

- Splay: 结构变化剧烈, 最差可达 $O(n)$
- AVL: `remove()` 时 $O(\log n)$, `insert()` 时可保证 $O(1)$
- Red-Black: 无论 `insert/remove`, 均不超过 $O(1)$

持久性: Persistent structures: 支持对历史版本的访问

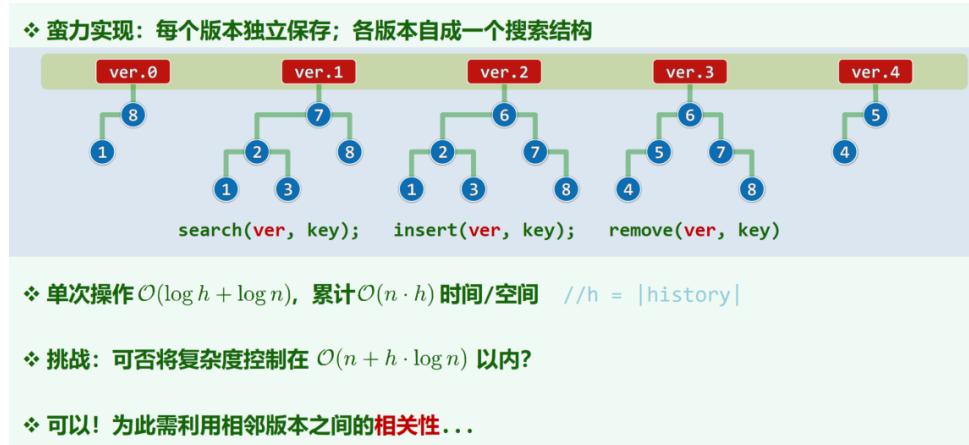


图 128 持久性

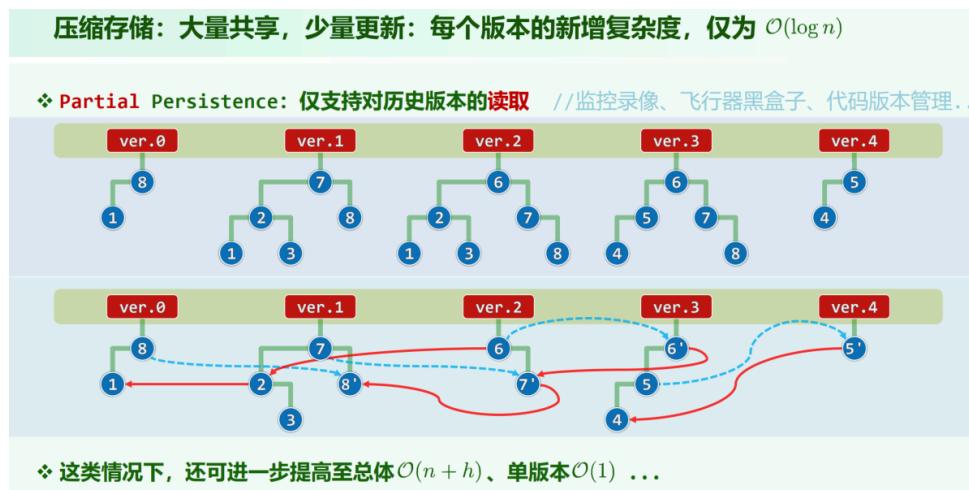


图 129 持久性

对于这样的版本控制，我们希望就树形结构的拓扑而言，相邻版本之间的差异不能超过 $O(1)$ ，而 Red-Black 树可以做到。

8.3.2 红黑树的结构

由红、黑两类节点组成的 BST，统一增设外部节点 NULL，使之成为真二叉树。

规则：

1. 树根：必为黑色
2. 外部节点：均为黑色
3. 红节点：只能有黑孩子（及黑父亲）
4. 外部节点：黑深度（黑的真祖先数目）相等
 - 亦即根（全树）的黑高度
 - 子树的黑高度，即后代 NULL 的相对黑深度

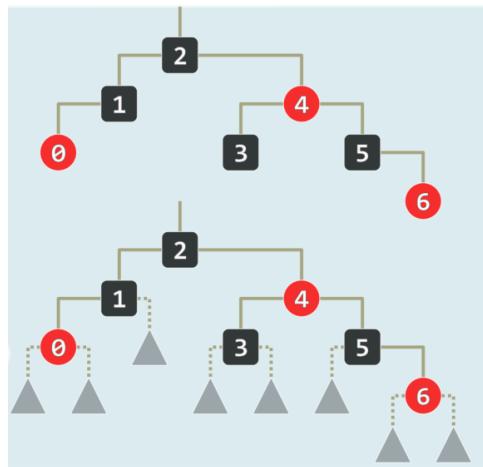


图 130 红黑树

将红节点提升至与其（黑）父亲等高，红边折叠起来——可以得到一棵等价的(2,4)树。

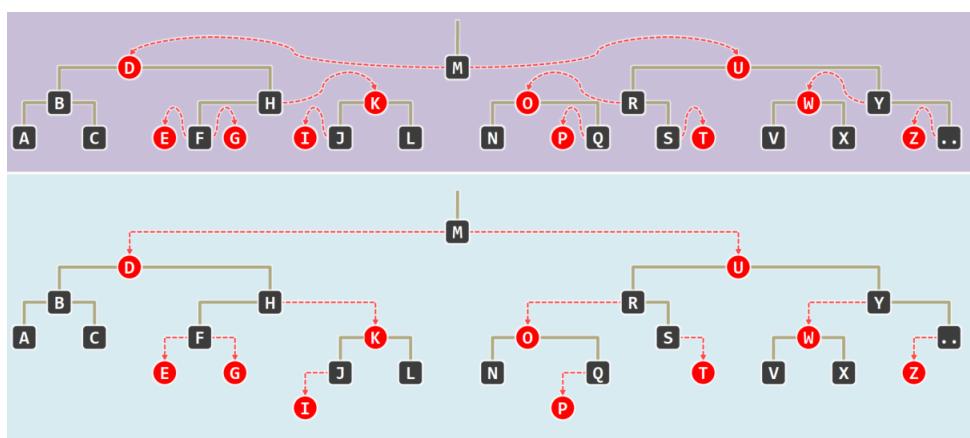


图 131 红黑树——等价的(2,4)树

将黑节点与其红孩子视作关键码，再合并为 B-树的超级节点。四种组合，分别对应于 4 阶 B-树的一类内部节点。

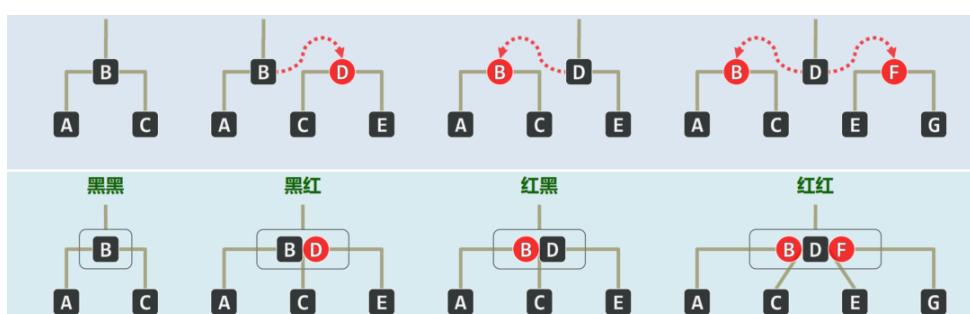


图 132 红黑树——等价的 B 树

包含 n 个内部节点的红黑树 T , 高度 $h = O(\log n)$:

$$\log_2(n+1) \leq h \leq 2\log_2(n+1)$$

若 T 高度为 h , 红/黑高度为 R/H , 则

$$H \leq h \leq H + R \leq 2H$$

若 T 所对应的 B-树为 T_B , 则 H 即是 T_B 的高度。 T_B 的每个节点, 都恰好包含 T 的一个黑节点。

$$H \leq \log_2\left(\frac{n+1}{2}\right) + 1 = \log_2(n+1)$$

从而红黑树是一棵 BBST。

RedBlack 模板类

```
template <typename T> class RedBlack : public BST<T> { //红黑树
public: //BST::search()等其余接口可直接沿用
    BinNodePosi<T> insert( const T & e ); //插入（重写）
    bool remove( const T & e ); //删除（重写）
protected:
    void solveDoubleRed( BinNodePosi<T> x ); //双红修正
    void solveDoubleBlack( BinNodePosi<T> x ); //双黑修正
    Rank updateHeight( BinNodePosi<T> x ); //更新节点 x 的高度（重写）
};

#define stature( p ) ( ( p ) ? ( p )->height : 0 ) //外部节点黑高度为 0, 以上递推
template <typename T> int RedBlack<T>::updateHeight( BinNodePosi<T> x )
    { return x->height = IsBlack( x ) + max( stature( x->lC ), stature( x-
>rC ) ); }
```

8.3.3 红黑树的插入

按 BST 规则插入关键码 e , $x = \text{insert}(e)$ 必为叶节点。

- 除非是首个节点 (根), x 的父亲 $p = x->\text{parent}$ 必存在
- 首先将 x 染红: $x->\text{color} = \text{isRoot}(x) ? \text{B} : \text{R}$
- 至此, 条件 1、2、4 依然满足; 但 3 不见得, 有可能出现双红/**double-red**: $p->\text{color} == x->\text{color} == \text{R}$
- 考查:
 - 祖父 $g = p->\text{parent}$ 必为黑
 - 叔父 $u = \text{uncle}(x) = \text{ sibling}(p)$

视 u 的颜色, 分两种情况处理:

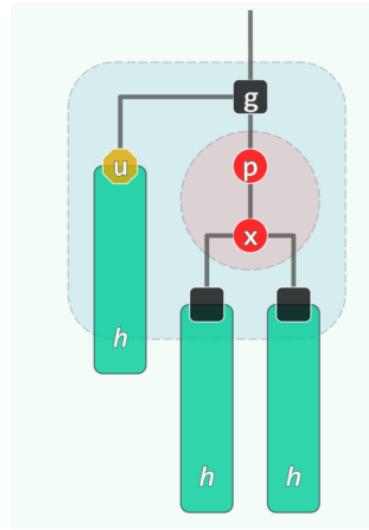


图 133 红黑树——插入

```
template <typename T> BinNodePosi<T> RedBlack<T>::insert( const T & e ) {
    // 确认目标节点不存在 (留意对 _hot 的设置)
    BinNodePosi<T> & x = search( e ); if ( x ) return x;
    // 创建红节点 x, 以 _hot 为父, 黑高度 = 0
    x = new BinNode<T>( e, _hot, NULL, NULL, 0 ); _size++;
    // 如有必要, 需做双红修正, 再返回插入的节点
    BinNodePosi<T> xOld = x; solveDoubleRed( x ); return xOld;
} //无论原树中是否存有 e, 返回时总有 x->data == e
```

双红修正

```
template <typename T> void RedBlack<T>::solveDoubleRed( BinNodePosi<T> x ) {
    if ( IsRoot( *x ) ) { //若已 (递归) 转至树根, 则将其转黑, 整树黑高度也随之递增
        { _root->color = RB_BLACK; _root->height++; return; } //否则...
        BinNodePosi<T> p = x->parent; //考查 x 的父亲 p (必存在)
        if ( IsBlack( p ) ) return; //若 p 为黑, 则可终止调整; 否则
        BinNodePosi<T> g = p->parent; //x 祖父 g 必存在, 且必黑
        BinNodePosi<T> u = uncle( x ); //以下视叔父 u 的颜色分别处理
        if ( IsBlack( u ) ) { /* ... u 为黑 (或 NULL) ... */ }
        else { /* ... u 为红 ... */ }
    }
}
```

8.3.3.1 RR-1: u->color == B——一次 3+4 重构+两点反转红黑【一蹴而就】

此时, x、p、g 的四个孩子 (可能是外部节点)

- 全为黑, 且
- 黑高度相同

按照 B 树理解如下图

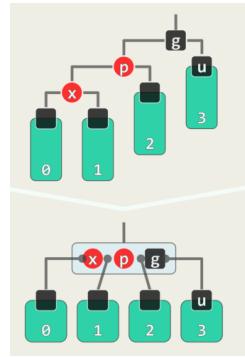


图 134 红黑树——插入——RR-1

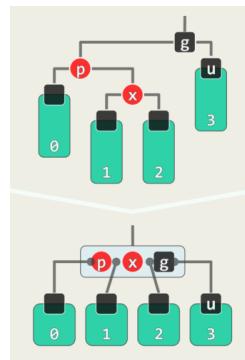


图 135 红黑树——插入——RR-1

局部“3+4”重构： b 转黑， a 或 c 转红。在某三叉节点中插入红关键码后，原黑关键码不再居中（RRB 或 BRR）。调整的效果，是将三个关键码的颜色改为 RBR。

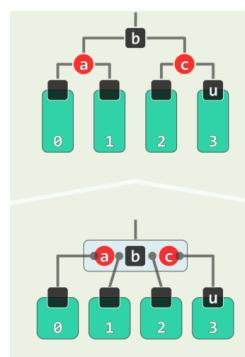


图 136 红黑树——插入——RR-1

如此调整，一蹴而就。

```

template <typename T> void RedBlack<T>::solveDoubleRed( BinNodePosi<T> x ) {
/* ..... */
if ( IsBlack( u ) ) { //u 为黑或 NULL
    // 若 x 与 p 同侧, 则 p 由红转黑, x 保持红; 否则, x 由红转黑, p 保持红
    if ( IsLChild( *x ) == IsLChild( *p ) ) p->color = RB_BLACK;
    else x->color = RB_BLACK;
    g->color = RB_RED; //g 必定由黑转红
    BinNodePosi<T> gg = g->parent; //great-grand parent
    BinNodePosi<T> r = FromParentTo( *g ) = rotateAt( x );
    r->parent = gg; //调整之后的新子树, 需与原曾祖父联接
} else { /* ... u 为红 ... */ }
}

