数据结构

Author: 经12-计18 张诗颖 2021011056

1 绪论

1. 工具：尺规计算机

2. 算法

要求：正确性、确定性、可行性、有穷性
好算法：正确、健壮、可读、效率
$Algorithms + DataStructures = Programs$
$(Algorithms + DataStructures) \times Efficiency = Computation$
算法分析：正确、成本 $T_A(P)$（算法A求解问题实例P的计算成本）

正确性证明：Loop Invariant, Convergence, Correctness

[EG/有穷性]：Hailstone序列（尚未证明或证否）

```
while (1 < n) {
    n % 2 ? n = 3 * n + 1 : n / 2;
    length++;
}
```

3. 计算模型

1. 图灵机 (Turing Machine)



- Tape：依次均匀地划分为单元格，各存有某一字符（初始为`#`），理想假设无限长
- Head：总是对准某一单元格，可读写；经过一个节拍可转向左右邻格
- Alphabet：字符的种类有限
- State：图灵机总是处于有限种状态中的某一种，约定`'h' = halt`

Transition Function：`(q, c; d, L/R, p)`（当前状态q，当前字符c，改写d，状态p）
[EG]：`(<, 1; 0, L, <)`
计算成本：从启动至停机所经历的节拍数目/Head累计的移动次数



2. RAM (Random Access Machine)
   - 寄存器顺序编号，总数没有限制
   - 可通过编号直接访问任意寄存器 call-by-rank
   - 每一基本操作（10个）仅需常数时间

```
R[i] <= c           R[i] <= R[R[j]]         R[i] <= R[j] + R[k]
R[i] <= R[j]        R[R[i]] <= R[j]         R[i] <= R[j] - R[k]
IF R[i] = 0 GOTO #          IF R[i] > 0 GOTO #
GOTO #                      STOP
```

计算成本：算法需要执行的基本操作次数

[EG]：Ceiling Division（除法上整）



1.1 渐进复杂度：大O记号

> Paul Bachmann, 1894: **T(n) = O(f(n))** iff $\exists c > 0$ s.t. $T(n) < c \cdot f(n)$ $\forall n >> 2$
> **T(n) = Ω(f(n))** iff $\exists c > 0$ s.t. $T(n) > c \cdot f(n)$ $\forall n >> 2$
> **T(n) = Θ(f(n))** iff $\exists c_1 > c_2 > 0$ s.t. $c_1 \cdot f(n) > T(n) > c_2 \cdot f(n)$ $\forall n >> 2$

O(1)：constant

O(log^{c}n)：poly-log，复杂度无限接近于常数（$\forall c > 0, logn = O(n^c)$），**[EP]**：逆Ackermann函数

O(logn)：对数，复杂度无限接近于常数，**[EP]**：二分查找、堆、词典查询、插入与删除

O(n^c)：polynomial，其中所有**O(n)**类函数称为**线性 (Linear function)**的（**[EP]**：数、图的遍历），复杂度已可令人满意

[EP/线性]：**O(nlog^{c}n)**：某些MST算法；

O(nloglogn)：某些三角剖分算法；

O(logn)：排序、EU\Huffman编码

[EP/平方]：**O(n^2)**：Dijkstra算法

[EP/立方]：**O(n^3)**：矩阵乘法

以上均为P问题（存在多项式算法的问题）

O(2^n)：exponential，通常属于有效算法到无效算法的分水岭，计算成本不可忍受

[EG]：2-Subset问题——**NP-complete**（不存在可在多项式时间内解答的算法）

> NP = Non-deterministic Polynomial

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	

1. 级数复杂度

- **算术级数**：与末项平方同阶
- **幂方级数**：比幂次高出一阶
- **几何级数**：与末项同阶

❖ **算术级数：与末项平方同阶** $T(n) = 1 + 2 + \ldots + n = \binom{n+1}{2} = \frac{n(n+1)}{2} = \mathcal{O}(n^2)$

❖ **幂方级数：比幂次高出一阶** $\sum_{k=0}^{n} k^d \approx \int_0^n x^d dx = \frac{x^{d+1}}{d+1}\Big|_0^n = \frac{n^{d+1}}{d+1} = \mathcal{O}(n^{d+1})$

$$T_2(n) = \sum_{k=1}^{n} k^2 = 1^2 + 2^2 + 3^2 + \ldots + n^2 = n(n+1)(2n+1)/6 = \mathcal{O}(n^3)$$

$$T_3(n) = \sum_{k=1}^{n} k^3 = 1^3 + 2^3 + 3^3 + \ldots + n^3 = n^2(n+1)^2/4 = \mathcal{O}(n^4)$$

$$T_4(n) = \sum_{k=1}^{n} k^4 = 1^4 + 2^4 + 3^4 + \ldots + n^4 = n(n+1)(2n+1)(3n^2+3n-1)/30 = \mathcal{O}(n^5)$$

❖ **几何级数：与末项同阶**

$$T_a(n) = \sum_{k=0}^{n} a^k = a^0 + a^1 + a^2 + a^3 + \ldots + a^n = \frac{a^{n+1}-1}{a-1} = \mathcal{O}(a^n), \quad 1 < a$$

$$T_2(n) = \sum_{k=0}^{n} 2^k = 1 + 2 + 4 + 8 + \ldots + 2^n = 2^{n+1} - 1 = \mathcal{O}(2^{n+1}) = \mathcal{O}(2^n)$$

- **几何分布**（收敛级数）：O(1)

> $(1-\lambda)\cdot[1+2\lambda+3\lambda^2+4\lambda^3+\ldots] = \frac{1}{1-\lambda} = O(1),\ 0<\lambda<1$

- **不收敛级数**

调和级数 $(\sum \frac{1}{k})$: $\Theta(logn)$
对数级数 $(\sum lnk)$: $\Theta(nlogn)$
$\sum k \cdot logk$: $O(n^2logn)$
$\sum k \cdot 2^k$: $O(n \cdot 2^n)$

❖ **调和级数：** $h(n) = \sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ ... \ + \frac{1}{n} = \ln n + \gamma + \mathcal{O}(\frac{1}{2n}) = \Theta(\log n)$

❖ **对数级数：** $\sum_{k=1}^{n} \ln k = \ln \prod_{k=1}^{n} k = \ln n! \approx (n + 0.5) \cdot \ln n - n = \Theta(n \cdot \log n)$

❖ **对数 + 线性 + 指数：** $\sum_{k=1}^{n} k \cdot \log k \approx \int_{1}^{n} x \ln x dx = \frac{x^2 \cdot (2 \cdot \ln x - 1)}{4} \Big|_1^n = \mathcal{O}(n^2 \log n)$

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

$$= n \cdot 2^{n+1} - \sum_{k=1}^{n} 2^k = n \cdot 2^{n+1} - (2^{n+1} - 2) = (n-1) \cdot 2^{n+1} + 2 = \mathcal{O}(n \cdot 2^n)$$

2. 复杂度分析

1. 常见的二层循环

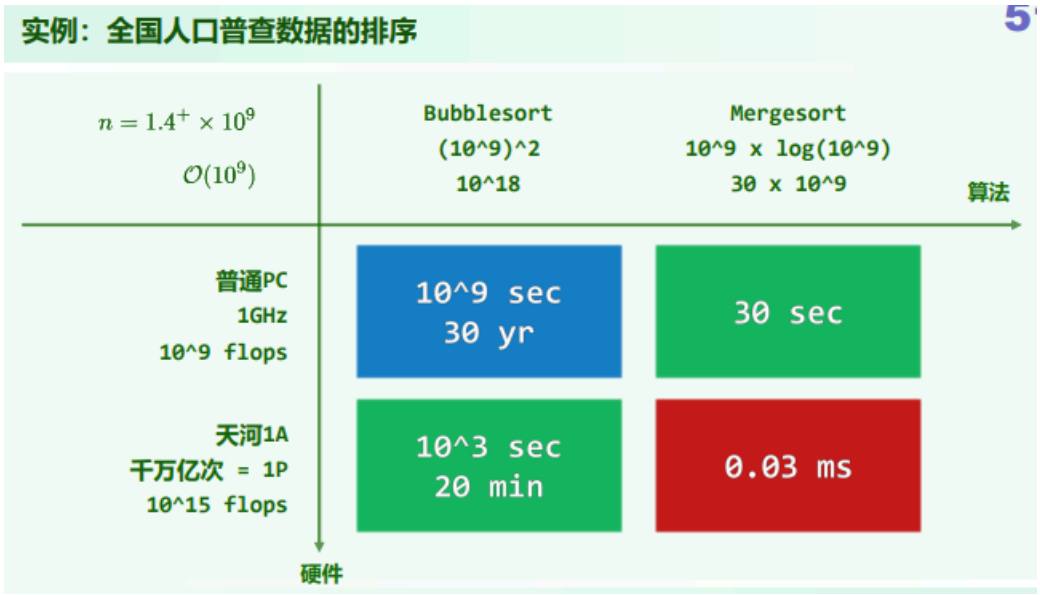
```
# O(n^2)
for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j) // 把n改成i，++j改成j+=const => 复杂度不变

# O(n)
for (int i = 1; i < n; i <<= 1)
    for (int j = 0; j < i; ++j)

# 更复杂的例子：O(nlogn)
for (int i = 0; i <= n; ++i)
    for (int j = 1; j < i; j += j)
# T(n)≈ln(x)在0到n上的积分
```

2. 封底估算：Back-Of-The-Envelope Calculation

> 1 day ≈ 10^5 sec
>
> 1生 ≈ 1 世纪 ≈ 3 × 10^9 sec
>
> 三生三世 ≈ 300yr ≈ 10^{10} sec
>
> 宇宙大爆炸至今 ≈ 4 × 10^{17} sec
>
> 普通PC：1GHz，10^9 flops
>
> 天河1A：1P，10^{15} flops



任何算法其时间复杂度一般是空间复杂度的上界（所以一般不首先考虑空间复杂度）

1.2 迭代与递归

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，他们各自所需时间之和为整体运行时间
递推方程：**[EG/Sum]** `T(n) = T(n - 1) + O(1)`，`base: T(0) = O(1)` => 求解

1. 减而治之

Decrease-and-conquer，将原问题划分为两个子问题，一个**平凡**，一个**规模缩减**
递归的潜在问题：空间复杂度爆炸

[EG/Reverse]:

递归版:

```
if (low < high) {
    swap(A[low], A[high]);
    reverse(A, low + 1, high - 1);
}
```

迭代版:

```
while (low < high)
    swap(A[low++], A[high--])
```

2. 分而治之

Divide-and-conquer，将原问题划分为若干个子问题，且子问题规模**大体相当**

[EG/BinaryRecursion]:

```
sum(int A[], int lo, int hi) { //[lo, hi)
    if (hi - lo < 2) return A[lo];
    int mi = (hi + lo) >> 1;
```

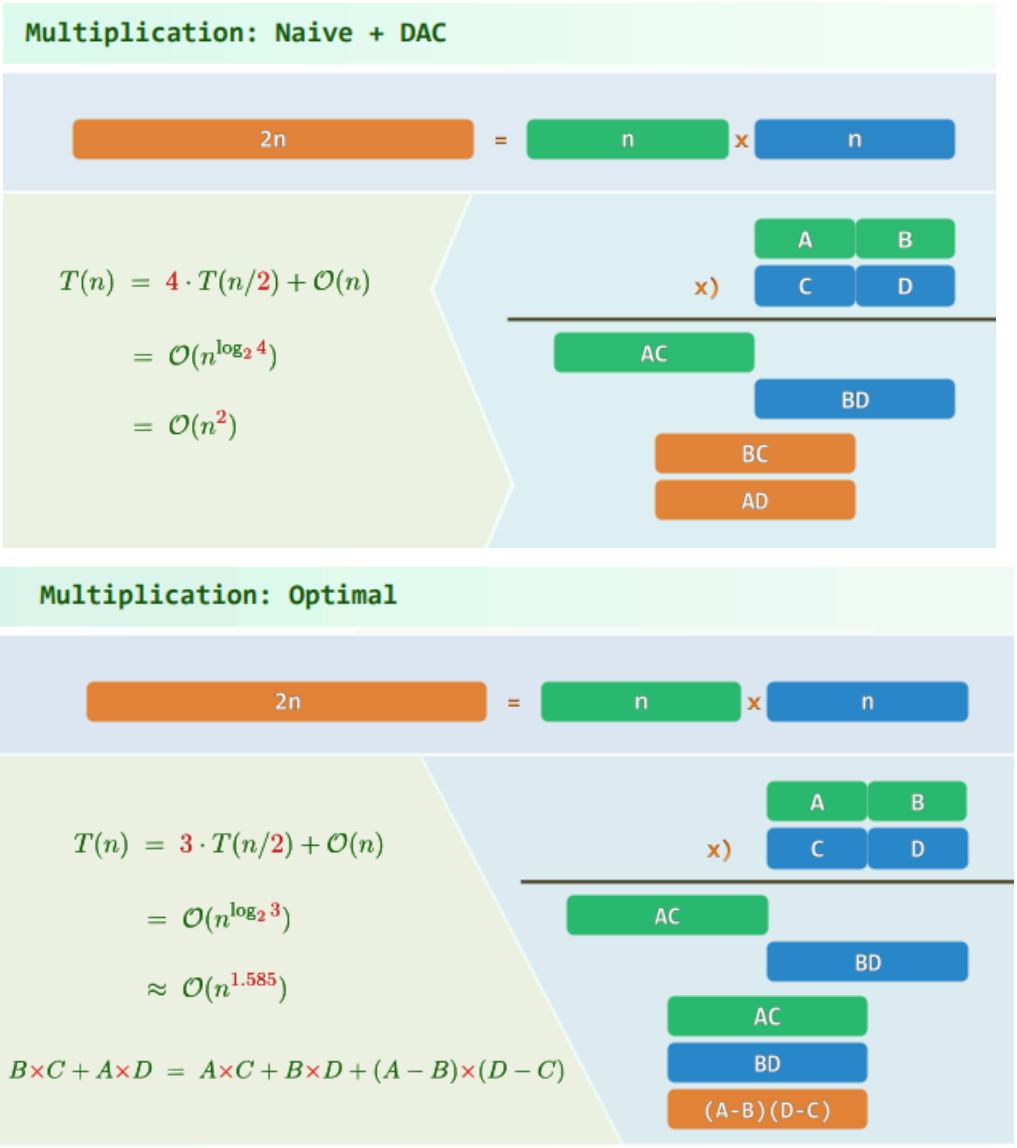
```
    return sum(A, lo, mi) + sum(A, mi, hi)
}
```

3. Master Theorem

分治策略对应的递推式通常形如：`T(n) = a·T(n/b) + O(f(n))`
[解释]：原问题被划分为**a**个规模均为**n/b**的子任务；任务的划分，解的合并耗时**f(n)**

- 若$f(n) = O(n^{log_b a - \epsilon})$，则$T(n) = \Theta(n^{log_b a})$
  [EP/kd-search]：`T(n) = 2·T(n/4) + O(1) = O(n^(1/2))`
- 若$f(n) = \Theta(n^{log_b a} \cdot log^k n)$，则$T(n) = \Theta(n^{log_b a} \cdot log^{k+1} n)$
  [EP/binary-search]：`T(n) = 1·T(n/2) + O(1) = O(logn)`
  [EP/mergesort]：`T(n) = 2·T(n/2) + O(n) = O(nlogn)`
  [EP/STL-mergesort]：`T(n) = 2·T(n/2) + O(nlogn) = O(nlog^2n)`
- 若$f(n) = \Omega(n^{log_b a + \epsilon})$，则$T(n) = \Theta(f(n))$
  [EP/quickSelect]：`T(n) = 1·T(n/2) + O(n) = O(n)`

4. EP：Multiplication



5. EP：总和最大区段

问题描述：从整数序列中，找出总和最大的区段（有多个时，短者优先）

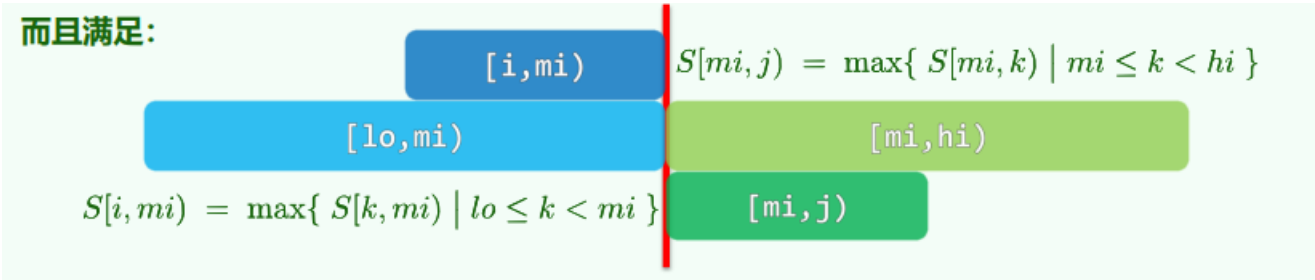
1. 蛮力算法：$O(n^3)$

```
int gs_BF(int A[], int n) { //Brute Force
    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;
}
```

2. 递增策略：$O(n^2)$

```
int gs_IC(int A[], int n) { //Incremental Strategy
    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;
}
```

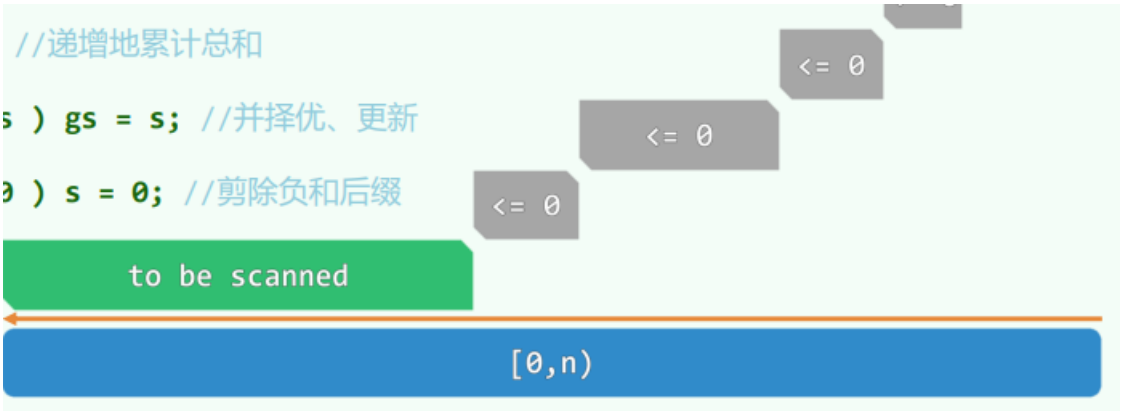
3. 分治策略：O(nlogn)




```
int gs_DC(int A[], int lo, int hi) { //Divide-and-Conquer
    if (hi - lo < 2) return A[lo]; //递归基
    int mi = (lo + hi) / 2;
    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;
    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)));
}
```

4. 减治策略：O(n)

   构思：考察**最短的**总和非正的后缀A[k, hi)，以及总和最大的区段GS(lo, hi) = A[i, j)，可以发现后者要么是前者的（真）**后缀**，要么与前者**无交**。




```
int gs_LS(int A[], int n) { //Linear Scan
    int gs = A[0], s = 0, i = n;
    while (0 < i--) { //在当前区间内
        s += A[i];
        if (gs < s) gs = s; //择优，更新
        if (s <= 0) s = 0; //剪除负和后缀
    }
    return gs;
}
```

1.3 动态规划

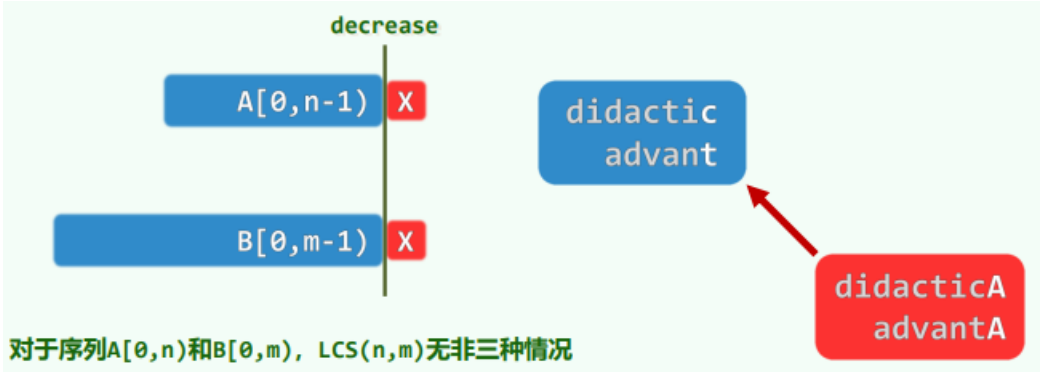
动态规划：Dynamic Programming，一种颠倒计算方法，自底而上迭代的方式。

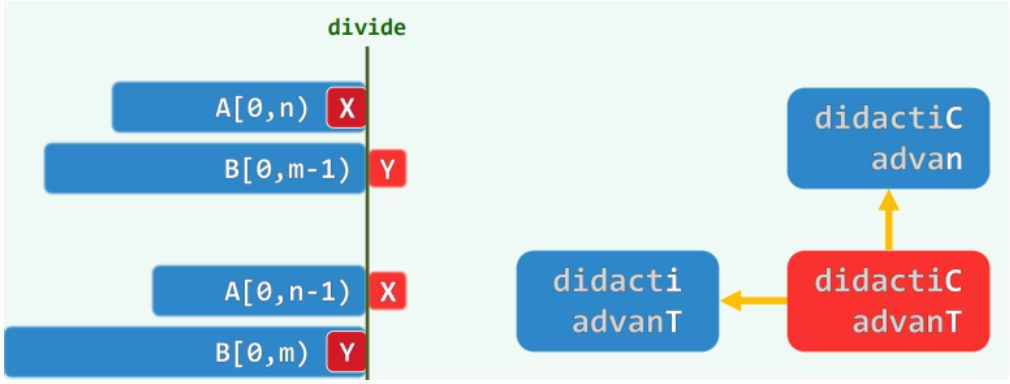
[EG/fibonacci]：T(n) = O(n)

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

1. EP：最长公共子序列

1. 减而治之——递归：复杂度$T(n) = O(2^n)$（当$n = m$时）



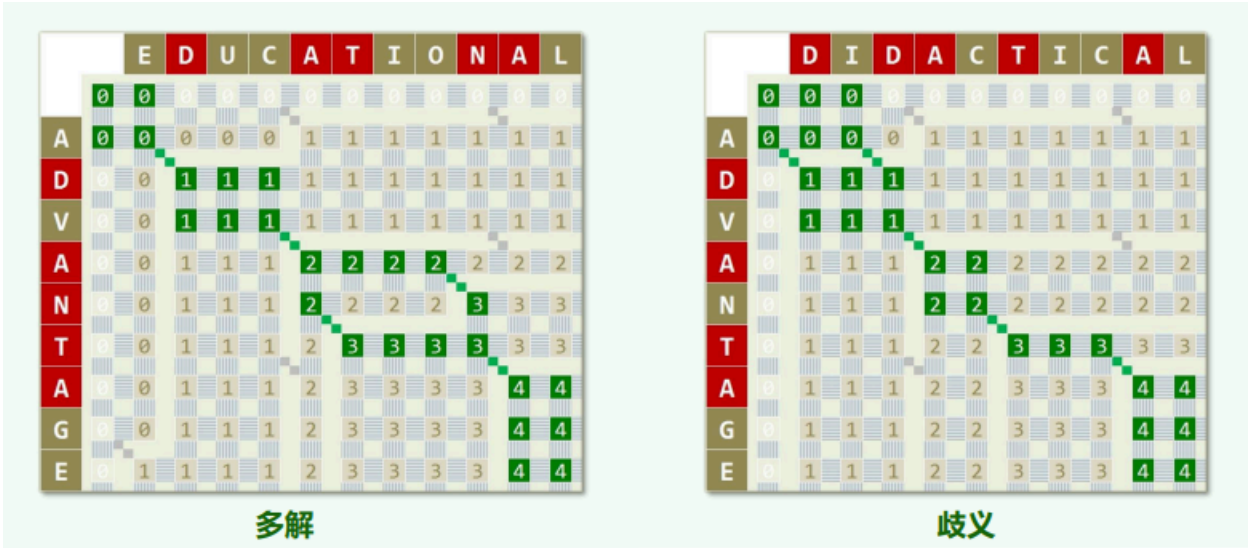


递归基：`n = 0`或`m = 0`取作空序列`""`（长度为零）

若`A[n - 1] = 'X' = B[m - 1]`，则取作：`LCS(n - 1, m - 2) + 'X'`

若`A[n - 1] ≠ B[m - 1]`，则在`LCS(n, m - 1)`与`LCS(n - 1, m)`中取更长者

2. 动态规划——递推：复杂度$T(n) = O(n \cdot m)$



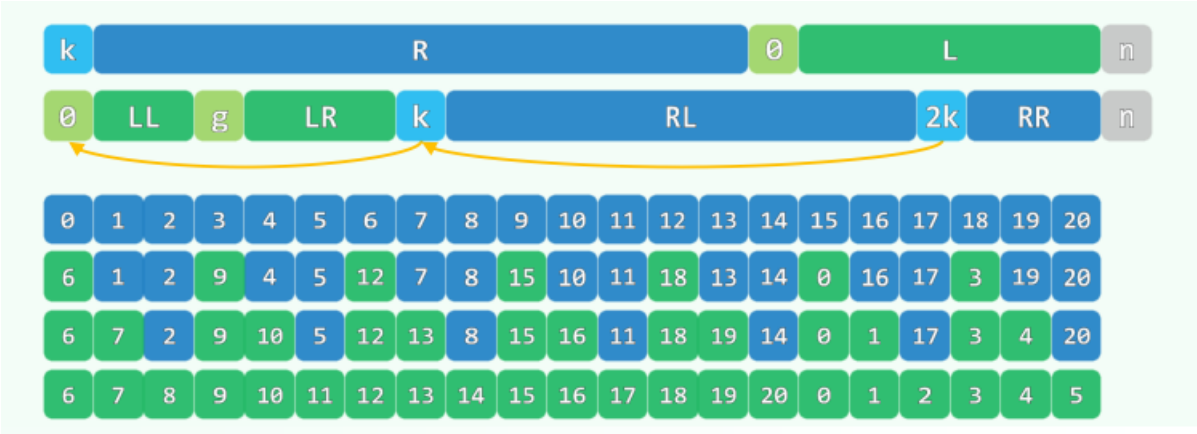

```
unsigned int lcs(char const * A, int n, char const * B, int m) {
    if (n < m) { //make sure m <= n
        swap(A, B);
        swap(n, m);
    }
    unsigned int* lcs1 = new unsigned int[m + 1]; //the current two rows are
    unsigned int* lcs2 = new unsigned int[m + 1]; //buffered alternatively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

2. EP：就地循环移位

问题描述：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个单位

1. 蛮力算法：`while (k--) shift(A, n, 0, 1);` 反复以1为间距循环左移，$T(n) = O(n * k)$
2. 迭代版：按照同余类依次左移，时间复杂度$T(n) = O(2n)$




```
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);
}
```

3. 倒置版：`reverse(A, k)`，`reverse(A + k, n - k)`，`reverse(A, n)`，时间复杂度$T(n) = O(3n)$，但是由于计算机Cache的存在，在实际实现时会更快

```
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)
```

```
}
```

2 向量

循秩访问 (Call-By-Rank)

抽象数据类型 (Abstract Data Type)：数据模型 + 定义在该模型上的一组操作
数据结构 (Data Structure)：基于某种特定语言，实现ADT的一整套算法
Application = **Interface** × Implementation
具体实现 (Implementation) + 实际应用 (Interface/Application)

2.1 基本语法

数组：元素由编号唯一指代并可以直接访问，称作**线性数组** (Linear Array)，访问方式称为**寻秩访问** (call-by-rank)，其中元素下标称为元素的**秩**（`using Rank = int;`）

向量：数组的抽象与泛化，由一组元素按线性次序封装而成；元素类型可以灵活选取

1. Vector ADT

`vec.begin()`：0秩迭代器
`vec.end()`：末秩迭代器

1. `size()`：返回元素总数
2. `get(r)`：返回秩为r的元素
3. `put(r, e)`：用e替换秩为r元素的数值
4. `insert(r, e)`：e作为秩为r的元素插入，原先后继依次后移
5. `remove(r)`：删除秩为r的元素，返回该元素原值
6. `disordered()`：判断所有元素是否已经按非降序排列
7. `sort()`：调整各元素的位置，使之按非降序排列
8. `find(a)`：返回目标元素a的秩（如果不存在返回-1）
9. `search(e)`：返回不大于e且秩最大的元素**（有序向量）**
10. `deduplicate(), uniquify()`：剔除重复元素**（无序/有序向量）**
11. `traverse()`：遍历向量并统一处理所有元素，参数为函数对象或函数指针

2. Vector的构造与析构

(typename=T)

`vec._capacity`：容量
`define DEFAULT_CAPACITY X`：默认初始容量X
`vec._size`：元素总数
`vec._elem`：指向元素数组的指针

1. 构造函数
   默认构造函数：`Vector(int c = DEFAULT_CAPACITY) {_elem = new T[_capacity = c]; _size = 0;}`

数组区间复制：`Vector(T const * A, Rank lo, Rank hi) {copyFrom(A, lo, hi);}`
向量区间复制：`Vector(Vector<T> const & V, Rank lo, Rank hi) {copyFrom(V._elem, lo, hi);}`
向量整体复制：`Vector(Vector<T> const & V) {copyFrom(V._elem, 0, V._size);}`
copyFrom()：`_elem = new T[_capacity = max(DEFAULT_CAPACITY, 2*(hi - lo))];`

2. 析构函数
`~Vector{delete [] _elem;}`

3. 动态空间管理

上溢 (overflow)：`_elem[]`不足以存放所有元素
下溢 (underflow)：`_elem[]`中的元素寥寥无几
装填因子 (load factor)：$\lambda = \frac{_size}{_capacity} << 50\%$

--扩容算法：`expand()`

```
template <typename T>
void Vector<T>::expand() {
    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) {_elem[i] = oldElem[i];} //复制
    delete [] oldElem;
} //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
```

--算法分析

平均分析 (average complexity)：根据各种操作出现概率的分布，计算成本加权平均
各种可能的操作独立考察，割裂了操作之间的相关性和连贯性，往往不能准确评判dsa
分摊分析 (amortized complexity)：连续实施足够多次操作，所需总体成本摊还至单次操作
忠实刻画可能出现的操作序列，更为精准地评判dsa

1. 容量递增策略：时间成本$O(n^2)$，每次操作的分摊成本$O(n)$，装填因子$\lambda \approx 100\%$
2. 容量加倍策略：时间成本$O(n)$，每次操作的分摊成本$O(1)$，装填因子$\lambda > 50\%$

4. 无序向量：基本操作的实现

1. 元素访问：重载`[]`运算符

```
template <typename T>
T & Vector<T>::operator[](Rank r) {
    return _elem[r];
}

template <typename T>
const T & Vector<T>::operator[](Rank r) const { // 仅限于右值
     return _elem[r];
}
```

2. 插入元素：`insert(r, e)`

```
template <typename T>
```

```
Rank Vector<T>::insert(Rank r, T const & e) {
    expand(); //如有必要，扩容
    for (Rank i = _size; i > r; i--)
        _elem[i] = _elem[i - 1];
    _elem[r] = e; _size++; return r;
}
```

3. 区间删除：`remove(lo, hi)`

```
template <typename T>
int Vector<T>::remove(Rank lo, Rank hi) { // [lo, hi)
    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++];
    _size = lo;
    shrink(); //若有必要，缩容
    return hi - lo; //返回被删除元素的数目
}
```

单元素删除：`remove(Rank r)`

```
template <typename T>
T Vector<T>::remove(Rank r) {
    T e = _elem[r];
    remove(r, r + 1);
    return e; //返回被删除元素
}
```

4. 判等器

```
template <typename K, typename V>
struct Entry {
    K key; V value; //键值
    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) {}; //拷贝
    bool operator==(Entry<K,V> const& e) {return key == e.key;}
    bool operator!=(Entry<K,V> const& e) {return key != e.key;}
}
```

有序向量：比较器（较判等器增加`<`，`>`）

5. 顺序查找：`find(e, lo, hi)`

```
template <typename T>
Rank Vector<T>::find(T const & e, Rank lo, Rank hi) const {
    while ((lo < hi--) && (e != _elem[hi])); //逆向查找
    return hi; //返回值＜lo即意味着失败；否则返回满足相等的最大秩
}
```

输入敏感 (input-sensitive)：最好$O(1)$，最差$O(n)$

6. 去重：`deduplicate()`

```
template <typename T>
Rank Vector<T>::deduplicate() {
    Rank oldSize = _size;
    for (Rank i = 1; i < _size; ) {
        if (find(_elem[i], 0, i) < 0) i++;
        else remove(i);
    }
    return oldSize - _size; //返回删除的元素个数
} //O(n^2)
```

7. 遍历：`traverse()`

```
//方法一：函数指针--只读或局部性修改
template <typename T>
void Vector<T>::traverse(void (*visit)(T&)) {
    for (Rank i = 0; i < _size; ++i)
        visit(_elem[i]);
}
//方法二：函数对象--全局性修改更便捷
template <typename T> template <typename VST>
void Vector<T>::traverse(VST & visit) {
    for (Rank i = 0; i < _size; ++i)
        visit(_elem[i]);
}
```

[EP/Increase]：

```
template <typename T>
struct Increase {
    virtual void operator()(T & e) {
        e++;
    }
}

template <typename T>
void increase(Vector<T> & V) {
    V.traverse(Increase<T>();)
}
```

5. 有序向量：基本操作的实现

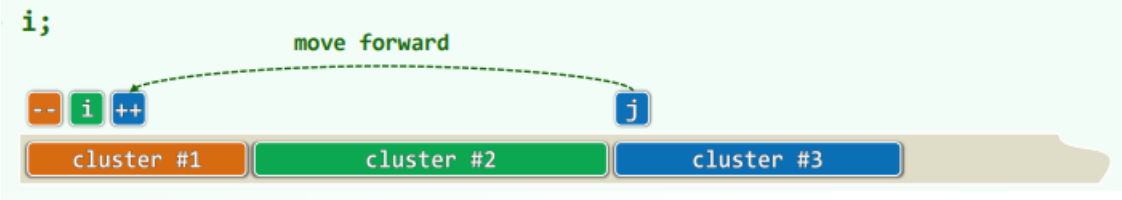
用**相邻逆序对**的数目可以度量向量的**紊乱**程度

判断是否有序：

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

```
}
```

1. 去重：`uniquify()`，复杂度$O(n)$




```
template <typename T>
int Vector<T>::uniquify() {
    Rank i = 0, j = 0;
    while (++j < _size) {
        if (_elem[i] != _elem[j])
            _elem[++i] = _elem[j];
    }
    _size = ++i;
    shrink();
    return j - i;
}
```

2.2 查找（有序向量）

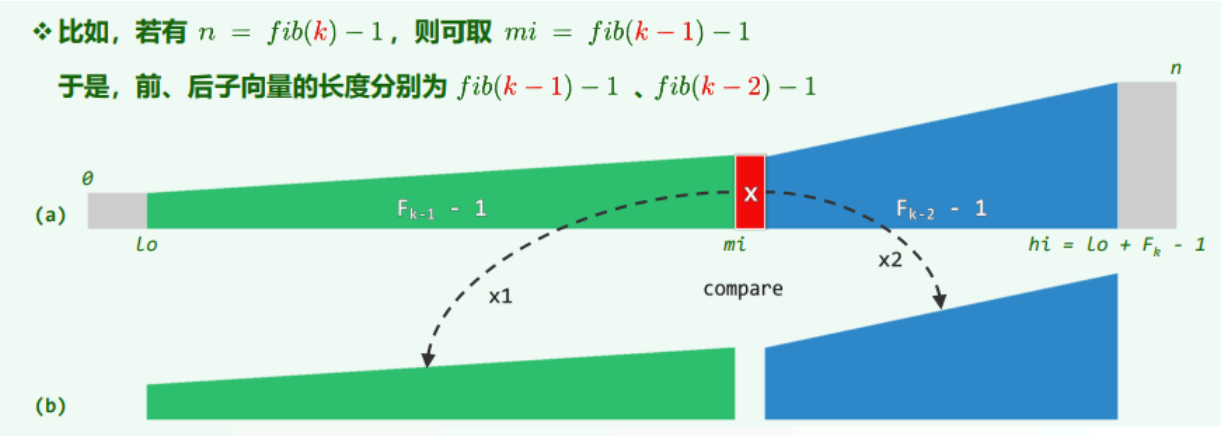
更精细地评估查找算法的性能：**查找长度**（search length，关键码的比较次数）

1. 二分查找（版本A）

1. binary_search
   平均查找长度 = $O(1.50 \cdot logn)$：左右分治复杂度≈1:2

```
template <typename T>
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;
        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 + 1, hi)
        else return mi; //命中
    }
    return -1; //查找失败
}
```

2. Fibonacci_search
   平均查找长度 = $O(1.44 \cdot logn)$




```
template <typename T>
static Rank fibSearch(T* S, T const & e, Rank lo, Rank hi) {
    for (Fib fib(hi - lo); lo < hi; ) { //Fib数列制表备查
        while (hi - lo < fib.get()) fib.prev(); //自后向前顺序查找轴点
        Rank mi = lo + fib.get() - 1; //确定形如Fib(k) - 1的轴点
        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; //查找失败
}
```

通用策略：在任何区间`[0, n)`内，总是选取$\lambda \cdot n$作为轴点，$0 \leq \lambda \leq 1$。二分查找对应于$\lambda = 0.5$，Fibonacci查找对应于$\lambda = \phi = 0.6180339...$
这类算法的渐近复杂度为$\alpha(\lambda) \cdot log_2 n = O(logn)$

> 递推式：$\alpha(\lambda) \cdot log_2 n = \lambda \cdot [1 + \alpha(\lambda) \cdot log_2(\lambda n)] + (1 - \lambda) \cdot [2 + \alpha(\lambda) \cdot log_2((1 - \lambda)n)]$
> 整理后：$\frac{-ln2}{\alpha(\lambda)} = \frac{\lambda \cdot ln\lambda + (1-\lambda) \cdot ln(1-\lambda)}{2-\lambda}$
> 当$\lambda = \phi = \frac{\sqrt{5}-1}{2}$时，有$\alpha(\lambda) = 1.440420...$最小

2. 二分查找（版本B）

相对于版本A，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) {
        Rank mi = (lo + hi) >> 1;
        e < S[mi] ? hi = mi; lo = mi; //区间分为[lo, mi)和[mi, hi)
    }
    return e == S[lo] ? lo : -1;
}
```