```

8.3.3.2 RR-2: u->color == R--上溢解决（无需旋转）：叔父染黑+祖父染红【递归上溯】

在 B-树中，等效于超级节点发生上溢。

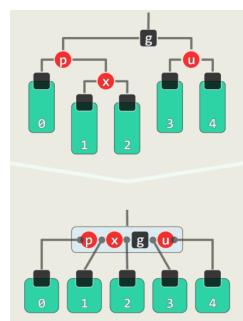


图 137 红黑树——插入——RR-2

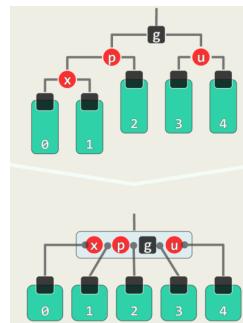


图 138 红黑树——插入——RR-2

p 与 u 转黑, g 转红: 在 B-树中, 等效于节点分裂, 关键码 g 上升一层。

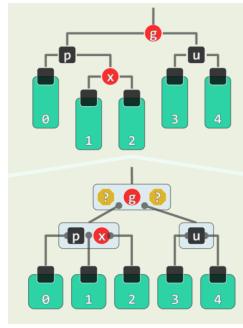


图 139 红黑树——插入——RR-2

可能继续向上传递——亦即， g 与 $\text{parent}(g)$ 再次构成双红。等效地将 g 视作新插入的节点，区分以上两种情况，如法处置。

g 若果真到达树根，则强行将其转为黑色（整树黑高度加一）。

```
template <typename T> void RedBlack<T>::solveDoubleRed( BinNodePosi<T> x ) {
    /* .... */
    if ( IsBlack( u ) ) { /* ... u 为黑（含 NULL） ... */
        else { //u 为红色
            p->color = RB_BLACK; p->height++; //p 由红转黑，增高
            u->color = RB_BLACK; u->height++; //u 由红转黑，增高
            g->color = RB_RED; //在 B-树中 g 相当于上交给父节点的关键码，故暂标记为红
            solveDoubleRed( g ); //继续调整：若已至树根，接下来的递归会将 g 转黑（尾递
        归）
    }
}
```

`RedBlack::insert()` 仅需 $O(\log n)$ 时间，至多 $O(\log n)$ 次染色和 $O(1)$ 次旋转。

	旋转	染色	此后
u 为黑	1 or 2	2	调整完成
u 为红	0	3	递归上溯

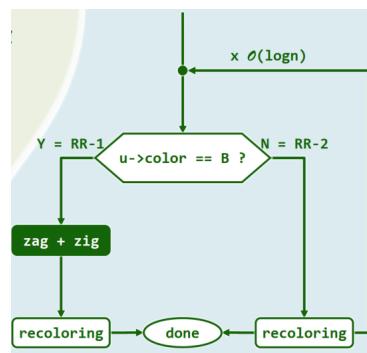


图 140 红黑树——插入——总结

8.3.4 红黑树的删除

首先按照 BST 常规算法，执行 `r = removeAt(x, _hot)`，实际被摘除的可能是 `x` 的前驱或后继 `w`，简捷起见，以下不妨统称作 `x`。

`x` 由孩子 `r` 接替，此时另一孩子 `k` 必为 `NULL`

- 但在随后的调整过程中 `x` 可能逐层上升
- 故需假想地、统一地、等效地理解为：
 - `k` 为一棵黑高度与 `r` 相等的子树，且
 - 随 `x` 一并摘除（尽管实际上从未存在过）

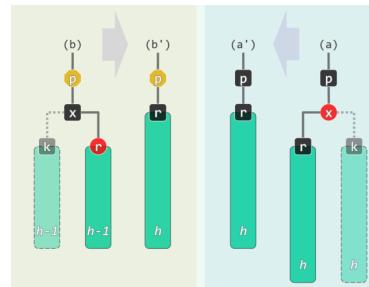


图 141 红黑树——删除

```
template <typename T> bool RedBlack<T>::remove( const T & e ) {
    BinNodePosi<T> & x = search( e ); if ( !x ) return false; // 查找定位
    BinNodePosi<T> r = removeAt( x, _hot ); // 删除 _hot 的某孩子，r 指向其接替者
    if ( ! ( --_size ) ) return true; // 若删除后为空树，可直接返回
    if ( ! _hot ) { // 若被删除的是根，则
        _root->color = RB_BLACK; // 将其置黑，并
        updateHeight( _root ); // 更新（全树）黑高度
        return true;
    } // 至此，原 x（现 r）必非根
    // 若父亲（及祖先）依然平衡，则无需调整
    if ( BlackHeightUpdated( * _hot ) ) return true;
    // 至此，必失衡
    // 若替代节点 r 为红，则只需简单地翻转其颜色
    if ( IsRed( r ) ) { r->color = RB_BLACK; r->height++; return true; }
    // 至此，r 以及被其替代的 x 均为黑色
    solveDoubleBlack( r ); // 双黑调整（入口处必有 r == NULL）
    return true;
}
```

完成 `removeAt()` 之后

- 条件 1、2 依然满足
- 但条件 3、4 却不见得

其一为红：

在原树中，考查 x 与 r

- 若 x 为红，则条件 3、4 自然满足
- 若 r 为红，则令其与 x 交换颜色，即可满足条件 3、4
一蹴而就。

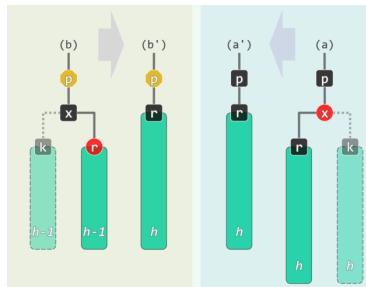


图 142 红黑树——删除——RB

双黑：

若 x 与 r 均黑 (double black)，则不然。

- 摘除 x 并代之以 r 后，全树黑深度不再统一（稍后可见，等效于 B-树中 x 所属节点下溢）
- 在新树中，考查 r 的父亲、兄弟
 - $p = r->parent$ //亦是原 x 的父亲
 - $s = sibling(r)$

以下分四种情况处理：

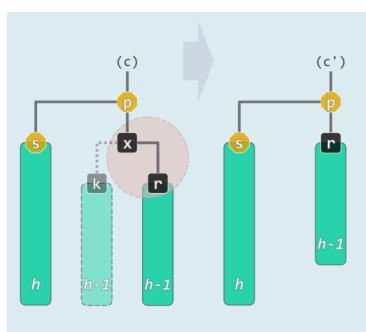


图 143 红黑树——删除——双黑

```
template <typename T> void RedBlack<T>::solveDoubleBlack( BinNodePosi<T> r ) {
    BinNodePosi<T> p = r ? r->parent : _root; if ( !p ) return; //r 的父亲
    BinNodePosi<T> s = (r == p->lc) ? p->rc : p->lc; //r 的兄弟
    if ( IsBlack( s ) ) { //兄弟 s 为黑
        BinNodePosi<T> t = NULL; //s 的红孩子 (若左、右孩子皆红，左者优先；皆黑时为
        NULL)
    }
}
```

```

if ( IsRed ( s->rc ) ) t = s->rc;
if ( IsRed ( s->lcc ) ) t = s->lcc;
if ( t ) { /* ... 黑 s 有红孩子: BB-1 ... */ }
else { /* ... 黑 s 无红孩子: BB-2R 或 BB-2B ... */ }
} else { /* ... 兄弟 s 为红: BB-3 ... */ }
}

```

8.3.4.1 BB-1: s 为黑，且(侄子是红的)至少有一个红孩子 t——下溢解决：一次“3+4”重构+三次染色【一蹴而就】

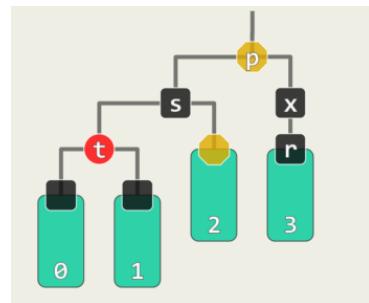


图 144 红黑树——删除——双黑——BB-1

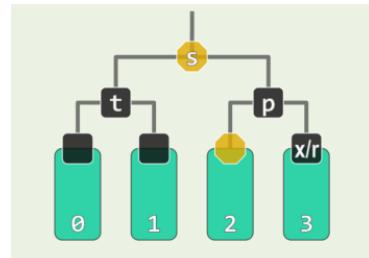


图 145 红黑树——删除——双黑——BB-1

“3+4”重构：

- t ~ a
- s ~ b
- p ~ c

重新着色：

- r 保持黑
- a、c 染黑
- b 继承 p 的原色

如此，红黑树性质在全局得以恢复。

如果按照 B 树理解：通过关键码的旋转，消除超级节点的下溢。在对应的 B-树中

- p 若为红，问号之一为黑关键码
- p 若为黑，必自成一个超级节点

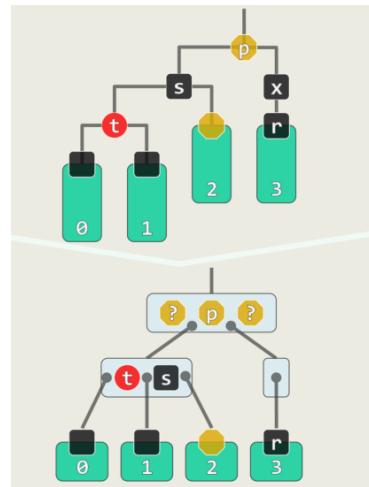


图 146 红黑树——删除——双黑——BB-1

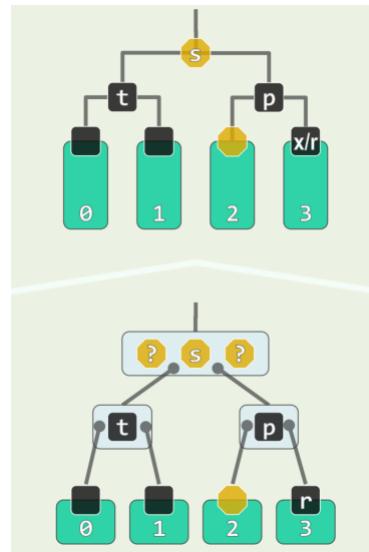


图 147 红黑树——删除——双黑——BB-1

```

if ( IsBlack( s ) ) { //兄弟 s 为黑
/* .... */
if ( t ) { //黑 s 有红孩子： BB-1
    RBColor oldColor = p->color; //备份 p 颜色，并对 t、父亲、祖父
    BinNodePosi<T> b = FromParentTo( *p ) = rotateAt( t ); //旋转
    if (HasLChild( *b )) { b->l->color = RB_BLACK; updateHeight( b->l ); }
    if (HasRChild( *b )) { b->r->color = RB_BLACK; updateHeight( b->r ); }
    b->color = oldColor; updateHeight( b ); //新根继承原根的颜色
}

```

```
} else { /* ... 黑 s 无红孩子: BB-2R 或 BB-2B ... */ }
} else { /* ... 兄弟 s 为红: BB-3 ... */ }
```

8.3.4.2 BB-2R: s 为黑, 且两个孩子均为黑; p 为红——下溢解决: 两个染色【一蹴而就】

- r 保持黑; s 转红; p 转黑
- 在对应的 B-树中, 等效于下溢节点与兄弟合并
- 红黑树性质在全局得以恢复——一蹴而就

失去关键码 p 后, 上层节点不会继而下溢。因为合并之前, 在 p 之左或右侧还应有一个黑关键码。

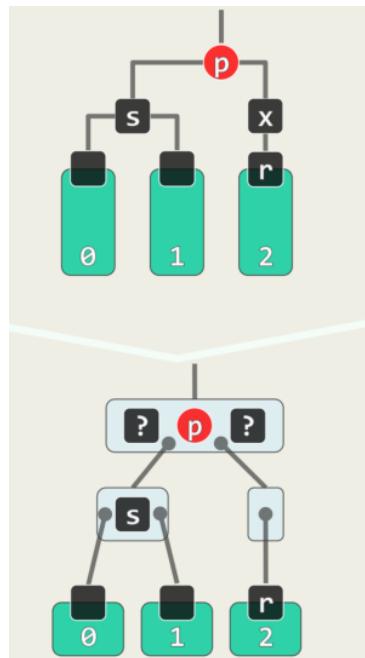


图 148 红黑树——删除——双黑——BB-2R

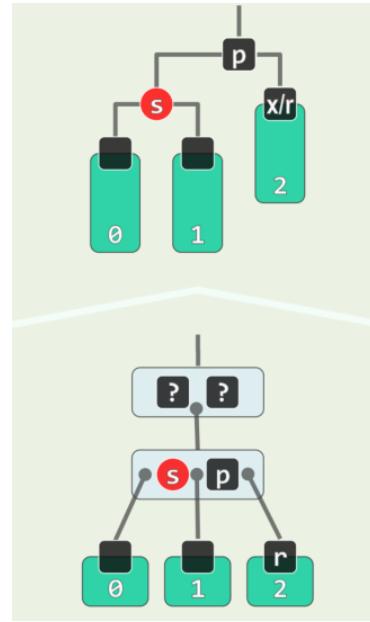


图 149 红黑树——删除——双黑——BB-2R

8.3.4.3 BB-2B: s 为黑, 且两个孩子均为黑; p 为黑——下溢解决: 一次染色【递归上溯】

- s 转红; r 与 p 保持黑
- 红黑树性质在局部得以恢复
- 在对应的 B-树中, 等效于下溢节点与兄弟合并
- 合并前, p 和 s 均属于单关键码节点

孩子的下溢修复后, 父节点继而下溢, 递归上溯 $O(\log n)$ 层

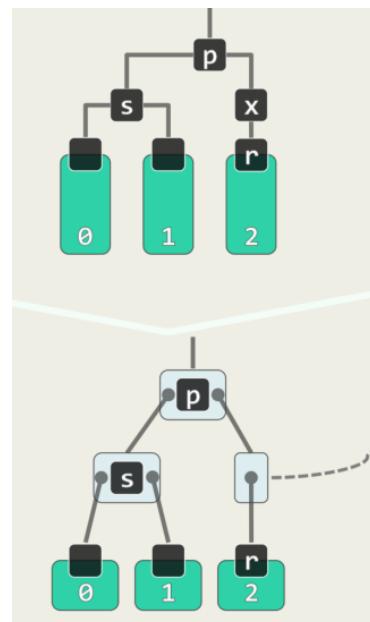


图 150 红黑树——删除——双黑——BB-2B

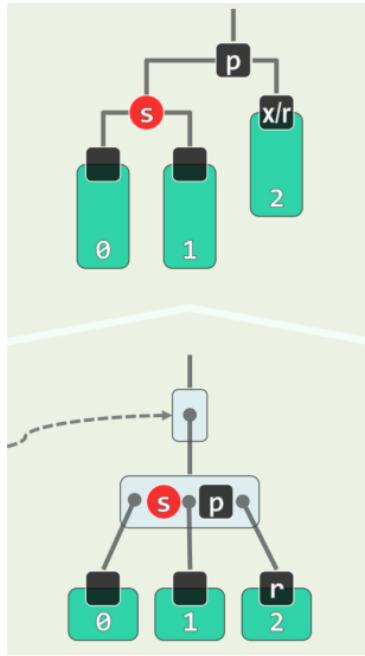


图 151 红黑树——删除——双黑——BB-2B

```

if ( IsBlack( s ) ) { //兄弟 s 为黑
    /* ..... */
    if ( t ) { /* ... 黑 s 有红孩子: BB-1 ... */ }
    else { /* 黑 s 无红孩子 */
        s->color = RB_RED; s->height--; //s 转红
        if ( IsRed( p ) ) //BB-2R: p 转黑, 但黑高度不变
            { p->color = RB_BLACK; }
        else //BB-2B: p 保持黑, 但黑高度下降; 递归修正
            { p->height--; solveDoubleBlack( p ); }
    }
} else { /* ... 兄弟 s 为红: BB-3 ... */ }
    
```

8.3.4.4 BB-3: s 为红（其孩子均为黑）——一次旋转+两次染色【化归成一蹴而就】

- 绕 p 单旋； s 红转黑， p 黑转红
- 黑高度依然异常，但 r 有了一个新的黑兄弟 s'
- 故转化为前述情况，而且 p 已转红，接下来
 - 绝不会是 BB-2B
 - 而只能是 BB-2R 或 BB-1
- 于是，再经一轮调整红黑树性质必然全局恢复。

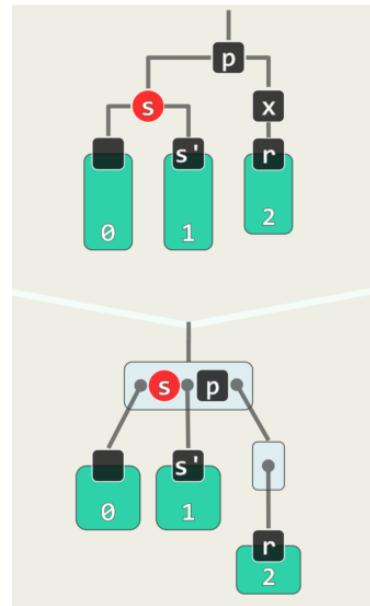


图 152 红黑树——删除——双黑——BB-3

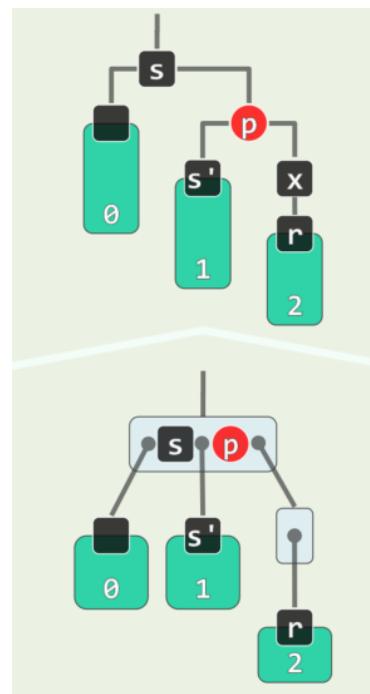


图 153 红黑树——删除——双黑——BB-3

```
if ( IsBlack( s ) ) { //兄弟 s 为黑
    if ( t ) { /* ... 黑 s 有红孩子: BB-1 ... */ }
    else { /* ... 黑 s 无红孩子: BB-2R 或 BB-2B ... */ }
} else { //兄弟 s 为红: BB-3
    s->color = RB_BLACK; p->color = RB_RED; //s 转黑, p 转红}
```

```

BinNodePosi<T> t = IsLChild( *s ) ? s->lc : s->rc; //取 t 与其父 s 同侧
_hot = p; FromParentTo( *p ) = rotateAt( t ); //对 t 及其父亲、祖父做平衡调整
solveDoubleBlack( r ); //继续修正 r—此时 p 已转红，故后续只能是 BB-1 或 BB-2R
}

```

RedBlack::remove()仅需 $O(\log n)$ 时间，至多 $O(\log n)$ 次染色和 $O(1)$ 次旋转。

	旋转	染色	此后
BB-1:黑 s 有红子 t	1or2	3	调整完成
BB-2R:黑 s 无红子， p 红	0	2	调整完成
BB-2B:黑 s 无红子， p 黑	0	1	递归上溯
BB-3:红 s	1	2	转为(1)或(2R), 调整完成

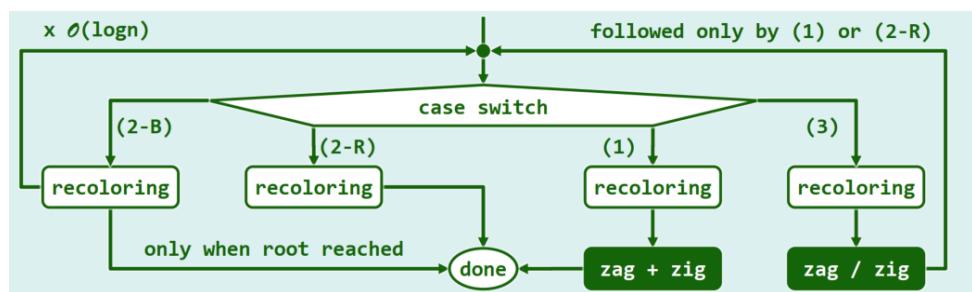


图 154 红黑树——删除——双黑——总结

九 词典 Dictionary

我们希望通过寻对象访问键值对，这就是字典的想法。

```
template <typename K, typename V> //key、 value
struct Dictionary {
    virtual Rank size() = 0;
    virtual bool put( K, V ) = 0;
    virtual V* get( K ) = 0;
    virtual bool remove( K ) = 0;
};
```

字典定义了接口（如获取、设置、删除键值对），而下面的散列和跳表提供了这些接口的具体实现。

哈希表通常提供更快的查找、插入和删除操作，但它们不支持有序键的高效操作。跳表提供有序存储和范围查询的能力，但在某些操作上可能不如哈希表高效。

9.1 散列 Hashtable

对于在很大范围 R 中的词条，但是往往只会用到其中的一小部分 N ，如果利用数组就会造成空间的浪费，因此可以利用散列。通过 Hash 函数，将关键字映射到表中一个位置来访问记录，以加快查找速度。

利用桶（规模为 M ）直接存放或间接指向一个词条。利用散列函数 $\text{hash}(k)$ 将词条关键码 k 转换为桶号，然后将词条存入桶中。如果两个词条的关键码相同，就会发生冲突，可以通过开放定址法、链地址法、再散列法等解决冲突。

这样一来，查找、插入、删除的期望复杂度是 $O(1)$ 。

9.1.1 冲突

冲突是指两个不同的关键码被映射到同一个桶中。当装填因子 $\lambda = \frac{N}{M}$ 越大，冲突越多，查找效率越低。

如果数据集固定且已知，可以实现完美散列，关键码不存在冲突。一般而言，要选取合适的散列函数，使得冲突尽可能少。

9.2 散列函数 Hash Function

散列函数的设计要求：

- 确定：对于同一个关键码，散列函数应该总是返回同一个值
- 快速：计算散列函数的时间应该尽可能短 $O(1)$
- 满射：充分利用散列空间
- 均匀：散列函数应该尽可能地将关键码均匀地散列到各个桶中，避免聚集现象

9.2.1 除余法

$$\text{hash}(k) = k \% M$$

非理想随机的时候， M 选取素数更好。

缺陷：

- 不论表长 M 怎么取，都有 $\text{hash}(0) = 0$
- 相邻关键码散列地址也相邻

9.2.2 MAD 法：Multiply-Add-Divide

$$\text{hash}(k) = (ak + b)\%M$$

9.2.3 其他散列函数

- 数字分析：选取关键码中的一部分作为散列地址
- 平方取中法：取关键码平方后的中间几位作为散列地址
- 折叠法：将关键码分割成几部分，然后将这几部分叠加起来作为散列地址
- 位异或法：将关键码的各个部分进行异或运算，得到的值作为散列地址

9.2.4 随机数

伪随机数算法： $\text{rand}(x + 1) = [\text{a} * \text{rand}(x)] \% M // M$ 素数， $\text{a} \% M != 0$

```
unsigned long int next = 1; //sizeof(long int) = 8
void srand(unsigned int seed) { next = seed; } //sizeof(int) = 4 or 8
int rand(void) { //1103515245 = 3^5 * 5 * 7 * 129749
    next = next * 1103515245 + 12345;
    return (unsigned int)(next/65536) % 32768;
}
```

从而 $\text{hash}(\text{key}) = \text{rand}(\text{key}) = (\text{rand}(0)\text{a}^{\text{key}})\%M$ 这样生成的结果是理想随机的。

9.2.5 hashCode 与多项式法

比较好的一种由字符串生成散列地址的方法是多项式法，即将字符串看作是一个多项式，然后将多项式的值作为散列地址。

```
static Rank hashCode( char s[] ) {
    Rank n = strlen(s); Rank h = 0;
    for ( Rank i = 0; i < n; i++ ) {
        h = (h << 5) | (h >> 27);
        h += s[i];
    } //乘以 32，加上扰动，累计贡献
    return h;
}
```

9.3 冲突解决：开放散列

开放散列是指当发生冲突时，不是将词条放入桶中，而是将词条放入其他的桶中。

9.3.1 多槽位

将每个桶扩展为一个槽位数组，每个槽位存放一个词条。当发生冲突时，依次检查槽位，直到找到一个空槽位，然后将词条放入其中。如果槽位数组满了，就需要扩容。

缺点是空间浪费，并且仍有情况需要一直扩容，不可控。

9.3.2 公共溢出区 Overflow Area

开辟一块连续空间，发生冲突的词条都放入其中。查找时，先在桶中查找，如果没有，再在溢出区查找。同一个关键码给出的词条用指针串起来。

缺点是溢出区查找效率低。

9.3.3 独立链 Linked-List Chaining / Separate Chaining

将每个桶扩展为一个链表，每个槽位存放一个链表的头结点。当发生冲突时，将词条插入到链表中。查找时，先在桶中查找，如果没有，再在链表中查找。

缺点是空间未必连续分布、系统缓存很难生效，并且节点的动态分配和释放会带来额外的开销。

9.4 冲突解决：封闭散列

只要有必要，任何散列桶都可以接纳任何词条；为每个词条，都需事先约定若干备用桶，优先级逐次下降。沿试探链（Probe Sequence/Chain），逐个转向下一桶单元，直到命中成功，或者抵达一个空桶而失败。

9.4.1 线性试探 Linear Probing

一旦冲突，则试探后一紧邻的桶，直到命中（成功），或抵达空桶（失败）。

新增非同义词之间的冲突，数据堆积（clustering）现象严重。但试探链连续，数据局部性良好，通过装填因子，冲突与堆积都可有效控制。

插入：新词条若尚不存在，则存入试探终止处的空桶。

懒惰删除：若词条存在，则将其标记为“已删除”，而非立即删除，以免破坏试探链。（空宅与故居）

```
template <typename K, typename V> int Hashtable<K, V>::probe4Hit(const K& k) {
    int r = hashCode(k) % M; //按除余法确定试探链起点
    while ( ( ht[r] && (k != ht[r]->key) ) || removed->test(r) )
        r = (r + 1) % M; //线性试探（跳过带懒惰删除标记的桶）
    return r; //调用者根据 ht[r]是否为空及其内容，即可判断查找是否成功
}
template <typename K, typename V> int Hashtable<K, V>::probe4Free(const K& k)
{
    int r = hashCode(k) % M; //按除余法确定试探链起点
    while ( ht[r] ) r = (r + 1) % M; //线性试探，直到空桶（无论是否带有懒惰删除标记）
    return r; //只要有空桶，线性试探迟早能找到
}
```

查找时，若遇到“已删除”标记，则继续试探下一桶。插入时，若遇到“已删除”标记，则将词条存入此处，而非继续试探下一桶。

重散列：装填因子过大时，重散列操作将桶数组容量倍增，同时将已有词条逐一移动到新桶中，以期平摊试探成本。

填装因子的计算： $\lambda = \frac{N}{M}$ ，其中 N 为词条总数， M 为桶数组容量。是一定要把懒惰删除词条减掉，才能统计真正有多少桶被占用了。

```
template <typename K, typename V> //随着装填因子增大，冲突概率、排解难度都将激增
void Hashtable<K, V>::rehash() { //此时，不如“集体搬迁”至一个更大的散列表
    int oldM = M; Entry<K, V>** oldHt = ht;
    ht = new Entry<K, V>*[ M = primeNLT( 4 * N ) ]; N = 0; //新表“扩”容量
    memset( ht, 0, sizeof( Entry<K, V>* ) * M ); //初始化各桶
    release( removed ); removed = new Bitmap(M); L = 0; //懒惰删除标记
    for ( int i = 0; i < oldM; i++ ) //扫描原表
        if ( oldHt[i] ) //将每个非空桶中的词条
            put( oldHt[i]->key, oldHt[i]->value ); //转入新表
    release( oldHt ); //释放—因所有词条均已转移，故只需释放桶数组本身
}
```