3. 二分查找（版本C）

较版本A和版本B优化了返回信息

返回信息：`search(e)`应该返回不大于e的最后一个元素，使得语句`V.insert(1 + V.search(e), e)`成立

```
template <typename T>
static Rank binSearch(T* S, T const & e, Rank lo, Rank hi) {
    while (lo < hi) {
        Rank mi = (lo + hi) >> 1;
        e < S[mi] ? hi = mi : lo = mi + 1; //[lo, mi)或[mi + 1, hi)
    } //出口时，必有S[lo = hi]
    return lo - 1;
}
```

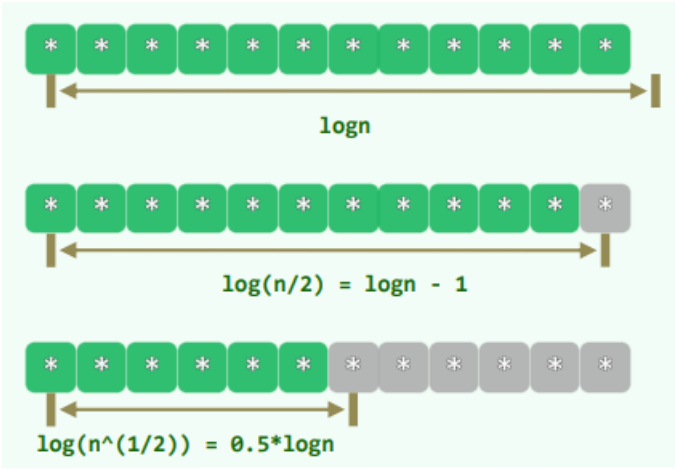
证明：① 不变性：`A[0, lo) ≤ e < A[hi, n)`，
在算法执行过程中的任意时刻，`A[lo - 1]`总是（截至目前已确认的）≤ e的最大者，`A[hi]`总是（截至目前已确认的）> e的最小者；算法结束时`A[lo - 1] = A[hi - 1]`是全局 ≤ e的最大者。
② 规模缩减

4. 插值查找

假设：已知有序向量中各元素随机分布的规律，如独立且均匀的随机分布
则`[lo, hi)`内各元素应大致呈线性趋势增长：$\frac{mi-lo}{hi-lo} \approx \frac{e-A[lo]}{A[hi]-A[lo]}$
因此，通过猜测轴点mi，可以极大地提高收敛速度：$mi \approx lo + (hi - lo) \cdot \frac{e-A[lo]}{A[hi]-A[lo]}$

算法分析：
最坏情况：$O(n)$
平均情况：$O(loglogn)$（每经过一次比较，待查区间宽度由n缩减为$\sqrt{n}$，有效字长$logn$减半）

> 插值查找：在字长意义上的**折半查找**
> 二分查找：在字长意义上的**顺序查找**



综合评价：
从$O(logn)$到$O(loglogn)$，优势并不明显
须引入乘法、除法运算
易受小扰动的干扰和畸形分布的“蒙骗”

实际可行的方法：首先通过插值查找将查找范围缩小，然后再进行二分查找进一步缩小范围，最后在10^2数据范围采用顺序查找

2.3 排序

排序器：统一入口

```
template <typename T>
void Vector<T>::sort(Rank lo, Rank hi)
```

- 起泡排序：bubbleSort：$O(n^2)$
- 归并排序：mergeSort：$O(n \cdot logn)$
- 选择排序：selectionSort：$O(n^2)$
- 插入排序：insertionSort：$O(n^2)$
- 堆排序：heapSort
- 快速排序：quickSort
- 希尔排序：shellSort

1. 起泡排序bubbleSort

复杂度：$O(n^2)$

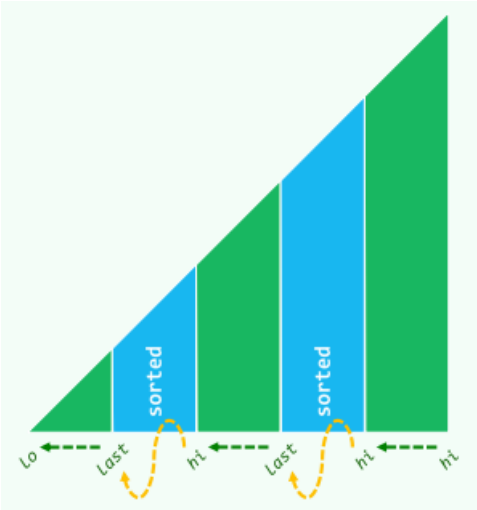
基本版：

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

提前终止版：

```
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; //仍未完全有序
            }
}
```

跳跃版：




```
template <typename T>
void Vector<T>::bubbleSort(Rank lo, Rank hi) {
    while (lo < (hi = bubble(lo, hi)); )
}
template <typename T>
Rank Vector<T>::bubble(Rank lo, Rank hi) {
    Rank last = lo;
    while (++lo < hi)
        if (_elem[lo - 1] > _elem[lo]) {
            last = lo; //逆序对只可能残留与[lo, last)
            swap(_elem[lo - 1], _elem[lo]);
        }
    return last; //返回最右侧的逆序对位置
}

// 2022 latest version
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[last = i]);
}
// A[lo, last) <= A[last, hi)
// A[last - 1] < A[last, hi)
```

算法评价

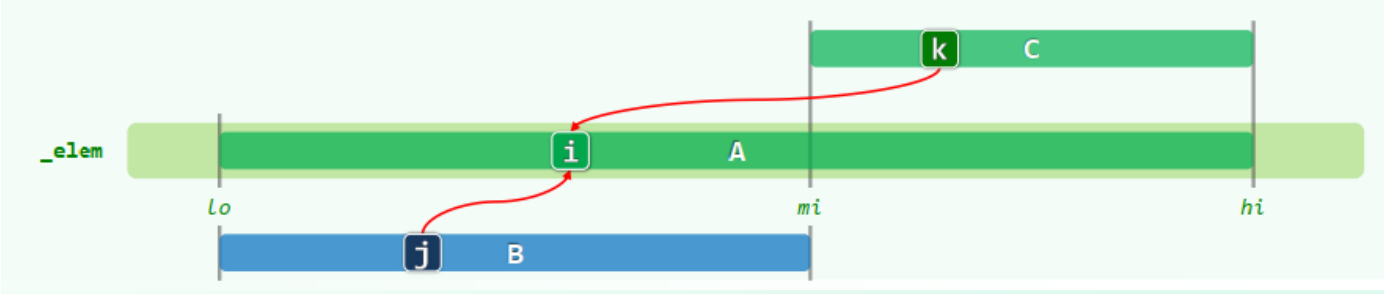
- 时间效率：最好$O(n)$，最坏$O(n^2)$
- 重复元素的**稳定性**：满足（起泡排序中唯有相邻元素才可以交换）

重复元素在输入、输出序列中的相对次序，是否保持不变?

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

2. 归并排序mergeSort

分而治之的思想，by J. von Neumann (1945)
复杂度：$O(n \cdot logn)$




```
template <typename T>
void Vector<T>::mergeSort(Rank lo, Rank hi) { //[lo, hi)
    if (hi - lo < 2) return;
    int mi = (lo + hi) >> 1;
    mergeSort(lo, mi);
    mergeSort(mi, hi);
    merge(lo, mi, hi);
}

template <typename T>
void Vector<T>::merge(Rank lo, Rank mi, Rank hi) {
    Rank i = 0; T* A = _elem + lo; //A = _elem[lo, hi), 就地
    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]; //复制自A的前缀
    Rank k = 0, lc = hi - mi; T* C = _elem + mi; //C[0, lc) = _elem[mi, hi), 就地

    while ((j < lb) && (k < lc)) //反复的比较B,C的首元素
        A[i++] = (B[j] <= C[k]) ? B[j++] : C[k++];
    while (j < lb) //若C先耗尽
        A[i++] = B[j++];
    delete [] B;
}
```

算法评价

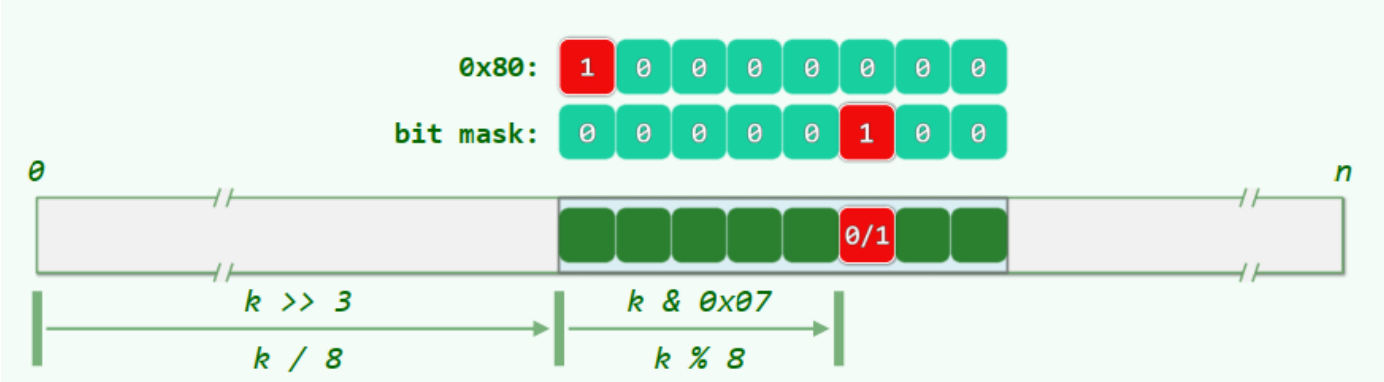
- 时间效率：$O(nlogn)$（递推公式：$T(n) = 2T(\frac{n}{2}) + O(n)$）
- **优点**
  只要实现恰当，可以保证**稳定性**
  可拓展性极佳，适宜于**外部**排序（海量网页搜索结果的归并）
  易于**并行**化

- **缺点**
  非**就地**，需要对等规模的辅助空间
  即便输入接近有序，仍需要$\Omega(nlogn)$时间

2.4 位图：数据结构

有限整数集：实现三个主要功能`bool test(int k);`，`void set(int k);`，`void clear(int k)`

1. 位图dsa实现




```
class Bitmap {
private:
    int N;
    unsigned char* M;
public:
    Bitmap(int n = 8) {
        M = new unsigned char[N = (n + 7) / 8];
        memset(M, 0, N);
    }
    ~Bitmap() {delete [] M; M = NULL;}
    void set(int k); //将k放入集合
    void clear(int k); //集合中去除k
    void bool test(int k); //返回集合中是否存在k
}

bool Bitmap::test(int k) {
    return M[k >> 3] & (0x80 >> (k & 0x07)); // k & 0x07 == k % 8 (bitmask)
}
void Bitmap::set(int k) {
    expand(k);
    M[k >> 3] |= (0x80 >> (k & 0x07));
}
void Bitmap::clear(int k) {
    expand(k);
    M[k >> 3] &= ~(0x80 >> (k & 0x07));
}
```

应用：小集合+大数据

`int A[n]`的元素均取自`[0, m)`，已知数据规模不大但重复度极高，如何去重？

Idea1. 先排序，在扫描——$T(n)=O(nlogn+n)$，$S(n)=O(4n)$

考虑$2^{24}=m<<n=10^{10}, S(n)=40GB$（24位无符号整数），内存消耗不可忍受

Idea2. 运用Bitmap存储所有数据是否存在的状态

总体运行时间 = $O(n+m)=O(n), S(n)=O(m)$

就上例而言，空间复杂度降至$m/8=2^{21}=2MB<<40GB$

```
Bitmap B(m); //S(m)
for (int i = 0; i < n; ++i) B.set(A[i]); //O(n), 将第A[i]为置为1
for (int i = 0; i < m; ++i)
    if (B.test(i)) ... //O(m), 判断A[i]是否存在
```

2. 筛法

```
void Eratosthenes(int n, char* file) {
    Bitmap B(n);
    B.set(0); B.set(1);
    for (int i = 2; i < n; ++i) {
        if (!B.test(i)) //说明i是素数
            for (int j = i * i; j < n; j += i) //可以改成i * i起点
                B.set(j);
    }
    B.dump(file);
}
```

内循环每趟迭代$O(n/i)$步，由**素数定理**外循环至多n/lnn趟，累计耗时$O(n\cdot logn)$

$$
\begin{aligned}
& n/2 + n/3 + n/5 + n/7 + n/11 + \ldots \\
< \; & n/2 + n/3 + n/4 + n/5 + n/6 + \ldots + n/(n/\ln n) \\
= \; & \mathcal{O}(n\cdot(\ln(n/\ln n)-1)) = \mathcal{O}(n\cdot\ln n - n\cdot\ln(\ln(n))) = \mathcal{O}(n\cdot\log n)
\end{aligned}
$$

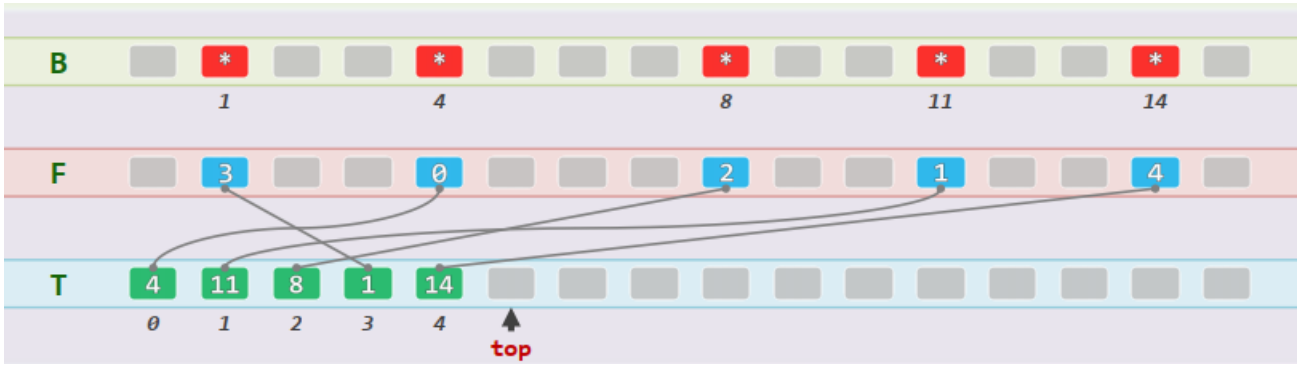
3. 快速初始化：校验环

结构：校验环（J. Hopcroft, 1974），将`B[]`拆分成一对等长的Rank型向量，有效位均满足：`T[F[k]] = k, F[T[k]] = k`，操作复杂度为$O(1)$

```
Rank F[m]; //From
Rank T[m]; Rank top = 0; //To及其栈顶指示

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

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



3 列表

循位置访问（Call-By-Position）

- **静态 vs 动态** 数据结构
  静态：仅读取，数据结构的内容及组成一般不变（`get`，`search`）
  数据空间整体创建或销毁
  数据元素的**物理存储次序**与其**逻辑次序**严格一致，可支持高效的静态操作
  动态：需写入，数据结构的局部或整体将改变（`put`，`insert`，`remove`）
  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**：采用动态储存策略的典型结构
  节点 (node)：列表中的元素，通过指针或引用彼此链接；在逻辑上构成一个线性序列
  前驱 (predecessor) / 后继 (successor)：相邻节点；没有前驱/后继的节点称作**首** (first/front) / **末** (last/rear) 节点

3.1 ListNode ADT

```
template <typename T> using ListNodePosi = ListNode<T>*;
template <typename T> struct ListNode {
    T data; //数值
    ListNodePosi<T> pred; //前驱
    ListNodePosi<T> succ; //后继
    ListNode() {}
    ListNode(T e, ListNodePosi<T> p = NULL, ListNodePosi<T> s = NULL)
        : data(e), pred(p), succ(s) {}
    ListNodePosi<T> insertAsPred(T const & e); //前插入
```

```
    ListNodePosi<T> insertAsSucc(T const & e); //后插入
};
```

1. `pred()`：当前节点前驱节点的位置
2. `succ()`：当前节点后继节点的位置
3. `data()`：当前节点所存数据对象
4. `insertAsPred(e)`：插入前驱节点，存入被引用对象e，返回新节点位置

```
template <typename T>
ListNodePosi<T> ListNode<T>::insertAsPred(T const & e) {
    ListNodePosi<T> x = new ListNode(e, pred, this);
    pred->succ = x; pred = x;
    return x;
}
```

5. `insertAsSucc(e)`：插入后继节点的，存入被引用对象e，返回新节点位置

3.2 List ADT

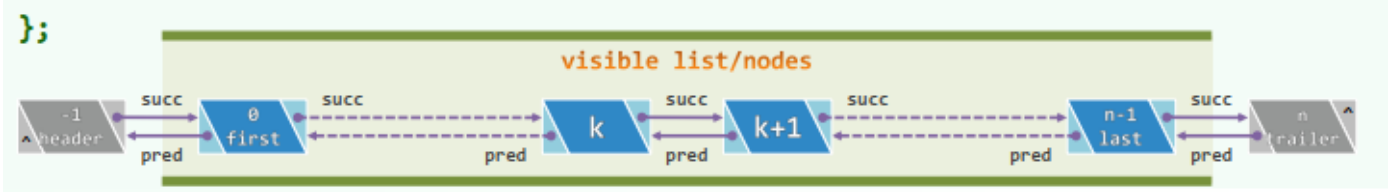
约定：头、首、末、尾节点的秩，分别理解为-1, 0, n-1, n

```
#include "listNode.h"
template <typename T> class List {
private:
    int _size;
    ListNodePosi<T> header, trailer; //哨兵

protected:
    void init() {
        header = new ListNode<T>;
        trailer = new ListNode<T>;
        header->succ = trailer; header->pred = NULL;
        trailer->pred = header; trailer->succ = NULL;
        _size = 0;
    }
    void copyNodes(ListNodePosi<T> p, int n) {
        init();
        while (n--) { //将起自p的n项依次作为末节点
            insertAsLast(p->data);
            p = p->succ;
        }
    }
    int clear() {
        int oldSize = _size;
        while (0 < _size) remove(header->succ);
        return oldSize;
    }
```

```
public:
    List(List<T> const & L) {
        copyNodes(L.first(), L._size);
    }
    ~List() {
        clear();
        delete header; delete trailer;
    }

    T operator[](Rank r) const { // assert: 0 <= r < size, T(r) = O(r)
        ListNodePosi<T> p = first(); //从首结点出发
        while (0 < r--) p = p->succ;
        return p->data;
    } //秩 == 前驱的总数
}
```



1. `size()`：报告列表当前的规模（节点总数）
2. `first(), last()`：返回首、末节点的位置
3. `insertAsFirst(e), insertAsLast(e)`：将e当作首、末节点插入
4. `insert(p, e), insert(e, p)`：将e当作节点p的直接**后继**、**前驱**插入

```
template <typename T>
ListNodePosi<T> List<T>::insert(T const & e, ListNodePosi<T> p) {
    _size++;
    return p->insertAsPred(e);
}
```

5. `remove(p)`：删除位置p处的节点，返回其中数据项

```
template <typename T>
T List<T>::remove(ListNodePosi<T> p) {
    T e = p->data; //备份
    p->pred->succ = p->succ;
    p->succ->pred = p->pred;
    delete p;
    _size--;
    return e; //返回备份的数值
}
```

6. `disordered()`：判断所有节点是否已按非降序排列
7. `sort()`：调整各节点的位置，使之按非降序排列

8. `find(e)`：查找目标元素e，失败时返回NULL

```
template <typename T> // assert: 0 <= n <= rank(p) < _size
ListNodePosi<T> List<T>::find(T const & e, int n, ListNodePosi<T> p) const {
    while (0 < n--) // 自后向前
        if (e == (p = p->pred) -> data)
            return p; // 在p的n个前驱中，返回等于e的最靠后者
    return NULL;
}

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

9. `search(e)`【有序列表】：查找e，返回不大于e且秩最大的节点

```
template <typename T>
ListNodePosi<T> List<T>::search(T const & e, int n, ListNodePosi<T> p) const {
    do {
        p = p->pred;
        n--;
    } while ((-1 < n) && (e < p->data));
    return p; // 失败时，返回区间左边界的前驱（可能是header）
} // 最好O(1),最坏O(n),平均O(n)；无法通过有序性提高查找效率
```

10. `deduplicate(), uniquify()`：针对列表/有序列表，踢除重复节点

```
//deduplicate(): O(n^2)
template <typename T>
int List<T>::deduplicate() {
    int oldSize = _size;
    ListNodePosi<T> p = first();
    for (Rank r = 0; p != trailer; p = p->succ)
        if (ListNodePosi<T> q = find(p->data, r, p))
            remove(q);
        else r++;
    return oldSize - _size;
}

//uniquify(): O(n)
template <typename T>
int List<T>::uniquify() {
    if (_size < 2) return 0;
    int oldSize = _size;
    ListNodePosi<T> p = first();
    ListNodePosi<T> q; // 各区段起点及其直接后继
    while (trailer != (q = p->succ)) // 反复考察紧邻的节点对(p, q)
        if (p->data != q->data) p = q;
        else remove(q);
```

```
    return oldSize - _size;
}
```

11. `traverse()`：遍历列表

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

3.3 排序

- 起泡排序：bubbleSort：$O(n^2)$
- 归并排序：mergeSort：$O(n \cdot logn)$
- 选择排序：selectionSort：$O(n^2)$
- 插入排序：insertionSort：$O(n^2)$
- 堆排序：heapSort
- 快速排序：quickSort
- 希尔排序：shellSort

1. 选择排序selectionSort

```
template <typename T>
void List<T>::selectionSort(ListNodePosi<T> p, int n) {
    ListNodePosi<T> head = p->pred, tail = p;
    for (int i = 0; i < n; ++i) tail = tail->succ; //待排序区间为(head, tail)
    while (1 < n) { // 反复从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 );
        tail = tail->pred; n--;
    }
}

template<typename T> // 起始于p的n各元素中选出最大者
ListNodePosi<T> List<T>::selectMax(ListNodePosi<T> p, int n) {
    ListNodePosi<T> max = p;
    for (ListNodePosi<T> cur = p; 1 < n; n--)
        if (! lt( (cur = cur->succ)->data, max->data)) // data >= max
            max = cur;
    return max;
}
// 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）
// 采用比较器!lt()或ge()，从而等效于后者优先
```

<u>性能分析</u>：

- 共迭代n次，在第k次迭代中
  `selectMax()`：$O(n-k)$

`swap()`：$O(1)$
故总体复杂度为$O(n^2)$

- 尽管如此，元素的**移动**操作远远少于起泡排序（选择排序的$O(n^2)$主要来自于元素的**比较**操作）
- 改进方向：利用高级的数据结构，`selectMax()`可改进至$O(logn)$，这样排序算法的复杂度可以降为$O(n \cdot logn)$

2. 循环节/Cycle

Definition：

> 任何一个序列$A[0,n)$，都可以分解为若干个循环节
> 任何一个序列$A[0,n)$，都对应于一个有序序列$S[0,n)$
> 元素$A[k]$在S中对应的秩，记作$r(A[k]) = r(k) \in [0,n)$
> 元素$A[k]$所属的循环节是：$A[k], A[r(k)], A[r(r(k))], ... A[r(...(r(k)))] = A[k]$
> 每个循环节，长度均不超过n
> 循环节之间，互不相交

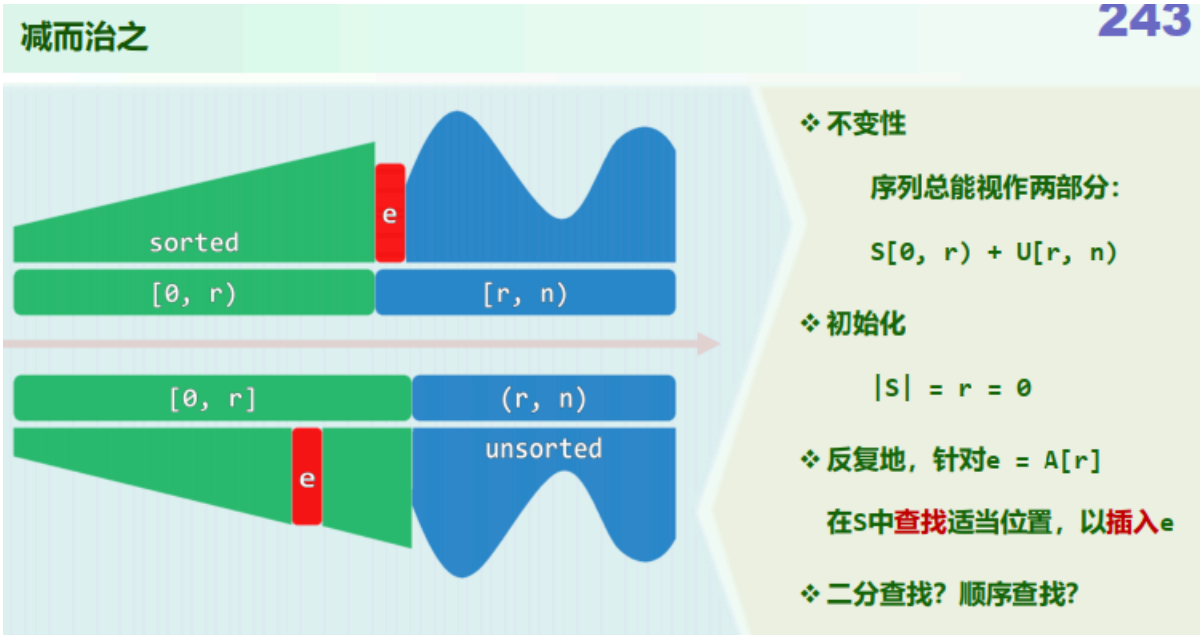
```
rank: 0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15
A[]: J  N  P  M  A  I  G  O  D  C  H  B  K  L  F  E
S[]: A  B  C  D  E  F  G  H  I  J  K  L  M  N  O  P
r[]: 9  13 15 12 0  8  6  14 3  2  7  1  10 11 5  4

   J  .  P  .  A  .  .  .  .  C  .  .  .  .  .  E
   .  N  .  .  .  .  .  .  .  .  .  B  .  L  .  .
   .  .  .  M  .  I  .  O  D  .  H  .  K  .  F  .
   .  .  .  .  .  .  G  .  .  .  .  .  .  .  .  .
```

Analysis：

1. 采用交换法，每迭代（`swap()`）一步，M都会脱离原属的循环节，自成一个循环节
   M原属的循环节，长度**恰好**减少一个单位，其余循环节保持不变
2. “M已经就位，无需交换”——这样的情况会出现c次，其中c表示最初有的循环节个数
   c：最大值为n，期望为$\Theta(logn)$

3. 插入排序insertionSort




```
template <typename T>
void List<T>::insertionSort(ListNodePosi<T> p, int n) {
    for (int r = 0; r < n; ++r) {
        insert( search(p->data, r, p), p->data );
        p = p->succ;
        remove(p->pred);
    } // n次迭代，每次O(r + 1)
} // 仅使用O(1)的辅助空间，属于就地算法
```

性能分析：

> 若各元素的取值系独立均匀分布，平均要做多少次元素比较?
> 考察$e = [r]$刚插入完成的那一时刻，此时的有序前缀$[0, r]$中，谁是e?
> 观察：其中的$r + 1$个元素都有可能，且概率均为$\frac{1}{r+1}$
> 因此，在刚完成的这次迭代中，为引入$S[r]$所花费时间的数学期望为：$1 + \sum_{k=0}^{r} \frac{k}{r+1} = 1 + \frac{r}{2}$
> 于是，总体时间的数学期望为 $\sum_{r=0}^{n-1}(1 + \frac{r}{2}) = O(n^2)$
>
> **Case Sensitive**
> best case = $O(n)$
> worst case = $O(n^2)$

4. 归并排序mergeSort

时间复杂度：$O(n \cdot logn)$

```
template <typename T>
void List<T>::mergeSort(ListNodePosi<T> & p, int n) {
    if (n < 2) return;
    ListNodePosi<T> q = p;
    int m  = n >> 1; //以中点为界
    for (int i = 0; i < m; ++i)
        q = q->succ;
    mergeSort(p, m);
```

```
    mergeSort(q, n - m); //子序列分别排序
    p = merge(p, m, *this, q, n - m); //归并
}

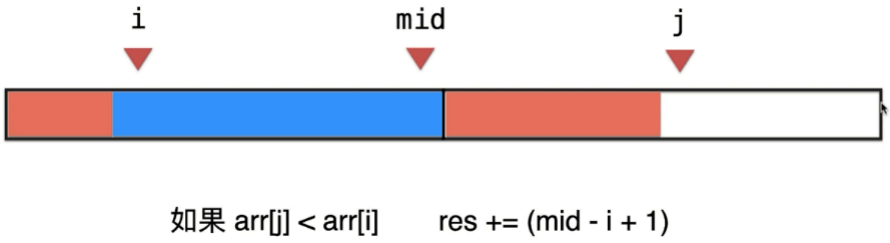
template <typename T>
ListNodePosi<T> List<T>::
    merge(ListNodePosi<T> p, int n, List<T> & L, ListNodePosi<T> q, int m) {
    ListNodePosi<T> pp = p->pred; //归并之后p或不再指向首节点，故需先记忆，以便返回前更新
    while ( (0 < m) && (q != p) ) //小者优先归入
        if ( (0 < n) && (p->data <= q->data) ) {
            p = p->succ;
            n--;
        }
    else {
        insert( L.remove( (q = q->succ)->pred ), p );
        m--;
    }
    return pp->succ;
} // 时间复杂度：O(n + m)，线性正比于节点总数
```

5. 逆序对/Inversion

Definition: $I(j) = \{0 \leq i < j | A[i] > A[j]\}$ （将逆序对统一记到**后者**的账上）

逆序对总数：$I = \sum_j \leq C_n^2 = O(n^2)$

- 对于BubbleSort来说，**交换**操作的次数恰好等于输入序列所含**逆序对**的总数
- 对于InsertionSort来说，针对$e = A[r]$的那一步迭代恰好需要做$I(r)$次**比较**
  若共含I个逆序对，则关键码比较次数为$O(I)$，运行时间为$O(n + I)$
- 任意给定一个序列，**统计**其中逆序对的总数：
  采用**归并排序**，仅需$O(n \cdot logn)$时间

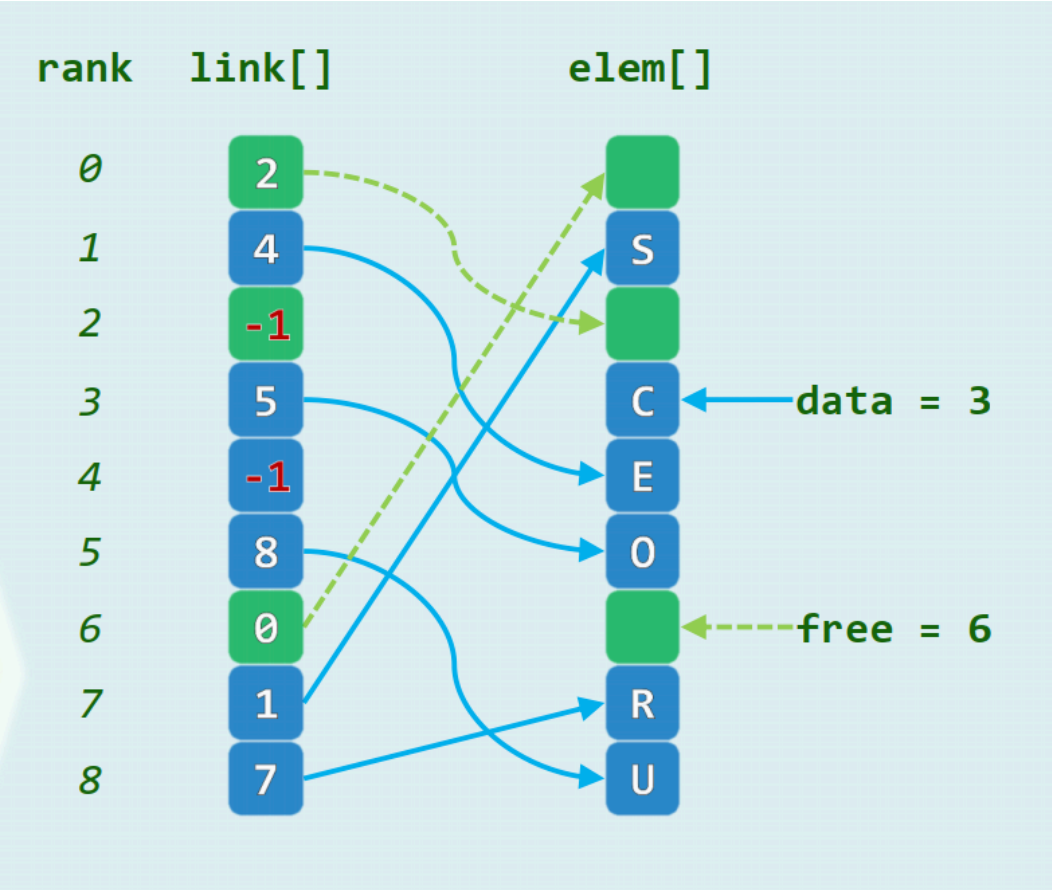


3.4 游标实现

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**

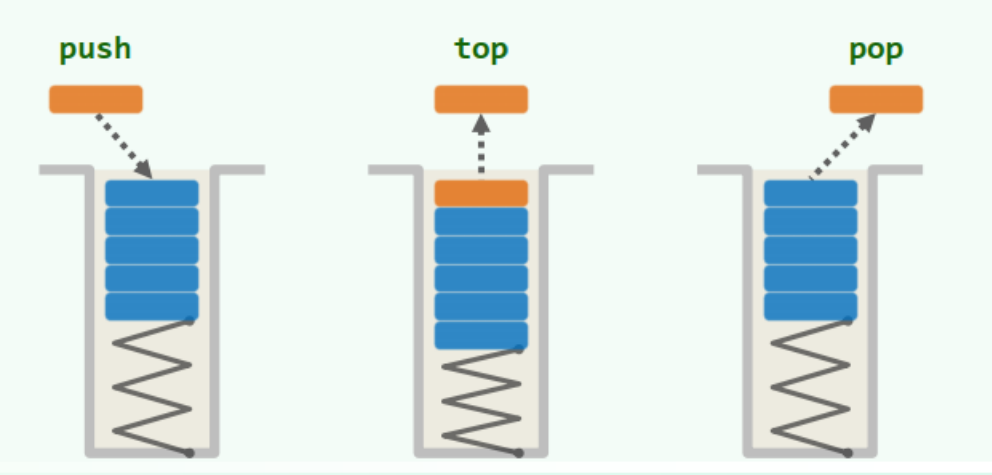
维护逻辑上**互补的**列表`data`和`free`
`insert()`操作的时间复杂度为$O(1)$，`remove()`操作的时间复杂度为$O(n)$
游标的数组大小是固定的、不可更改的



3.5 Java和Python的列表实现

4 栈与队列

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）



4.1 栈及其基本运用

基本接口

`size()` / `empty()`：判断大小 / 判断是否为空
`push()`：入栈
`pop()`：出栈
`top()`：查顶

扩展接口：`getMax()`……

1. 属于序列的特例，可以直接基于向量或列表派生

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

2. 也可以通过继承`List`来实现
3. 如此实现的栈的各接口，均只需$O(1)$时间

递归



空间复杂度：递归算法所需的空间，主要取决于递归的**深度**，而非递归实例总数

1. 消除递归：尾递归

定义：在递归实例中，作为**最后**一步的递归调用

> 是最简单的递归模式
> 一旦抵达递归基，便会引发**一连串**的return（且返回地址相同），调用栈相应地**连续**pop
> 编译器一般可以**自动识别**并代为改写为迭代形式
> 改用迭代形式：时间复杂度有**常系数**改进，空间复杂度或有**渐近**改进

以阶乘为例：

```
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
```

2. 进制转换

实现难点：位数m不确定，如何使得空间不浪费，且**自低而高**得到数位、**自高而低**输出

```
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) {
        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
}

int 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
}
```

3. 括号匹配

算法思路：顺序扫描表达式，用栈记录已扫描的部分——反复迭代（遇到`(`则进栈，遇到`)`则出栈）

```
bool paren(const char exp[], int lo, int hi) { // exp[lo, hi)
    Stack<char> S;
    for (int i = lo; i < hi; ++i) {
        if ('(' == exp[i]) S.push(exp[i]);
        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（遇到右括号时栈已空，必不匹配
    }
    return S.empty();
}
```

实际上，若仅考虑一种括号，只需一个**计数器**足矣：`S.size()`

> 一旦转负，则为适配（右括号多余）；
> 最后不为零，则不匹配（左括号多余）；
> 最后归零，即为匹配

扩展：多类括号
只需约定“括号”的**通用**格式，而不必实现**固定**括号的类型与数量
[EX/html]：`<body> | </body>`，`<h1> | </h1>`，`<font> | </font>`，`<p> | </p>`...