插入

```
template <typename K, typename V> bool Hashtable<K, V>::put( K k, V v ) {
    if ( ht[ probe4Hit( k ) ] ) return false; //雷同元素不必重复插入
    int r = probe4Free( k ); //为新词条找个空桶（只要装填因子控制得当，必然成功）
    ht[ r ] = new Entry<K, V>( k, v ); ++N; //插入
    if ( removed->test( r ) ) { removed->clear( r ); --L; } //懒惰删除标记
    if ( (N + L)*2 > M ) rehash(); //若装填因子高于 50%，重散列
    return true;
}
```

删除

```
template <typename K, typename V> bool Hashtable<K, V>::remove( K k ) {
    int r = probe4Hit( k ); if ( !ht[r] ) return false; //确认目标词条确实存在
    release( ht[r] ); ht[r] = NULL; --N; //清除目标词条
    removed->set(r); ++L; //更新标记、计数器
    if ( 3*N < L ) rehash(); //若懒惰删除标记过多，重散列
    return true;
}
```

9.4.2 双向平方试探 Quadratic Probing

试探链改成平方试探，即每次试探的步长为 $1^2, 2^2, 3^2, c\dots$ ，而不是 $1, 2, 3, c\dots$ 。这样可以避免线性试探中的堆积现象。但这样试探链不会遍历完全剩余系， M 是素数的时候，只会有 $\lceil \frac{M}{2} \rceil$ 被取到。

为了可以遍历完全剩余系，取 M 是模 4 余 3 的素数，这样它的正负平方可以遍历完全剩余系。而模 4 余 1 的素数，它的正负平方只能遍历到一半。

9.5 桶排序 Bucket sort

对于 $(0, m]$ 的整数，可以借助散列表排序。

将他们存入散列表，然后依次遍历散列表，将非空桶中的词条按照关键码顺序输出。

也可以存在同义词，同义词再进行排序即可（每个桶表示一个区间，每次将元素插入对应的桶中）。同义词可以用链表储存。可以保存存入的次序，这对后面的基数排序有作用。这被称为稳定性。

9.5.1 例：MaxGap

任意 n 个互异点均将实轴分为 $n-1$ 段有界区间，其中的哪一段最长？

线性算法：

1. 找到最左点、最右点 $O(n)$
2. 将有效范围均匀地划分为 $n - 1$ 段（ n 个桶） $O(n)$
3. 通过散列，将各点归入对应的桶 $O(n)$
4. 在各桶中，动态记录最左点、最右点 $O(n)$
5. 算出相邻（非空）桶之间的“距离” $O(n)$
6. 最大的距离即 MaxGap

9.6 基数排序 Radix Sort

词典排序 (lexicographic order)：自 k_1 到 k_t (低位优先)，依次以各域为序做一趟桶排序。

时间成本是 $O(t(n + M))$ ，其中 M 是基数， t 是关键码的个数。

```
typedef unsigned int U; //约定：类型 T 或就是 U；或可转换为 U，并依此定序

template <typename T> void List<T>::radixSort( ListNodePosi<T> p, int n ) {
    ListNodePosi<T> head = p->pred; ListNodePosi<T> tail = p;
    for ( int i = 0; i < n; i++ ) tail = tail->succ; //待排序区间为(head, tail)
    for ( U radixBit = 0x1; radixBit && (p = head); radixBit <<= 1 ) //以下反复
        地
        for ( int i = 0; i < n; i++ ) //根据当前基数位，将所有节点
            radixBit & U (p->succ->data) ? //分拣为前缀 (0) 与后缀 (1)
                insert( remove( p->succ ), tail ) : p = p->succ;
} //为避免 remove()、insert() 的低效率，可拓展 List::move(p,tail) 接口，将节点 p 直
接移至 tail 之前
```

9.7 计数排序 Counting Sort

基数排序中反复做的桶排序，亦属“小集合 + 大数据”类型，是否可以更快。计数排序可以优化基数排序中的桶排序。

仍以纸牌排序为例 ($n \gg m = 4$)。假设已按点数排序，以下（稳定地）按花色排序。

1. 经过分桶，统计出各种花色的数量。

2. 自前向后扫描各桶，依次累加即可确定各套花色所处的秩区间。
3. 自后向前扫描每一张牌，对应桶的计数减一，即是其在最终有序序列中对应的秩。

该方法的时间复杂度是 $O(n + m)$ ，但是需要额外的空间。

9.8 跳转表 SkipList

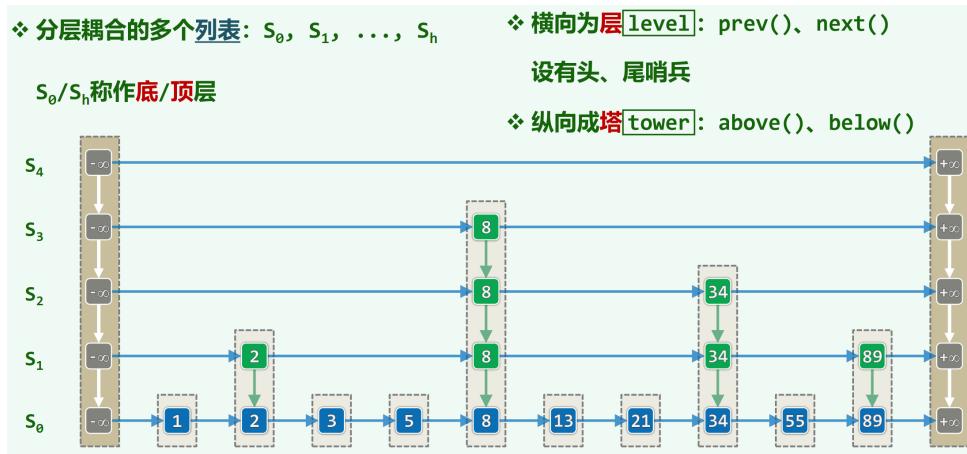


图 155 跳转表

跳转表是一种特殊的链表，它的每个节点都有一个指向下一个节点的指针，也有一个指向下一层的指针。这样一来，可以在每一层中进行二分查找，然后跳转到下一层，再进行二分查找，直到最后一层。

跳转表的查找时间复杂度是 $O(\log n)$ ，插入和删除的时间复杂度是 $O(\log n)$ 。

先定义四联节点，包括前驱、后继、上邻、下邻。

```
template <typename T> using QNodePosi = QNode<T>*; //节点位置
template <typename T> struct QNode { //四联节点
    T entry; //所存词条
    QNodePosi<T> pred, succ, above, below; //前驱、后继、上邻、下邻
    QNode( T e = T(), QNodePosi<T> p = NULL, QNodePosi<T> s = NULL,
    QNodePosi<T> a = NULL, QNodePosi<T> b = NULL ) //构造器
        : entry(e), pred(p), succ(s), above(a), below(b) {}
    QNodePosi<T> insert( T const& e, QNodePosi<T> b = NULL ); //将 e 作为当前节点
    的后继、 b 的上邻插入
};
```

由四联节点构成四联表，作为跳转表的一层。

```
template <typename T> struct Quadlist { //四联表
    Rank _size; //节点总数
    QNodePosi<T> header, trailer; //头、尾哨兵
    void init(); int clear(); //初始化、 清除
    Quadlist() { init(); } //构造
```

```

~Quadlist() { clear(); delete header; delete trailer; } //析构
T remove( QNodePosi<T> p ); //删除 p
QNodePosi<T> insert( T const & e, QNodePosi<T> p, QNodePosi<T> b =
NULL );//将 e 作为 p 的后继、 b 的上邻插入
};

```

从四联表和字典继承到跳转表。

```

template < typename K, typename V > struct Skiplist : public Dictionary<K, V>,
public List< Quadlist< Entry<K, V> >*> {
    Skiplist() { insertAsFirst( new Quadlist< Entry<K, V> > ); } //至少有一层
空列表
    QNodePosi< Entry<K, V> > search( K ); //由关键码查询词条
    Rank size() { return empty() ? 0 : last()->data->size(); } //词条总数
    Rank height() { return List::size(); } //层高，即 Quadlist 总数
    bool put( K, V ); //插入 (Skiplist 允许词条重复，故必然成功)
    V * get( K ); //读取
    bool remove( K ); //删除
};

```

9.8.1 性能

空间上，各层塔高符合几何分布： $P_i = \left(\frac{1}{2}\right)^i$ ，塔高的期望是2。故总空间复杂度为 $O(n)$ 。

9.8.2 插入与删除

先查询到插入的位置，在紧邻右边的位置建塔。

```

template <typename K, typename V> bool Skiplist<K, V>::put( K k, V v ) {
    Entry< K, V > e = Entry< K, V >( k, v ); //待插入的词条（将被同一塔中所有节点
共用）
    QNodePosi< Entry<K, V> > p = search( k ); //查找插入位置： 新塔将紧邻其右，逐
层生长
    ListNodePosi< Quadlist< Entry<K, V> >*> qlist = last(); //首先在最底层
    QNodePosi< Entry<K, V> > b = qlist->data->insert( e, p ); //创建新塔的基座
    while ( rand() & 1 ) { //经投掷硬币，若新塔需再长高，则
        /* ... 建塔 ... */
        while ( p->pred && !p->above ) p = p->pred; //找出不低于此高度的最近前驱
        if ( !p->pred && !p->above ) { //若该前驱是 header，且已是顶层，则
            insertAsFirst( new Quadlist< Entry<K, V> > ); //需要创建新的一层
            first()->data->header->below = qlist->data->header;
            qlist->data->header->above = first()->data->header;
        }
        p = p->above; qlist = qlist->pred; //上升一层，并在该层
        b = qlist->data->insert( e, p, b ); //将新节点插入 p 之后、 b 之上
    }
    return true; //Dictionary 允许重复元素，故插入必成功
} //体会：得益于哨兵的设置，哪些环节被简化了？

```

建塔时，每上升一层，都要重新组织指针，并且决定是否向上一层是随机的。

删除时，先查询到删除的位置，然后逐层删除。

```
template <typename K, typename V> bool SkipList<K, V>::remove( K k ) {
    /* ... 1. 预备 ... */
    QNodePosi< Entry<K, V> > p = search( k ); //查找目标词条
    if ( !p->pred || (k != p->entry.key) ) return false; //若不存在，直接返回
    ListNodePosi< Quadlist< Entry<K, V> >*> qlist = last(); //从底层 Quadlist
    begin
        while ( p->above ) { qlist = qlist->pred; p = p->above; } //升至塔顶
        /* ... 2. 拆塔 ... */
        do { QNodePosi< Entry<K, V> > lower = p->below; //记住下一层节点，并
            qlist->data->remove( p ); //删除当前层节点，再
            p = lower; qlist = qlist->succ; //转入下一层
        } while ( qlist->succ ); //直到塔基
        /* ... 3. 删除空表 ... */
        while ( (1 < height()) && (first()->data->_size < 1) ) { //逐层清除
            List::remove( first() );
            first()->data->header->above = NULL;
        } //已不含词条的 Quadlist (至少保留最底层空表)
        return true; //删除成功
    } //体会：得益于哨兵的设置，哪些环节被简化了？
}
```

9.8.3 查找

跳转表的查找从左边的塔顶开始，如果当前节点的后继是空，或者后继的关键码大于目标关键码，就向下一层。如果当前节点的后继的关键码等于目标关键码，就返回当前节点的后继。

```
template <typename K, typename V> //关键码不大于 k 的最后一个词条（所对应塔的基座）
QNodePosi< Entry<K, V> > SkipList<K, V>::search( K k ) {
    for ( QNodePosi< Entry<K, V> > p = first()->data->header; ; ) //从顶层的首
    节点出发
        if ( (p->succ->succ) && (p->succ->entry.key <= k) ) p = p->succ; //尽可能右移
        else if ( p->below ) p = p->below; //水平越界时，下移
        else return p; //验证：此时的 p 符合输出约定（可能是最底层列表的 header）
    } //体会：得益于哨兵的设置，哪些环节被简化了？
}
```

比较复杂的是估算该过程的复杂度。

跳转分为横向跳转和纵向跳转。

先分析纵向跳转。对于跳表高度，随着 k 的增加，第 k 层为空的概率急剧上升
 $P(|S_k| = 0) = (1 - p^k)^n$ 。从而跳表高度的期望是 $O(\log n)$ 的，从而纵向跳转的期望是 $O(\log n)$ 的。

再分析横向跳转。可以注意到在同一水平列表中，横向跳转所经节点必然依次紧邻，而且每次抵达都是塔顶。塔的高度理想随机，沿同一层跳转的次数呈几何分布。从而每个高度横向跳转的期望是 $O(1)$ 的。

综上，跳转表的查找时间复杂度是 $O(\log n)$ 。

十 优先级队列 Priority Queue

循优先级访问：对存入的数据约定一定的优先级，每次访问是严格按照优先级的。

1. 在应用中会出现需要循优先级访问的例子：

- 离散事件模拟
- 操作系统：任务调度/中断处理/MRU/...
- 输入法：词频调整

2. 希望能够：

- 快速找到极值元素：须反复地、快速地定位
- 集合组成：可动态变化
- 元素优先级：可动态变化

3. 作为底层数据结构所支持的高效操作是很多高效算法的基础

- 内部、外部、在线排序
- 贪心算法：Huffman 编码、Kruskal
- 平面扫描算法中的事件队列

```
template <typename T> struct PQ { //priority queue
    virtual void insert( T ) = 0;
    virtual T getMax() = 0;
    virtual T delMax() = 0;
}; //作为 ADT 的 PQ 有多种实现方式，各自的效率及适用场合也不尽相同
```

- Stack 和 Queue，都是 PQ 的特例——优先级完全取决于元素的插入次序；
- Steap 和 Queap，也是 PQ 的特例——插入和删除的位置受限。

对于前面的 `vector`、`sorted_vector`、`list`、`sorted_list`、`BBST`。若只需查找极值元，则不必维护所有元素之间的全序关系，偏序足矣。

因此有理由相信，存在某种更为简单、维护成本更低的实现方式，使得各功能接口的时间复杂度依然为 $O(\log n)$ ，而且实际效率更高。

10.1 完全二叉堆 Heap

由于完全二叉树的特性，我们可以把一棵完全二叉树存在一个线性的列表中。并且可以通过秩的代数运算得到父亲和儿子。

```
#define Parent(i) ( ((i) - 1) >> 1 )
#define LChild(i) ( 1 + ((i) << 1) )
#define RChild(i) ( (1 + (i)) << 1 )
```

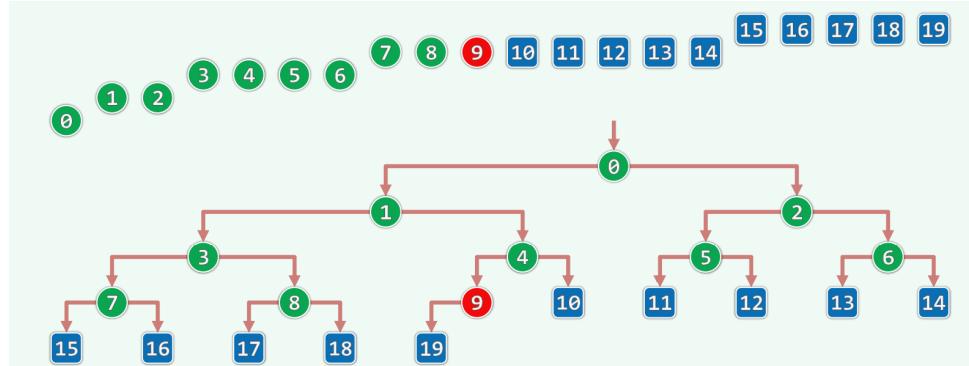


图 156 完全二叉堆的存储结构

```

template <typename T> struct PQ_CmplHeap : public PQ<T>, public Vector<T> {
    PQ_CmplHeap( T* A, Rank n ) { copyFrom( A, 0, n ); heapify( _elem, n ); }
    void insert( T );
    T getMax();
    T delMax();
};

// 该结构的实现包括下面的一些函数
template <typename T> Rank percolateDown( T* A, Rank n, Rank i ); //下滤
template <typename T> Rank percolateUp( T* A, Rank i ); //上滤
template <typename T> void heapify( T* A, Rank n ); //Floyd 建堆算法

```

该完全二叉堆满足堆序性，即：只要 $0 < i$ ，必满足 $H[i] \leq H[Parent(i)]$ ，故 $H[0]$ 即是全局最大。

```
template <typename T> T PQ_CmplHeap<T>::getMax() { return _elem[0]; }
```

现在讨论如何在动态操作后，仍维护堆序性。

10.1.1 插入：逐层上滤

```

template <typename T> void PQ_CmplHeap<T>::insert( T e ) //插入
{
    Vector<T>::insert( e );
    percolateUp( _elem, _size - 1 ); //先接入，再上滤
}

```

在插入一个元素时，我们把它放在队列的最后。每次与父亲对比，如果比父亲大，就 swap，直到不能替换。

```

template <typename T> Rank percolateUp( T* A, Rank i ) { //0 <= i < _size
    while ( 0 < i ) { //在抵达堆顶之前，反复地
        Rank j = Parent( i ); //考查[i]之父亲[j]
        if ( lt( A[i], A[j] ) ) break; //一旦父子顺序，上滤旋即完成；否则
        swap( A[i], A[j] ); i = j; //父子换位，并继续考查上一层
    } //while
    return i; //返回上滤最终抵达的位置
}

```

该算法的效率是 $O(\log n)$ 。

10.1.2 删除：割肉补疮 + 逐层下滤

每次删除的是最顶层的节点，为了保证完全二叉树的结构，我们把最后一个节点放到顶层，然后逐层下滤。

```
template <typename T> T PQ_CmplHeap<T>::delMax() { //取出最大词条
    swap( _elem[0], _elem[ --_size ] ); //堆顶、堆尾互换 (_size 递减不致引发
    shrink())
    percolateDown( _elem, _size, 0 ); //新堆顶下滤
    return _elem[_size]; //返回原堆顶
}
```

相当于去掉最顶层的节点后，变成两颗子树的合并。取来最后一个节点放在顶层，并逐层下滤即可。

```
template <typename T> Rank percolateDown( T* A, Rank n, Rank i ) { //0 <= i <
n
    Rank j; //i 及其（至多两个）孩子中，堪为父者
    while ( i != ( j = ProperParent( A, n, i ) ) ) //只要 i 非 j，则
        swap( A[i], A[j] ), i = j; //换位，并继续考察 i
    return i; //返回下滤抵达的位置（亦 i 亦 j）
}
```

10.1.3 批量建堆

10.1.3.1 自上而下的上滤

相当于每次插入在最后插入元素，然后上滤，直到插入所有元素。

```
PQ_CmplHeap( T* A, Rank n )
{ copyFrom( A, 0, n ); heapify( _elem, n ); }
```

不断调用上滤函数即可。

```
template <typename T> void heapify( T* A, const Rank n ) { //蛮力
    for ( Rank i = 1; i < n; i++ ) //按照逐层遍历次序逐一
        percolateUp( A, i ); //经上滤插入各节点
}
```

这种方法是低效的，最坏情况下每个节点都需上滤至根。耗时是 $O(n \log n)$ 的。这足以全排序，一定有更好的方法。

10.1.3.2 自下而上的下滤

任意给定堆 H_1 和 H_2 ，以及节点 p 。为得到堆 $H_1 \cup p \cup H_2$ ，只需将 H_1 和 H_2 的根当作 p 的孩子，再对 p 下滤。

```
template <typename T> //Robert Floyd, 1964
void heapify( T* A, Rank n ) { //自下而上
    for ( Rank i = n/2 - 1; -1 != i; i-- ) //依次
        percolateDown( A, n, i ); //经下滤合并子堆
} //可理解为子堆的逐层合并，堆序性最终必然在全局恢复
```

类似自下而上的归并排序。

二者的区别在于，前者（自上而下的上滤）每一个节点都要经过其深度次操作，但是后者（自下而上的下滤）每一个节点只要经过其高度次操作。

从而该算法的效率是 $O(n)$ 的。

10.2 堆排序 Heap Sort

`selectionSort()`的想法是，从后向前排序，每次把前缀最大者交换到后缀的最前端。这样，每次交换后，前缀都是有序的。复杂度是 $O(n^2)$ 的。

这个过程可以被堆优化。做一定的预处理，将前缀建成堆，然后每次取出堆顶，放到后缀的最前端。这样，每次取出后，前缀都是有序的。复杂度是 $O(n \log n)$ 的。

```
template <typename T> void Vector<T>::heapSort( Rank lo, Rank hi ) { //就地堆排
    T* A = _elem + lo; Rank n = hi - lo; heapify( A, n ); //待排序区间建堆,
    O(n)
    while ( 0 < --n ) //反复地摘除最大元并归入已排序的后缀，直至堆空
        { swap( A[0], A[n] ); percolateDown( A, n, 0 ); } //堆顶与末元素对换后下
    滤
}
```

具体实现是就地建堆，对顶就是第一个元素，然后每次取出堆顶，放到后缀的最前端，再对新堆顶下滤。

10.3 锦标赛树 Tournament Tree

锦标赛树是完全二叉堆的等效方法。

10.3.1 胜者树

锦标赛树的每一个节点都是一个比赛，每一个节点都有一个胜者。每一个节点的胜者都是其左右孩子的胜者中的较小者。从而树的根部就是最小者，类似于堆顶。

```
TournamentSort()
CREATE a tournament tree for the input list
while there are active leaves
- REMOVE the root
- RETRACE the root down to its leaf
- DEACTIVATE the leaf
- REPLAY along the path back to the root
```

每次取出胜者后，沿着胜者的路径，重新比赛，可以算出新的胜者。

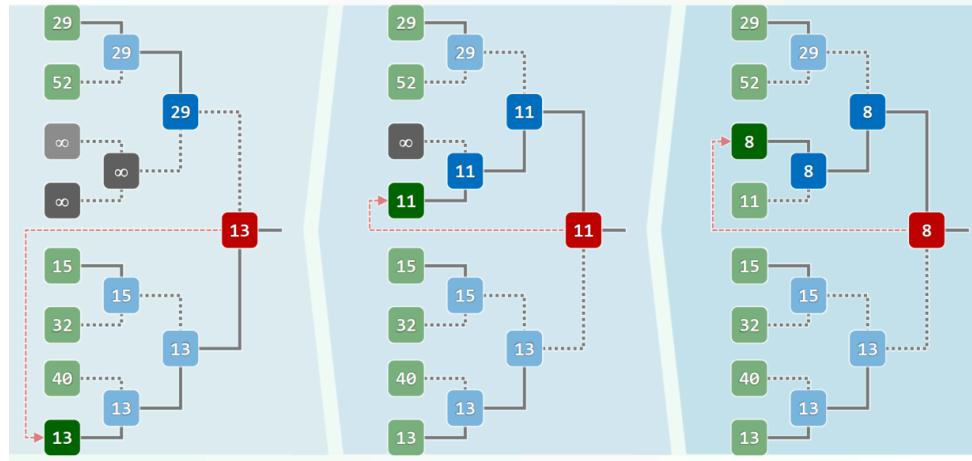


图 157 锦标赛树的更新

锦标赛树的 `create()` 是 $O(n)$ 的，`replay()` 是 $O(\log n)$ 的。

从而如果利用锦标赛树排序，是 $O(n \log n)$ 的。

在锦标赛树中，进行 k 次迭代选取的话，一共需要 $O(k \log n)$ 的时间。在渐进意义上与完全二叉堆是一样的。但是由于占用空间和堆局部性不好的原因，常系数差别较大。堆的每次下滤不一定到最底层，而胜者树一定会遍历所有层。

10.3.2 败者树

胜者树重赛过程中，须交替访问沿途节点及其兄弟，这造成了比较大的开销。

而败者树内部的节点记录比赛的败者，增设根的父亲，记录冠军。

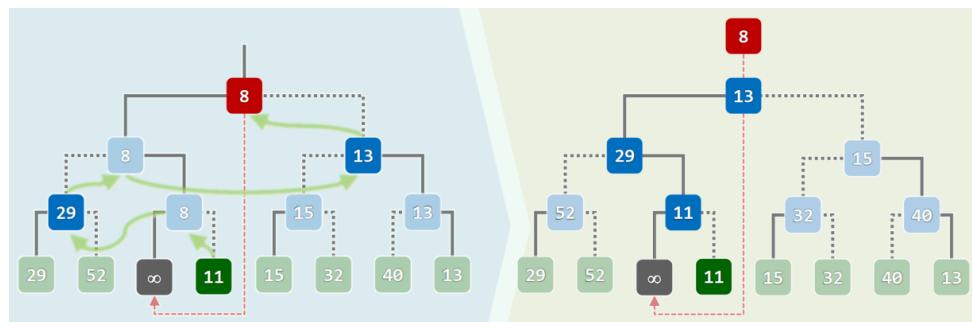


图 158 败者树的储存

在构造时，每个节点都会留下败者，向上抛出胜者：每次比较两个节点（下方节点抛给的胜者），把败者放在父亲上，把胜者向上抛出。

而更新时，就直接沿着胜者的路径，重新比赛，可以算出新的胜者。不需要再访问兄弟节点了。

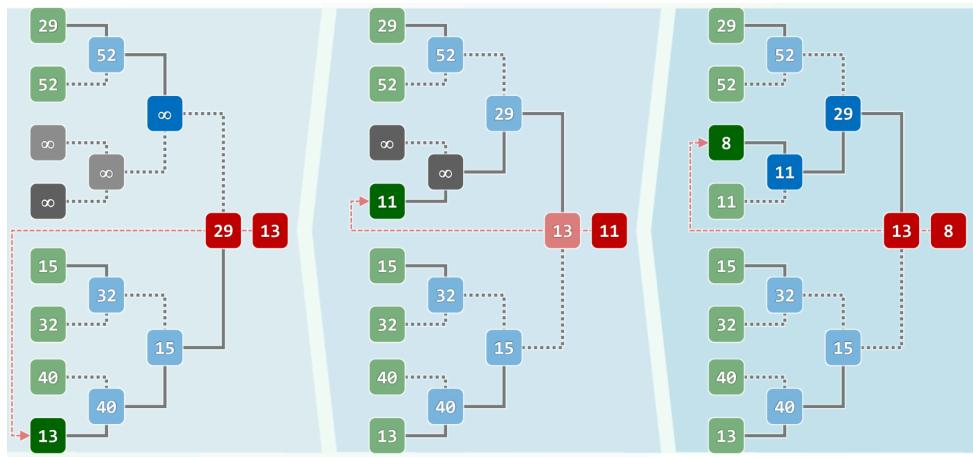


图 159 败者树的更新

十一 串 string/char[]

用 `char[]` 表示串，主要讨论串的匹配。

11.1 蛮力算法 BF

从前往后遍历，每次匹配失败则推进到下一个位置。

通常建议用双移动的指针实现。

```
int match( char * P, char * T )
{
    size_t n = strlen(T), i = 0;
    size_t m = strlen(P), j = 0;
    while ( j < m && i < n ) //自左向右逐次比对（可优化）
        if ( T[i] == P[j] ) { i++; j++; } //若匹配，则转到下一对字符
        else { i -= j-1; j = 0; } //否则，T回退、P复位
    return i-j; //最终的对齐位置：藉此足以判断匹配结果
}
```

通过移动同时移动 `i` 和 `j`，逐位比较，之后 `i` 回退，`j` 复位，继续比较。

这种算法的时间复杂度为 $O(mn)$ ，其中 m 为模式串长度， n 为文本串长度。

11.2 KMP 算法

一次比较在第 j 位失配，已经掌握了前 j 位的信息，可以利用这些信息，跳过一些不必要的比较。

例如，一次比较后，就可以直接跳转到下图的位置，而不必从头开始比较。

```
R E G R E S S ...
R E G R E T S
      R E G R E T S
```