4. 栈混洗 Stack Permutation

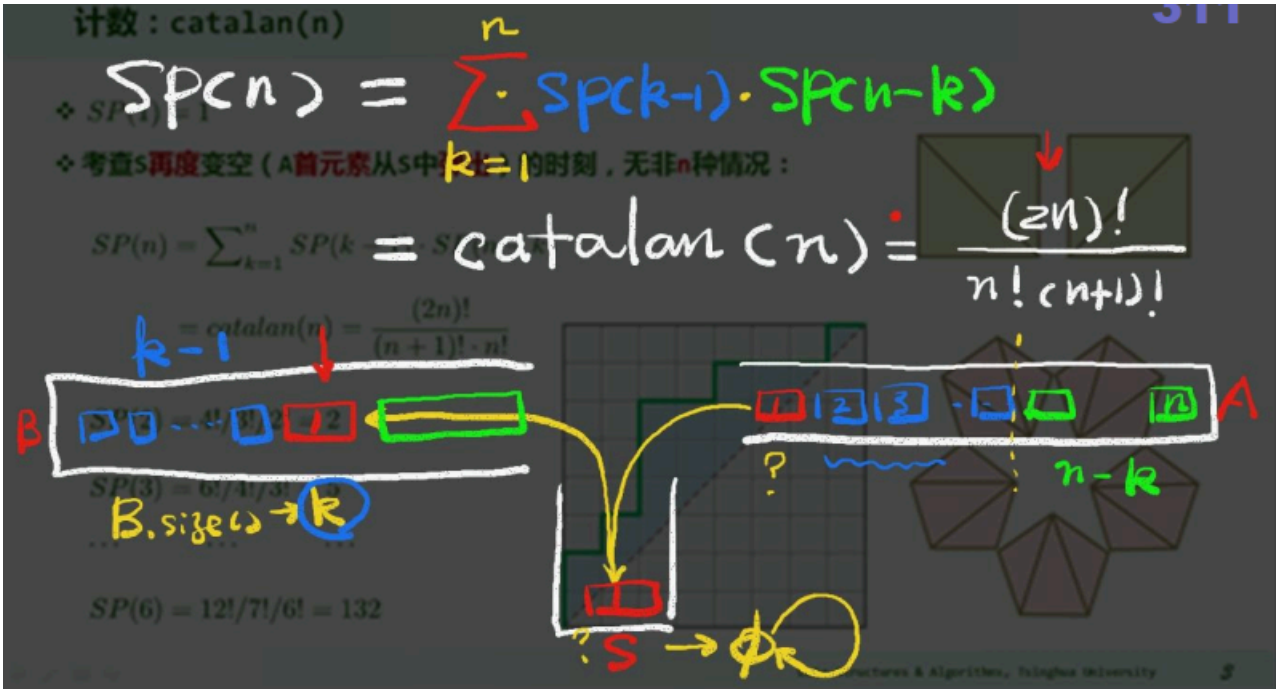
栈混洗：将栈A中的元素全部转入栈B中（`S.push(A.pop()); B.push(S.pop())`）

> **Notation**: $< a_1, a_2, a_3, \dots, a_n]$

- **计数：SP(n)**

  同一输入序列，可有多种栈混洗。一般地，对于长度为n的序列，混洗总数$SP(n) = ?$

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



- **甄别禁形**

  > **禁形**：对任何$1 \le i < j < k \le n, [\dots, k, \dots, i, \dots, j, \dots >$ **必非**栈混洗

  - **充要性**（Knuth, 1968）：A permutation is a stack permutation **iff** it does NOT involve the permutation 312
  - 如此，可得一个$O(n^3)$的甄别算法
  - 如此，可得一个$O(n^2)$的甄别算法：
    $[p_1, p_2, p_3, \dots, p_n >$是$< 1, 2, 3, \dots, n]$的栈混洗， **当且仅当**对于任意$i < j$，不含模式
    $[\dots, j+1, \dots, i, \dots, j, \dots >$
  - $O(n)$算法：直接借助栈A、B和S，模拟混洗过程，每次`S.pop()`之前，检测S是否已空；或需弹出的元素在S中，却非顶元素：

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

4.2 表达式求值

1. 中缀表达式求值

思路：自左向右扫描表达式，用栈记录已扫描的部分（含已执行运算的结果）
栈的**顶部**存在**可优先计算**的子表达式 `?` 该子表达式推展；计算其数值；计算结果进栈 `:` 当前字符进栈，转入下一字符

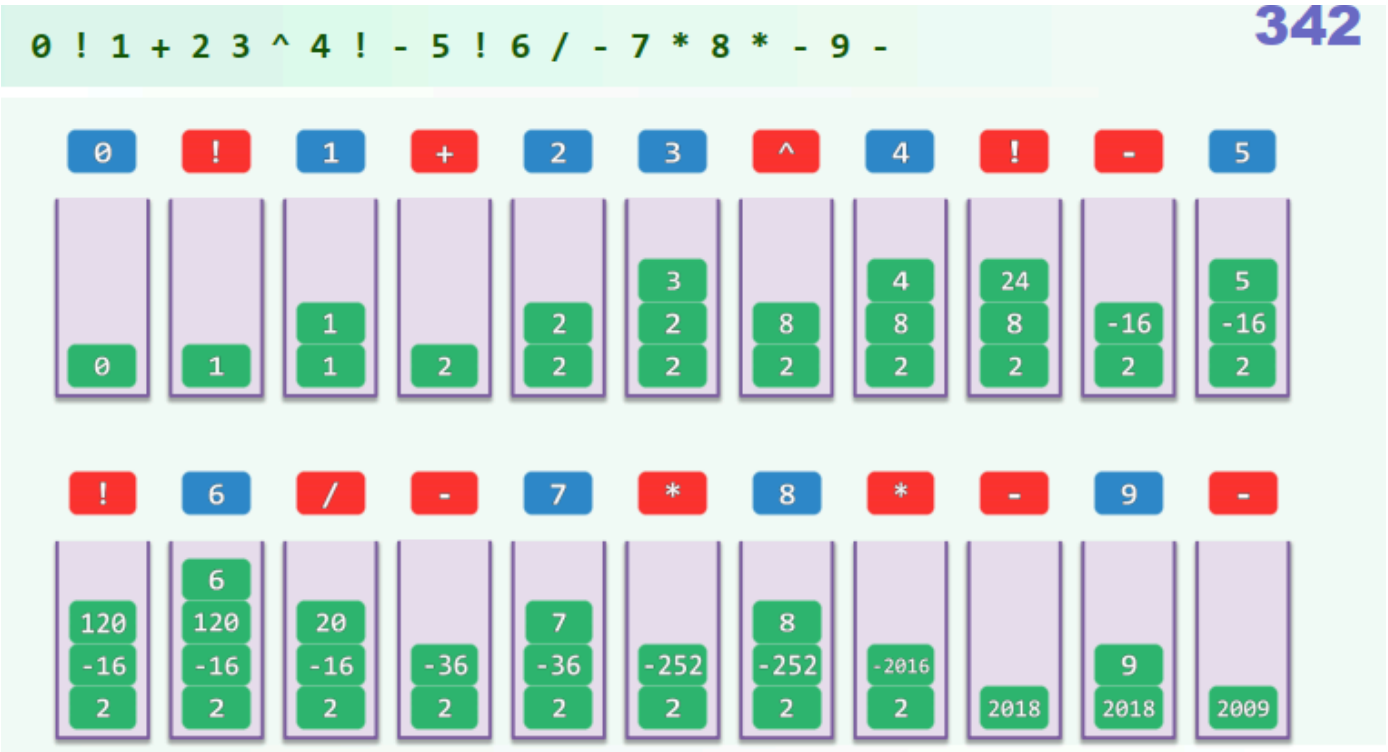
```
double evaluate(char* S, char* RPN) { // S保证语法正确
    Stack<double> opnd; // 运算数栈 Operand Stack
    Stack<char> optr; // 运算符栈 Operator
    optr.push('\0');
    while (!optr.empty()) {
        if (isdigit(*S)) // 若为操作数
            readNumber(S, opnd); // 读入
        else // 若为运算符，则视其与栈顶运算符之间优先级的高低
            switch(priority(optr.top(), *S)) { //见“优先级表”
                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 '=': // 优先级相等（当前运算符为右括号，或尾部哨兵'\0'）
                    optr.pop(); // 脱括号并接收下一个字符
                    S++;
                    break;
                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
                    }
                    break;
            }
    }
    return opnd.pop(); // 弹出并返回最后的计算结果
}
```

```
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 --------------|
```

2. 逆波兰表达式 Reverse Polish Notation / RPN

> 逆波兰表达式：J. Lukasiewicz (1878 ~ 1956)
> 由运算符 (operator) 和操作数 (operand) 组成的表达式中，**不需要括号** (parenthesis-free)，即可表示带优先级的运算关系
> 作为补偿，须额外引入一个起分割作用的**元字符**（比如空格）
>
> 亦称作**后缀式 (postfix)**
>
> 应用：PostScript (1985)，支持设备独立的图形描述
> （1 × 解释器 + 5 × 栈）× RPN语法

```
// pseudo code
引入栈S  //存放操作数
逐个处理下一个元素
    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  //无需估计优先级
返回栈顶
```




```
double evaluate(char* S, char* RPN) { // RPN转换
    while (!optr.empty()) {
        if (isdigit(*S)) {
            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 '>': {
                    char op = optr.pop();
                    append(RPN, op);
                }
                   /*...*/
            }
    }
}
```

4.3 队列

队列（queue）：受限的序列

> 只能在队尾插入 / 查询：`enqueue()`, `rear()`
> 只能在队头删除 / 查询：`dequeue()`, `front()`
>
> 先进先出 (FIFO): First in First out
> 后进后出 (LILO): Last in Last out
>
> 扩展接口：`getMax()`...

基本接口

`bool empty()`：判空
`void enqueue(T const& e)`：队尾插入
`T dequeue()`：队头删除
`T front()`：查询队头元素
`int size()`：查询队列长度

1. 属于序列的特例，可以直接基于向量或列表派生

```
template <typename T>
class Queue : public List<T> {
public:
    void enqueue(T const & e) { insertAsLast(e); } // 入队
    T dequeue() { return remove(first()); } // 出队
    T & front() { return first()->data; } //队首查询
}
```

2. 如此实现的队列接口，均只需$O(1)$时间

1. 资源循环分配

```
// RoundRobin
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); //接受服务，
    Q.enqueue(e); //然后重新入队
}
```

2. 银行服务模型

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

```
// 参数：nWin == 窗口/队列的数目，servTime == 营业时长
struct Customer { // 顾客类
    int window; //所处窗口（队列）
    unsigned int time; //服务市场
};

void simulate(int nWin, int servTime) {
    Queue<Customer>* windows = new Queue<Customer>[nWin];
    for (int now = 0; now < servTime; now++) {
        Customer c;
        c.time = 1 + rand() % 50;
        c.window = bestWindow(windows, nWin); // 找出最佳/最短服务窗口
        windows[c.window].enqueue(c); //新顾客加入对应的队列
        for (int i = 0; i < nWin; ++i) {
            if (!windows[i].empty()) {
```

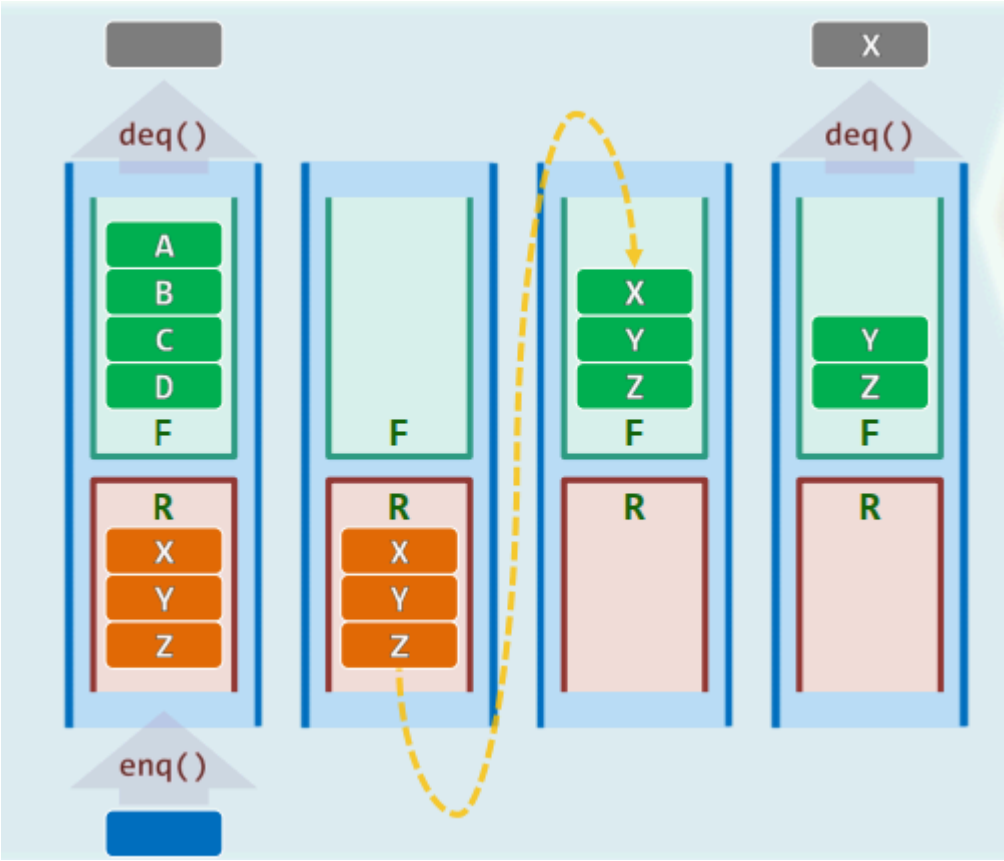
```
                if (--windows[i].front().time <= 0) // 队首顾客接受服务
                    windows[i].dequeue(); //服务完毕则出列，有后继顾客接替
            }
        }
    }
    delete [] windows; // 释放所有队列
}
```

3. 双栈当队

Queue = Stack × 2

```
def Q.enqueue(e):
    R.push(e);
def Q.dequeue(): // 0 < Q.size()
    if (F.empty())
        while (!R.empty())
            F.push(R.pop());
    return F.pop();
```

> Best / Worse case: $O(1)/O(n)$
>
> **Amortized Cost** (of any operation sequence involving **n** items) = $4n = O(n)$
> **Aggregate Cost** = $3e + d$ (e denotes `enqueue()` times, d denotes `dequeue()` times)
> the amortized cost for each OPERATION is $\frac{3e+d}{e+d} < 3$
> **Amortization by Potential**:
> Consider the k^{th} operation
> Define $\phi_k = |R_k| - |F_k|$
> Then $A_k = T_k + \phi_k - \phi_{k-1} \equiv 2$
> Hence $2n = T(n) + \phi_n - \phi_0 > T(n) - n$
> $T(n) < 3n = O(n)$

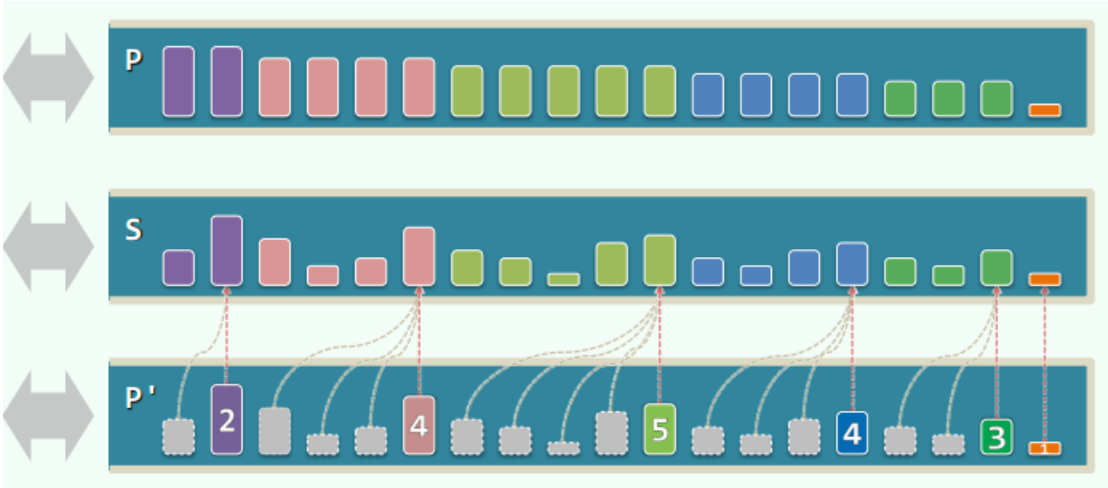


4. Steap + Queap

Steap = Stack + Heap

> 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)
```

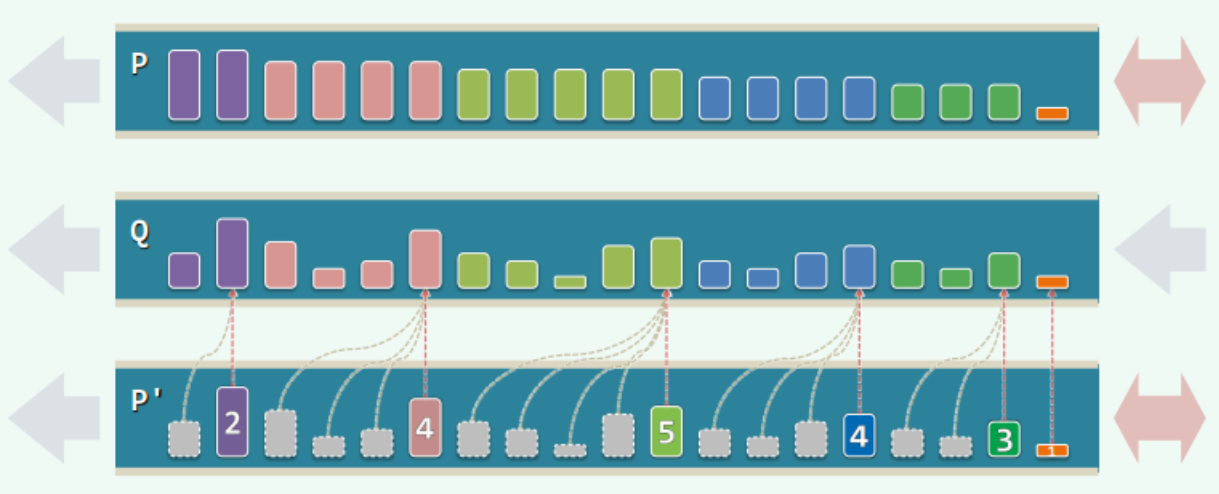


改进：构造p'队列，记录相同最大值的数量大小，从后往前扫描时只需累加并合并即可，分摊复杂度可以下降。

Queap = Queue + Heap

> P中每个元素，都是Q中对应后缀的最大者

```
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;
}
*Double Ending Queue (Deq)*
```

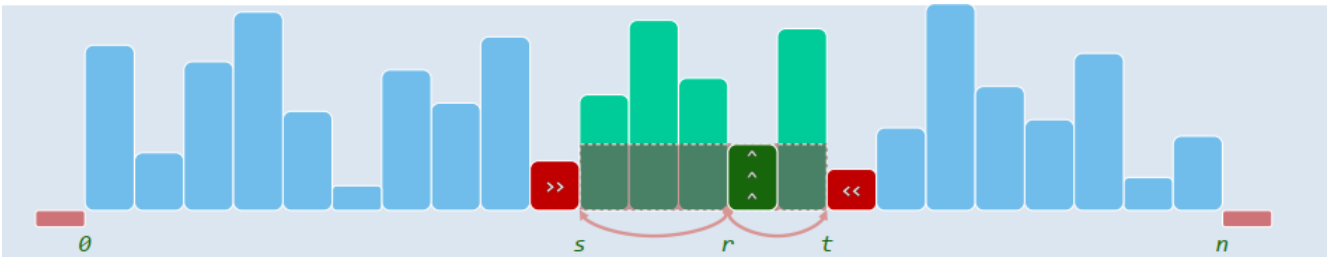


5. 直方图内最大矩形

> Maximum: 全局最大值
> Maximal: 局部极大值

- Maximal rectangle supported by H[r]: $maxRect(r) = H[r] \cdot (t(r) - s(r))$
  where $\begin{cases} s(r) = max\{k | 0 \le k < r \ and \ H[k-1] < H[r]\} \\ t(r) = min\{k | r < k \le n \ and \ H[r] > H[k]\} \end{cases}$

  > 依次扫描所有n块矩形，将其高度作为r，然后在确认唯一s和t计算得到每个r得到的最大矩形面积。从所有$A(r)$中选取最大值为全局最大矩形
  >
  > 复杂度：$O(n^2)$



- Using **Stack**

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

// Determine t(r) by another scan in the REVERSED direction
// By amortization, time complexity is O(n)
```

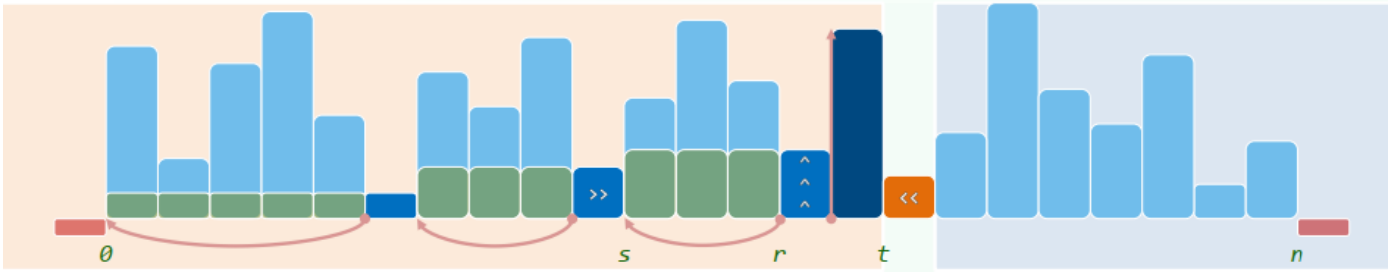
> 维护一个单调递增的栈（栈顶元素即为待进栈$s(r)$）
> 逆序再扫描一遍可以求出$t(r)$，从而可以求解面积$A(r)$

- Problem: $t(r)$ can't be calculated ON-LINE && space complexity is $O(n)$-too much!
  compute **BOTH** $s(r)$ and $t(r)$ by a **SINGLE** scan

  One-Pass Scan:

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

> 每次进栈可以确定$s(r)$，每次出栈可以确定$t(r)$，因而依次扫描就可以计算出$s(r)$、$t(r)$从而可以计算出面积$A(r)$



5 二叉树

一种半线性的数据结构（在确定某种次序之后，具有线性特征）

树

- 树是**极小连通图、极大无环图**
  $T=(V;E)$，其中节点数$n=|V|$，边数$e=|E|$
- 指定任一节点$r\in V$作为**根**后，T称作**有根树**
- 相对于T，T_i称作以r_i为根的**子树**（subtree rooted at r_i），记作$T_i=subtree(r_i)$

- r_i称作r的**孩子** (child)，r_i之间互称**兄弟** (sibling)，r为其**父亲** (parent)，$d = degree(r)$为r的（出）**度**（degree）

  > $e = \sum_{v \in V} degree(v) = n - 1 = \theta(n)$，故在衡量相关复杂度时，可以$n$作为参照

- 若指定T_i作为T的第i棵子树，r_i作为r的第i个孩子，则T称作**有序树**
- V中的$k+1$个节点，通过V中的k条边依次相连，构成一条**路径/通路**（path）

  > $\pi = (v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$

  路径**长度**即所含**边**数：$|\pi| = k$
  环路（cycle/loop）：$v_k = v_0$

  > 如果覆盖所有节点各一次，则称作周游（tour）

- **连通图**：节点之间均有路径（connected）。若连通图不含环路，则称为**无环图**（acyclic）
- 树的任一节点v与根之间存在**唯一**路径

  > $path(v, r) = path(v)$
  > 不致歧义时，路径、节点和子树棵**相互指代**：$path(v)$~v~$subtree(v)$

  v的深度：$depth(v) = |path(v)|$

  $path(v)$上节点均为v的**祖先**（ancestor），v是它们的**后代**（descendent），其中除自身以外，是**真**（proper）祖先/后代。

  根节点是所有节点的**公共祖先**，深度为0。没有后代的节点称作**叶子**（leaf）。所有叶子深度中的最大者，称作（子）树（根）的**高度**。

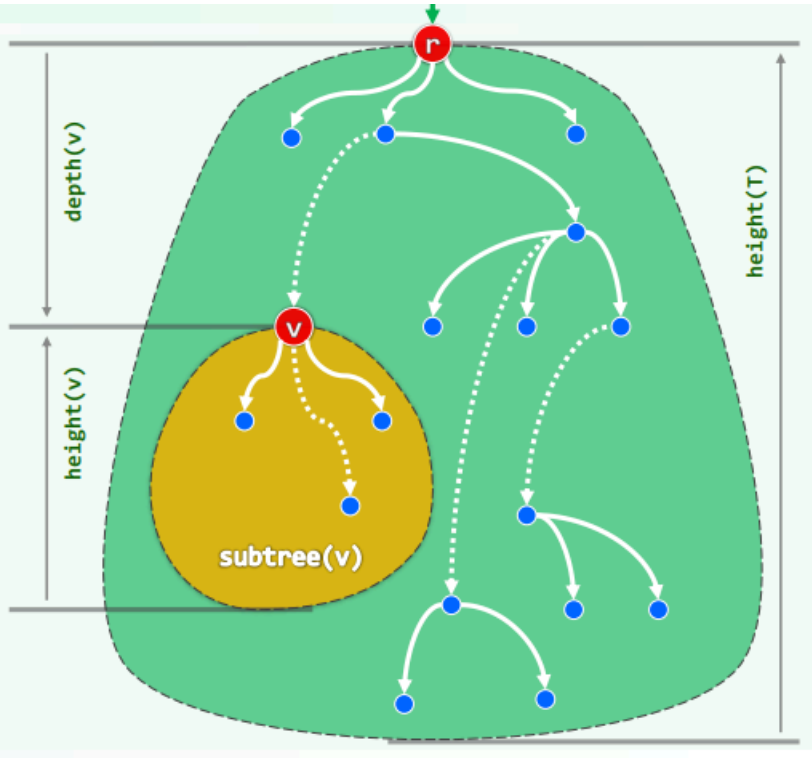
  > $height(v) = height(subtree(v))$

  特别地，**空树**的高度取作-1

  > $depth(v) + height(v) \leq height(T)$

  于是以$|path(v)|$为指标，可对所有节点做**等价类**划分

- **半线性**：在任一深度，v的祖先/后代若存在，则**必然/未必**唯一



二叉树

二叉树：有根有序树

- 节点度数**不超过2**
- 孩子（子树）可以**左**、**右**区分（隐含有序）
  `lc()`~`lSubtree()`
  `rc()`~`rSubtree()`
- 基数：设度数为0，1和2的节点各有n_0，n_1和n_2个
  边数：$e = n - 1 = n_1 + 2n_2$
  叶节点数：$n_0 = n_2 + 1$，注意：n_1与n_0无关！
  节点数：$n = n_0 + n_1 + n_2 = 1 + n_1 + 2n_2$

  > 特别地，当$n_1 = 0$时，有
  > $e = 2n_2, n_0 = n_2 + 1 = \frac{(n+1)}{2}$
  > **此时，节点度数均为偶数，不含单分支节点**

- **满树**
  深度为k的节点，至多有2^k个
  n个节点、高h的二叉树满足$h + 1 \le n \le 2^{h+1} - 1$

  > 特殊情况：
  > $n = h + 1$：退化为一条单链
  > $n = 2^{h+1} - 1$：即所谓的**满二叉树**（full binary tree）

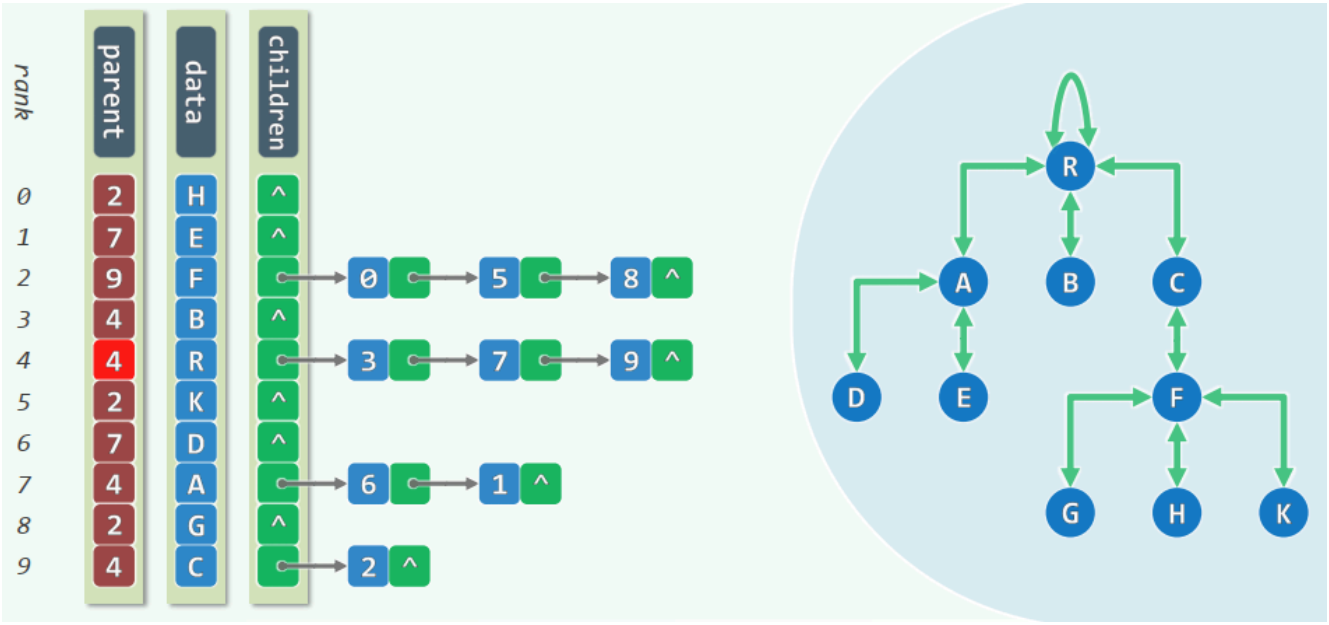
- **真二叉树**
  通过引入$n_1 + 2n_0$个**外部节点**，可使原有节点度数统一为2，由此可以将任一二叉树转化为**真二叉树**（proper binary tree）。

  > 如此转换之后，全树自身的复杂度并未实质增加

二叉树/多叉树的描述

1. 二叉树的描述

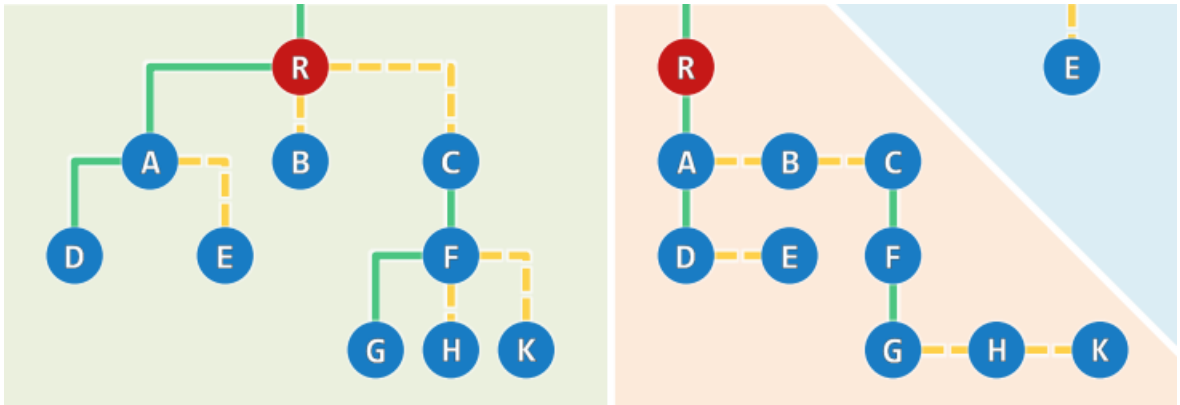
   `root()`：根节点
   `parent()`：父节点
   `firstChild()`：长子
   `nextSibling()`：兄弟
   `insert(i, e)`：将e作为第i个孩子插入
   `remove(i)`：删除第i个孩子（及其后代）
   `traverse()`：遍历



2. 多叉树的描述：长子-兄弟表示法

   有根且**有序**的多叉树，均可转化并表示为二叉树

   - 长子 ~ 左孩子：`firstChild()` ~ `lc()`
   - 兄弟 ~ 右孩子：`nextSibling()` ~ `rc()`



5.1 二叉树的实现

基本接口

1. BinNode模板类

```
template <typename T> using BinNodePosi = BinNode<T>*;
template <typename T> struct BinNode {
    BinNodePosi<T> parent, ls, rc; //父亲、孩子
    T data; int height;
    int size() { // 后代总数
        int s = 1; //计入本身
        if (lc) s += lc->size();
        if (rc) s += rc->size();
        return s;
    }
    BinNodePosi<T> insertAsLc(T const & e) { //作为左孩子插入新节点
        return lc = new BinNode(e, this);
    }
    BinNodePosi<T> insertAsRC(T const & e) { //作为有孩子插入新节点
        return rc = new BinNode(e, this);
    }
    BinNodePosi<T> succ(); //（中序遍历意义下）当前节点的直接后继
    template <typename VST> void travLevl(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
}
```

2. BinTree模板类

```
#define stature(p) ( (p) ? (p)->height : -1) //节点高度——空树~-1

template <typename T> class BinTree {
protected:
    int _size; //规模
    BinNodePosi<T> _root; //根节点
    virtual int updateHeight(BinNodePosi<T> x) { //更新节点x的高度
        return x->height = 1 + max( stature(x->lc), stature(x->rc) );
    }
    void updateHeightAbove(BinNodePosi<T> x); { //更新x及其祖先的高度
        while (x) { // O(n = depth(x))
            updateHeight(x);
            x = x->parent; //可优化
        }
    }

public:
    int size() const { return _size; } //规模
    bool empty() const { return !_root; } //判空
    BinNodePosi<T> root() const { return _root; } //树根

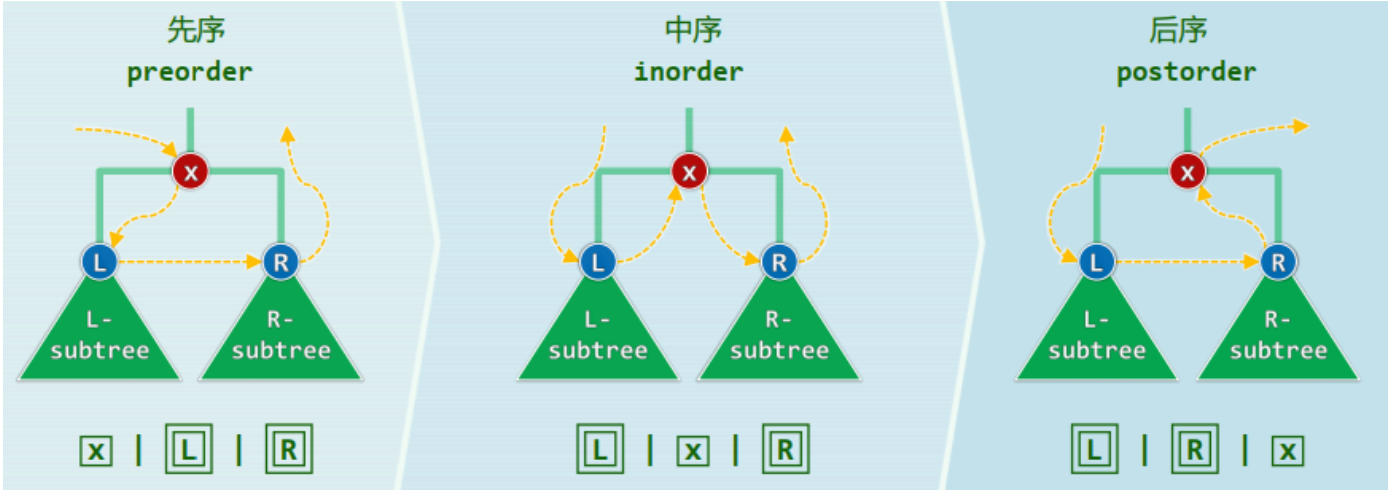
    //孩子接入
    BinNodePosi<T> insert(BinNodePosi<T> x, T const & e); //右孩子
    BinNodePosi<T> insert(T const & e,  BinNodePosi<T> x) { //左孩子
        _size++;
        x->insertAsLC(e);
```

```
        updateHeightAbove(x);
        return x->lc;
    }

    //子树接入
    BinNodePosi<T> attach(BinTree<T>* &S, BinNodePosi<T> x); //接入左子树
    BinNodePosi<T> attach(BinNodePosi<T> x, BinTree<T>* &S) { //接入右子树
        if (x->rc = S->_root)
            x->rc->parent = x;
        _size += S->_size;
        updateHeightAbove(x);
        S->root = NULL;
        S->_size = 0;
        release(S); S = NULL;
        return x;
    }

    //子树删除
    int remove(BinNodePosi<T> x) {
        FromParentTo(*x) = NULL;
        updateHeightAbove(x->parent); //更新祖先高度（其余节点亦不变）
        int n = removeAt(x);
        _size -= n;
        return n;
    }
    static int removeAt(BinNodePosi<T> x) {
        if (!x) return 0;
        int n = 1 + removeAt(x->lc) + removeAt(x->rc);
        release(x->data); release(x);
        return n;
    }

    //子树分离
    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
    }
}
```



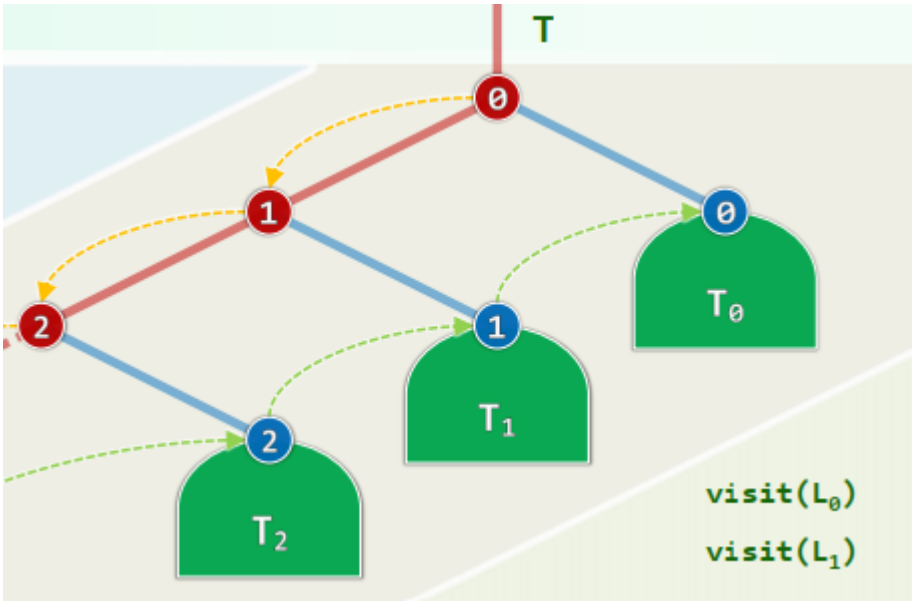
以下遍历其实都是深度优先搜索

1. 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)
```

Problem: 使用**默认**的Call Stack，允许的递归**深度**有限
改进：藤缠树

> 沿着左侧藤，自上而下访问藤上节点，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




```
template <typename T, typename VST>
void travPre(BinNodePosi<T> x, VST & visit) {
    Stack<BinNodePosi<T>> S; //辅助栈
    while (true) {
```

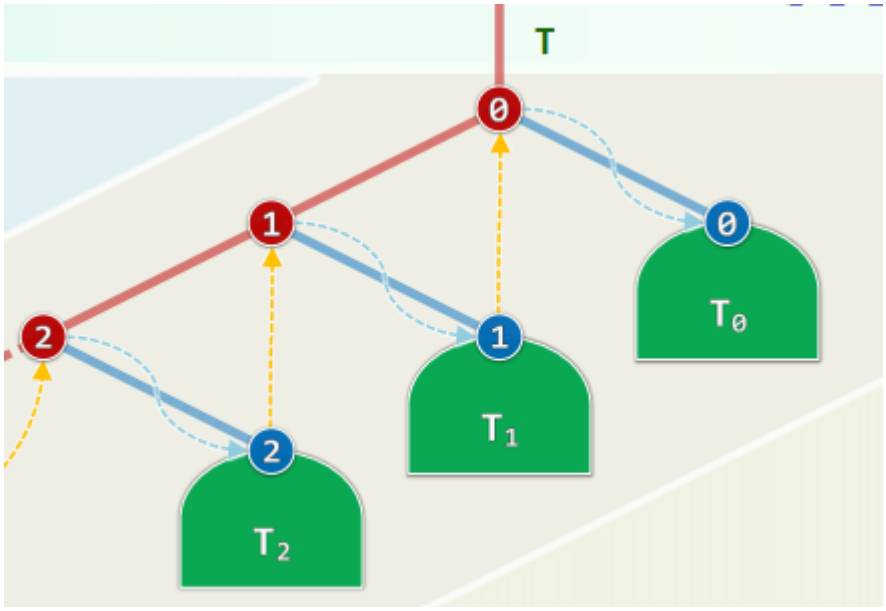
```
        while (x) {
            visit(x->data);
            S.push(x->rc);
            x = x->lc;
        }
        if (S.empty()) break; //栈空即退出
        x = S.pop();
    } //#pop = #push = #visit = O(n) = 分摊O(1)
}
```

2. 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);
} //T(n) = T(a) + O(1) + T(n - a - 1) = O(n)
```