快速右移+决不后退：这满足，模式串长为 j 的前缀的一部分真前缀等于真后缀，且要保证真前缀的长度最大。

11.2.1 `next[]` 表

用这样的方式构造查询表 `next[]`。

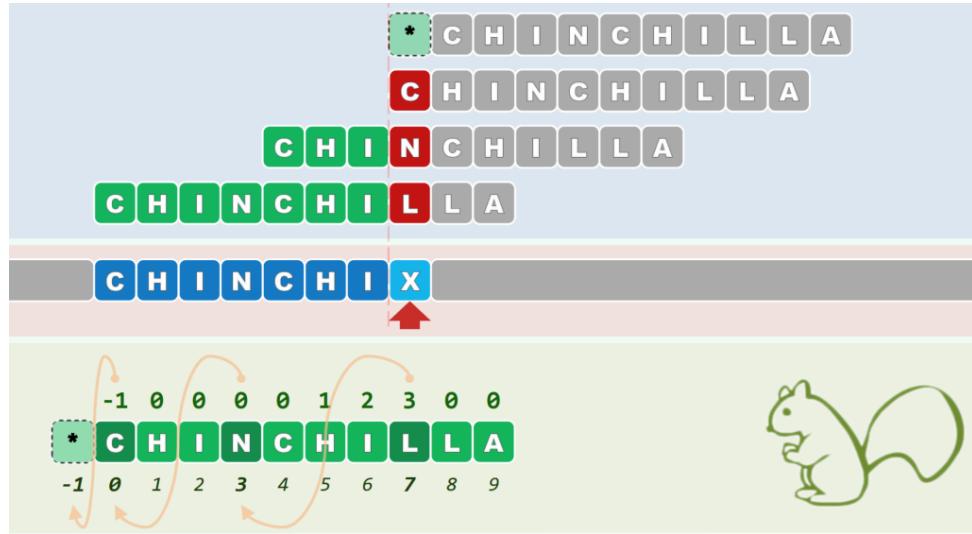


图 160 next[] 表

用数学语言描述是：

$$\forall j \geq 1, N(P, j) = \{0 \leq t < j \mid P[0, t) = P[j-t, j)\}$$

$$\text{next}[j] = \max\{N(P, j)\}$$

下面写出构造的递推关系：

$$\text{next}[j+1] = \text{next}[j] + 1 \Leftrightarrow P[j] = P[\text{next}[j]]$$

这是因为，如果 $P[j] = P[\text{next}[j]]$ ，那么 $P[j+1] = P[\text{next}[j]+1]$ ，所以 $P[j+1]$ 的真前缀长度为 $P[j]$ 的真前缀长度加一。

如果 $P[j] \neq P[\text{next}[j]]$ ，就向前递推，看 $P[J]$ 与 $P[\text{next}[\text{next}[j]]]$ 是否相等。如果相等，就可令 $\text{next}[j+1] = \text{next}[\text{next}[j]] + 1$ ，否则继续递推。直到找到一个相等的，或者 j 递推到 0 ($\text{next}[0] = -1$)。

```
int* buildNext( char* P ) {
    size_t m = strlen(P), j = 0;
    int* next = new int[m];
    int t = next[0] = -1;
    while ( j < m - 1 )
        if ( 0 > t || P[t] == P[j] ) { // 匹配
            ++t; ++j; next[j] = t; // 则递增赋值
        } else // 否则
            t = next[t]; // 继续尝试下一值得尝试的位置
    return next;
}
```

这样以来 KMP 算法(Knuth-Morris-Pratt)就可以写成：

```

int match( char * P, char * T ) {
    int * next = buildNext(P);
    int n = (int) strlen(T), i = 0;
    int m = (int) strlen(P), j = 0;
    while ( j < m && i < n ) //可优化
        if ( 0 > j || T[i] == P[j] ) {
            i++; j++;
        } else
            j = next[j];
    delete [] next;
    return i - j;
}

```

通过生成模式串的 `next[]` 表，可以在匹配失败时，直接跳转到 `next[]` 表中的位置，而不必从头开始比较。

11.2.2 分摊分析

KMP 算法的时间复杂度为 $O(m + n)$ ，其中 m 为模式串长度， n 为文本串长度。

其中，生成 `next[]` 表的时间复杂度为 $O(m)$ ，匹配的时间复杂度为 $O(n)$ 。

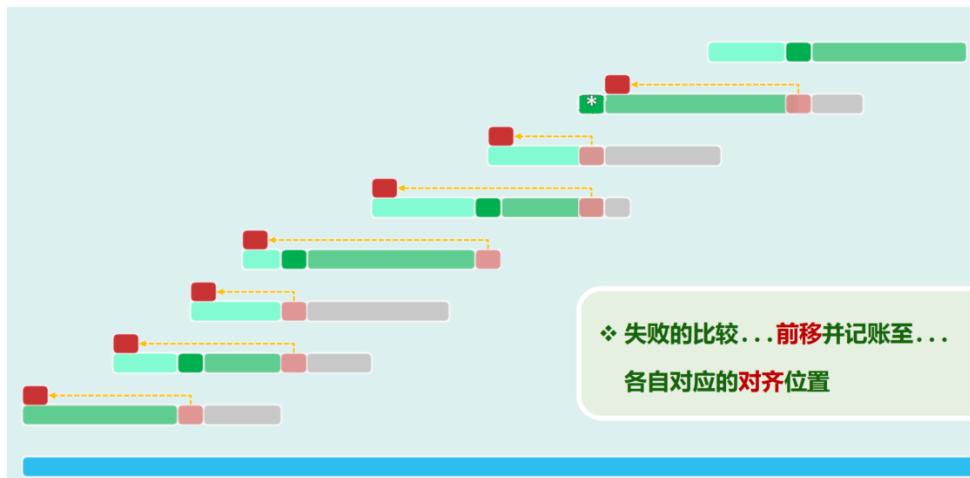


图 161 KMP 算法的分摊分析

考虑上图，浅绿色是不言自明的比较、深绿色是消耗时间的比较。将红色即失配部分，前移后，可以看到所有深绿与红色的部分之和，为 $O(n)$ 的。

也可以构造计步器 $k = 2i - j$ ，这个计步器是严格增加的，最大值是 $2n - 1$ ，所以时间复杂度为 $O(n)$ 。

```

while ( j < m && i < n ) //k 必随迭代而单调递增，故也是迭代步数的上界
    if ( 0 > j || T[i] == P[j] )
        { i++; j++; } //k 恰好加 1

```

```
else
    j = next[j]; //k至少加1
```

11.3 优化的 KMP 算法

按照刚才的方法，每次失配的下一次对比没有用到目前正在比较的 $P[j]$ 的信息。

如果该位置是字符 c 导致失配，那么下一次比较，除了刚才的前后缀相同以外，保证前缀的下一位不再是 c 。这样就可以吸取刚才的教训。

11.3.1 $\text{next}[]$ 表

$$\forall j \geq 1, N(P, j) = \{0 \leq t < j \mid P[0, t) = P[j - t, j) \text{ 且 } P[t] \neq P[j]\}$$
$$\text{next}[j] = \max\{N(P, j)\}$$

代码实现如下

```
int* buildNext( char* P ) {
    size_t m = strlen(P), j = 0;
    int* next = new int[m]; int t = next[0] = -1;
    while ( j < m - 1 ) {
        if ( 0 > t || P[t] == P[j] ) {
            if ( P[++t] != P[++j] )
                next[j] = t;
            else //P[next[t]] != P[t] == P[j]
                next[j] = next[t];
        } else
            t = next[t];
    }
    return next;
}
```

该算法单次匹配概率越大（字符集越小），优势越明显。

11.4 BM 算法

Boyer-Moore 提出了两种策略，BC 策略和 GS 策略。

11.4.1 BC 策略

每次匹配从末字符开始，从后向前匹配。如果失配，就考察前面的字符是否有造成失配的，如果有，就将其移动过来，否则就移动整个模式串。



图 162 BC 策略

11.4.1.1 Bad-Character 规则

如果失配的字符在模式串中，就将模式串移动到该字符的右侧，否则移动整个模式串。

画家算法：将第 j 位对应的字符 c 处的 $bc[c]$ 赋值成 j ，如果 c 不在模式串中，就赋值成 -1 。从小到大遍历后， $bc[*]$ 中储存的就是每个字符最后出现的位置。

```
int * buildBC( char * P ) {
    int * bc = new int[ 256 ];
    for ( size_t j = 0; j < 256; j++ ) bc[j] = -1;
    for ( size_t m = strlen(P), j = 0; j < m; j++ )
        bc[ P[ j ] ] = j; //painter's algorithm
    return bc;
} //O( s + m )
```

只用存储最后一个而不用全部存储的原因是，每次移动可以保证在移动中间的位置都是不合法的，只要一直比较并且移动，就可以保证不会出现不合法的情况。

11.4.1.2 复杂度分析

最好情况是 $O(\frac{n}{m})$ ，最坏情况是 $O(mn)$ 。

单次匹配概率越小，性能优势越明显，需单次比较，即可排除 m 个对齐位置，一次移动 m 个对齐位置。

单次匹配概率越大的场合，性能越接近于蛮力算法。

11.4.2 GS 策略

仿照 KMP，记忆好后缀。相当于 KMP 从后颠倒。

11.4.2.1 Good-Suffix 规则

扫描比对中断于 $T[i + j] \neq P[j]$ 时， $U = P(j, m)$ 必为好后缀。下一对齐位置满足：

1. U 重新与 $V(k) = P(k, m + k - j)$ 匹配 (经验)

2. $P[k] \neq P[j]$ (教训)

11.4.3 综合性能

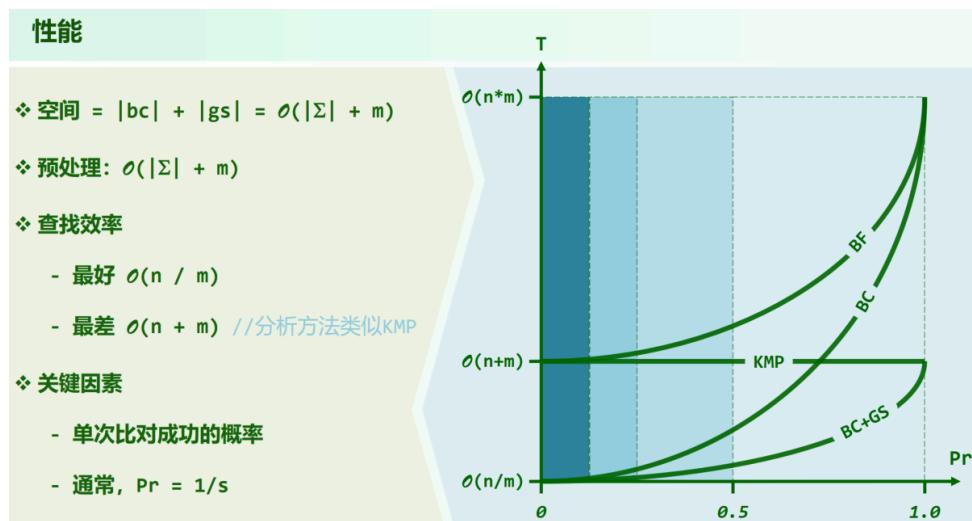


图 163 字符串匹配的综合性能

11.5 Karp-Rabin 算法

将字符串看成是一个数字，通过哈希函数将字符串转换成数字，然后比较数字是否相等。

通过散列，将指纹压缩至存储器支持的范围，但指纹相同，原串却未必匹配。

11.5.1 快速指纹计算

每次计算指纹时，只需要减去最高位，然后乘以基数，再加上新的最低位即可。这样可以在 $O(1)$ 的时间内完成递推。

十二 排序

12.1 快速排序 Quick Sort

选择一个轴点，将小于轴点的元素放在轴点左边，大于轴点的元素放在轴点右边，然后递归地对左右两个子序列进行快速排序。 $\text{sorted}(S) = \text{sorted}(S_L) + \text{pivot} + \text{sorted}(S_R)$

$\text{pivot: } \max[\text{lo}, \text{mi}] \leq \text{pivot} < \min(\text{mi}, \text{hi})$

排好的轴点就是排序之后的位置，此后不会再动。快速排序就是将所有元素逐个转换为轴点的过程。

```
template <typename T> void Vector<T>::quickSort( Rank lo, Rank hi ) {
    if ( hi - lo < 2 ) return;
    Rank mi = partition( lo, hi ); //能否足够高效?
    quickSort( lo, mi );
    quickSort( mi + 1, hi );
}
```

12.1.1 partition——LUG 版

每次选取一个轴点的候选，从前缀和后缀交替扫描，将小于轴点的元素交换到前缀，大于轴点的元素交换到后缀，直至交换至前缀与后缀的交界处。

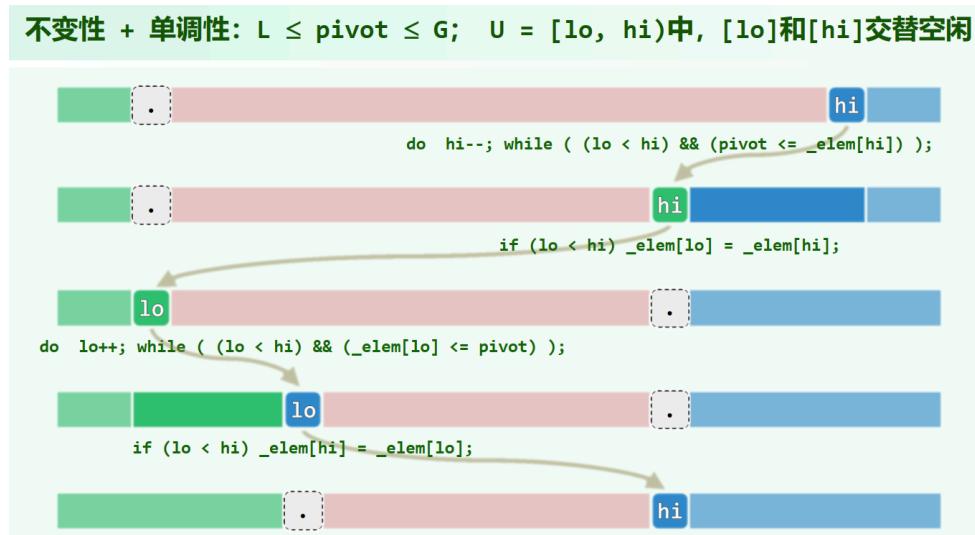


图 164 LUG 版 partition

交替的方法是：从前缀开始，如果当前元素小于候选者，则继续向后扫描；如果当前元素大于候选者，则从后缀开始，向前扫描，直至找到一个小于轴点的元素，将其交换至前缀，然后继续向后扫描。

```

template <typename T> Rank Vector<T>::partition( Rank lo, Rank hi ) { // [lo,
hi)
    swap( _elem[lo], _elem[lo + rand() % (hi-lo)] ); // 随机交换
    T pivot = _elem[lo]; // 经以上交换，等效于随机选取候选轴点
    while ( lo < hi ) { // 从两端交替地向中间扫描，彼此靠拢
        do hi--; while ( (lo < hi) && (pivot <= _elem[hi]) ); // 向左拓展 G
        if (lo < hi) _elem[lo] = _elem[hi]; // 凡 小于 轴点者，皆归入 L
        do lo++; while ( (lo < hi) && (_elem[lo] <= pivot) ); // 向右拓展 L
        if (lo < hi) _elem[hi] = _elem[lo]; // 凡 大于 轴点者，皆归入 G
    } // assert: lo == hi or hi+1
    _elem[hi] = pivot; return hi; // 候选轴点归位；返回其秩
}