藤缠树

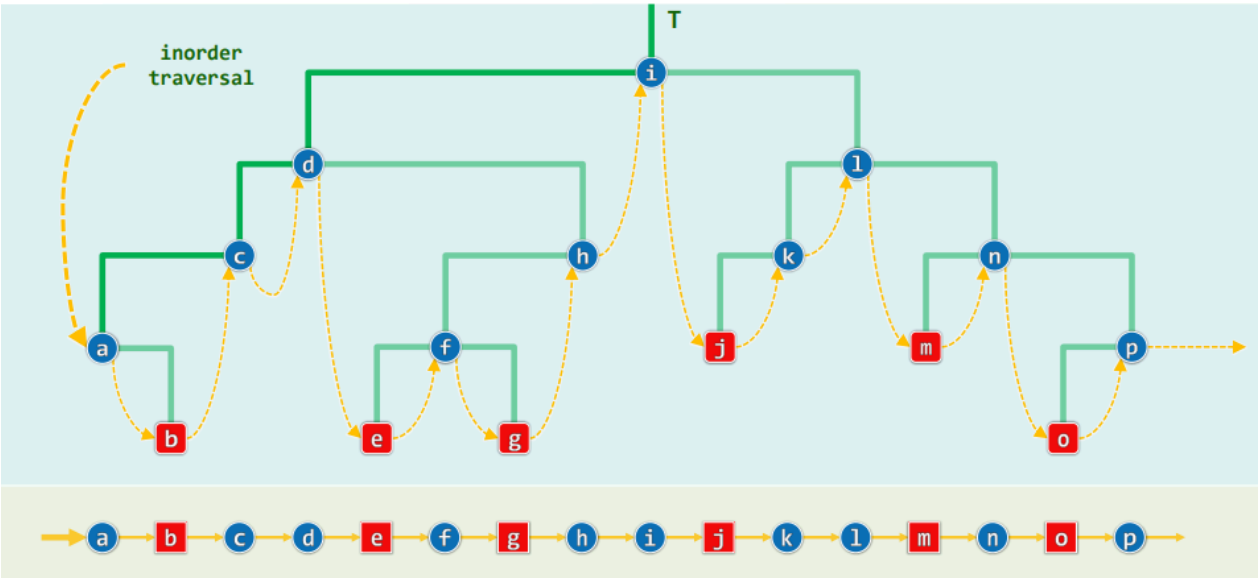
> 沿着左侧藤，遍历可自底而上分解为$d+1$步迭代：访问藤上节点，再遍历其右子树
> 各右子树的遍历彼此独立，自成一个子任务




```
template <typename T, typename V>
void travIn(BinNodePosi<T> x, V & visit) {
    Stack<BinNodePosi<T>> S; //辅助栈
    while (true) {
        while (x) {
            S.push(x);
            x = x->lc;
        }
        if (S.empty()) break;
        x = S.pop();
```

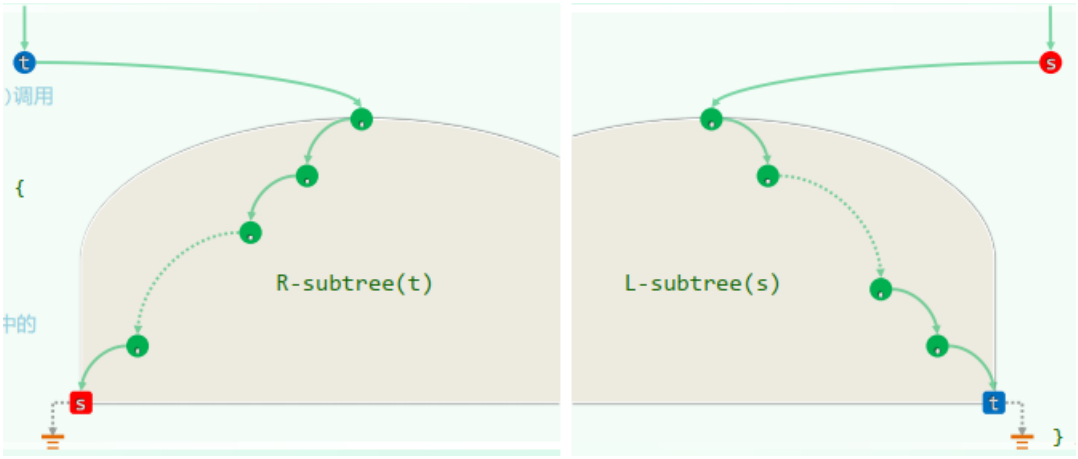
```
        visit(x->data);
        x = x->rc;
    }
}
```

- **效率**：分摊分析

> 每次迭代，都恰有一个节点出栈并被访问
> 每个节点入栈一次且仅一次
> 每次迭代所需要的时间...——分摊$O(1)$
>
> 
>
>
> 分摊复杂度：$O(n)$

后继与前驱

```
// for (BinNodePosi<T> t = first(); t; t = t->succ()) ...
// 在中序遍历移一下的直接后继
template <typename T>
BinNodePosi<T> BinNode<T>::succ() {
    BinNodePosi<T> s = this;
    if (rc) {
        s = rc; //若有右孩子，则直接后继必是右子树中的最小节点
        while (HasLChild(*s))
            s = s->lc;
    }
    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
        s = s->parent; //最后再朝右上移动一步
    }
    return s;
} //两种情况下，时间复杂度分别为当前节点的高度与深度，不过O(h)
```

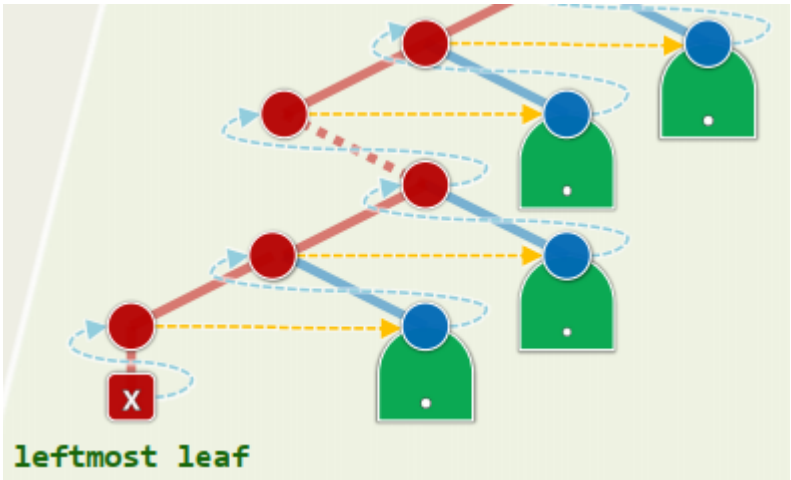


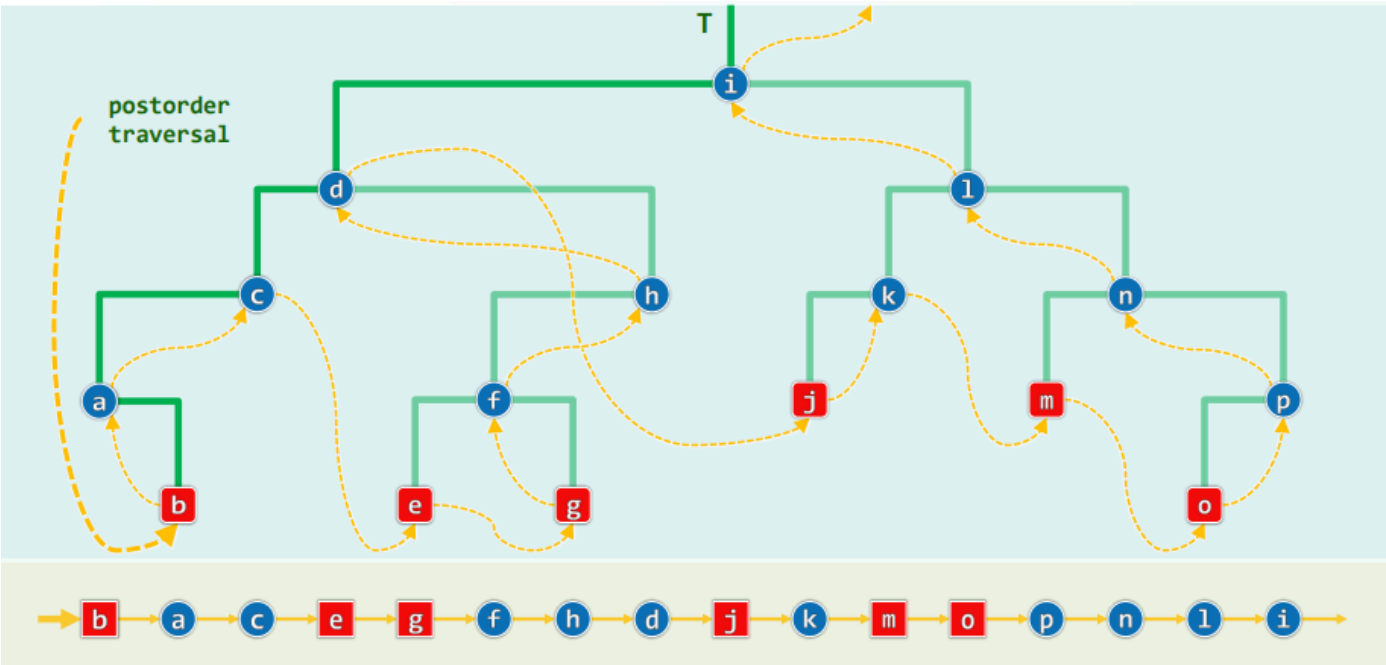
3. 后序遍历

```
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);
} //T(n) = T(a) + T(n - a - 1) + O(1) = O(n)
```

藤缠树

> 从根出发下行：尽可能沿**左**分支，实不得已才沿**右**分支
> 最后一个节点必定是叶子，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






```
template <typename T, typename V>
void travPost(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()) {
        if (S.top() != x->parent) { //即栈顶为右兄，在右兄子树中找到最靠左的叶子
            while (x = S.top())
                if (HasLChild(*x)) { //尽可能向左，在此之前
                    if (HasRChild(*x)) //若有右孩子，则
                        S.push(x->rc); //优先入栈
                    S.push(x->lc); //然后转向左孩子
                } else
                    S.push(x->rc); //实不得已，转向右孩子
            S.pop(); //返回之前，弹出栈顶的空节点
        }
        x = S.pop(); //弹出栈顶（即前一节点的后继）以更新x
        visit(x->data);
    }
}
```

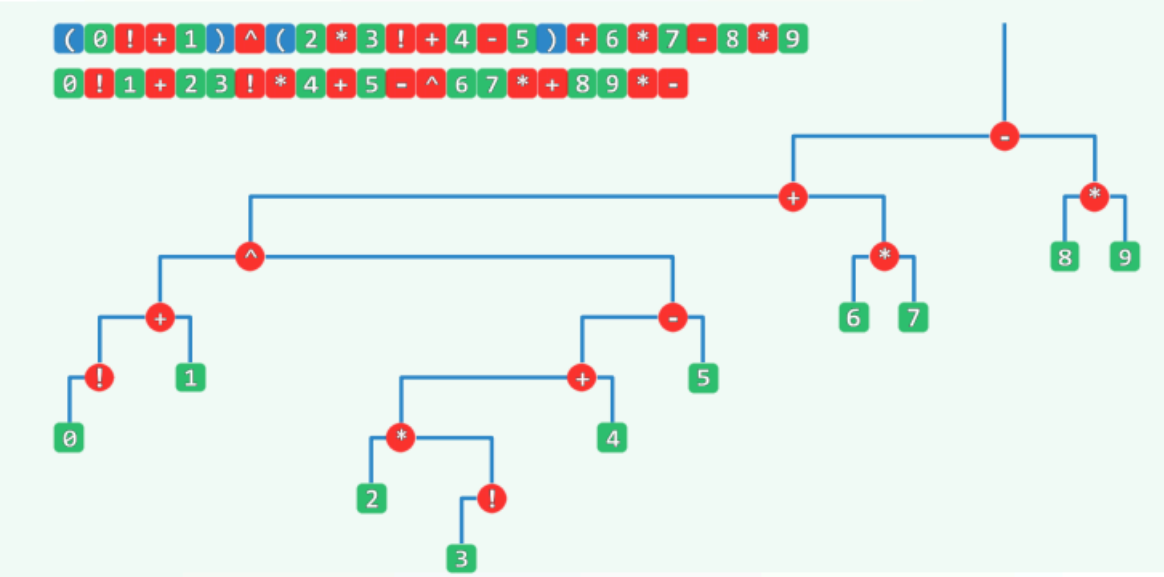
- **正确性**：数学归纳法

  > 每个结点出栈后：以之为根的**子树**已经**完全**遍历，而且其右兄弟r若存在，必恰在**栈顶**
  > 此时正开始遍历子树r

- **效率**：分摊分析

  > 每次迭代，都有一个节点出栈并被访问
  > 每个节点入栈一次且仅一次
  > 每次迭代所需时间...
  >
  > 分摊复杂度：$O(n)$

表达式树（RPN）

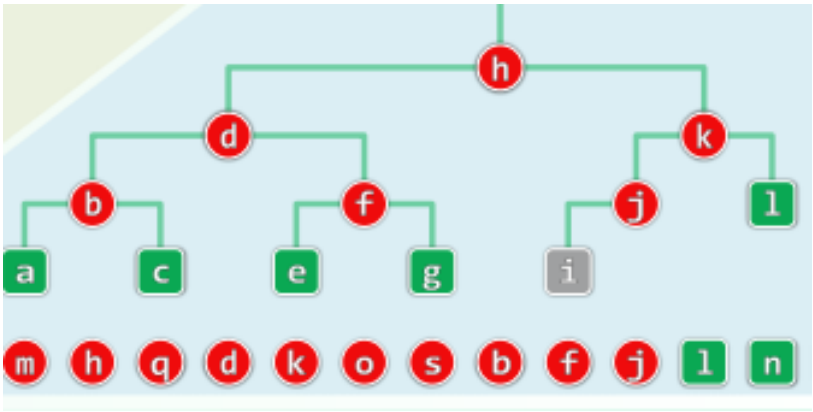


> RPN实际上是Expression Tree的后序遍历

4. 层次遍历（广度优先）

> **概念区分**：满二叉树、完全二叉树、真二叉树

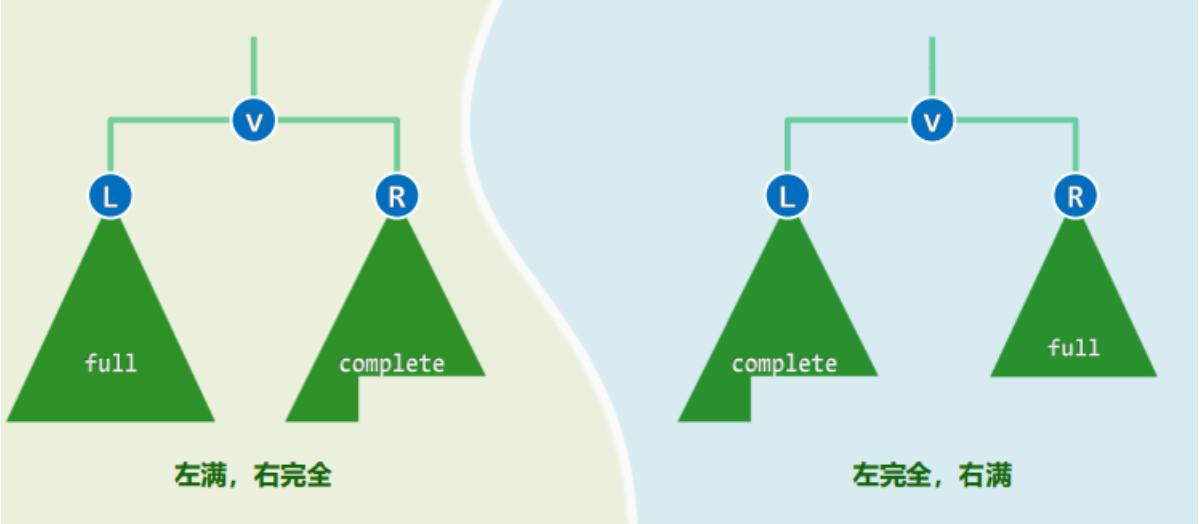
> **完全二叉树**
> 叶节点仅限于**最低两层**
> 底层叶子，均居于次底层叶子左侧（相当于LCA）
> 除末节点的父亲，内部节点均有**双子**
>
> 叶节点：不致少于内部节点，但至多多出一个




```
template <typename T> template <typename VST>
void BinNode<T>::travLevel( VST & visit ) { //二叉树层次遍历
    Queue< BinNodePosi<T> > Q; //引入辅助队列
    Q.enqueue( this ); //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(); //取出队首节点，并随即
        visit( x->data ); //访问之
        if ( HasLChild( * x ) ) Q.enqueue( x->lc ); //左孩子入队
        if ( HasRChild( * x ) ) Q.enqueue( x->rc ); //右孩子入队
    }
}
```

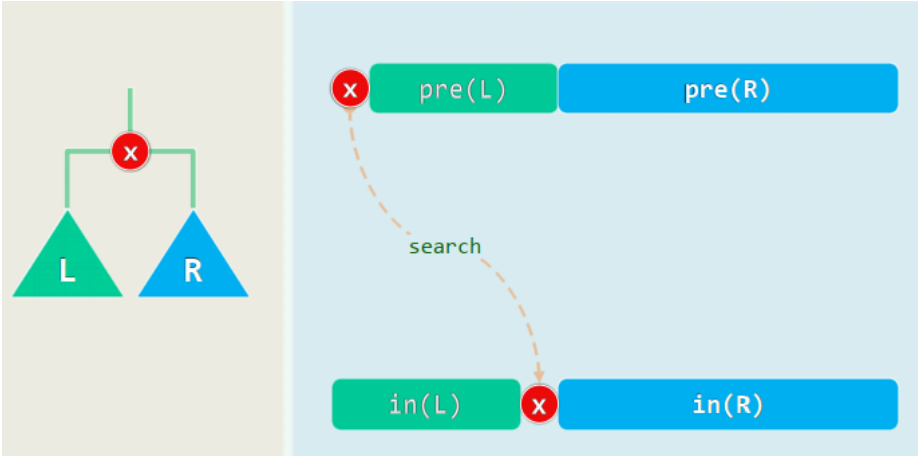
> 考察遍历过程中的n步迭代...
> 前$[n/2]-1$步迭代中，均有**右**孩子入队
> 前$[n/2]$步迭代中，都有**左**孩子入队
> 累计至少$n-1$次入队
>
> 辅助队列的规模：
> 先增后减，**单峰**且**对称**
> 最大规模 = $[n/2]$，最大规模可能出现2次

- 树的层次遍历 => 向量的顺序遍历
- 完全二叉树与满树

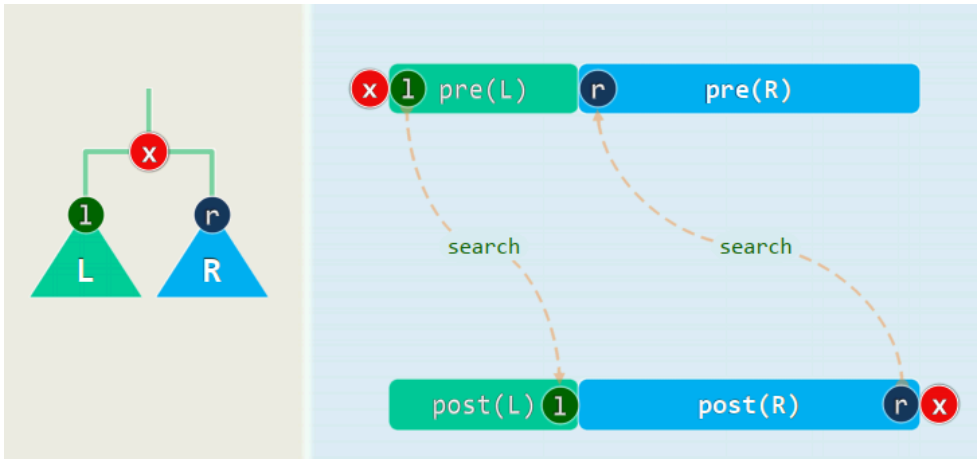


5.2 二叉树的重构

1. 先序 | 后序 => 中旭



2. 先序 <=> 后序



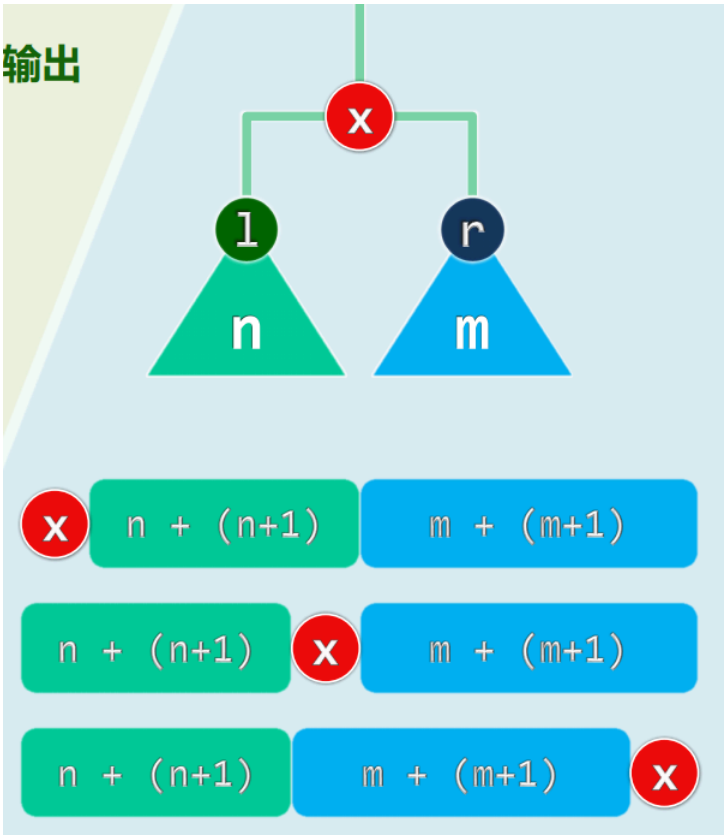
3. 增强序列

假想地认为，每个**NULL**也是“真实”节点，并在遍历时一并输出
每次递归返回，同时输出一个事先约定的元字符“^”

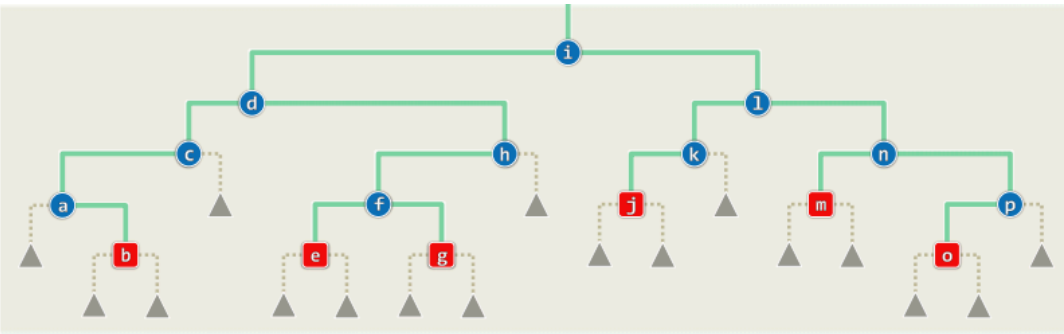
若将遍历序列表示为一个Iterator，则可将其定义为`Vector<BinNode<T>*>`，于是在**增强的**遍历序列中，这类“节点”可统一记作**NULL**

可归纳证明：在增强的先序、中序、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

如此，通过对增强序列**分而治之**，即可重构原树



实例：

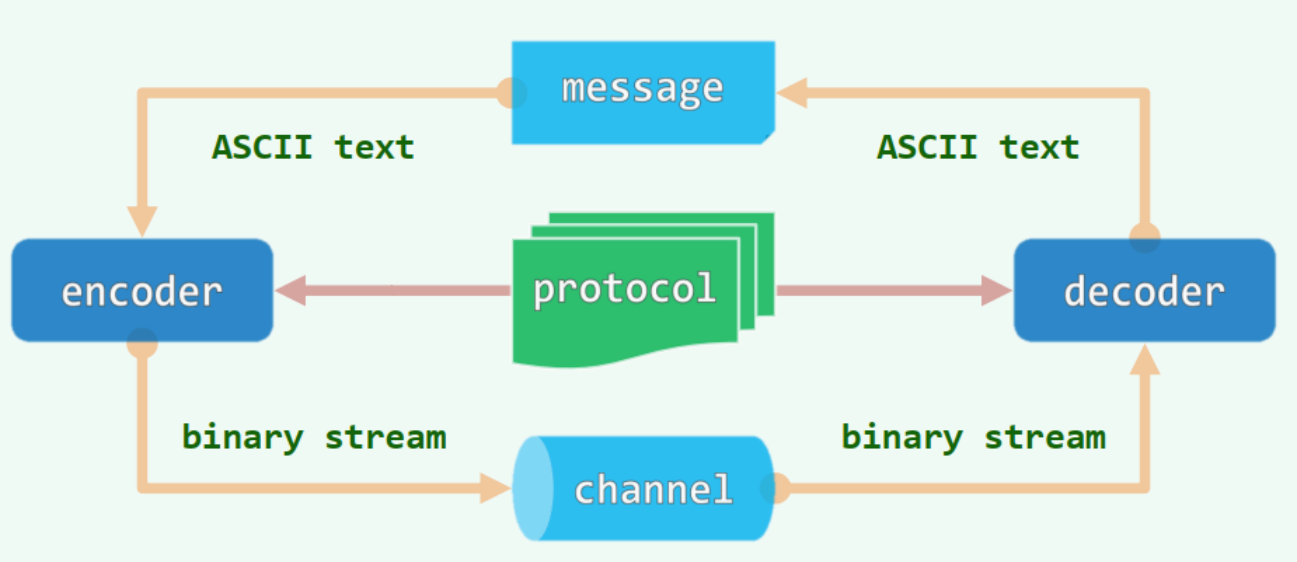


5.3 Huffman编码树

编码

- 二进制编码

  组成数据文件的字符来自字符集$\sum$
  字符被赋予互异的二进制串
- 文件的大小取决于：字符的数量 × 各字符编码的长短
- 通讯带宽有限时，如何对各字符编码使得文件最小？



1. PFC编码 (prefix-free code)

将$\sum$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
不同字符的编码**互不为前缀**，故不致歧义

平均编码长度：$ald(T) = \sum_{x \in \sum} \frac{depth(v(x))}{|\sum|}$
对于特定的$\sum$，$ald()$最小者即为最优编码树T_{opt}

> **真**完全树是**最优**编码树

2. Huffman编码：最优带权编码树

> 已知各字符的**期望频率**，构造最优编码树
> 频率最**高/低**的（超）字符，应尽可能放在**高/低**处
> 故此，通过**适当**的交换，同样可以缩短$wald(T)$

文件长度正比于平均带圈深度$wald(T) = \sum_x rps(x) \times w(x)$

伪代码：

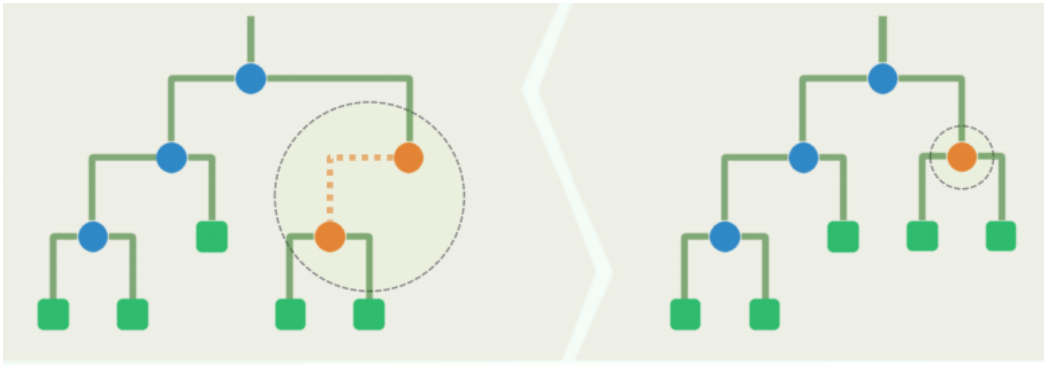
> 贪婪策略：频率**低**的字符优先引入，位置亦更**低**
> 为每个字符创建一棵单节点的数，组成森林**F**，按照出现频率，对数排序
> while （F中的树不止一棵）
> 取出频率最小的两棵树：T_1和T_2
> 将它们合并成一棵新树T，并令：
> $lc(T) = T_1$ 且$rc(T) = T_2$
> $w(root(T)) = w(root(T_1)) + w(root(T_2))$
>
> 尽管贪心策略未必总能得到最优解，但如上算法的确能够得到最优编码树**之一**

正确性

1. 双子性

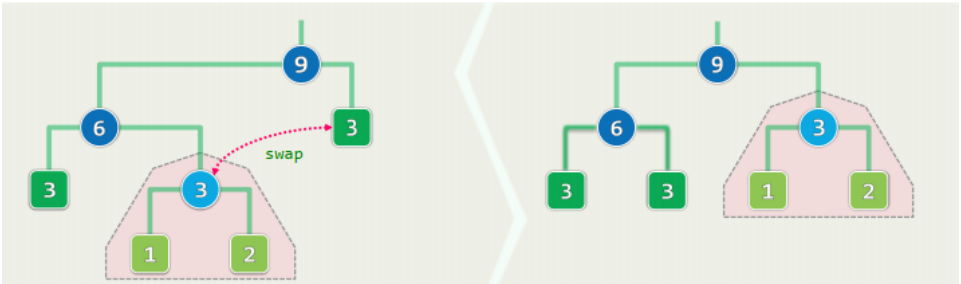
   最优编码树的特征

   > 首先，每一内部节点都有**两个**孩子——节点度数均为偶数（0或2），即**真**二叉树
   > 否则，将1度节点**替换**为其唯一的孩子，则新书的$wald$将**更小**



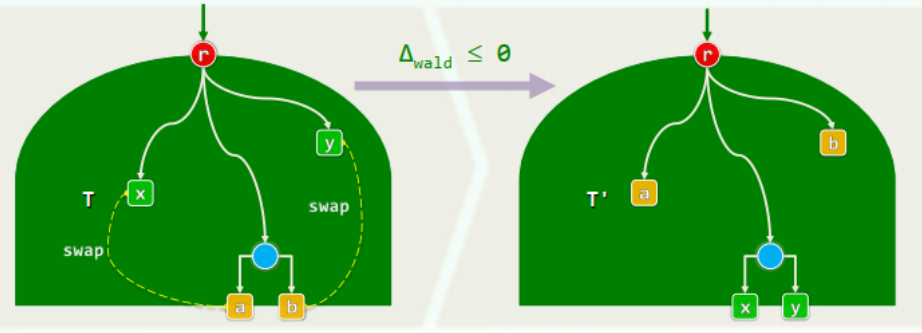
2. 不唯一性

   > 对任一内部节点而言，左右子树**互换**之后$wald$不变
   > 为消除这种歧义，可以（比如）明确要求**左**子树的频率更**低**



3. 层次性

> 出现频率最低的字符x和y，必在某棵最优编码树中处于最底层，且互为兄弟
> 否则，任取一棵最优编码树，并在其最底层任取一对兄弟a和b，于是，a和x、b和y 交换之后，$wald$绝不会增加



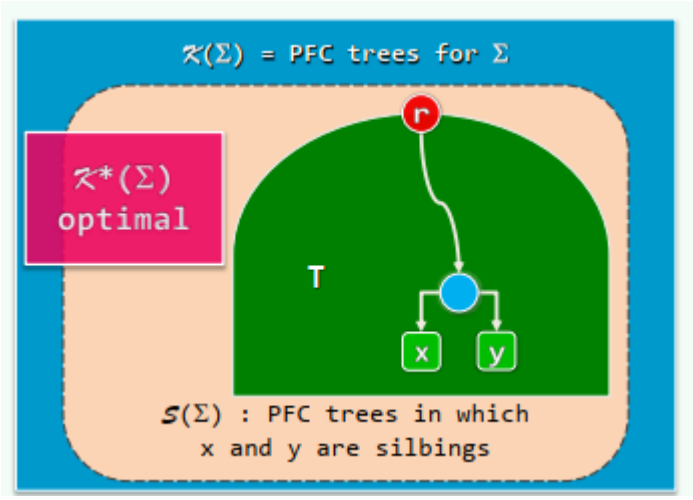
4. 数学归纳和定差

对$|\sum|$做归纳可证：Huffman算法所成的必然是**一棵**最优编码树

$|\sum| = 2$时显然
设算法在$|\sum| < n$时均正确。现在设$|\sum| = n$，取$\sum$中频率最低的x，y（不放设二者互为兄弟），令$\sum' = (\sum \setminus \{x, y\}) \cup \{z\}, w(z) = w(x) + w(y)$

定差：对于$\sum'$的**任一**编码树T'，只要为z添加孩子x和y，即可得到$\sum$的一棵编码树T，且满足$wd(T) - wd(T') = w(x) + w(y) = w(z)$

因此，只要T'是$\sum'$的最优编码树，则T也是$\sum$的最优编码树（之一）



3. Huffman编码树：算法实现

Huffman（超）字符

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

Huffman（子）树

```
using HuffTree = BinTree<HuffChar>;
```

Huffman森林

```
using HuffForest = List<HuffTree*>;
```

更高效的数据结构实现方式：
*（得益于已定义的**统一接口**，支撑Huffman算法的这些底层数据结构可直接**彼此替换**）*

```
using HuffForest = PQ_List<HuffTree*>; //基于列表的优先级队列
using HuffForest = PQ_CompHeap<HuffTree*>; //完全二叉堆
using HuffForest = PQ_LeftHeap<HuffTree*>; //左式堆
```

构造编码树：

```
HuffTree* generateTree(HuffForest * forest) {
    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，准备合并T1和T2
        S->insert(HuffChar('^', // 根节点权重，取作T1与T2之和
           T1->root()->data.weight + T2->root()->data.weight));
        S->attach(T1, S->root());
        S->attach(S->root(), T2);
        forest->insertAsLast(S); // T1与T2合并后，重新插回森林
    } // assert: 循环结束时，森林中唯一的那棵树即是Huffman编码树
    return forest->first()->data;
}

//查找最小超字符
HuffTree* minHChar(HuffForest * forest) {
    /*如何优化算法*/
}
```

构造编码表：

```
#include "Hashtable.h"
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
        {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->lc);}
    if (HasRChild(*v)) //Right = 1
        {code->set(length); generateCT(code, length+1, table, v->rc);}
} //总体O(n)
```

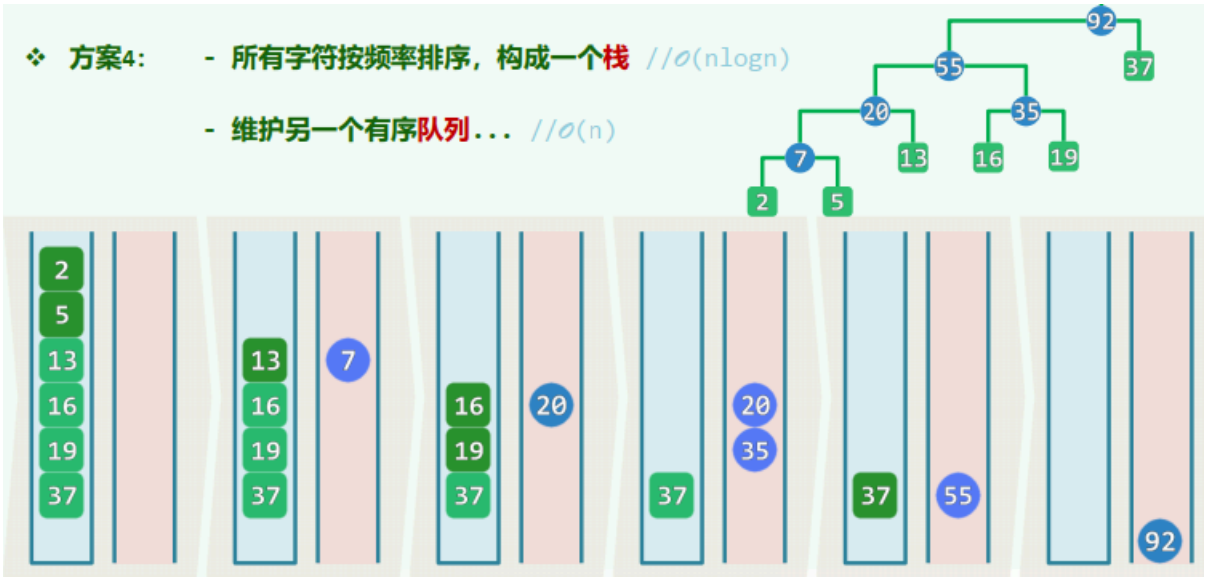
优化：向量+列表+优先级队列

方案1：$O(n^2)$
初始化时，通过排序得到一个**非升序向量**：$O(nlogn)$
每次（从**后**端）取出频率最低的两个节点：$O(1)$
将合并得到的新树插入向量，并保持有序：$O(n)$

方案2：$O(n^2)$
初始化时，通过排序得到一个**非降序列表**：$O(nlogn)$
每次（从**前**端）取出频率最低的两个节点：$O(1)$
将合并得到的新树插入列表，并保持有序：$O(n)$

方案3：$O(nlogn)$
初始化时，将所有数组织为一个**优先级队列**：$O(n)$
取出频率最低的两个节点，合并得到的新树插入队列：$O(logn) + P(logn)$