```

12.1.1.1 时间复杂度

最好的情况下，每次都几乎均匀地划分成两个子序列，递归树的深度为 $O(\log n)$ ，每层的时间复杂度为 $O(n)$ ，总的时间复杂度为 $O(n \log n)$ 。

最坏的情况下，每次都只能划分成一个子序列，递归树的深度为 $O(n)$ ，每层的时间复杂度为 $O(n)$ ，总的时间复杂度为 $O(n^2)$ 。

采用随机选取（Randomization）、三者(low,high,mid)取中（Sampling）之类的策略，降低最坏情况的概率，而无法杜绝。但是数据是非理想随机的，所以这些策略可能有效。

12.1.1.2 空间复杂度

空间复杂度即是递归栈的深度，最好情况下，递归栈的深度为 $O(\log n)$ ，空间复杂度为 $O(\log n)$ 。

模拟进栈过程：

```

#define Put( K, s, t ) { if ( 1 < (t) - (s) ) { K.push(s); K.push(t); } }
#define Get( K, s, t ) { t = K.pop(); s = K.pop(); }
template <typename T> void Vector<T>::quickSort( Rank lo, Rank hi ) {
    Stack<Rank> Task; Put( Task, lo, hi ); // 类似于对递归树的先序遍历
    while ( !Task.empty() ) {
        Get( Task, lo, hi ); Rank mi = partition( lo, hi );
        if ( mi-lo < hi-mi ) { Put( Task, mi+1, hi ); Put( Task, lo, mi ); }
        else { Put( Task, lo, mi ); Put( Task, mi+1, hi ); }
    } // 大|小任务优先入|出栈，可保证（辅助栈）空间不过 O(logn)
}

```

小任务优先出栈，大任务优先入栈，可保证辅助栈空间不过 $O(\log n)$ 。

12.1.1.2.1 递归深度分析

下面我们证明：最坏情况递归 $\Omega(n)$ 层，概率极低；平均情况递归 $O(\log n)$ 层，概率极高。

对于一个区间，我们定义好的轴点为居中占比为 λ 的部分的轴点。事实上对于除非过于侧偏的 pivot，都会有效地缩短递归深度。

```
|<--(1-lambda)/2-->|<--lambda-->|<--(1-lambda)/2-->|
|<----坏轴点----->||<--好轴点-->||<----坏轴点----->|
```

于是，在任何一条递归路径上，好轴点决不会多于

$$d(n, \lambda) = \log_{\frac{2}{n+1}} n$$

这是因为之后递归的所有好节点会出现在当前区间的中间占比为 $\lambda + \frac{1-\lambda}{2}$ 的部分。

以 $\lambda = 0.5$ 为例， $d(n, 0.5) = \log_{\frac{4}{3}} n \approx 2.41 \log n$ 。这意味着同时，深入 $\frac{1}{\lambda} d(n, \lambda)$ 层后，即可期望出现 $d(n, \lambda)$ 个好轴点——从而在此之前终止递归。

下面证明任何一条递归路径的长度，只有极小的概率超过

$$D(n, \lambda) = \frac{2}{\lambda} d(n, \lambda)$$

事实上此概率

$$\begin{aligned} &\leq \sum_{k=0}^{D(n, \lambda)} (1-\lambda)^k \lambda^{D-k} \\ &= 2^{-D(n, \lambda)} \sum_{k=0}^{D(n, \lambda)} (2\lambda)^k \\ &\leq 2^{-4D} \left(e \frac{D}{d} \right)^d = 16^{-d} (4e)^d \\ &\approx n^{-1.343} \end{aligned}$$

当 $n = 10^6$ 时，递归深度不超过 D 的概率 $\geq 1 - n^{-0.343} \approx 99.12\%$ 。从而可以说复杂度为 $O(\log n)$ 是极高概率发生（occurring w.h.p）的。

12.1.1.2.2 比较次数分析

记期望的比较次数为 $T(n)$ ，是一个期望值，下面分析 $T(n)$ 的递归表达式。

1. 递推分析

设 $T(n)$ 为 n 个元素的序列的期望比较次数，是 `partition` 与递归任务的期望之和，

$$T(0) = T(1) = 0,$$

$$\begin{aligned} T(n) &= n - 1 + \frac{1}{n} \sum_{k=0}^{n-1} (T(i) + T(n-i-1)) \\ &= n - 1 + \frac{2}{n} \sum_{k=0}^{n-1} T(i) \\ &\approx 2n \ln n \end{aligned}$$

2. 后向分析

设经排序后得到的输出序列为： $\{a_1, a_2, \dots, a_i, \dots, a_j, \dots, a_n\}$ 。

这一输出与具体使用何种算法无关，故可使用 Backward Analysis。

比较操作的期望次数应为

$$T(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} P(i, j)$$

亦即，每一对 $\langle a_i, a_j \rangle$ 在排序过程中接受比较之概率的总和。

`quickSort` 的过程及结果，可理解为：按某种次序，将各元素逐个确认为 `pivot`。

若 $k \in [0, i) \cup (j, n]$ ，则 a_k 早于或晚于 a_i 和 a_j 被确认，均与 $P(i, j)$ 无关。实际上， $\langle a_i, a_j \rangle$ 接受比较，当且仅当在 $\{a_i, \dots, a_j\}$ 中， a_i 或 a_j 率先被确认。

从而

$$\begin{aligned} T(n) &= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} P(i, j) \\ &= \sum_{j=0}^{n-1} \sum_{i=0}^{k-1} P(i, j) \\ &= \sum_{j=0}^{n-1} \sum_{i=0}^{k-1} \frac{2}{j - i + 1} \\ &\approx \sum_{j=0}^{n-1} 2(\ln(j) - 1) \\ &\leq 2n \ln n \end{aligned}$$

Vector sorter	#compare	#move	cache-friendly
Insertionsort	$\mathcal{O}(n) \sim \mathcal{O}(n^2)$ expected- $\mathcal{O}(n^2)$	$\mathcal{O}(n) \sim \mathcal{O}(n^2)$ expected- $\mathcal{O}(n^2)$	
Selectionsort	$\Theta(n^2)$	$\mathcal{O}(n)$	
Heapsort	$2.0 * n \log n$	$1.0 * n \log n$	<code>percolateDown(): X[i] X[j]</code>
Mergesort	$1.0 * n \log n$	$1.5 * n \log n$	<code>merge(): ✓[i] ✓[j] ✓[k]</code>
Quicksort	expected- $1.386 * n \log n$ w.h.p. ($1.386/2 = 0.694$) * $n \log n$	expected- $\sqrt{\log n}$	$\checkmark \checkmark$ pivot ✓[lo] ✓[hi]

图 165 排序算法的对比

12.1.2 `partition`--DUP 版

有大量元素与轴点雷同时

- 切分点将接近于 `lo`
- 划分极度失衡

- 递归深度接近于 $O(n)$
- 运行时间接近于 $O(n^2)$

移动 lo 和 hi 的过程中，同时比较相邻元素，若属于相邻的重复元素，则不再深入递归。但一般情况下，如此计算量反而增加，得不偿失。

DUP 的想法就是在 LUG `partition` 的比较上，将两头指针移动的判定条件的判定条件由 \leq 改为 $<$ ，从而使相邻元素不再重复比较。

```
template <typename T> Rank Vector<T>::partition( Rank lo, Rank hi ) { // [lo, hi)
    swap( _elem[lo], _elem[lo + rand() % (hi-lo)] ); // 随机交换
    T pivot = _elem[lo]; // 经以上交换，等效于随机选取候选轴点
    while ( lo < hi ) { // 从两端交替地向中间扫描，彼此靠拢
        /* 与 LUG 版仅改变符号 */
        do hi--; while ( (lo < hi) && (pivot < _elem[hi]) ); // 向左拓展 G
        if (lo < hi) _elem[lo] = _elem[hi]; // 凡不大于轴点者，皆归入 L
        do lo++; while ( (lo < hi) && (_elem[lo] < pivot) ); // 向右拓展 L
        if (lo < hi) _elem[hi] = _elem[lo]; // 凡不小于轴点者，皆归入 G
    } // assert: lo == hi or hi+1
    _elem[hi] = pivot; return hi; // 候选轴点归位；返回其秩
}
```

- 可以正确地处理一般情况同时复杂度并未实质增高；
- 遇到连续的重复元素时
 - lo 和 hi 会交替移动
 - 切分点接近于 $(lo+hi)/2$
- 由 LUG 版的勤于拓展、懒于交换，转为懒于拓展、勤于交换

12.1.3 `partition`— LGU 版

在形式上将 `partition` 化简，全部用 `swap` 实现。呈现形式是 Pivot + L + G + U。用这样 的方式，可以大大简化代码。



图 166 LGU 版 `partition`

采用 `swap` 实现滚动拓展，而非平移拓展

```
template <typename T> Rank Vector<T>::partition( Rank lo, Rank hi ) { // [lo, hi)
    swap( _elem[ lo ], _elem[ lo + rand() % ( hi - lo ) ] ); // 随机交换
    T pivot = _elem[ lo ]; Rank mi = lo;
    for ( Rank k = lo + 1; k < hi; k++ ) // 自左向右考查每个 [k]
        if ( _elem[ k ] < pivot ) // 若 [k] 小于轴点，则将其
            swap( _elem[ ++mi ], _elem[ k ] ); // 与 [mi] 交换， 向右扩展
    swap( _elem[ lo ], _elem[ mi ] ); // 候选轴点归位（从而名副其实）
    return mi; // 返回轴点的秩
}
```

12.2 k-selection

k-selection：在任意一组可比较大小的元素中，由小到大，找到次序为 k 者。亦即，在这组元素的非降排序序列 S 中，找出 $S[k]$ 。

median：长度为 n 的有序序列 S 中，元素 $S[\lfloor \frac{n}{2} \rfloor]$ 称作中位数。

12.2.1 众数 Majority

无序向量中，若有一半以上元素同为 m ，则称之为众数。

12.2.1.1 充分条件

如果获取了中位数，只需要验证该数是否为众数即可，若不是，则无众数。

```
template <typename T> bool majority( Vector<T> A, T & maj )
{ return majEleCheck( A, maj = median( A ) ); }
```

mode：众数若存在，则亦必频繁数。

```
template <typename T> bool majority( Vector<T> A, T & maj )
{ return majEleCheck( A, maj = mode( A ) ); }
```

同样地：`mode()` 算法难以兼顾时间、空间的高效。

12.2.1.2 减而治之——丢掉前缀

若在向量 A 的前缀 P ($|P|$ 为偶数) 中，元素 x 出现的次数恰占半数，则 A 有众数，仅当对应的后缀 $A - P$ 有众数 m ，且 m 就是 A 的众数。

证：

1. 若 $x = m$ ，则在排除前缀 P 之后， m 与其它元素在数量上的差距保持不变
2. 若 $x \neq m$ ，则在排除前缀 P 之后， m 与其它元素在数量上的差距不致缩小

从而可以按照这样的方法，不断地剪掉前缀、保存候选者，进行扫描。

```
template <typename T> T majCandidate( Vector<T> A ) {
    T maj;
    for ( Rank c = 0, i = 0; i < A.size(); i++ )
```

```

if ( 0 == c ) {
    maj = A[i]; c = 1;
} else
    maj == A[i] ? c++ : c--;
return maj;
}

```

最后验证候选者是否为众数。

```

template <typename T> bool majority( Vector<T> A, T & maj )
{ return majEleCheck( A, maj = majEleCandidate( A ) ); }

```

12.2.2 QuickSelect

下图是对 k-selection 的尝试，采用了不同的方法。

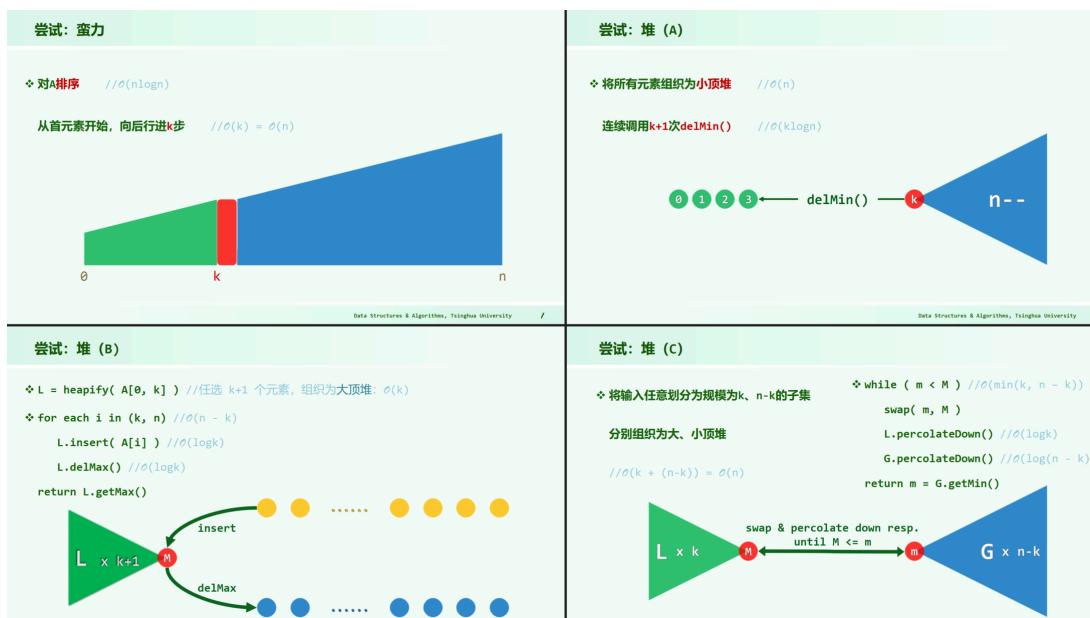


图 167 尝试

希望能找到 $O(n)$ 的算法。

采用快速排序的想法，每次选取一个轴点，将小于轴点的元素放在轴点左边，大于轴点的元素放在轴点右边。直到轴点的秩为 k 。

```

template <typename T> void quickSelect( Vector<T> & A, Rank k ) {
    for ( Rank lo = 0, hi = A.size(); lo < hi; ) {
        Rank i = lo, j = hi; T pivot = A[lo]; //大胆猜测
        while ( i < j ) { //小心求证:  $O(hi - lo + 1) = O(n)$ 
            do j--; while ( (i < j) && (pivot <= A[j]) ); if ( i < j ) A[i] =
A[j];
            do i++; while ( (i < j) && (A[i] <= pivot) ); if ( i < j ) A[j] =

```

```

A[i];
} //assert: quit with i == j or j+1
A[j] = pivot;
if ( k <= j ) hi = j; //suffix trimmed
if ( i <= k ) lo = i; //prefix trimmed
} //A[k] is now a pivot
}

```

在期望上讲，复杂度为 $O(n)$ ，但是最坏情况下，复杂度为 $O(n^2)$ 。

可以用递推的方式证明：

$$\begin{aligned}
 T(n) &= (n-1) + \frac{1}{n} \sum_{k=0}^{n-1} \max\{T(k), T(n-k-1)\} \\
 &= (n-1) + \frac{1}{n} \sum_{k=0}^{n-1} T(\max\{k, n-k-1\}) \\
 &\leq (n-1) + \frac{2}{n} \sum_{k=\frac{n}{2}}^{n-1} T(k)
 \end{aligned}$$

其中

$$T(1) = 0, T(2) = 1$$

可以归纳

$$T(n) < 4n$$

从而，该算法在期望上讲，复杂度为 $O(n)$ 。

12.2.3 LinearSelect

`LinearSelect` 是 `QuickSelect` 的改进版，采用了中位数的中位数的思想。

先找局部中位数，取这些中位数的中位数，得到较好的猜测位置。由这个中位数进行分割，递归地进行查找。

```

def linearSelect( A, n, k ):
    Let Q be a small constant
    1. if ( n = |A| < Q ) return trivialSelect( A, n, k )
    2. else divide A evenly into n/Q subsequences (each of size Q)
    3. Sort each subsequence and determine n/Q medians //e.g. by insertionsort
    4. Call linearSelect() to find M, median of the medians //by recursion
    5. Let L/E/G = { x </=> M | x in A }
    6. if (k < |L|) return linearSelect(A, |L|, k)
       if (k < |L|+|E|) return M
    return linearSelect(A+|L|+|E|, |G|, k-|L|-|E|)

```

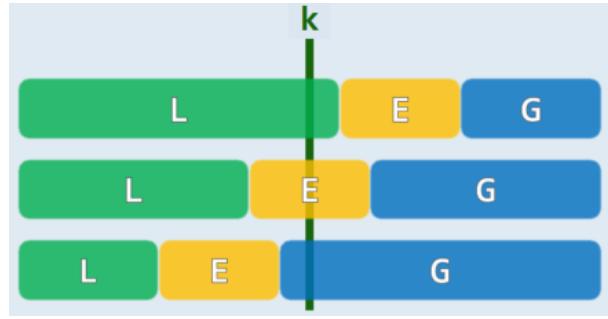


图 168 LinearSelect

将 `linearSelect()` 算法的运行时间记作 $T(n)$

- 第 0 步: $O(1) = O(Q \log Q)$, 递归基: 序列长度 $|A| \leq Q$
- 第 1 步: $O(n)$, 子序列划分
- 第 2 步: $O(n) = Q^2 \times \frac{n}{Q}$, /子序列各自排序, 并找到中位数
- 第 3 步: $T\left(\frac{n}{Q}\right)$, 从 $\frac{n}{Q}$ 个中位数中, 递归地找到全局中位数
- 第 4 步: $O(n)$, 划分子集 L/E/G, 并分别计数 —— 一趟扫描足矣
- 第 5 步: $T\left(\frac{3n}{4}\right)$, 如下图, 至少有 $\frac{1}{4}$ 被排除

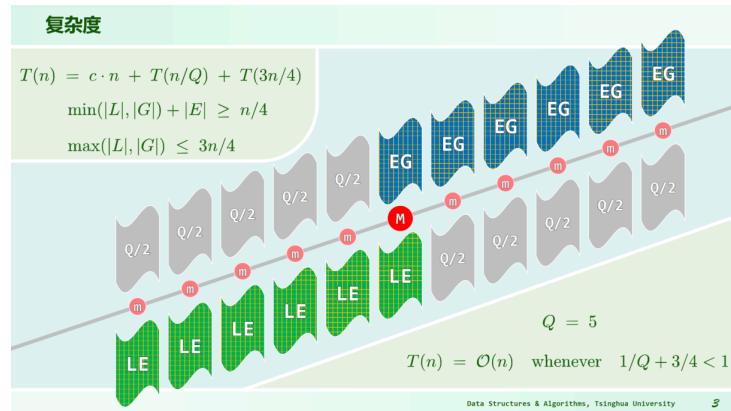


图 169 LinearSelect——复杂度

从而总复杂度是线性的, 并且选取 $Q = 5$ 。

由于常系数过大($\sim > 40$), 理论价值比应用价值高。

12.3 Shell Sort

注意到每交换一个逆序对, 总逆序对数量一定严格减少。

Shell 排列考虑将线性序列理解成矩阵, 按列进行排序。

递减增量 (diminishing increment)

- 由粗到细: 重排矩阵, 使其更窄, 再次逐列排序 (h-sorting/h-sorted)
- 逐步求精: 如此往复, 直至矩阵变成一列 (1-sorting/1-sorted)

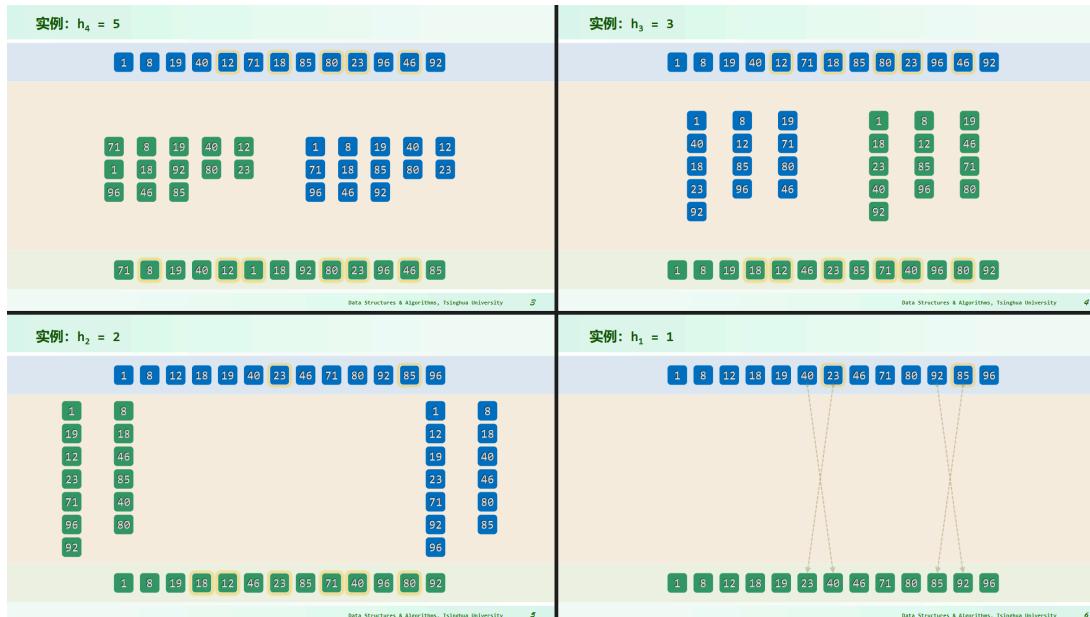


图 170 Shell Sort

```
template <typename T> void Vector<T>::shellSort( Rank lo, Rank hi ) {
    for ( Rank d = 0x7FFFFFFF; 0 < d; d >= 1 ) //PS Sequence: 1, 3, 7, 15,
31, ...
    for ( Rank j = lo + d; j < hi; j++ ) { //for each j in [lo+d, hi)
        T x = _elem[j]; Rank i = j; //within the prefix of the subsequence of
[j]
        while ( (lo + d <= i) && (x < _elem[i-d]) ) //find the appropriate
            _elem[i] = _elem[i-d]; i -= d; //predecessor [i]
        _elem[i] = x; //where to insert [j]
    }
} //0 <= lo < hi <= size <= 2^31
```

对每一列进行插入排序。事实上，Shell 排序很适合并行化。

对于 Shell 排序，选择合适的增量序列是很重要的。

12.3.1 Shell 序列

Shell 给出的是 $\{2^k\}$ 的序列，但是这样的序列并不是最优的。最坏情况要达到 $O(n^2)$ 。

反例是：考查由子序列 $A = \text{unsort}[0, 2N-1]$ 和 $B = \text{unsort}[2N-1, 2N)$ 交错而成的序列。

在做 2-sorting 时， A 、 B 各成一列；故此后必然各自有序。最后一次 1-sorting 仍需 $O(n^2)$ 。

根源在于， H_{shell} 中各项并不互素，甚至相邻项也非互素。

这里不加证明的给出几个引理：

LEM L:

如下图，设有向量 $X[\theta, m+r]$ 和 $Y[\theta, r+n]$ ，且满足：对任何 $\theta \leq j < r$ ，都有 $Y[j] \leq X[m+j]$ 。在 X 和 Y 分别（按非降次序）排序并转换为 X' 和 Y' 后，对任何 $\theta \leq j < r$ ，依然有 $Y'[j] \leq X'[m+j]$ 成立。

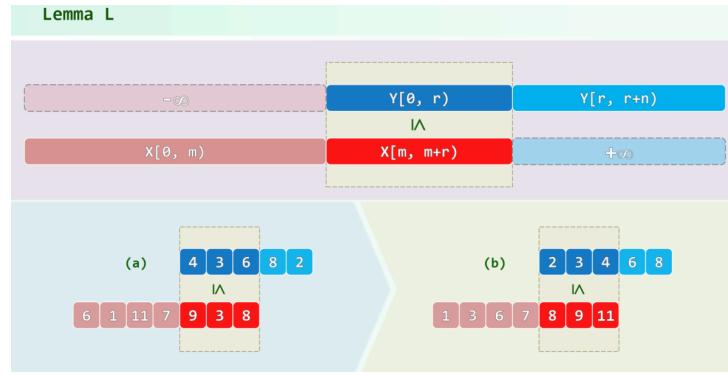


图 171 LEM L

THM K(Knuth):

A g-ordered sequence REMAINS g-ordered after being h-sorted.

证明用到了 LEM L。其中 h-ordered 是指，当排列成长为 h 的矩阵时，每一列都是有序的，即 $S[i] \leq S[i + h]$ 。

由此可以得到，经过一次 h-sorting 和 g-sorting 的序列可以保持 h-ordered 和 g-ordered。

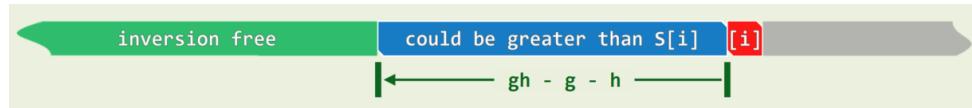


图 172 Inversion

线性组合：对于任意 $m, n \in N$ ，既是 h-sorting 又是 g-ordered 的序列称为 (g, h) -ordered 的，它是 $(mg + nh)$ -ordered 的序列。

由数论的小性质可知（一些 Bezout 定理的应用），对于互素的 g, h ，对于任意 $k > gh - g - h$ ， k -ordered 的序列必然是 (g, h) -ordered 的。

这就意味着

$$i - j > gh - g - h \Rightarrow S[i] \geq S[j]$$

所以除了左侧的 $gh - g - h$ 个元素，元素都要比 $S[i]$ 大。所以逆序对的数量不超过 $n \cdot (gh - g - h)$ 。

12.3.2 PS 序列

Papernov & Stasevic 给出序列：

$$H_{\text{PS}} = \{2^k - 1\} = H_{\text{shell}} - 1$$

需要

- $O(\log n)$ 次外部迭代
- $O(n^{3/2})$ 的复杂度

12.3.3 Pratt 序列

Pratt 给出序列:

$$H_{\text{Pratt}} = \{2^i 3^j \mid i, j \in N\}$$

复杂度是 $O(n \log^2 n)$ 。

12.3.4 Sedgewick 序列

Sedgewick 给出序列:

$$H_{\text{Sedgewick}} = \{9 \times 4^i - 9 \times 2^i + 1, 4^i - 3 \times 2^i + 1\}$$

最坏复杂度是 $O(n^{\frac{4}{3}})$, 平均是 $O(n^{\frac{7}{6}})$ 。是实践中最好的序列。