方案4：$O(nlogn)$
所有字符按频率排序，构成一个**栈**：$O(nlogn)$
维护另一个有序**队列**：$O(n)$



5.4 下界：代数判定树

问题P的难度：若问题P存在算法，则所有算法中**最低**的复杂度成为P的难度

多种角度估算时间、空间复杂度：

最好 / best-case
最坏 / worst-case
平均 / average-case
分摊 / amortized

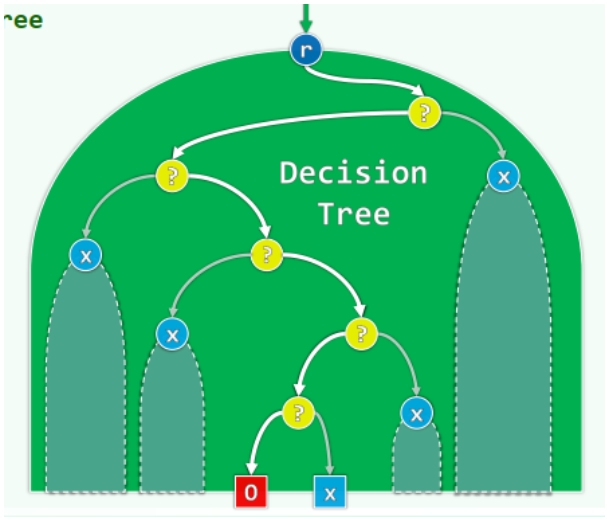
其中，对最坏情况的估计最保守、最稳妥。因此，首先应考虑**最坏情况最优**的算法（worst-case optimal）

基于比较的算法（Comparison-Based Algorithm）：任何CBD在最坏情况下，都需要$\Omega(nlogn)$时间才能完成排序

1. 判定树

每个CBA算法都对应于一棵Algebraic Decision Tree
每一可能的输出，都对应于**至少**一个叶节点
每一次运行过程，都对应于起始于根的某条路径

代数判定树（Algebraic Decision Tree）
针对“比较-判定”式算法的计算模型
给定输入的规模，将所有可能的输入所对应的一系列判断表示出来

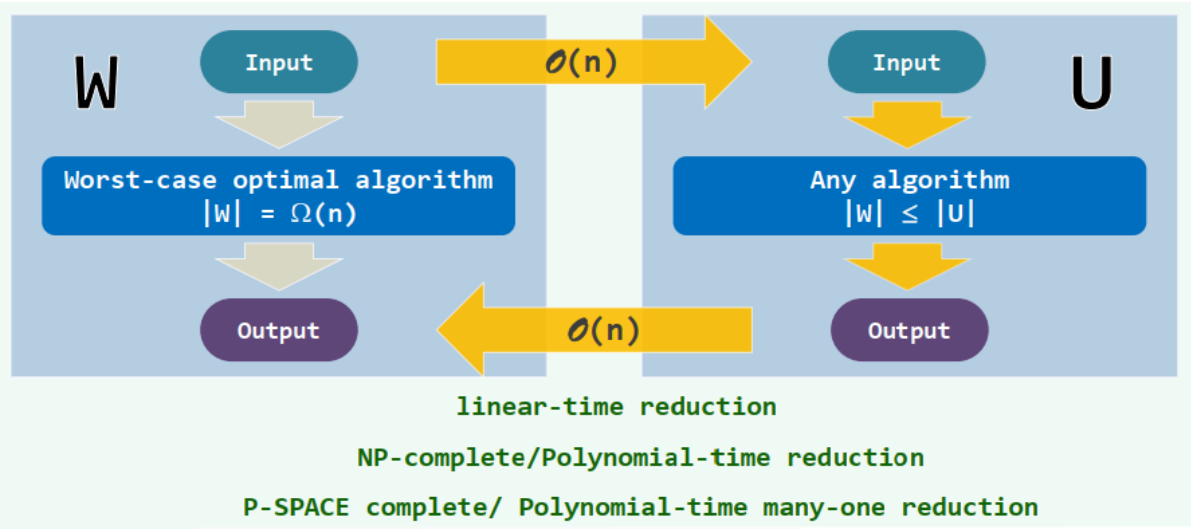


代数判定:
使用某一常数次数代数多项式将任意一组关键码作为变量，对多项式求值
根据结果的符号，确定算法推进方向

> 叶节点深度 ~ 比较次数 ~ 计算成本
> 树高 ~ 最坏情况时的计算成本
> 树高的下界 ~ 所有CBA的时间复杂度下界

2. 下界：归约

线性归约（Linear-Time Reduction）：除了（代数）判定树，归约（reduction）也是确定下界的有利工具



实例:

> [Element Uniqueness] 任意n个**实数**中，是否包含雷同？——$O(nlogn)$
> $EU \leq_N Closest\ Pair$
> [Integer Element Uniqueness] 任意n个**整数**中，是否包含雷同？——$O(nlogn)$
> $IED \leq_N Segment\ Intersection\ Detection$
> [Set Disjointness] 任意一对集合A和B，是否存在**公共**元素？——$O(nlogn)$
> $SD \leq_N Diameter$
> [Red-Blue Matching] 平面上任给n个红色点和n个蓝色点，如何用**互不相交**的线段配对联结——$O(nlogn)$
> $Sorting \leq_N Red - Blue\ Matching$
>
> $Sorting \leq_N Huffman\ Tree \leq_N Optimal\ Encoding\ Tree$

5.5 其它应用

1. Graph/Tree: Diameter / Eccentricity / Radius / Center
2. Knights of the Round Table / Travelling Knight

6 二叉搜索树 BST

call-by-Key：循**关键码**访问
数据集中的数据项，统一地表示和实现为词条（entry）形式

```
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) {};
    bool operator< ( Entry<K, V> const & e ) { return key < e.key; }
    bool operator> ( Entry<K, V> const & e ) { return key > e.key; }
    bool operator==( Entry<K, V> const & e ) { return key == e.key; }
    bool operator!=( Entry<K, V> const & e ) { return key != e.key; }
}
```

假设：没有重复关键码

顺序性：任一节点均**不小/大于**其**左/右**后代 <==> 任一节点均**不小/大于**其左/右**孩子**

> 顺序性只是对**局部**特征的刻画，却可导出BST的**整体**特征

单调性：对**树高**做数学归纳，可以证明BST的**中序**遍历序列，必然**单调**非降

基本接口

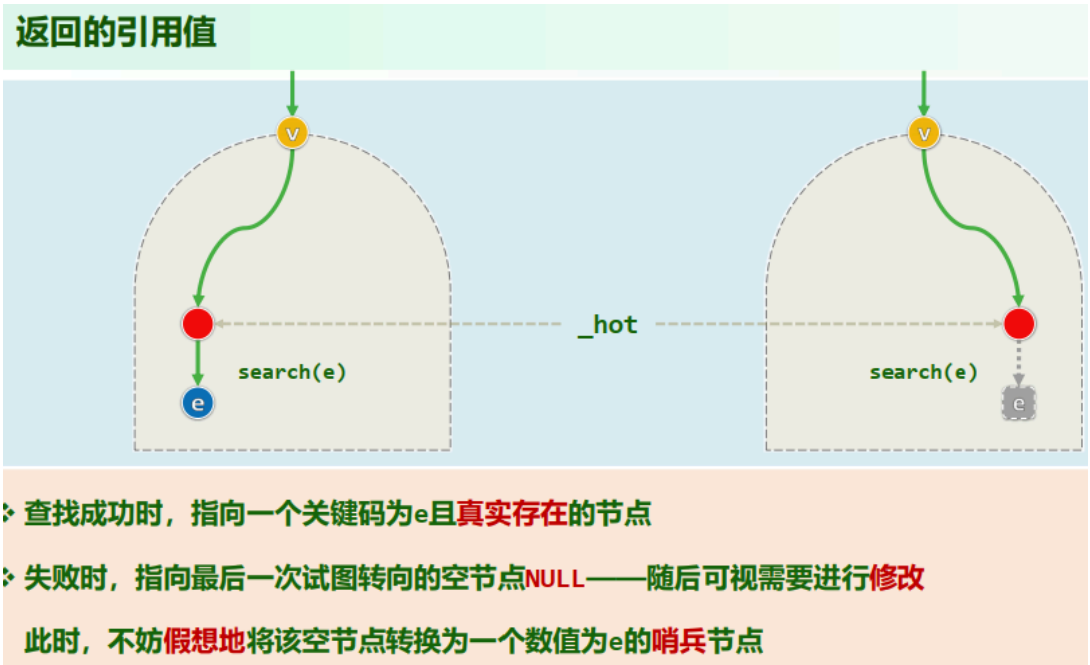
```
template <typenme T> class BST : public BinTree<T> {
public: //以virtual修饰，以便派生类重写
    virtual BinNodePosi<T> & search(const T &); //查找
    virtual BinNodePosi<T> insert(const T &); //插入
    virtual bool remove(const T &); //删除
```

```
protected:
    BinNodePosi<T> _hot; //命中节点的父亲
    BinNodePosi<T> connect34( //3+4重构，稍晚再详解
        BinNodePosi<T>, BinNodePosi<T>, BinNodePosi<T>,
        BinNodePosi<T>, BinNodePosi<T>, BinNodePosi<T>, BinNodePosi<T>);
    BinNodePosi<T> rotateAt( BinNodePosi<T> ); //旋转调整
}
```

6.1 查找

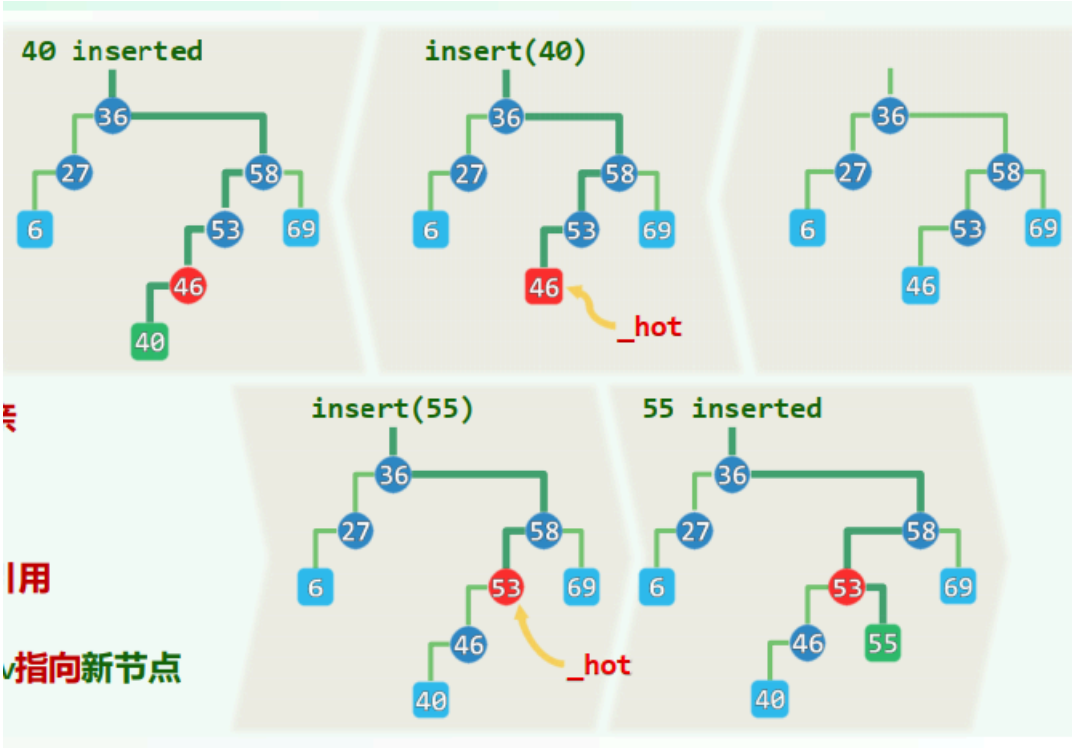
减而治之：对照中序遍历序列，整个过程可视作有序向量的**二分查找**

```
template <typename T>
BinNodePosi<T> & BST<T>::search(const T & e) {
    if (!_root || e == _root->data) { //空树，或恰好在树根命中
        _hot = Null;
        return _root;
    }
    for (_hot = _root; ; ) {
        BinNodePosi<T> & v = (e < _hot->data) ? _hot->lc : _hot->rc;
        if (!v || e == v->data) return v;
        _hot = v; //无论命中或失败，_hot均指向v的父亲（v是根时，hot为NULL）
    }
}
```



6.2 插入

> 先借助`search(e)`确定插入位置及方向
> 若e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
> 于是，只需令_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
} //时间主要消耗于search(e)和updateHeightAbove(x)；均线性正比于x的深度，不超过树高
```

6.3 删除

```
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 ); //分两大类情况实施删除，更新全树规模
    _size--; //更新全树规模
    updateHeightAbove( _hot ); //更新_hot及其历代祖先的高度
    return true;
} //累计O(h)时间：search()、updateHeightAbove()；还有removeAt()中可能调用的succ()

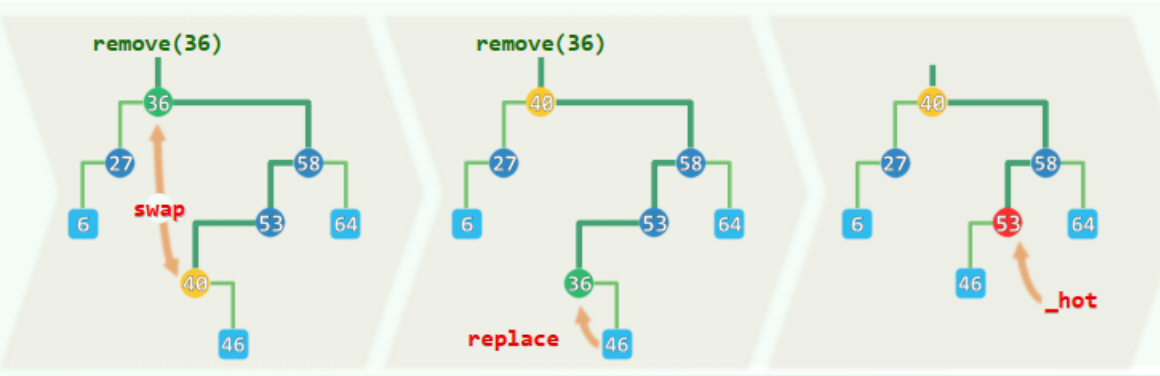
template <typename T>
static BinNodePosi<T> removeAt(BinNodePosi<T> & x, BinNodePosi<T> & hot) {
    BinNodePosi<T> w = x; //实际被摘除的节点，初值同x
    BinNodePosi<T> succ = NULL; //实际被删除节点的接替者
    if ( !HasLChild( *x ) ) succ = x = x->rc; //左子树为空
    else if ( ! HasRChild( *x ) ) succ = x = x->lc; //右子树为空
    else {
```

```
        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
    }

    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
```

主要难点：双分支情况下的删除

> 调用`BinNode::succ()`找到x的直接后继（必无左孩子），交换x进而问题转换为删除w，可按前一情况处理
> 尽管顺序性在中途曾**一度不合**，但最终必将重新恢复



6.4 平衡 BBST

1. 期望树高

> BST的主要接口`search()`、`insert()`和`remove()`的运行时间在最坏情况下，均线性正比于其高度$O(h)$
> 在最坏情况下，二叉搜索树可能彻底**退化**为列表
> 下面给出此类较坏情况下的**概率**分析

- **随机生成**：n个词条按随机排列一次插入

  > 设各排列出现的概率**均等**（$1/n!$），则BST的**平均**树高为$\Theta(logn)$
  > 即便计入`remove()`，也可通过随机使用`succ()`和`pred()`，避免逐渐**倾侧**的趋势

- **随机组成**：n个互异节点，在遵守顺序性的前提下，随机确定拓扑连接关系

  > 由n个互异节点随机**组成**的BST，若共计$S(n)$棵，则有：
  > $S(n)=\sum_{k=1}nS(k-1)\cdot S(n-k)=catalan(n)=\frac{(2n)!}{(n+1)!n!}$
  > 假设所有BST**等概率**地出现，则其**平均**高度为$\Theta(\sqrt{n})$
  > 在Huffman编码之类的应用中，二叉树（尽管还不是BST）的确是逐渐**拼合**而成的

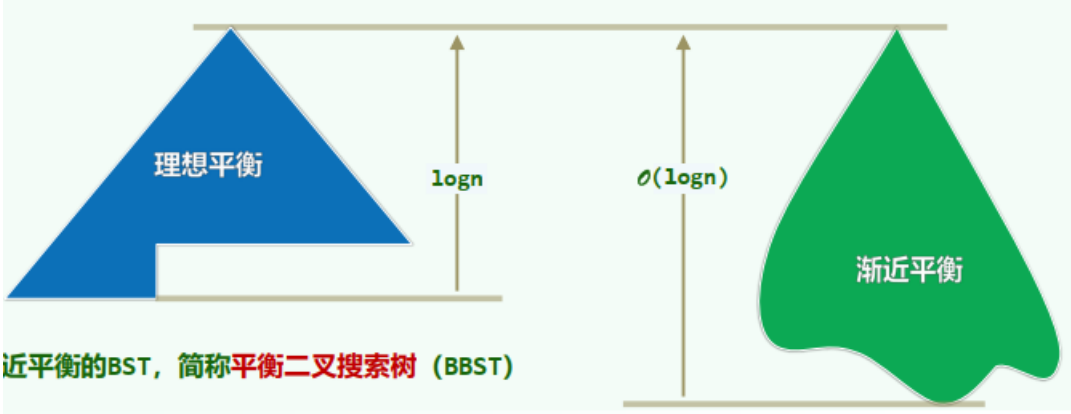
2. 理想与渐进平衡

理想平衡：由n个节点组成的二叉树，高度不低于log_2n，达到该下界时称作理想平衡

> 大致相当于**完全树**甚至**满树**：叶节点只能出现与最底部的两层——条件过于苛刻

渐近平衡：高度渐近地不超过$O(logn)$，即可接受

渐进平衡的BST，简称平衡二叉搜索树



3. 等价变换

- **上下可变**：连接关系不尽相同，承袭关系可能颠倒
- **左右不乱**：中序遍历序列完全一致，全局单调非降

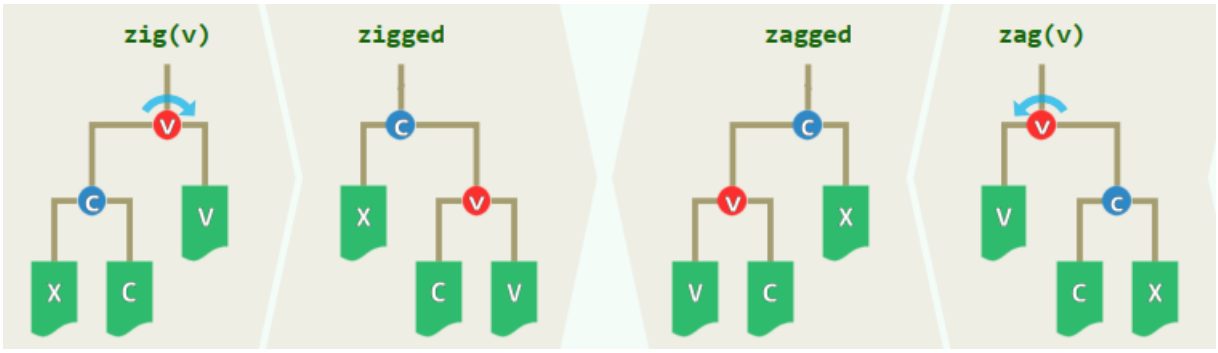
各种BBST都可视作BST的某一子集，相应地满足精心设计的**限制条件**

> 单次动态修改操作后，至多$O(logn)$处局部不再满足限制条件（可能相继违反，未必同时）
> 可在$O(logn)$时间内，使局部重新满足

- **等价变换 + 旋转调整**

  刚刚失衡的BST，必可迅速转换为一棵等价的BBST——为此，只需$O(logn)$甚至$O(1)$次旋转

  > zig和zag：仅涉及**常数**个节点，只需调整其间的连接关系，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



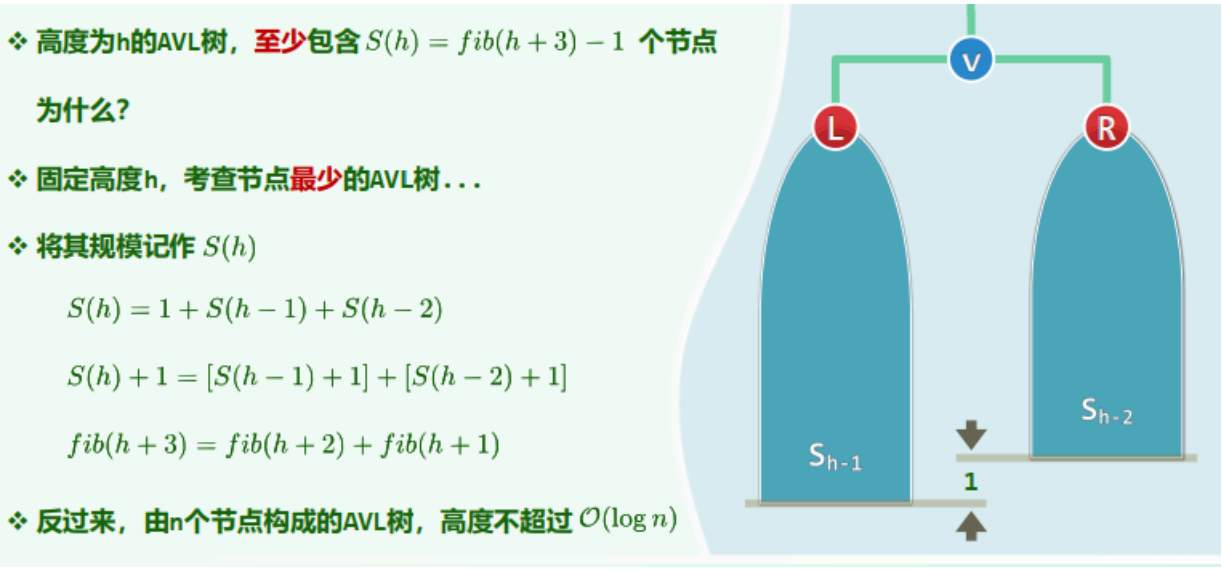
6.5 AVL树

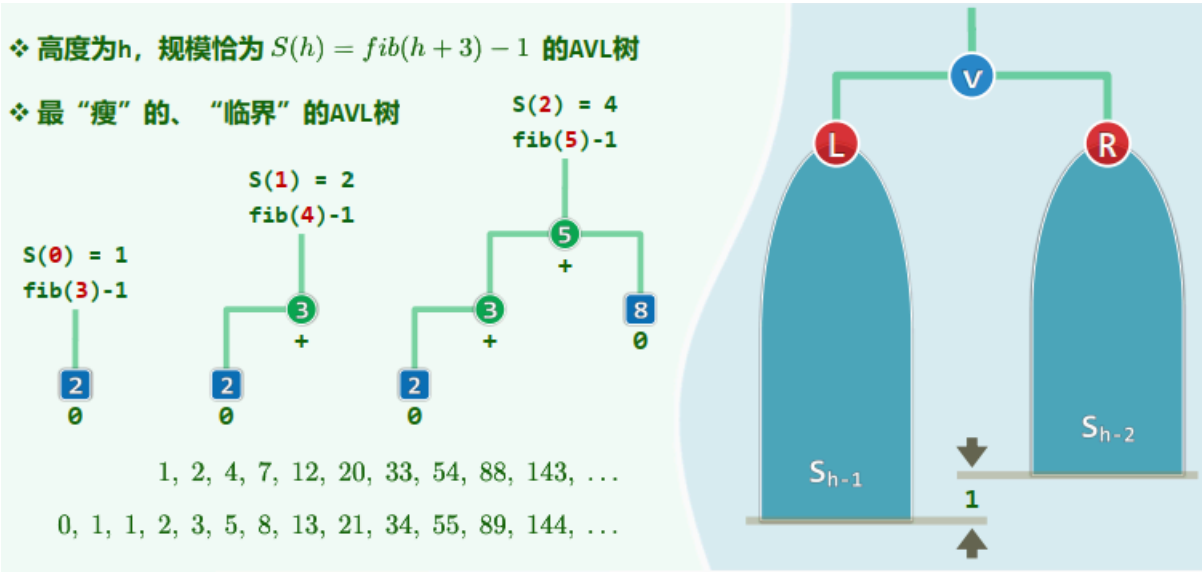
> 平衡因子Balance Factor: $balFac(v) = height(lc(v)) - height(rc(v))$
> $\forall v \in AVL, |balFac(v)| \leq 1$

AVL树未必**理想平衡**，但必然**渐近平衡**

渐近平衡

> 高度为h的AVL树，至少包含$S(h) = fib(h+3) - 1$个节点

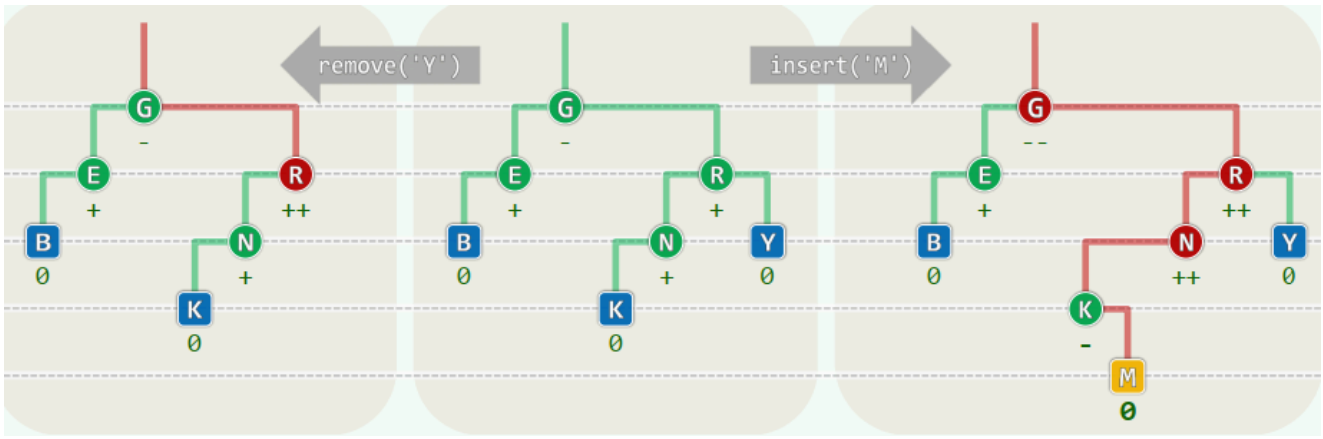




重平衡

失衡

> 插入：从**祖父**开始，每个祖先都有可能失衡，且**可能同时**失衡，但只需调整**一次**
> 删除：从**父亲**开始，每个祖先都有可能失衡，且**至多一个**，但可能需要调整**多次**



重平衡

> 局部性：所有旋转都在**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$O(logn)$次*

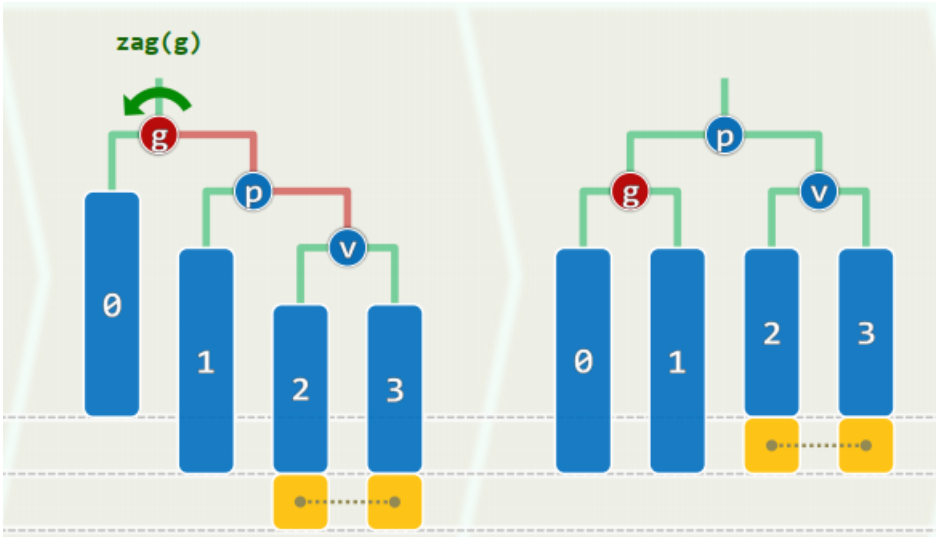
```
#define Balanced(x) (stature((x).lc) == stature((x).rc)) //理想平衡
#define BalFac(x) (stature((x).lc) - stature((x).rc)) //平衡因子
#define AvlBalanced(x) ((-2 < BalFac(x)) && (BalFac(x) < 2)) //AVL平衡条件

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）
};
```

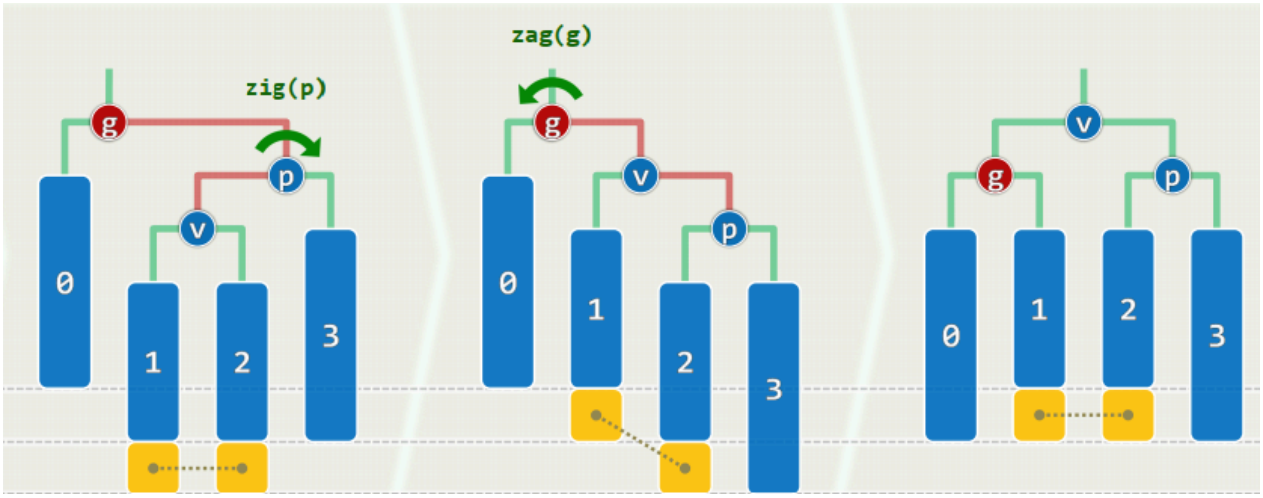
1. 插入

> 同时可有**多个**失衡节点，最低者g不低于x的祖先
> g经单旋调整后复衡，子树**高度复原**
> 更高祖先也必平衡，全树复衡

- **单旋**：黄色方块**恰好**存在其一



- **双旋**

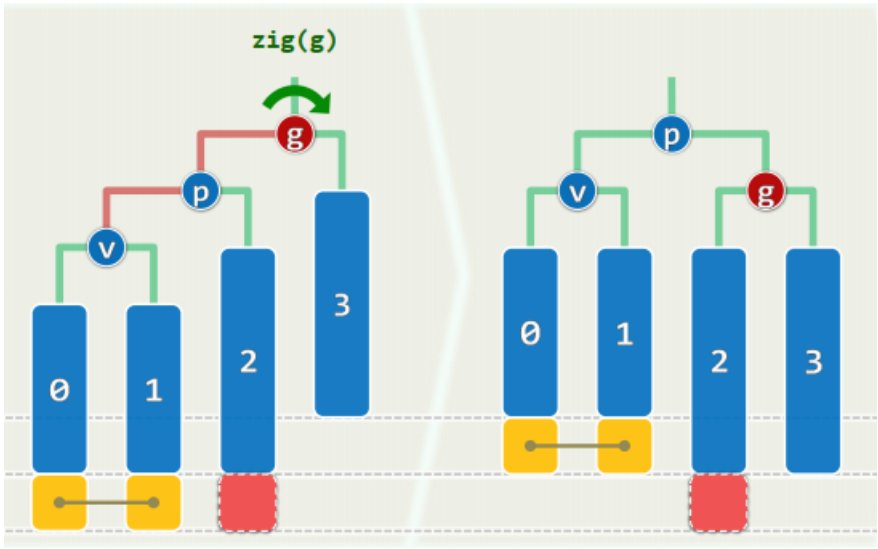


```
template <typename T>
BinNodePosi<T> AVL<T>::insert( const T & e ) {
    BinNodePosi<T> & x = search( e );
    if ( x ) return x; //若目标尚不存在
    BinNodePosi<T> xx = x = new BinNode<T>( e, _hot );
    _size++; //则创建新节点
        // 此时，若x的父亲_hot增高，则祖父有可能失衡
    for ( BinNodePosi<T> g = _hot; g; g = g->parent ) //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（即便g未失衡，高度亦可能增加）
    return xx; //返回新节点位置
}
```

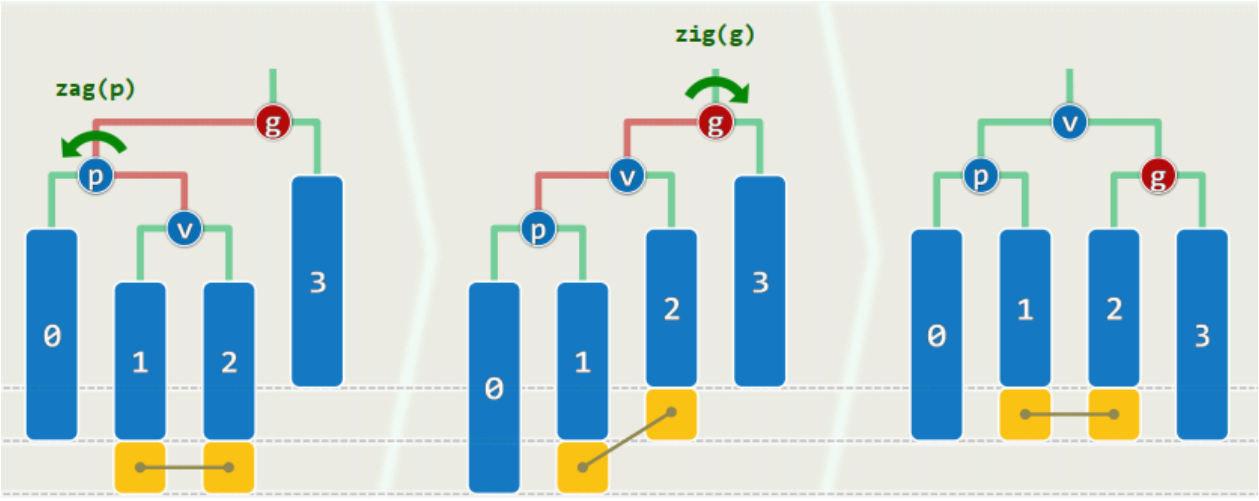
2. 删除

> 同时**至多一个**失衡节点g，**首个**可能就是x的父亲_hot
> 复衡后子树高度**未必**复原，更高祖先仍可能**随之**失衡
> 失衡可能持续向上**传播**，最多需做$O(logn)$次调整

- **单旋**：黄色方块**至少**存在其一，红色方块可有可无



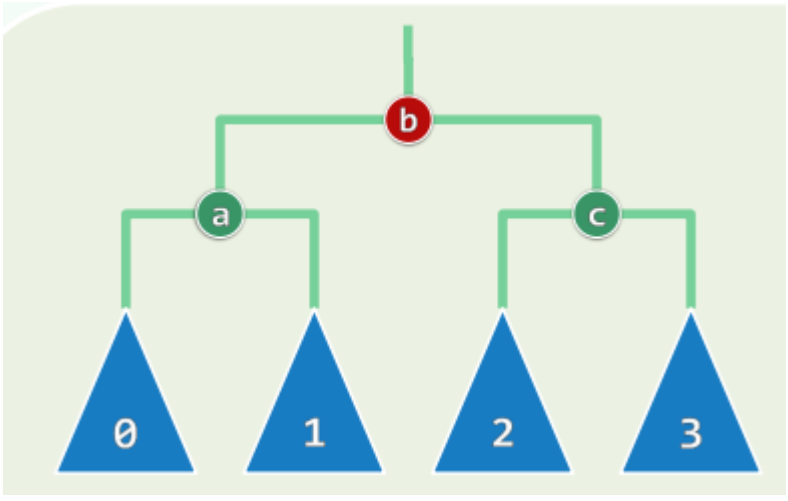
- **双旋**




```
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O(logn)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
}
```

3. (3+4)-重构

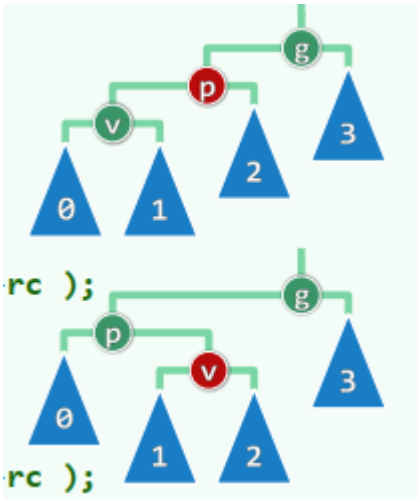
> 设g为**最低**的失衡节点，沿**最长**分支考察祖孙三代：g~p~v，按**中序**遍历次序，**重命名**为：a < b < c
> 它们总共拥有四个子树（或为空），按**中序**遍历次序，**重命名**为：$T_0 < T_1 < T_2 < T_3$
> 将原先以g为根的子树S，替换为一棵新子树S`
> 等价变换，保持中序遍历次序：$T_0 < a < T_1 < b < T_2 < c < T_3$




```
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;
}
```

统一调整:




```
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; //向上联接
            return connect34( v, p, g, v->lc, v->rc, p->rc, g->rc );
        } else { //zig-zag
            v->parent = g->parent; //向上联接
            return connect34( p, v, g, p->lc, v->lc, v->rc, g->rc );
        }
    else { /*.. zag-zig & zag-zag ..*/ }
}
```

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$O(logn)$

$O(n)$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$\Omega(logn)$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$\Omega(logn)$

7 BST Application

Range Query

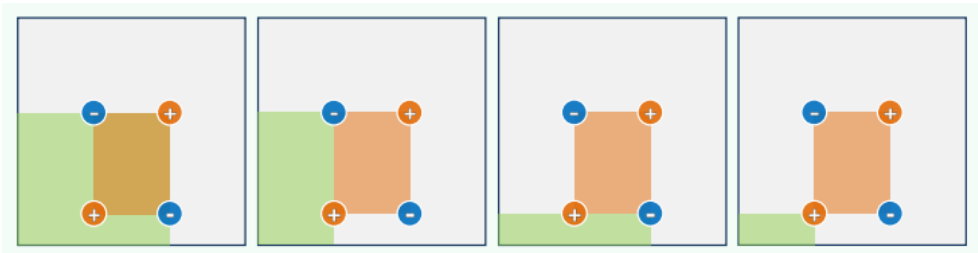
> Counting and Reporting

- **1D Range Query**

  Binary Search + enumerating: $O(logn + r)$

- **2D Planar Range Query**

  Inclusion-Exclusion Principle: query $O(logn)$ + storage $\Theta(n^2)$

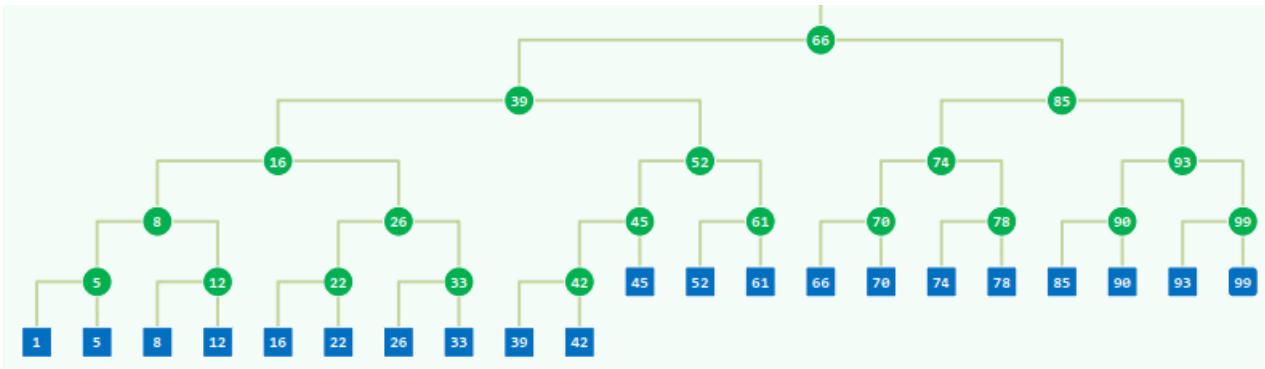


disjoint subtrees

1. kd-Tree: 1D

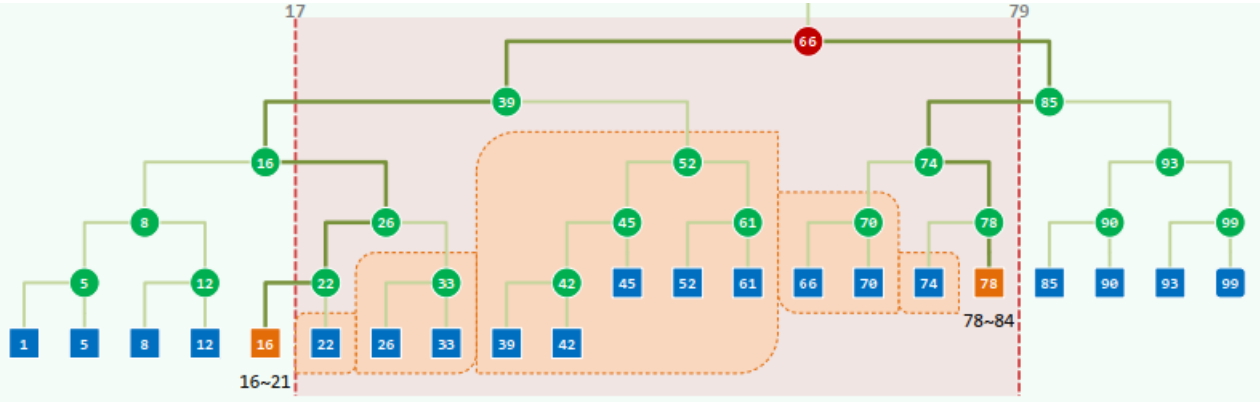
结构：a complete (balanced) BST

> $\forall v, v.key = min\{u.key | u \in v.rTree\} = v.succ.key$
> $\forall u \in v.lTree,\ u.key < v.key,\ \forall u \in v.rTree,\ u.key \geq v.key$
>
> `search(x)`: return the MAXIMUM key not greater than x



disjoint subtrees

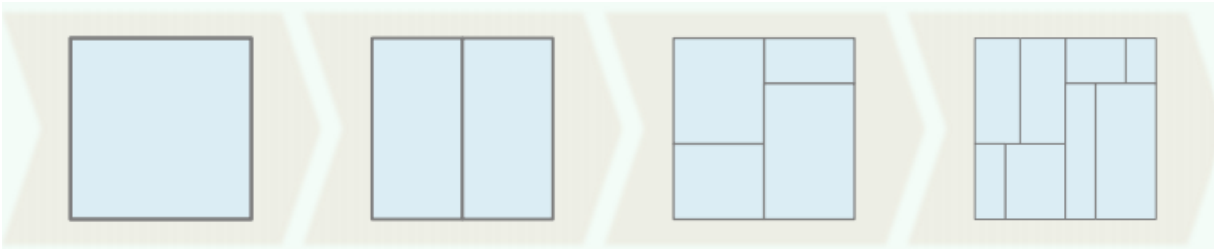
> Starting from the LCA (lowest common ancestor), traverse path(16) and path(78) once more resp.
> All R/L-turns along path(16)/path(78) are ignored and the R/L subtree at each L/R-turn is reported



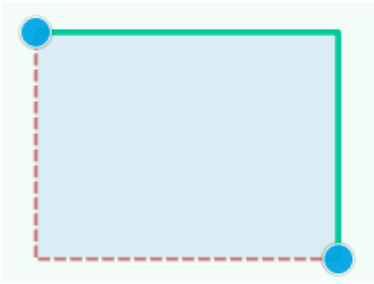
Complexity:
Query: $O(logn)$
Preprocessing: $O(nlogn)$
Storage: $O(n)$

2. kd-Tree: 2D

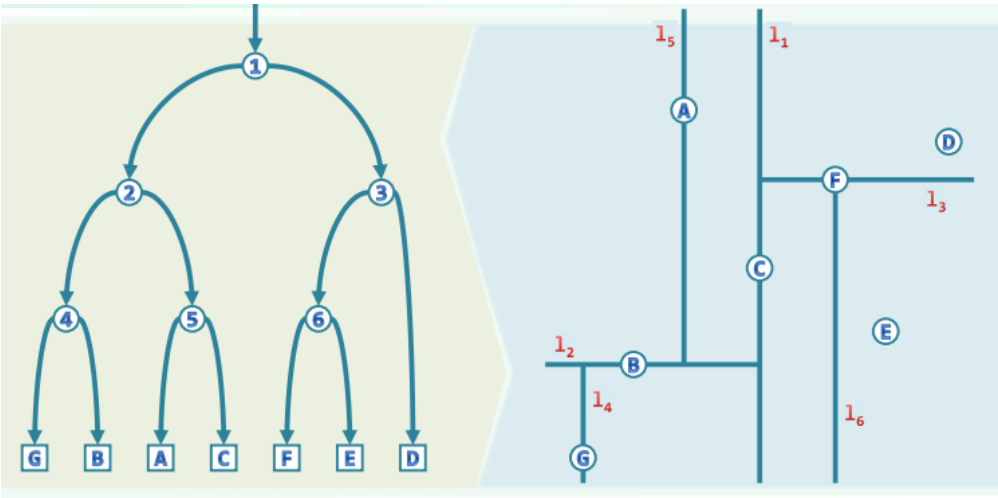
To extend the BBST method to planar GRS, we
- **divide** the plane recursively and
- **arrange** the regions into a kd-tree



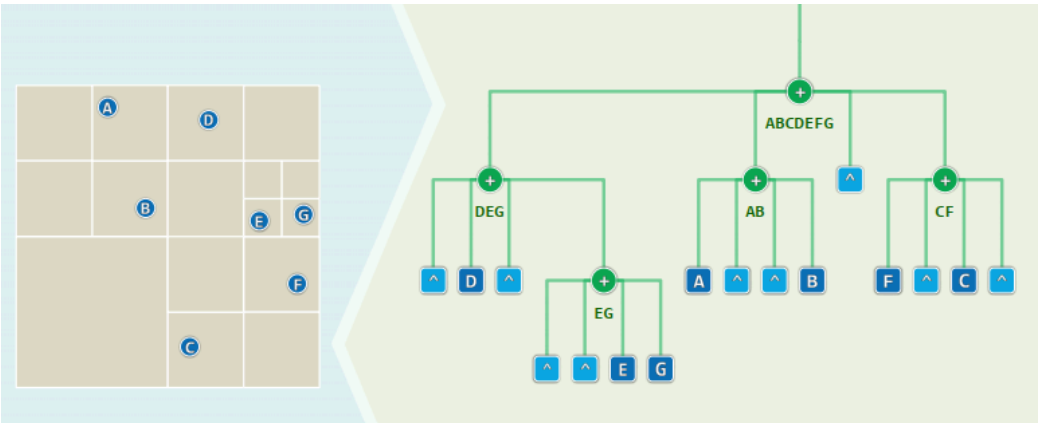
Definition: each region is defined to be **open/closed** on the **left-lower/right-upper** sides



Example:



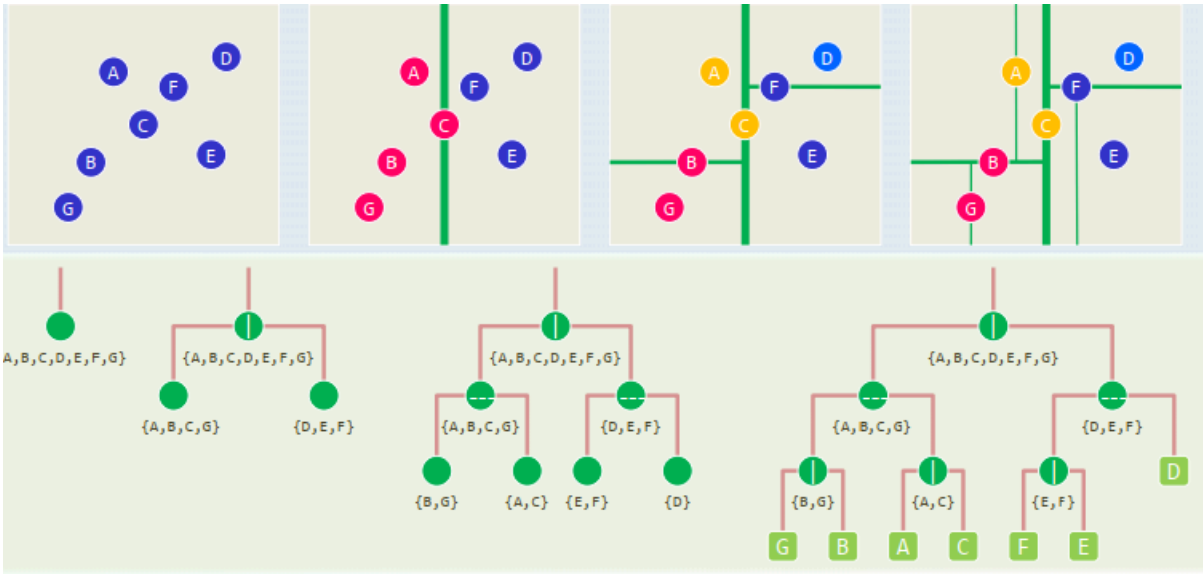
Quadtree



- **buildKdTree(P, d)**

  这个实现方式不是很好，参考PA3-5

```
// construct a 2d-(sub)tree for point (sub)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 )
```



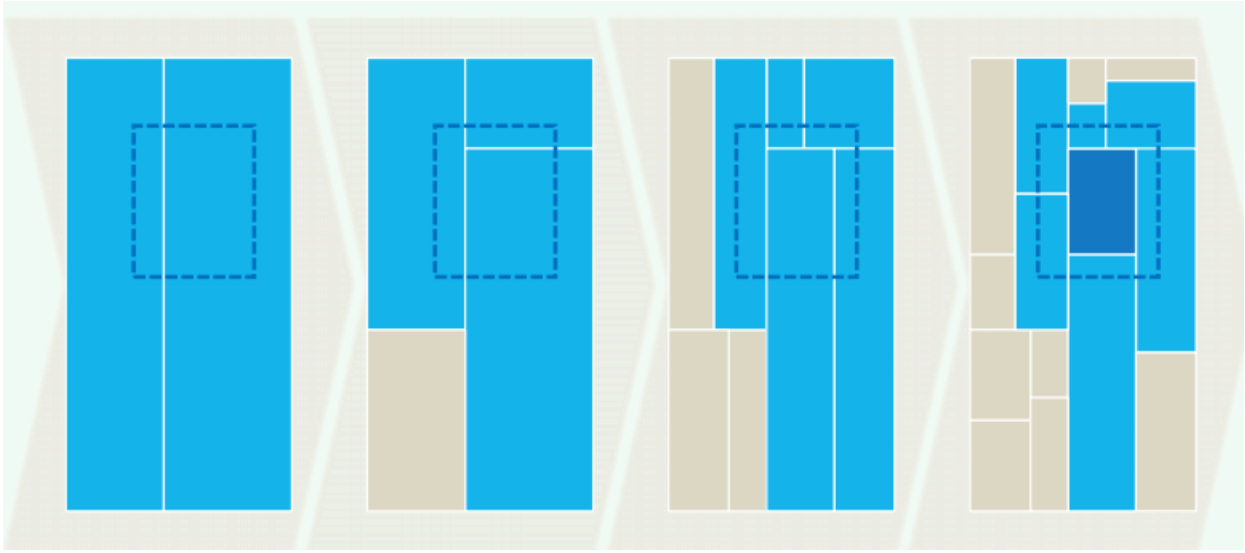
> each **node** => each rectangular **sub-region** of the plane
> each of these subsets is called a **canonical subset**
> for each internal node X with children L and R,
> $region(X) = region(L) \cup region(R)$
> sub-regions of nodes at a same depth
> never **intersect** with each other and their **union** covers the entire plane

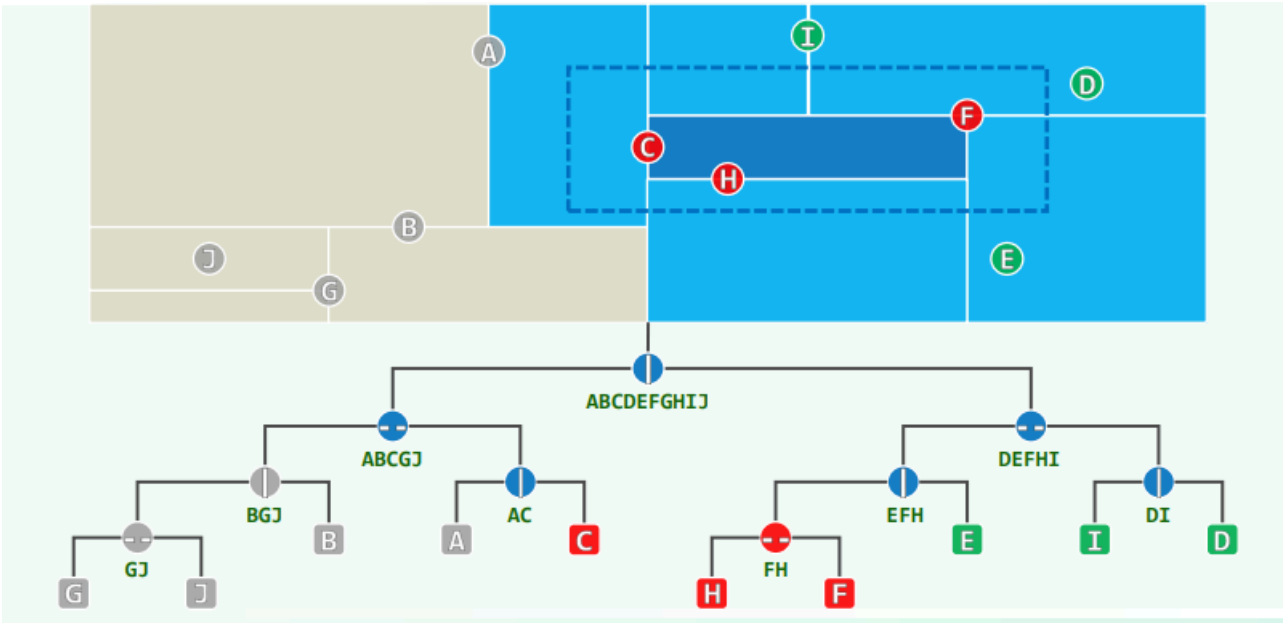
- **Query**

```
if ( isLeaf( v ) )
    if ( inside( v, R ) ) report(v)
        return

if ( region( v->lc ) ⊆ R )
    reportSubtree( v->lc )
else if ( region( v->lc ) ∩ R ≠ Φ)
    kdSearch( v->lc, R )

if ( region( v->rc ) ⊆ R )
    reportSubtree( v->rc )
else if ( region( v->rc ) ∩ R ≠ Φ)
    kdSearch( v->rc, R )
```





3. Complexity

- **Preprocessing**

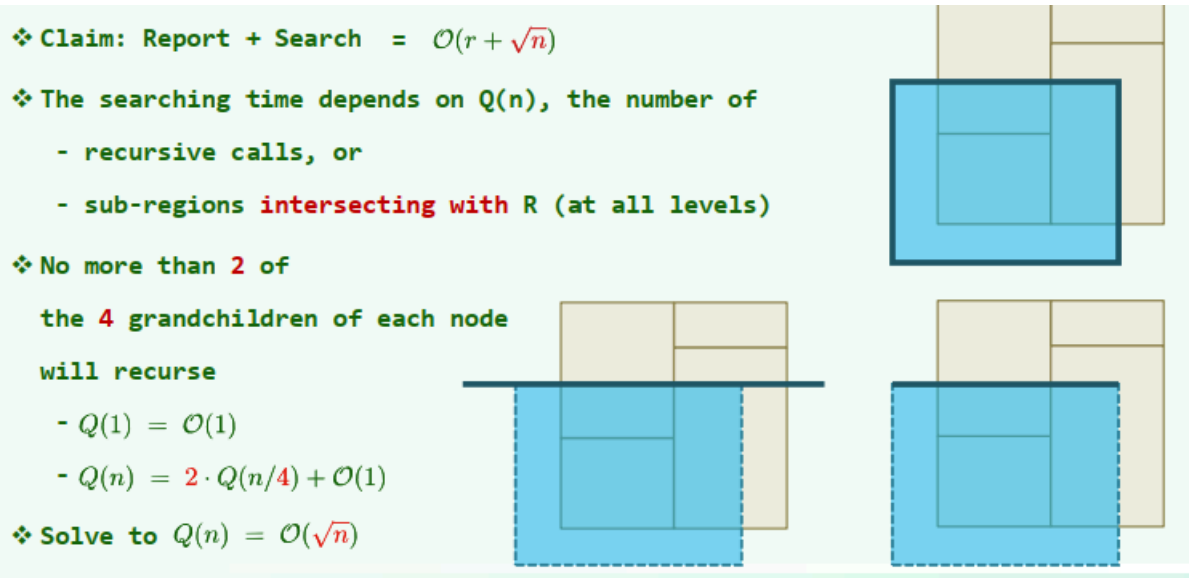
  $T(n) = 2 * T(n/2) + O(n) = O(nlogn)$

- **Storage**

  the tree has a height of $O(logn)$, Storage = $O(n)$

- **Query Time**

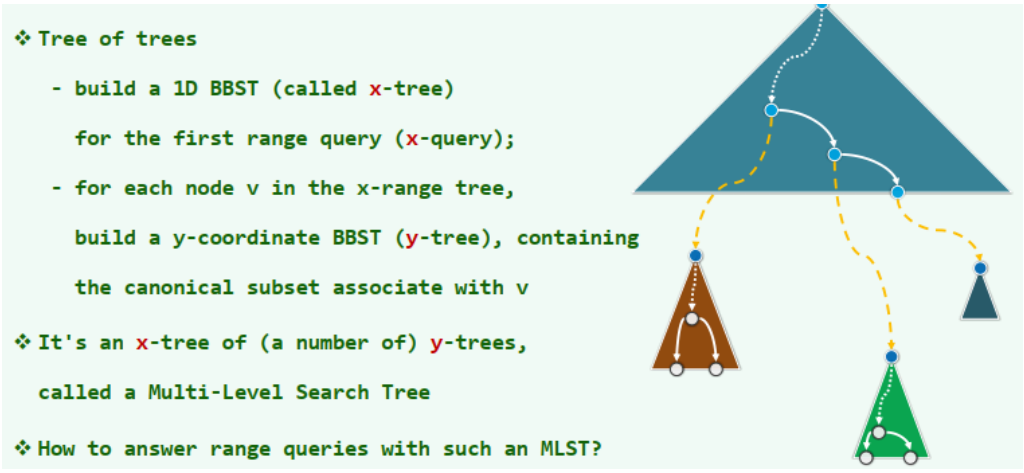
  $O(r + \sqrt{n})$



beyond 2D: $O(r+n^{1-\frac{1}{d}})$

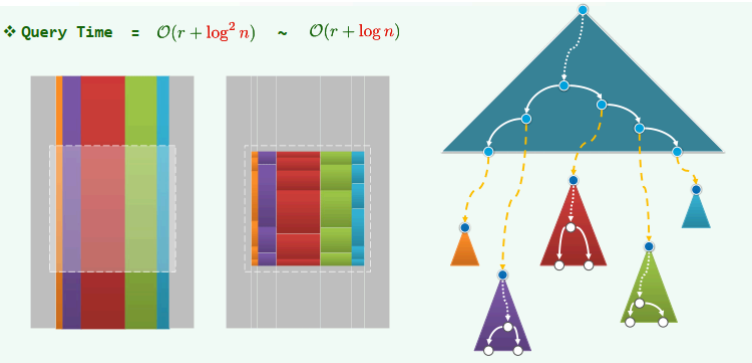
Multi-Level Search Tree

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



Complexity:
Query Time = $O(r+log^2n) \sim O(r+logn)$

> **2-level search tree**:
> build-tree in $O(nlogn)$ time
> each input point is stored in $O(logn)$ y-trees
> needs $O(nlogn)$ space
>
> **x**-range query needs $O(logn)$ time to locate the $O(logn)$ nodes
> then for each of these nodes, a **y**-range search is invoked, which needs $O(logn)$ time



- Query Algorithm

> Determine the canonical subsets of points that satisfy the first query.
> Find out from each canonical subset which points lie within the **y**-range.

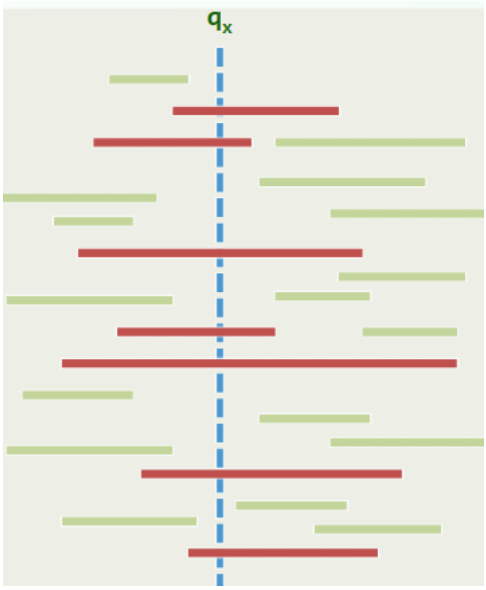
> **Beyond 2D**:
> a d-level tree for S uses $O(n \cdot log^{d-1} n)$ storage
> can be constructed in $O(n \cdot log^{d-1} n)$ time
> each orthogonal range query of S can be answered in $O(r + log^d n)$ time

Range Tree

Interval Tree

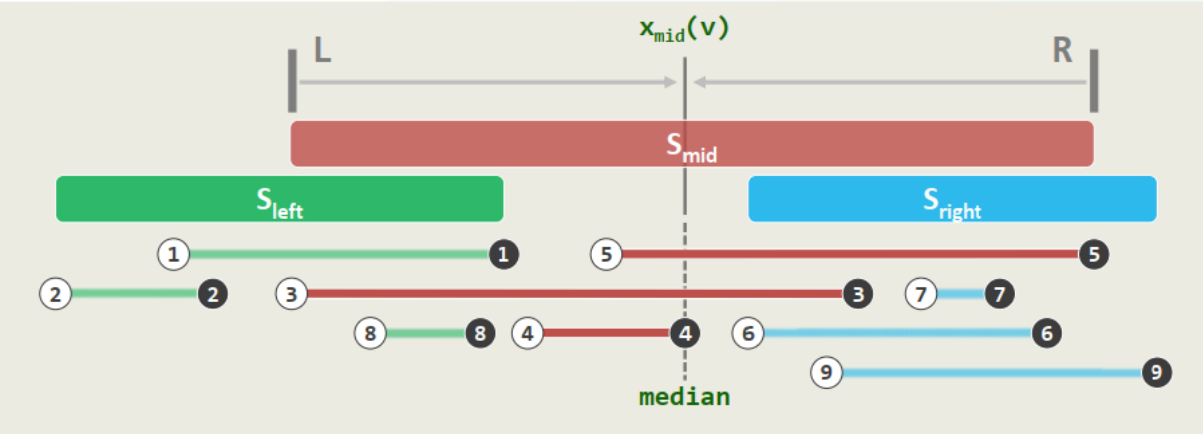
Stabbing Query:

> Given a set of intervals in general position on the **x**-axis: $S = \{s_i = [x_i, x_i' | 1 \leq i \leq n]\}$ and a query point q_x
> Find all intervals that contain q_x: $\{s_i = [x_i, x_i'] | x_i \leq q_x \leq x_i'\}$



1. Partitioning

> **Median**:
> Let $P = \partial S$ be the set of all endpoints
> Let $x_{mid} = median(P)$ be the median of P



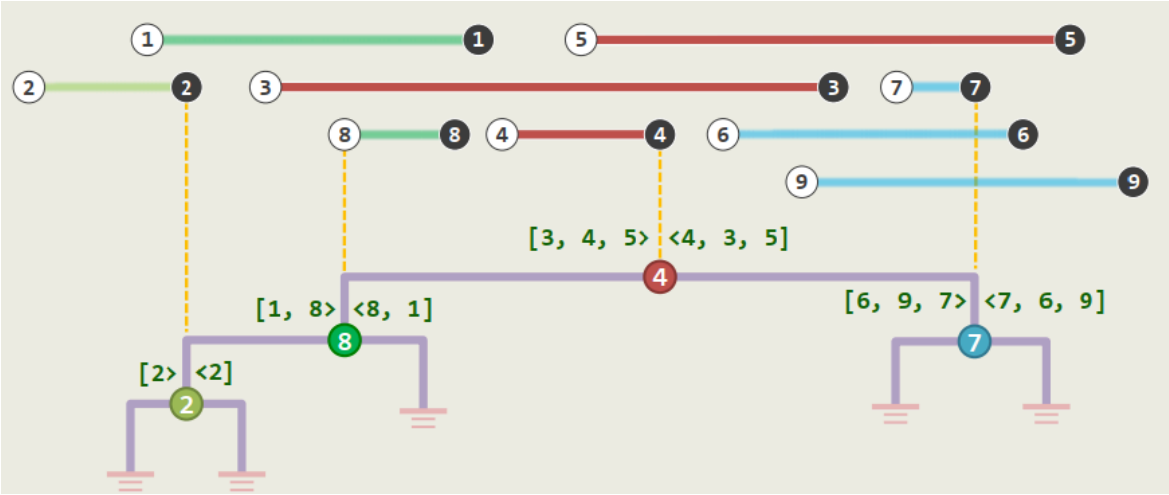
- All intervals can be then categorized into 3 subsets:
  $S_{left} = \{ S_i \mid x_i' < x_{mid} \}$ $S_{mid} = \{ S_i \mid x_i \le x_{mid} \le x_i' \}$ $S_{right} = \{ S_i \mid x_{mid} < x_i \}$
- $S_{left/right}$ will be recursively partitioned until they are empty (leaves)

> $max\{|S_{left}|, |S_{right}|\} \le n/2$
> Best case: $|S_{mid}| = mid$
> Worst case: $|S_{mid}| = 1$

2. Construction

Complexity: $O(nlogn)$
Storage: $O(n)$

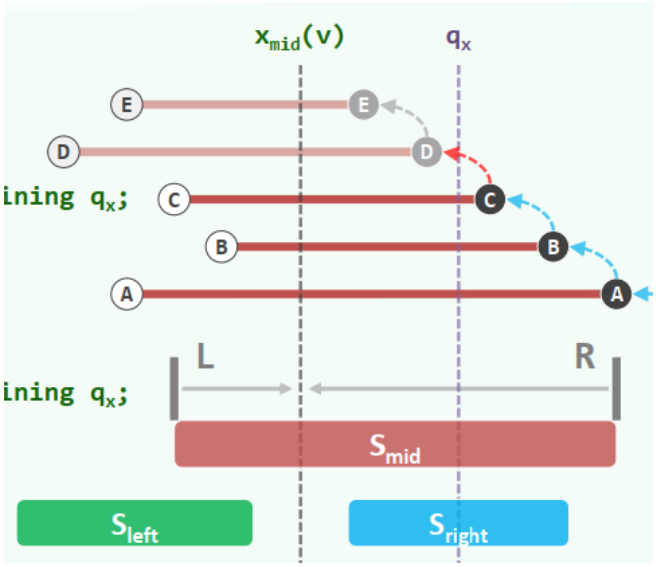
Hint: avoid repeatedly sorting

> each segment appears twice (one in each list)
>
> 
>

3. queryIntervalTree

```
if (!v) return; //base
if ( q_x < x_mid(v) )
    report all segments of S_mid(v) containing q_x;
    queryIntervalTree( lc(v), q_x );
else if ( x_mid(v) < q_x )
    report all segments of S_mid(v) containing q_x;
    //（优化）preprocessing: 对右端点进行排序
    queryIntervalTree( rc(v), q_x );
else //with a probability ≈ 0
```

```
    report all segments of S_mid( v ); //both rc(v) & lc(v) can be ignored
```



Query Time: $O(r + logn)$

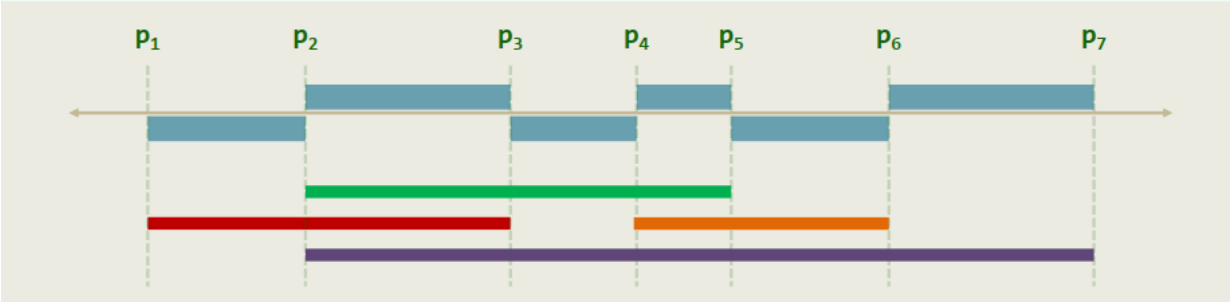
Segment Tree

> Stabbing Query

Discretization 离散化

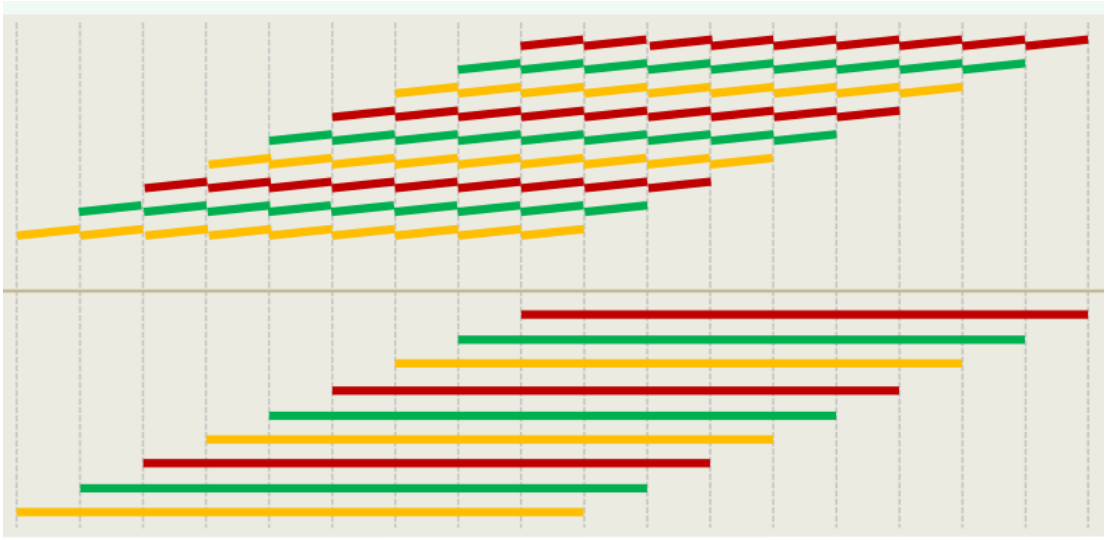
> Let $I = \{[x_i, x_i'] | i = 1, 2, 3, \ldots, n\}$ be n intervals on the x-axis
> sort all the endpoints into $p_1, p_2, p_3, \ldots, p_m, m \leq 2n$
> $m + 1$ **elementary intervals** are hence defined as: $(-\infty, p_1], (p_1, p_2], \ldots, (p_{m-1}, p_m], (p_m, +\infty)$

```
Within each EI, all stabbing queries share a same output
∴ If we sort all EI's into a vector and
  store the corresponding output with each EI, then ...
```




```
∴ Once a query position is determined,  //by an O(logn) time binary search
  the output can then be returned directly  //O(r)
```

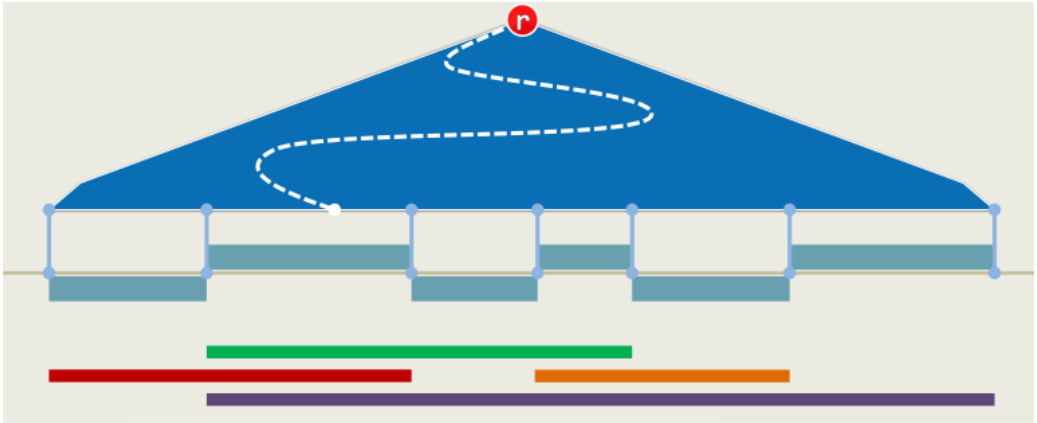
> **Worst Case**:
> every interval spans $\Omega(n)$ EI's and a total space of $\Omega(n^2)$ is required



Sorted Vector --> BBST

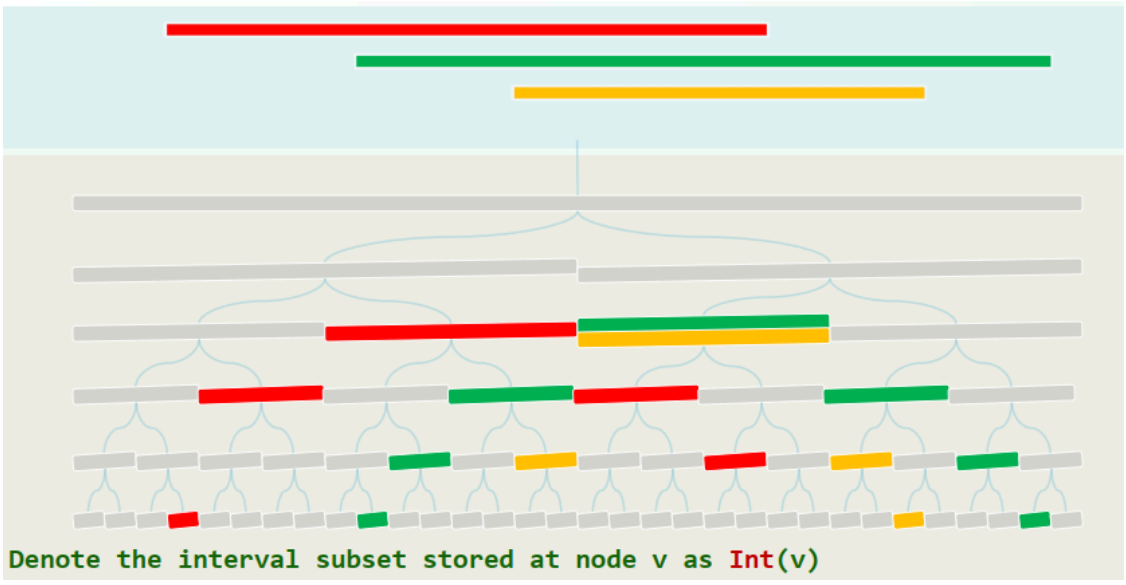
For each leaf v,
denote the corresponding elementary interval as $R(v)$ //range of domination
denote the subset of intervals spanning $R(v)$ as $Int(v)$ and store $Int(v)$ at v

To find all intervals containing q_x, we can
find the leaf v whose $R(v)$ contains q_x //$O(logn)$ time for a BBST
and then report $Int(v)$ //$O(1+r)$ time



Store each interval at $O(logn)$ common ancestors by greedy merging

Canonical Subsets with $O(nlogn)$ Space



1. BuildSegementTree

> Sort all endpoints in I before determining all the EI's: $O(nlogn)$
> 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)**

```
//InsertSegment(v, s)
if ( R(v) ⊆ s ) //greedy by top-down
    store s at v and return;
if ( R( lc(v) ) ∩ s ≠ Φ ) //recurse
    InsertSegment( lc(v), s );
if ( R( rc(v) ) ∩ s ≠ Φ ) //recurse
    InsertSegment( rc(v), s );

At each level, ≤ 4 nodes are visited (2 stores + 2 recursions)
// O(logn) time
```

2. Query

```
//Query(v, q_x)
report all the intervals in Int(v)
if ( v is a leaf )
    return
if ( qx ∈ Int( lc(v) ) )
    Query( lc(v), qx )
else
    Query( rc(v), qx )
```

> **Complexity**:$O(r + logn)$
> Only one node is visited per level, altogether $O(logn)$ nodes
> At each node v
> the CS $Int(v)$ is reported in time: $1 + |Int(v)| = O(1 + rv)$
> Reporting all the intervals: costs $O(r)$ time

3. Conlusion

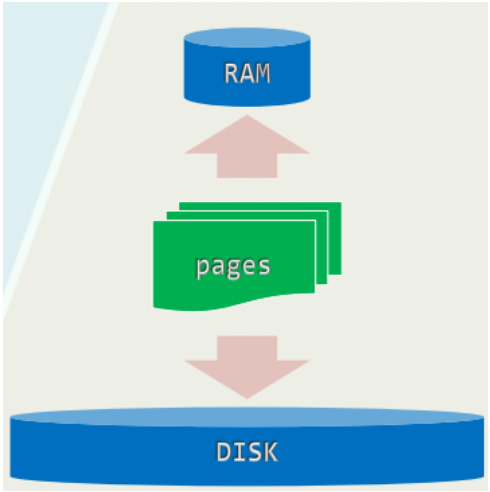
a segment tree of **size**: $O(nlogn)$
can be built in $O(nlogn)$ **time**
which reports all intervals containing a query point in $O(r+logn)$ **time**

8 高级搜索树

存储器现状

> **CPU Register**: [cycles] = 0, [sec] = ns
> **SRAM/cache**: [cycles] = 4~75, [sec] = ns
> **DRAM/main memory**: [cycles] = 10^2, [sec] = ns
> **DISK**: [cycles] = 10^7, [sec] = ms
>
> 不同类型的存储器，容量、访问速度差异悬殊
> 实际运行时间主要取决于：相邻存储级别之间数据传输（I/O）的**速度**与**次数**
>
> Definition（内存 / 外存）：所有相对于当前存储器级别更低的都叫做“外存”，反之称为“内存”
>
> **分级存储**：批量访问
> 以**页**（page）或**块**（block）为单位，借助缓冲区，可大大缩短字节的**平均**访问时间

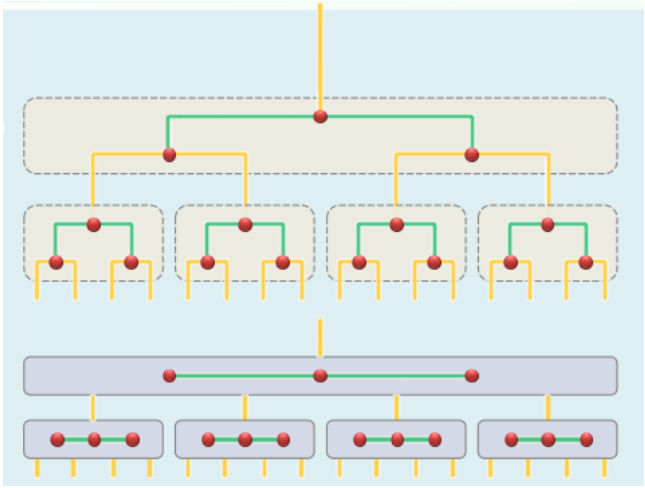
```
#include <cstdio>
#define BUFSIZ 512 //缓冲区默认容量
int setvbuf ( //定制缓冲区
    FILE* fp, //流
    char* buf, //缓冲区
    int _Mode, //_IOFBF | _IFLBF | _IONBF
    size_t size); //缓冲区容量
int fflush(FILE* fp); //强制清空缓冲区
```



8.1 B树

基本结构

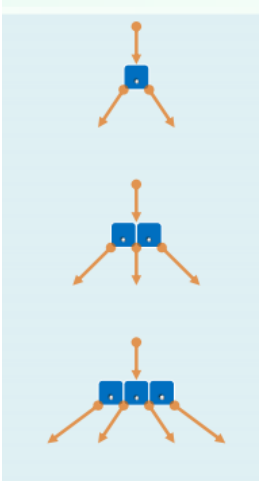
每**d代**合并为**超级节点**，$m=2^d$路，$m-1$个关键码——逻辑上与BBST**完全等价**



> **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
> 普通AVL：若有$n = 1G$个记录，每次查找需要$log_2 10^9 \approx 30$次I/O操作
> B树：充分利用外存的**批量访问**，每下降一层，都以**超级节点**为单位，读入一组关键码
> 目前多数数据库系统采用$m = 200 - 300$。取$m = 256$，则每次查找只需要$log_{256} 10^9 \leq 4$次I/O

$([m/2], m)$-树：以$(2, 4)$-树为例

> 各节点的分支数，可能是2,3或4
> 各节点所含key的数目，可能是1,2或3

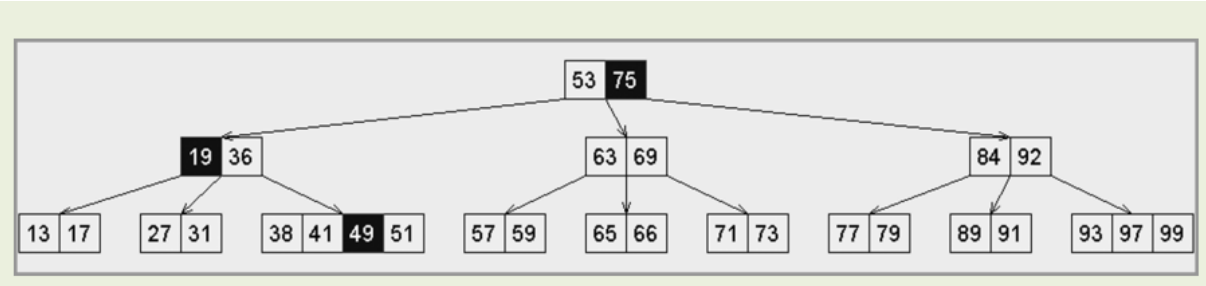


```
//BTNode
template <typename T> struct BTNode { //B-树节点
    BTNodePosi<T> parent; //父
    Vector<T> key; //关键码（总比孩子少一个）
    Vector< BTNodePosi<T> > child; //孩子

    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
    }
};
//BTree
template <typename T> using BTNodePosi = BTNode<T>*; //B-树节点位置
```

```
template <typename T>
class BTree { //B-树
protected:
    int _size; int _m; BTNodePosi<T> _root; //关键码总数、阶次、根
    BTNodePosi<T> _hot; //search()最后访问的非空节点位置
    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
};
```

1. 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
}
```

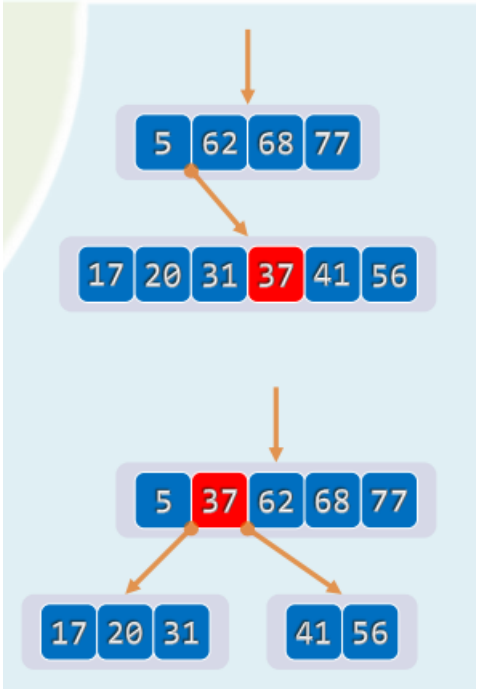
最大树高：含N个关键码的m阶B树，高度$h \leq O(log_m N)$

> 推导：$log_m(N+1) \leq h \leq 1 + [log_{m/2}\frac{N+1}{2}]$
> 取$m = 256$，树高约降低至$\frac{1}{7}$ ~ $\frac{1}{8}$

2. 插入

```
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
}
```

分裂：关键码上升一层，并分裂以所得的两个节点为左右孩子



再分裂：若上溢节点的父亲本就饱和，则在接纳被提升的关键码之后也将上溢，逐层**向上传播**，总体执行时间正比于分裂次数$O(h)$

分裂至根节点：B树高度增加，且新生的树根仅有**两个**分支

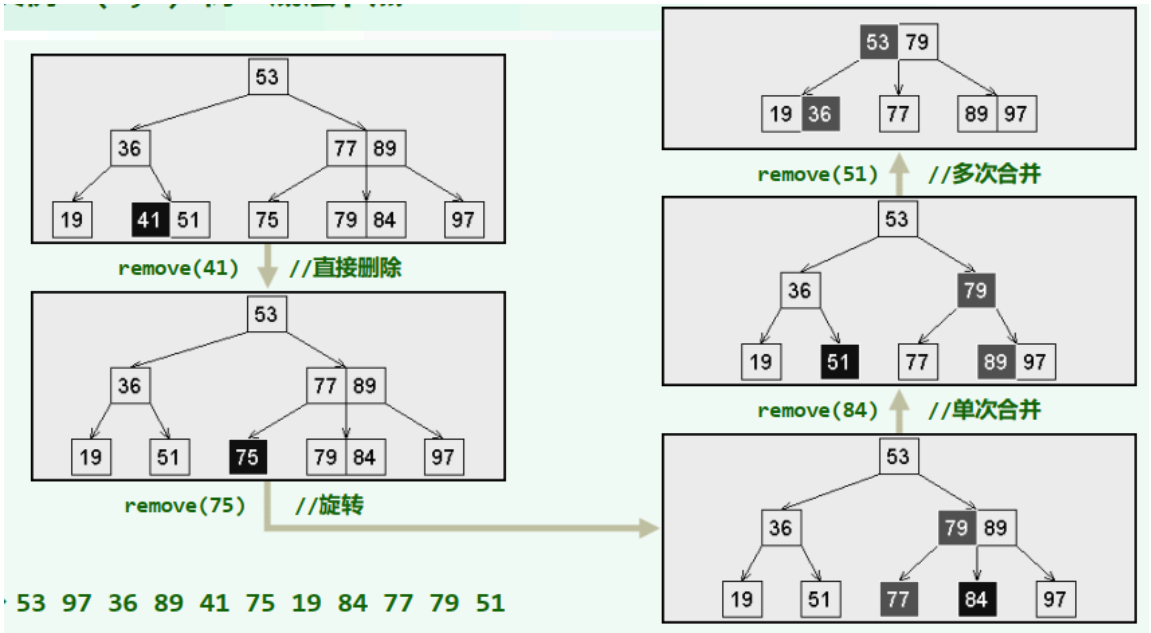
```
//上溢修复
template <typename T>
void BTree<T>::solveOverflow( BTNodePosi<T> v ) {
    while ( _m <= v->key.size() ) { //除非当前节点不再上溢
        Rank s = _m / 2; //轴点（此时_m = key.size() = child.size() - 1）
        BTNodePosi<T> u = new BTNode<T>(); //注意：新节点已有一个空孩子
        for ( Rank j = 0; j < _m - s - 1; j++ ) { //分裂出右侧节点u（效率低可改进）
            u->child.insert( j, v->child.remove( s + 1 ) ); //v右侧_m-s-1个孩子
            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 );
        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
```

3. 删除

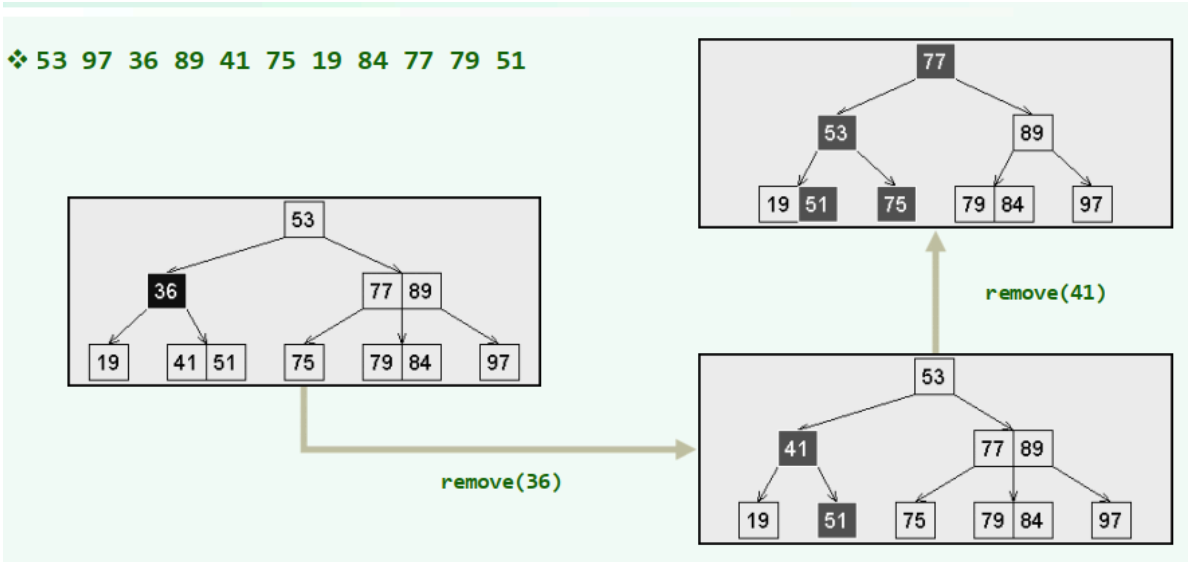
```
//确保目标在叶子中
```

```
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;
    }
        //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
}
```

$(2,3)$-树：底层节点



$(2,3)$-树：非底层节点



代码实现：

下溢修复 713

```
template <typename T> void BTree<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
```

下溢修复：情况#1：旋转（向左兄弟借关键码） 714

```
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
```

下溢修复：情况#3：合并（1/2） 715

```
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;
   /* ... TBC ... */
```

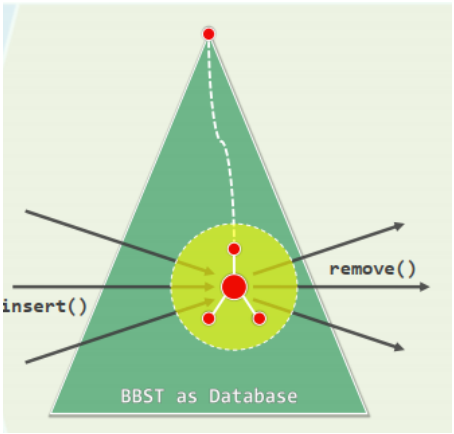
下溢修复：情况#3：合并（2/2） 716

```
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 */ }
```

思考：B树的插入删除优先策略非对称（插入：split，删除：rotate 而不是 merge）

8.2 红黑树 Red-Black Tree

> **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
> Red-Black Tree保证无论insert / remove，结构变化均不超过$O(1)$



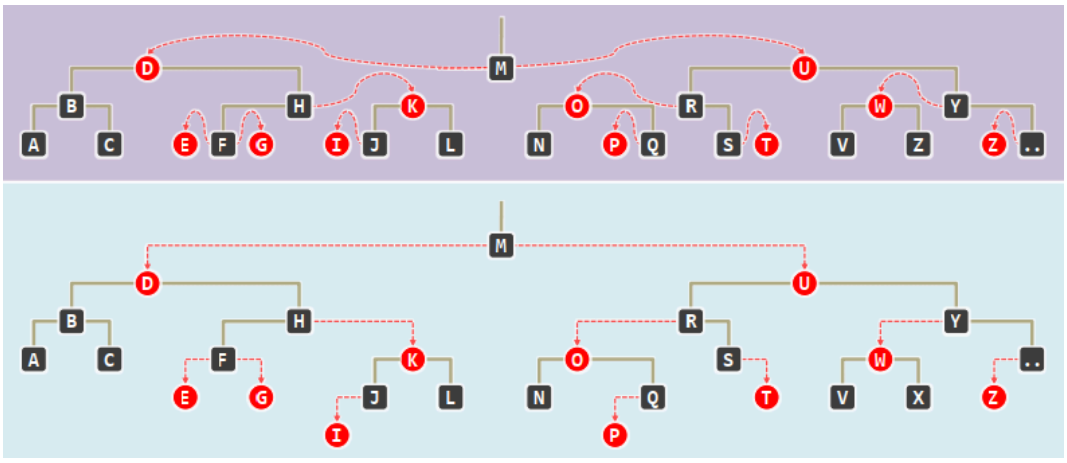
> **持久性 Persistent Structures**：支持对历史版本的访问
> Partial Persistence：仅支持对历史版本的**读取**
> 每个版本的新增复杂度，仅为$O(logn)$，甚至$O(1)$...!

1. 结构

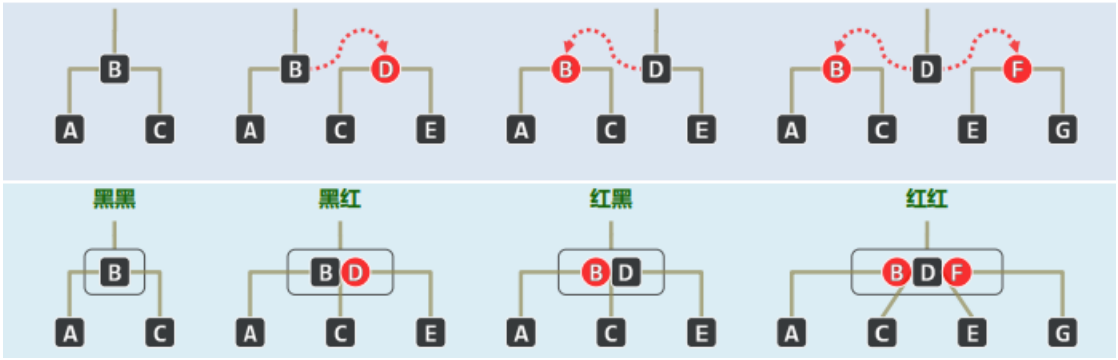
由红、黑两类节点组成的BST，统一增设外部节点NULL，使之成为**真二叉树**

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
>
> *（一种直观理解方式：）*



于是，每棵红黑树都对应于一棵**（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**（无非四种组合）



红黑树 ∈ BBST

- 包含n个内部节点的红黑树T，高度$h = O(logn)$，实际上有$log_2(n+1) \le h \le 2 \cdot log_2(n+1)$
- 若T高度为h，红/黑高度为R/H，则$H \le h \le R + H \le 2 \cdot H$
  其中，若T所对应的B-树为T_B，则H即是T_B的高度，且T_B的每个节点，都**恰好**包含T的**一个**黑节点

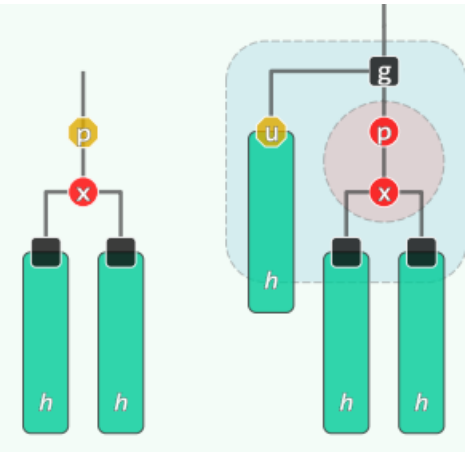
```
template <typename T> class RedBlack : public BST<T> { //红黑树
public: //BST::search()等其余接口可直接沿用
    BinNodePosi<T> insert( const T & e ); //插入（重写）
    bool remove( const T & e ); //删除（重写）
protected:
    void solveDoubleRed( BinNodePosi<T> x ); //双红修正
    void solveDoubleBlack( BinNodePosi<T> x );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

```
    int updateHeight( BinNodePosi<T> x ); //更新节点x的高度（重写）
};

template <typename T>
int RedBlack<T>::updateHeight( BinNodePosi<T> x ) {
    return x->height = IsBlack( x ) + max( stature( x->lc ), stature( x->rc ) );
}
```

2. 插入

> 首先按照BST规则插入关键码e，首先将x染红，可能违反规则3：**双红 (double-red)**
> 考察：祖父`g = p->parent`和叔父`u = uncle(x) = sibling(p)`

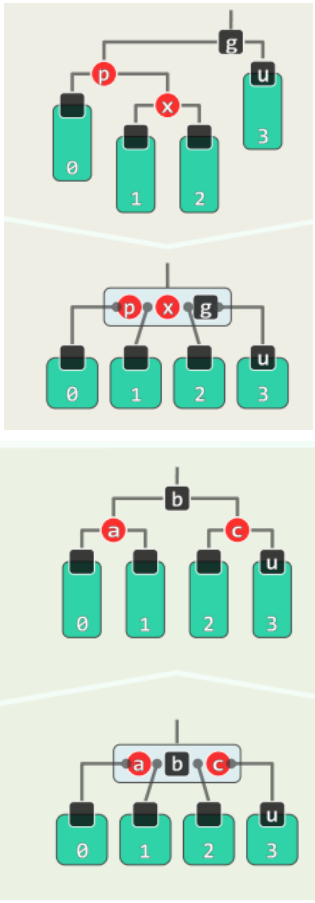



```
template <typename T> BinNodePosi<T>
RedBlack<T>::insert( const T & e ) {
    // 确认目标节点不存在（留意对_hot的设置）
    BinNodePosi<T> & x = search( e ); if ( x ) return x;
    // 创建红节点x，以_hot为父，黑高度 = 0
    x = new BinNode<T>( e, _hot, NULL, NULL, 0 ); _size++;
    // 如有必要，需做双红修正，再返回插入的节点
    BinNodePosi<T> xOld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

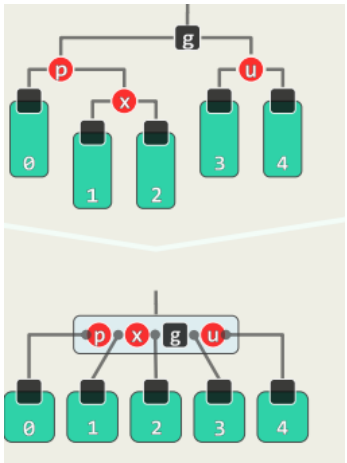
template <typename T>
void RedBlack<T>::solveDoubleRed( BinNodePosi<T> x ) {
    if ( IsRoot( *x ) ) //若已（递归）转至树根，则将其转黑，整树黑高度也随之递增
        { _root->color = RB_BLACK; _root->height++; return; } //否则...
    BinNodePosi<T> p = x->parent; //考查x的父亲p（必存在）
    if ( IsBlack( p ) ) return; //若p为黑， 则可终止调整；否则
    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）
        // 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
```

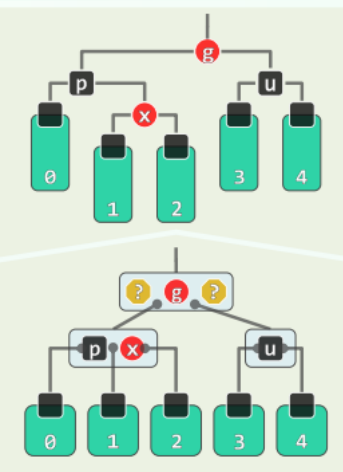
```
    }
    else { //u为红
        p->color = RB_BLACK; p->height++; //p由红转黑，增高
        u->color = RB_BLACK; u->height++; //u由红转黑，增高
        g->color = RB_RED; //在B-树中g相当于上交给父节点的关键码，故暂标记为红
        solveDoubleRed( g ); //继续调整：若已至树根，接下来的递归会将g转黑（尾递归）
    }
}
```

- **RR-1：u->color == B**
  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
  此时，进行局部“3+4”重构，将三个关键码的颜色改为**RBR**即可



- **RR-2：u->color == R**
  在B-树中，等效于超级节点发生**上溢**
  `p`与`u`转黑，`g`转红，节点**分裂**，关键码`g`上升一层





如果“双红”调整不断向上传递到树根，则强行将`g`转为黑色，整棵树的黑高度加一。

- **复杂度分析**

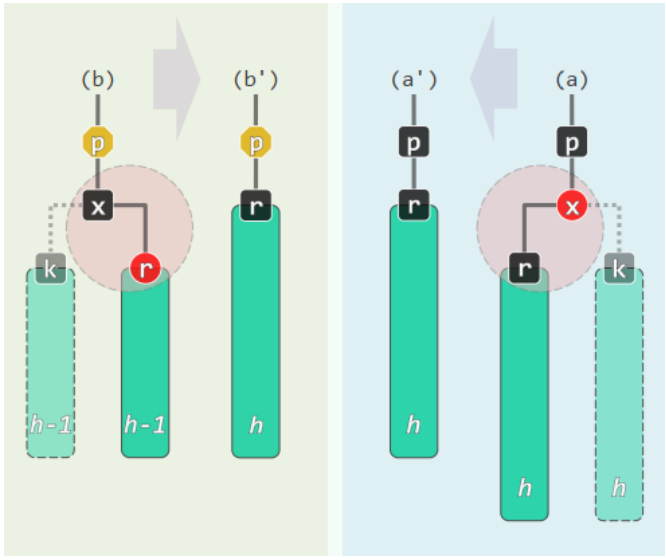
  重构、染色均只需常数时间，故只需统计其总次数

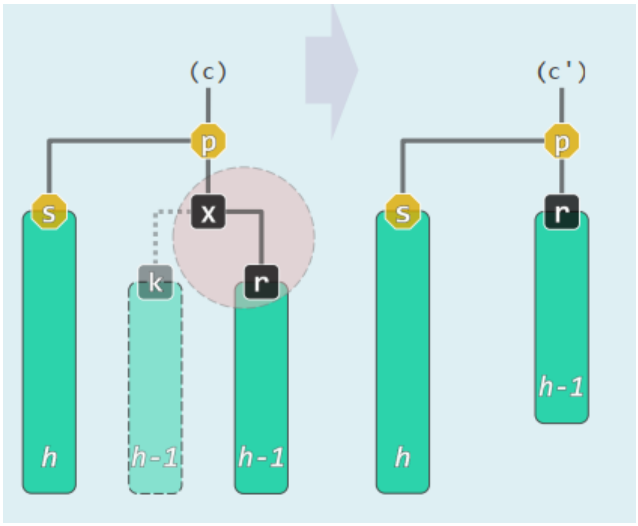
  `RedBlack::insert()`仅需$O(logn)$时间，期间至多做$O(logn)$次染色、$O(1)$次旋转

	旋转	染色	此后
u为黑	1~2	2	调整随即完成
u为红	0	3	可能再次双红 但必上升两层

3. 删除

> 首先按照BST常规算法，执行`r = removeAt(x, _hot)`
> `x`由孩子`r`接替，此时另一孩子`k`必为NULL。假想将另一孩子`k`理解为一棵黑高度与`r`相等的子树，且随`x`一并摘除。
> 可能违反规则3、4：
> 若`x`为红，则自然满足
> 若`r`为红，则令其与`x`交换颜色即可
> 若`x`与`r`**双黑 (double black)**，摘除`x`并代之以`r`后，全树黑深度不再统一（相当于B-树中`x`所属节点发生**下溢**，考察`r`的父亲`p = r->parent`、兄弟`s = sibling(r)`






```
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
    if ( BlackHeightUpdated( * _hot ) )
        return true;  //若父亲（及祖先）依然平衡，则无需调整
    // 至此，必失衡
    if ( IsRed( r ) ) {  // 若替代节点r为红，则只需简单地翻转其颜色
        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;
}
```

```
template <typename T>
```

```
void RedBlack<T>::solveDoubleBlack( BinNodePosi<T> r ) {
    BinNodePosi<T> p = r ? r->parent : _hot; 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->lc ) ) t = s->lc;
        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->lc->color = RB_BLACK;
                updateHeight( b->lc );
            }
            if (HasRChild( *b )) {
                b->rc->color = RB_BLACK;
                updateHeight( b->rc );
            }
            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
            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 { //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，故只能是BB-1或BB-2R
    }
}
```

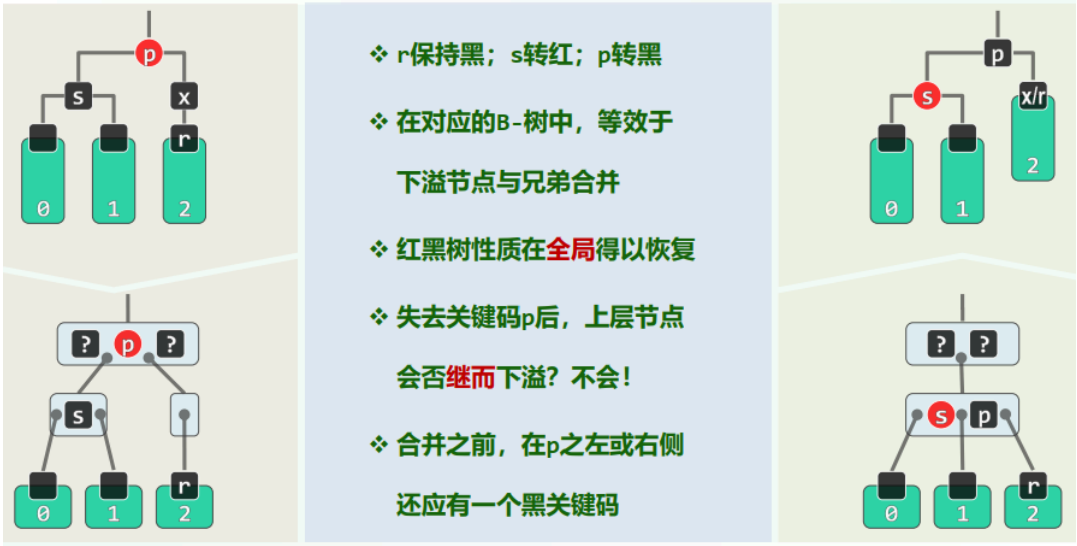
- **BB-1：s为黑，且至少有一个红孩子t**

  “3+4”重构：`r`保持黑，`a`，`c`染黑，`b`继承`p`的原色



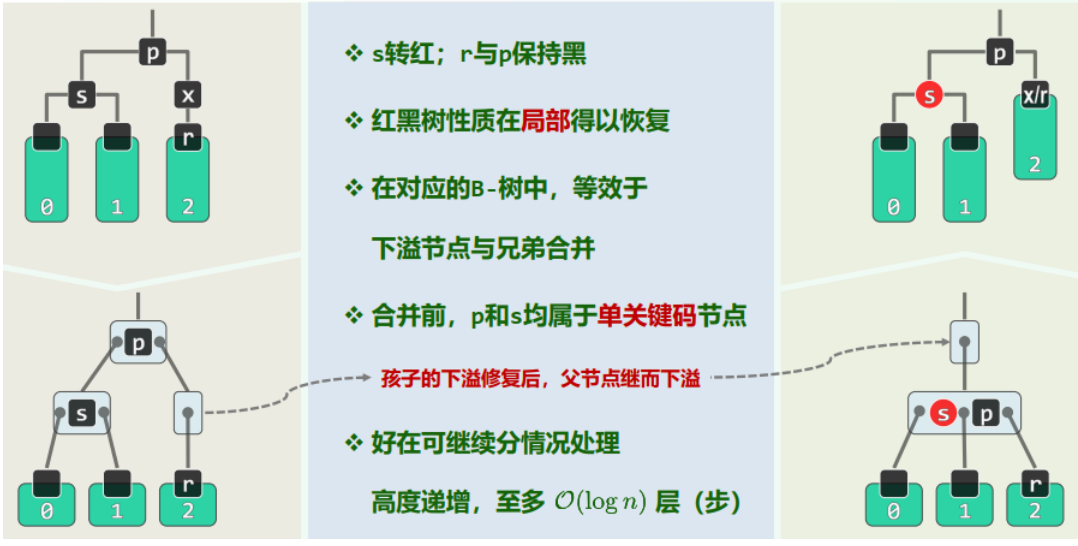
- **BB-2：s为黑，且两个孩子均为黑**
  - **p为红（BB-2R）**

    `r`保持黑，`s`转红，`p`转黑。且能保证失去关键码`p`之后，上层节点不会继续下溢，这是因为合并之前在`p`之左或者右侧还应有一个黑关键码



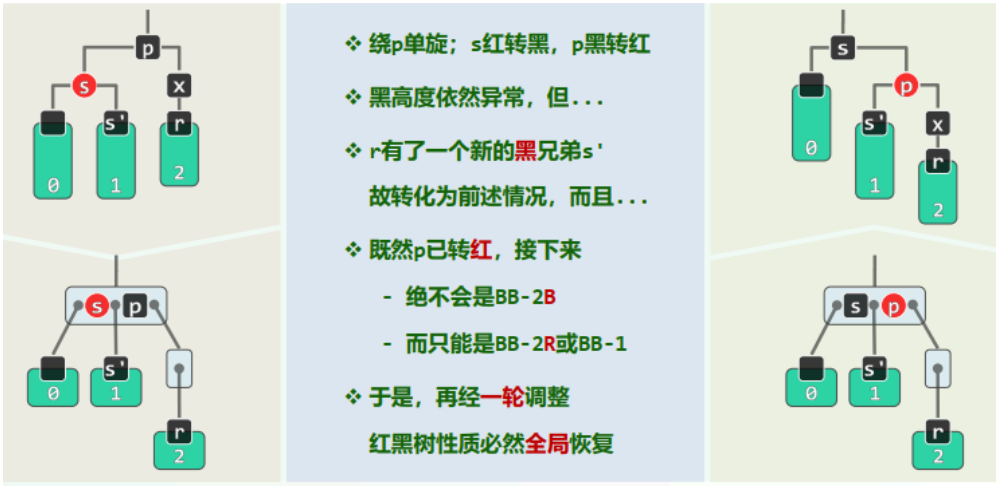
  - **p为黑（BB-2B）**

    `r`与`p`保持黑，`s`转红。孩子的下溢修复后，父节点继而下溢。



- **BB-3：s为红（其孩子均为黑）**

  `s`红转黑，`p`黑转红（绕`p`单旋）。此时`r`有了一个新的黑兄弟`s'`，故而转化为前述情况。



- **复杂度分析**

  仅需$O(logn)$时间，其中$O(logn)$次重染色，$O(1)$次旋转

	旋转	染色	此后
（1）黑s有红子t	1~2	3	调整随即完成
（2R）黑s无红子，p红	0	2	调整随即完成
（2B）黑s无红子，p黑	0	1	必再次双黑，但将上升一层
（3）红s	1	2	转为(1)或(2R)

8.3 伸展树 Splay Tree

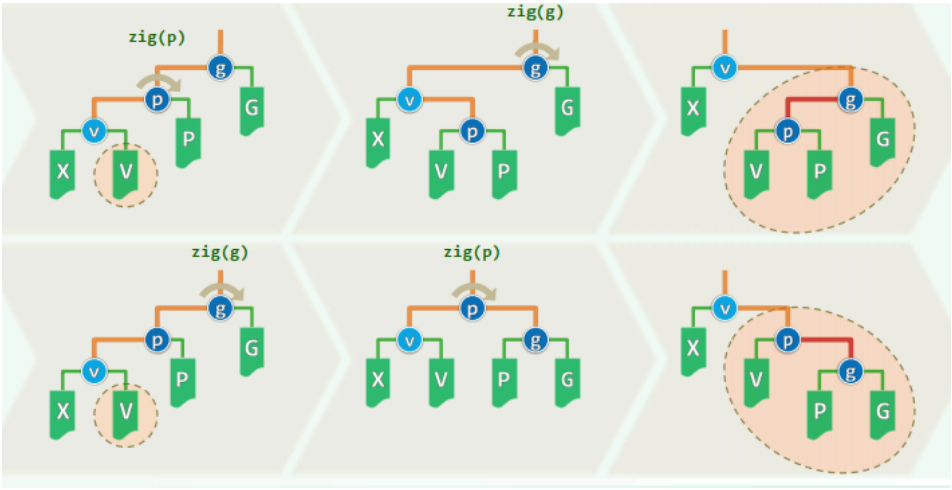
局部性 / Locality：刚被访问的**数据**，极有可能**很快**地**再次**被访问

伸展树：

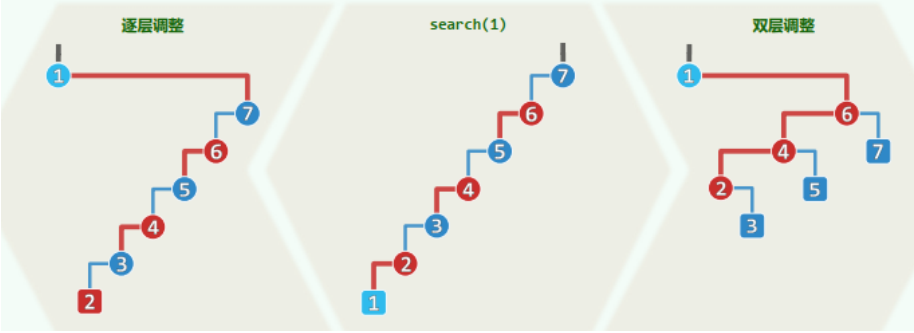
> **自适应链表** self-adjusting list：节点一旦被访问，随即移到**最前端**
> **伸展树** self-adjusting binary tree：逐层伸展，使得BST的节点一旦被访问，随即调整到**树根**（或其附近），（实现方法：zig + zag）
>
> Worst-Case: 倒序访问退化为链表的顺序树，$\Omega(n^2)$，分摊$\Omega(n)$

1. 双层伸展

> 反复考察**祖孙三代**，根据它们的相对位置，经**两次旋转**，使v上升两层，称为（子）树根
> *（下图中第一行为“俗”办法，第二行为“好”办法）*



这种结构可以使得最坏的情况并不**持续性发生**（如下图所示，节点访问之后，对应路径的长度随机**折半**）。在这种情况下，伸展操作分摊仅需$O(logn)$。



要是v只有父亲，没有祖父，此时必有$v.parent() == T.root()$，只需要做**单次**旋转。但是这种情况最多在最后出现**一次**

2. 数学证明

定义伸展树S的势能：

$$\phi(S) = log(\Pi_{v\in S} size(v)) = \sum_{v\in S} log(size(v)) = \sum_{v\in S} rank(v) = \sum_{v\in S} logV$$

直觉：越**平衡/倾斜**的树，势能越**小/大**

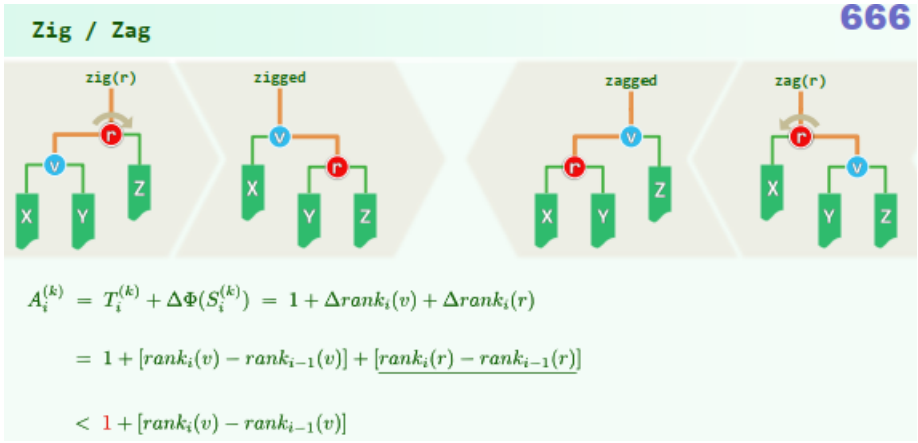
单链：$\phi(S) = logn! = O(nlogn)$
满树：$\phi(S) = O(n)$

$$\begin{aligned}\Phi(S) &= \log\prod_{d=0}^{n}(2^{h-d+1}-1)^{2^d} \le \log\prod_{d=0}^{n}(2^{h-d+1})^{2^d} \\ &= \log\prod_{d=0}^{h}2^{(h-d+1)\cdot 2^d} = \sum_{d=0}^{h}(h-d+1)\cdot 2^d = (h+1)\cdot\sum_{d=0}^{h}2^d - \sum_{d=0}^{h}d\cdot 2^d \\ &= (h+1)\cdot(2^{h+1}-1) - [(h-1)\cdot 2^{h+1}+2] = 2^{h+2}-h-3 = \mathcal{O}(n)\end{aligned}$$

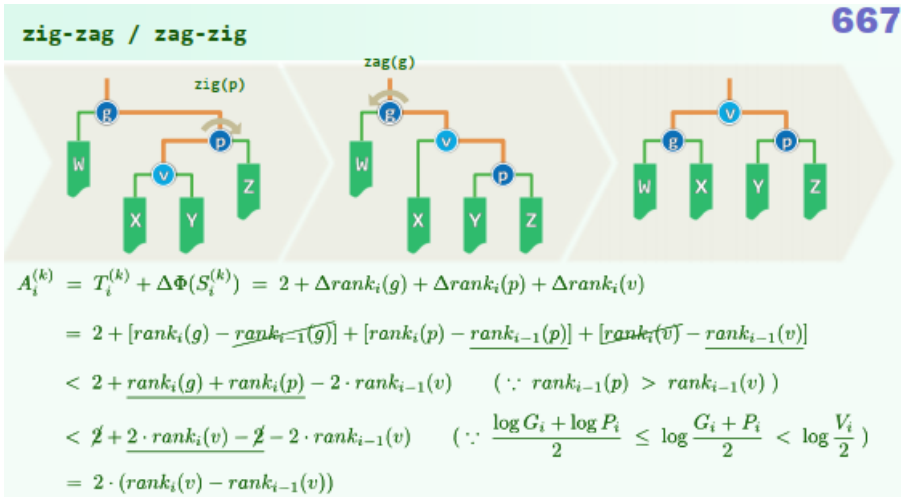
考察对伸展树S的$m >> n$次连续访问，记$A^{(k)} = T^{(k)} + \triangle\phi^{(k)}, k = 0, 1, 2...m$
则有：$A - O(nlogn) \le T = A - \triangle\phi \le A + O(nlogn)$
故若能证明$A = O(mlogn)$，则必有$T = O(mlogn)$。

尽管$T^{(k)}$的变化幅度可能很大，我们可以证明：$A^{(k)}$都不致超过节点v的势能变化量，即：
$O(rank^{(k)}(v) - rank^{(k-1)}(v) = O(logn))$

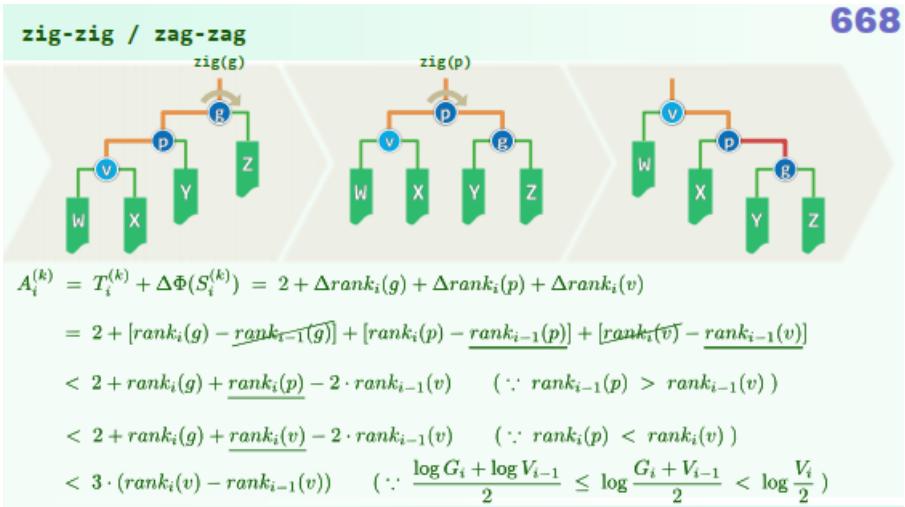
事实上，$A^{(k)}$不过是v的若干次连续伸展操作的时间成本的累计，这些操作无非**三种**情况...



$$
\begin{aligned}
A_i^{(k)} &= T_i^{(k)} + \Delta\Phi(S_i^{(k)}) = 1 + \Delta rank_i(v) + \Delta rank_i(r) \\
&= 1 + [rank_i(v) - rank_{i-1}(v)] + [rank_i(r) - rank_{i-1}(r)] \\
&< 1 + [rank_i(v) - rank_{i-1}(v)]
\end{aligned}
$$



$$
\begin{aligned}
A_i^{(k)} &= T_i^{(k)} + \Delta\Phi(S_i^{(k)}) = 2 + \Delta rank_i(g) + \Delta rank_i(p) + \Delta rank_i(v) \\
&= 2 + [rank_i(g) - rank_{i-1}(g)] + [rank_i(p) - rank_{i-1}(p)] + [rank_i(v) - rank_{i-1}(v)] \\
&< 2 + rank_i(g) + rank_i(p) - 2 \cdot rank_{i-1}(v) \qquad (\because rank_{i-1}(p) > rank_{i-1}(v)) \\
&< 2 + 2 \cdot rank_i(v) - 2 - 2 \cdot rank_{i-1}(v) \qquad (\because \frac{\log G_i + \log P_i}{2} \le \log \frac{G_i + P_i}{2} < \log \frac{V_i}{2}) \\
&= 2 \cdot (rank_i(v) - rank_{i-1}(v))
\end{aligned}
$$



$$
\begin{aligned}
A_i^{(k)} &= T_i^{(k)} + \Delta\Phi(S_i^{(k)}) = 2 + \Delta rank_i(g) + \Delta rank_i(p) + \Delta rank_i(v) \\
&= 2 + [rank_i(g) - rank_{i-1}(g)] + [rank_i(p) - rank_{i-1}(p)] + [rank_i(v) - rank_{i-1}(v)] \\
&< 2 + rank_i(g) + rank_i(p) - 2 \cdot rank_{i-1}(v) \qquad (\because rank_{i-1}(p) > rank_{i-1}(v)) \\
&< 2 + rank_i(g) + rank_i(v) - 2 \cdot rank_{i-1}(v) \qquad (\because rank_i(p) < rank_i(v)) \\
&< 3 \cdot (rank_i(v) - rank_{i-1}(v)) \qquad (\because \frac{\log G_i + \log V_{i-1}}{2} \le \log \frac{G_i + V_{i-1}}{2} < \log \frac{V_i}{2})
\end{aligned}
$$

综合评价：
局部性强、缓存命中率极高时（即$k << n << m$）

> 效率甚至可以更高——自适应的$O(logk)$
> 任何**连续的m次**查找，仅需$O(mlogk + nlogn)$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

3. 算法实现

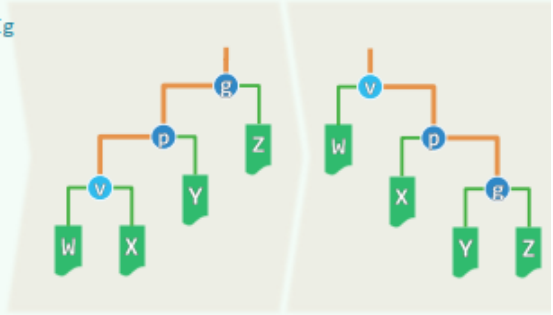
```
template <typename T> class Splay : public BST<T> { //由BST派生
```

```
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）
};

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 );
    }
    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
}
```

伸展算法（举例：zig-zig）：

```
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 */ }
```



1. 查找算法

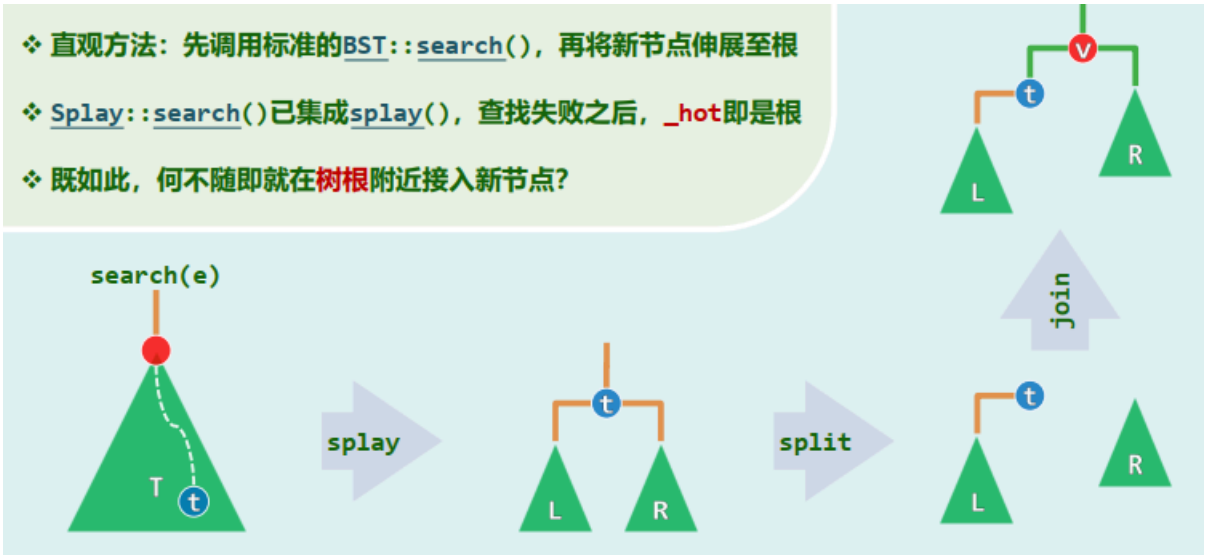
> 伸展树的查找， 与常规`BST::search()`不同：很可能会改变树的拓扑结构，不再属于**静态**操作

```
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;
}
```

2. 插入算法

> 直观方法：先调用标准的`BST::search()`，再将新节点伸展至根
> `Splay::search()`已集成`splay()`，查找失败之后， _hot即是根
> 既如此，何不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 ); //更新规模及t与_root的高度，插入成功
    return _root;
} //无论如何， 返回时总有_root->data == e
```

3. 删除算法

> 直观方法：调用BST标准的删除算法，再将_hot伸展至根
> 注意到，`Splay::search()`成功之后，目标节点即是树根
> 既如此，何不随即就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(t); //记下左、右子树后，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为左子树
    }
    _size--; if ( _root ) updateHeight( _root ); //更新记录
    return true; //删除成功
}
```

9 词典

9.1 散列 / Hash

循对象访问

> entry = (key, value)
>
> **Map / Dictionary**: 词条的集合
> 关键码禁止/允许**雷同**
> `get(key)`, `put(key, value)`, `remove(key)`
>
> 关键码未必可定义大小，元素类型较BST更多样
> 查找对象不限于最大/最小词条，接口功能较PQ更强大

```
//Dictionary
template <typename K, typename V>
struct Dictionary {
```

```
    virtual int size() = 0;
    virtual bool put(K, V) = 0;
    virtual V* get(K) = 0;
    virtual bool remove(K) = 0;
};
```

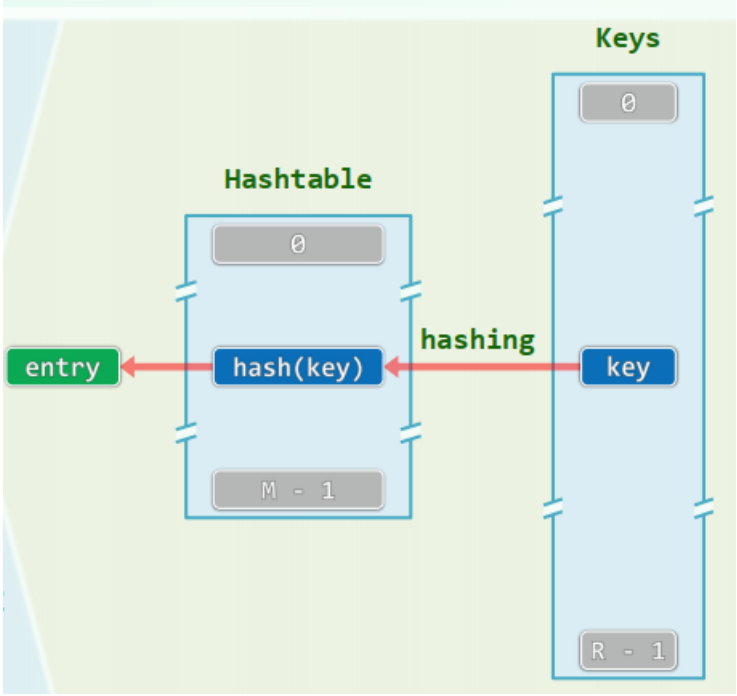
Notation

R: real-world scenario (可能的所有rank，e.g. 可能的所有合法电话)
N: 实际使用的rank，e.g.实有的固定电话

散列表/散列函数

桶（bucket）：直接存放或间接指向一个词条

> Bucket array ~ Hashtable
> 容量：M
> 满足：$N < M << R$
> 空间：$O(N+M) = O(N)$
>
> 定址/杂凑/散列
> 根据词条的key（未必可比较）**“直接”**确定散列表入口（无论表有多长）
>
> **散列函数**：`hash(): key |-> &entry`
> **“直接”**：$expected - O(1) \neq O(1)$



1. 冲突

装填因子（load factor）：$\lambda = N/M$

> λ越大，空间利用率越高，冲突的情况越严重
> 通过降低λ，冲突程度将会有所改善，但只要数据集在**动态**变化，就无法彻底杜绝

> **完美散列**（perfect hashing）：实现单射的散列
> 采用两级散列模式
> 仅需$O(n)$空间
> 关键码之间互不冲突
> 最坏情况下的查找时间也不过$O(1)$

在**装填因子**确定之后， **散列策略**的选取将至关重要， **散列函数**的设计也很有讲究

2. 设计Hash算法

- **确定** determinism：同意关键码总是被映射到同一地址
- **快速** efficiency：expected-$O(1)$
- **满射** surjection：尽可能充分地利用整个散列空间
- **均匀** uniform：关键码映射到散列表各位置的**概率**尽量接近，有效避免聚集（clustering）现象

- **除余法**：$hash(key) = key$
  选取M为**素数**，数据对散列表的覆盖最**均匀**和**充分**：考虑周期性
  （对于理想随机序列，表长是否为素数无关紧要）

  > **蝉**的哲学：经长期自然选择，生命周期取**素数**

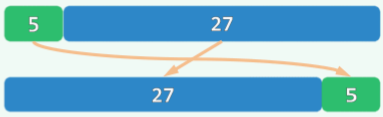
  缺陷：

  - 不动点：无论表长M取值如何，总有$hash(0) \equiv 0$
  - 相关性：$[0, R)$的关键码尽管系平均分配至M个桶，但**相邻**关键码的散列地址也必**相邻**
- **MAD法**（Multiply-Add-Divide）

  $hash(key) = (a \times key + b) \% M, M\ prime, a > 1, b > 0, and\ M$不是$a$的因数
- **数字分析**（selecting digits）：抽取key中的某几位，构成地址
- **平方取中**（mid-square）：去key^{2}中间若干位，构成地址
  e.g. $hash(123) = middle(123 \times 123) = 15219 => 512$
- **折叠法**（folding）：将key分隔成**等宽**的若干段，其**总和**作为地址
  e.g. $hash(123456789) = 123 + 456 + 789 = 1368$
  e.g. $hash(123456789) = 123 + 654 + 789 = 1566$
- **位异或法XOR**：将key分割成**等宽**的二进制端，经**异或**运算得到地址
  e.g. $hash(110011011b) = 110\ \ XOR\ \ 011\ \ XOR\ \ 011 = 110b$
- **随机数法**
- **多项式法**：String/Object to Integer

$$
\begin{aligned}
& hashCode(\text{“}x_{n-1} \ldots x_3 x_2 x_1 x_0\text{”}) \\
= & \ x_{n-1} \cdot a^{n-1} + \ldots + x_2 \cdot a^2 + x_1 \cdot a^1 + x_0 \\
= & \ (\ldots((x_{n-1} \cdot a + x_{n-2}) \cdot a) + \ldots + x_1) \cdot a + x_0
\end{aligned}
$$

```
static size_t hashCode( char s[] ) {
   unsigned int h = 0;
   for ( size_t n = strlen(s), i = 0; i < n; i++ )
      { h = (h << 5) | (h >> 27);  h += ( int ) s[i];  }
   return ( size_t ) h;
```



3. 处理冲突：开放散列

Open Hashing (necessarily closed addressing)

多槽位

- **多槽位** Multiple Slots

  > 桶单元细分成若干**槽位**
  > 存放（与同一单元）冲突的词条

  只要槽位数目不太多，依然可以保证$O(1)$的时间效率
  然而，槽位过细会造成空间**浪费**；且无论多细，计算情况下仍然可能**不够**

公共溢出区

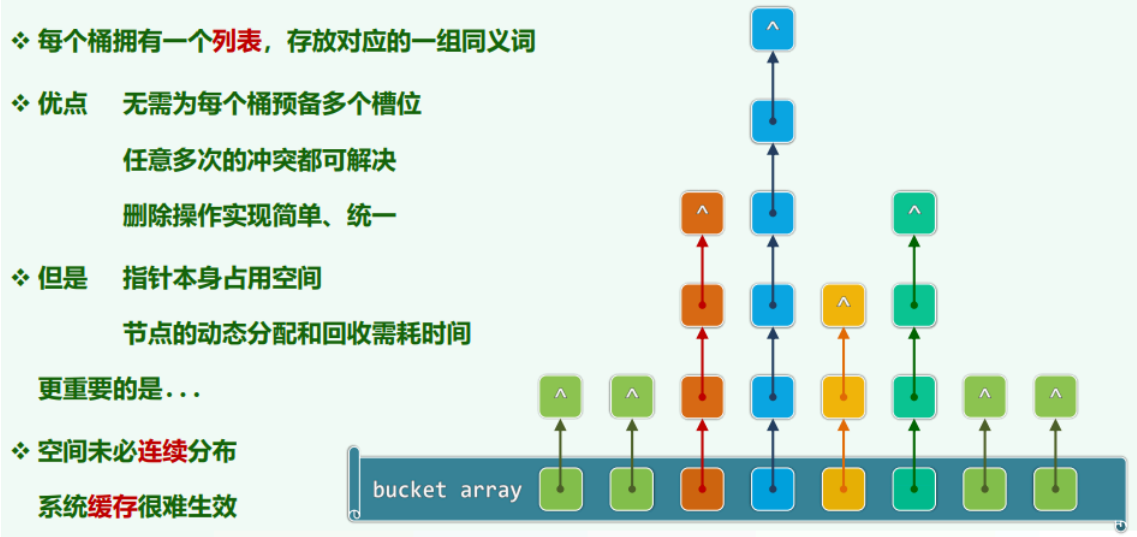
- **公共溢出区** Overflow Area

  > 单独开辟一块连续空间，发生冲突的词条， **顺序**存入此区域

  但是一旦发生冲突，最坏情况下处理冲突词条所需的时间将正比于溢出区的**规模**

独立链

- **独立链** Linked-List Chaining / Separate Chaining



4. 处理冲突：封闭散列

Closed Hashing (necessarily open addressing): 只要有必要，任何散列桶都可以接纳**任何**词条

开放定址

- **开放定址**

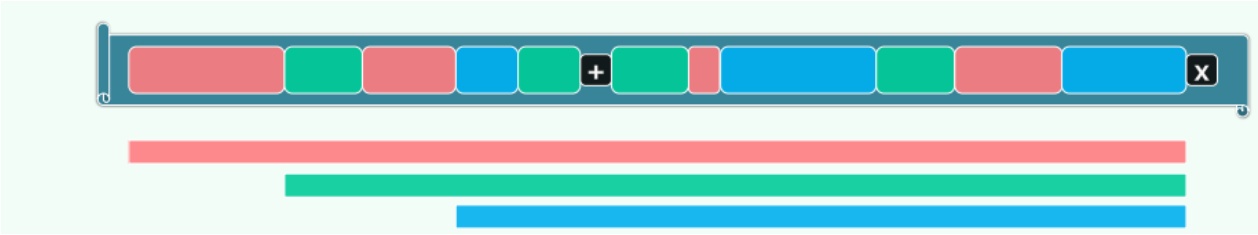
- Probe Sequence / Chain
  为每个词条，都需**事先约定**多干**备用桶**，优先级逐次下降
- 查找算法（线性试探）

  > 沿试探链，逐个转向**紧邻**的桶，直到命中（成功）；或者抵达一个空桶（存在则必能找到？） 而失败

在散列表内部解决冲突，无需附加的指针、链表或溢出区等，整体结构保持简洁
但是，新增**非同义词**之间的冲突；数据堆积（clustering）现象严重
通过装填因子，冲突与堆积都可有效控制——缓存生效，所以可以很快

问题：插入 + 删除

> 插入：新词条若尚不存在，则存入试探终止处的空桶
> **试探链**：可能因而彼此**串接、 重叠**（未必是synonym）
>
> 删除：简单地清楚命中的桶？
> 经过它的试探链都将因此**断裂**，导致后续词条**丢失**——明明存在，却访问不到



懒惰删除

- **懒惰删除**

```
Bitmap* removed; //用Bitmap懒惰地标记被删除的桶
int L; //被标记桶的数目
```

> **查找**词条时，被视作“必不匹配的非空桶”，试探链在此得以延续
> **插入**词条时，被视作“必然匹配的空闲桶”，可以用来存放新词条

```
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
}
```

重散列

- **重散列** Rehashing

```
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
    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
}
```

> 懒惰删除的算法使得装填因子的真实值被低估，因而“扩”容采用$4\times N$

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;
```

```
}
```

平方试探

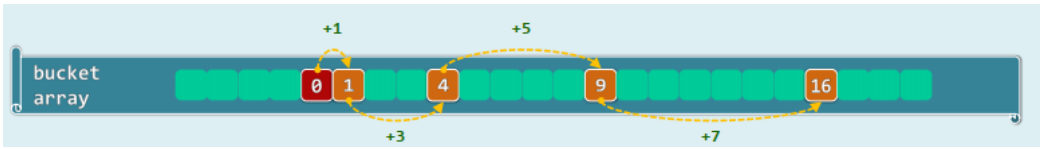
- **平方试探** Quadratic Probing

  Definition: 以**平方数**为距离，确定下一试探桶单元

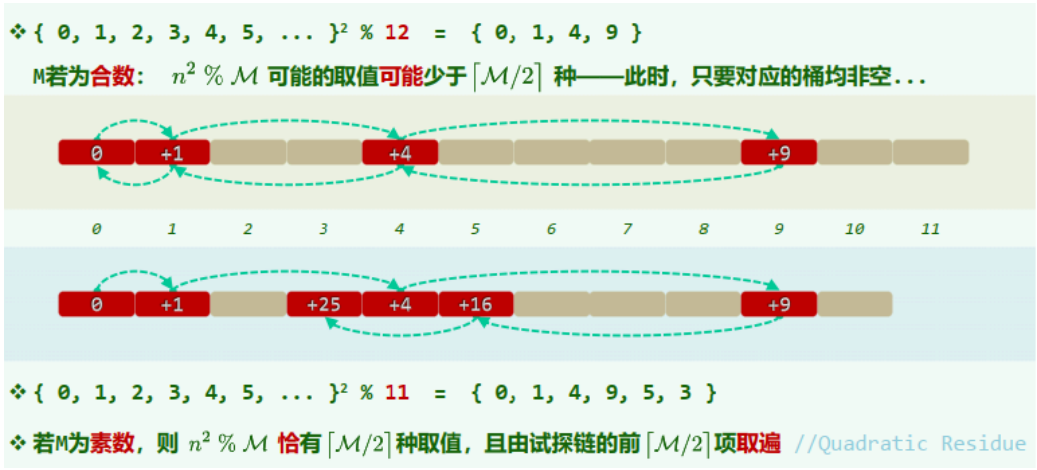
  > 数据聚集现象有所缓解
  > 试探链上，各桶间距**线性递增**
  > 一旦冲突，可“聪明”地跳离**是非之地**
  > 对于大散列表，**I/O**操作有所增加

  ```
  [hash(key) + 1] % M
  [hash(key) + 4] % M
  [hash(key) + 9] % M
  [hash(key) + 16] % M
  ```

  Image unavailable in the original source: image-20221111101527420.png

  素数表长时，只要$\lambda < 0.5$就一定能够找出；否则，不见得

双向平方试探

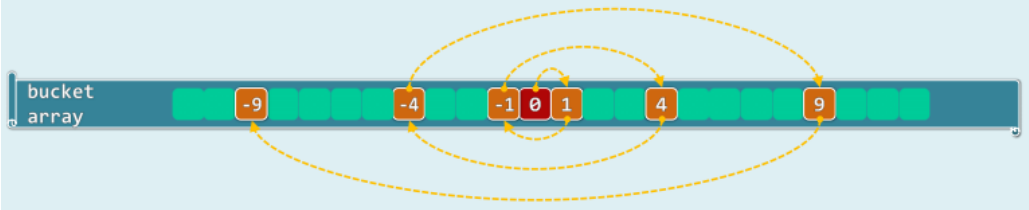
- **双向平方试探**

  Definition: **交替**地沿两个方向试探，均按**平方**确定距离

  ```
  [hash(key) + 1] % M
  [hash(key) + 4] % M
  [hash(key) + 9] % M
  [hash(key) + 16] % M

  [hash(key) - 1] % M
  [hash(key) - 4] % M
  [hash(key) - 9] % M
  ```

```
[hash(key) - 16] % M
```



M \ ± i²	-36	-25	-16	-9	-4	-1	0	1	4	9	16	25	36
5					1	4	0	1	4				
7				5	3	6	0	1	4	2			
11		8	6	2	7	10	0	1	4	9	5	3	
13	3	1	10	4	9	12	0	1	4	9	3	12	10

两类素数：$4k+1$和$4k+3$两类。表长取素数$M=4k+3$则**必然可以**保证试探链的前M项均互异

Two-Square Theorem of Fermat

❖ 任一素数 p 可表示为一对整数的平方和，当且仅当

$$p \equiv 1 \pmod 4$$

❖ 只要注意到：

$$\begin{aligned}(u^2+v^2)\cdot(s^2+t^2) &= (us+vt)^2+(ut-vs)^2\\ &= (us-vt)^2+(ut+vs)^2\end{aligned}$$

$$\begin{aligned}(2^2+3^2)\cdot(5^2+8^2) &= (10+24)^2+(16-15)^2\\ &= (10-24)^2+(16+15)^2\end{aligned}$$

❖ 就不难推知：

- 自然数 n 可表示为一对整数的平方和，当（且仅当）
- 它的每一 $M=4\cdot k+3$ 类的素因子均为偶数次方

双散列 Double Hashing

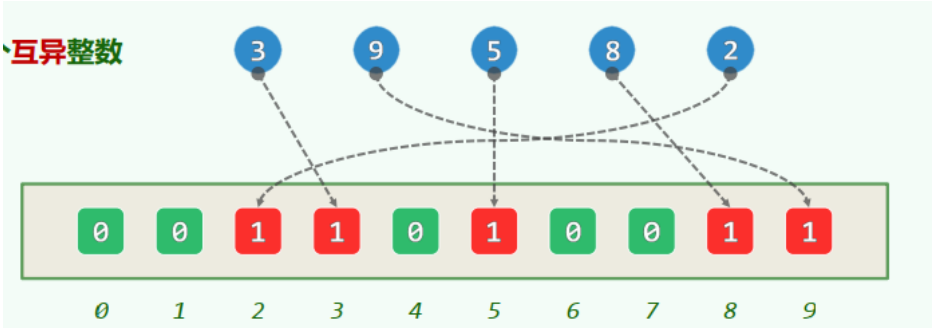
> 预先约定第二散列函数：$hash_2(key,i)$
> 冲突时，由其偏移**增量**，确定下一试探位置：$[hash(key)+\sum_{i=1}^{k} hash_2(key,i)]\%M$
>
> - 线性试探：$hash_2(key,i)\equiv 1$
> - 平方试探：$hash_2(key,i)=2i-1$

9.2 桶排序

1. 基本算法实现

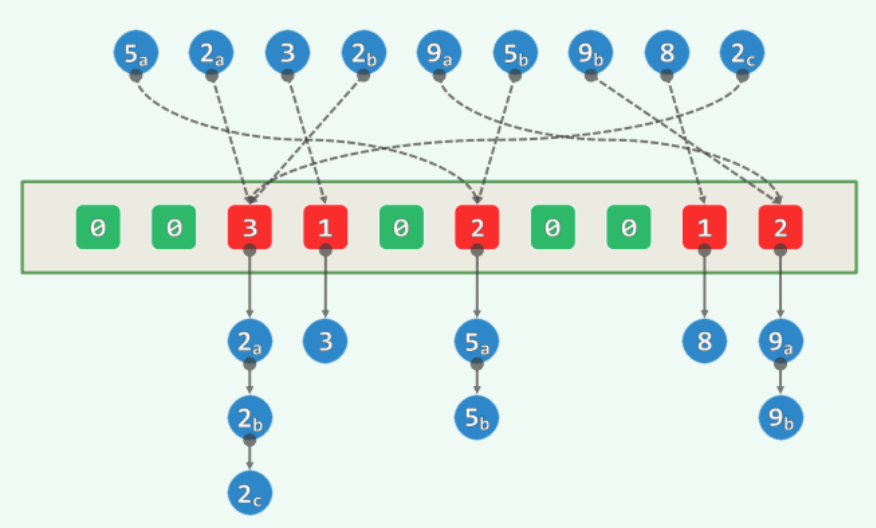
1. 对$[0,m)$内$n(<m)$个**互异**的整数借助散列表$H[]$进行排序

> 空间：$O(m)$
> 时间：$O(n)$
>
> **初始化**：for i = 0 to m - 1, let $H[i]$ = 0
> **映射**：for each key in the input, let $H[key]$ = 1
> **枚举**：for i = 0 to m - 1, output i if $H[i]$ = 1



2. 允许重复（可能$m << n$），依然可以使用Hash——没一簇同义词各成一个链表

> 空间：$O(m+n)$
> 时间：$O(n)$



2. 最大缝隙 MaxGap

> 任意n个互异点均将实轴氛围$n-1$段有界区间，其中的哪一段**最长**?

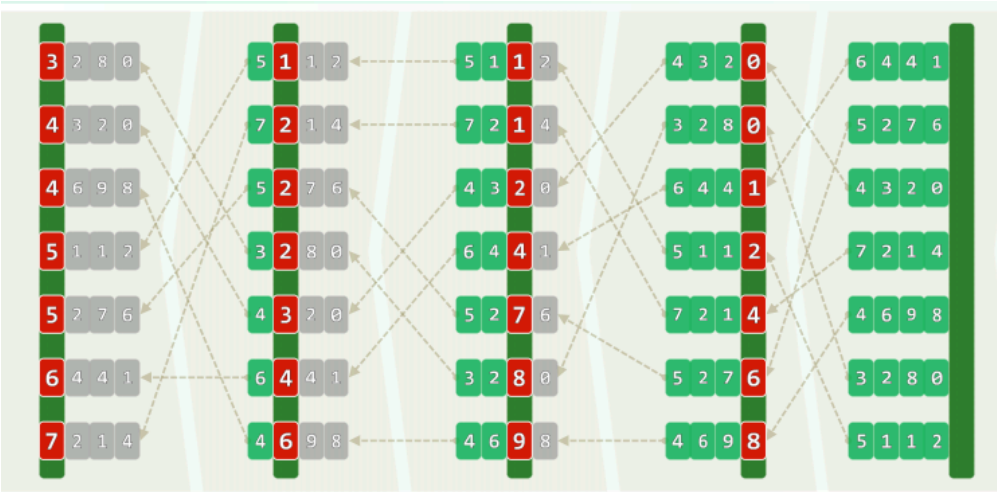
线性算法:



9.3 基数排序 Radixsort

> 有时，关键码由多个**域**组成： k_d , k_{t-1} , ... , k_1 // (suit, point) in bridge
> 若将各域视作**字母**，则关键码即**单词**——按**词典**的方式排序（lexicographic order）

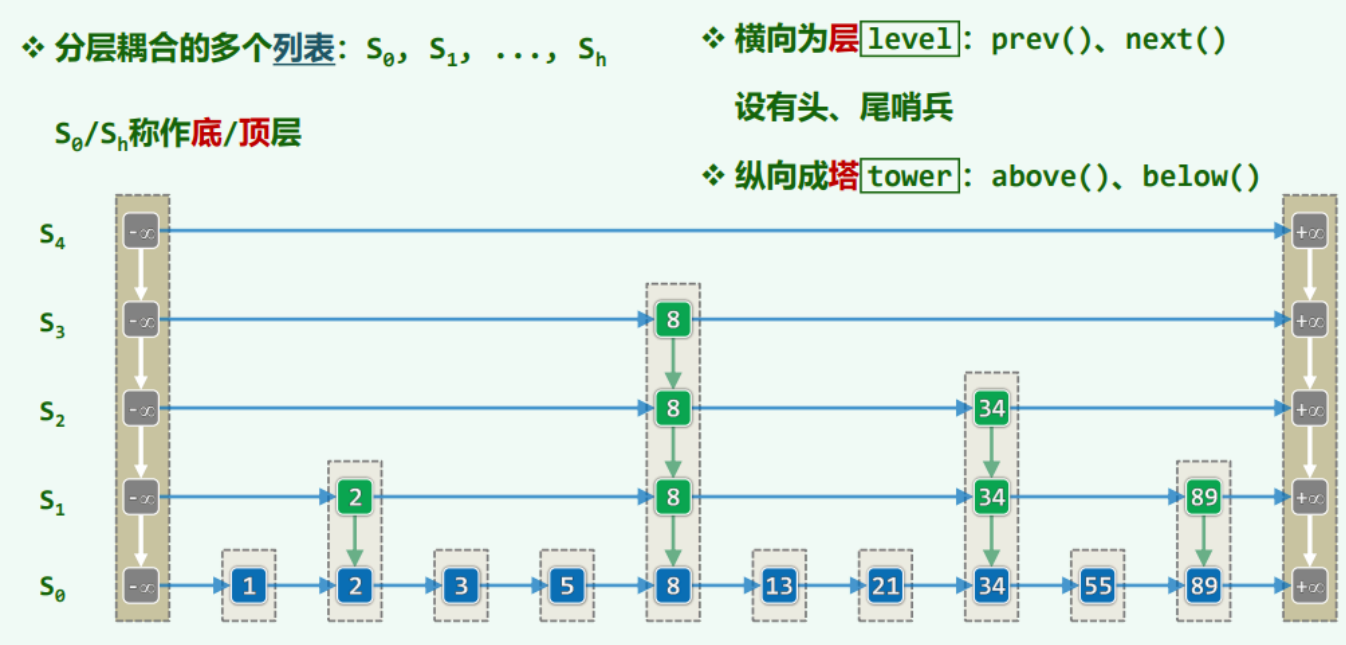
1. 自k_1到k_t（**低位**优先），依次以各域为序做一趟**桶排序**



```
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
```

9.4 跳转表 Skip List




```
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
};

template <typename T> struct Quadlist { //四联表
    int _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
};

template < typename K, typename V >
struct Skiplist :
public Dictionary<K, V>, public List< Quadlist< Entry<K, V> >* > {
    Skiplist() //即便为空，也有一层空列表
        { insertAsFirst( new Quadlist< Entry<K, V> > ); };
    QNodePosi< Entry<K, V> > search( K ); //由关键码查询词条
    int size() {return empty() ? 0 : last()->data->size();} //词条总数
    int 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
};
```

空间性能

逐层随机减半：S_k中的每个关键码，在S_{k+1}中依然出现的概率，均为$p = 1/2$

各塔的高度符合**几何分布**：$Pr(h = k) = p^{k-1} \cdot (1 - p)$

期望的塔高：$E(h) = \frac{1}{1-p} = 2$，即期望空间总和为$expected - O(n)$

1. 查找

> 时间复杂度：$O(logn)$

```
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？
```

时间性能

查找时间取决于**横向、纵向**的累计跳转次数

整张表的期望高度为$O(logn)$，查找中**纵向**跳转的次数，累计$expected - O(logn)$

（每一个塔的期望高度为$O(2) = O(1)$）

事实上，查找中横向跳转的次数也为$O(logn)$

❖ **观察：** **一座塔高度不低于|低于k 的概率** $= p^k \mid 1 - p^k$



❖ **引理：** **随着k 的增加，S_k 为空/非空的概率急剧上升/下降**

$$Pr(|S_k| = 0) = (1 - p^k)^n \geq 1 - n \cdot p^k \qquad Pr(|S_k| > 0) \leq n \cdot p^k$$

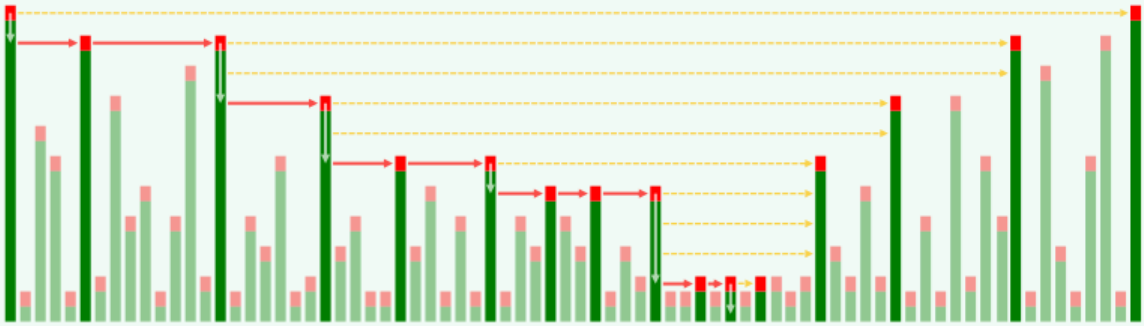
❖ **推论：** **跳转表高度$h = \mathcal{O}(\log n)$的概率极大**

❖ **比如：** **若 $p = 1/2$ ，则第 $k = 3 \cdot \log_{1/p} n$ 层非空（当且仅当 $h \geq k$）的概率为**

$$Pr(|S_k| > 0) \leq n \cdot p^k = n \cdot n^{-3} = 1/n^2 \longmapsto 0$$

❖ **结论：** **查找过程中，纵向跳转的次数，累计不过** expected-$\mathcal{O}(\log n)$

❖ 观察：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



❖ 于是：若将沿同一层列表跳转的次数记作Y，则它符合几何分布 $Pr(Y=k) = (1-p)^k \cdot p$

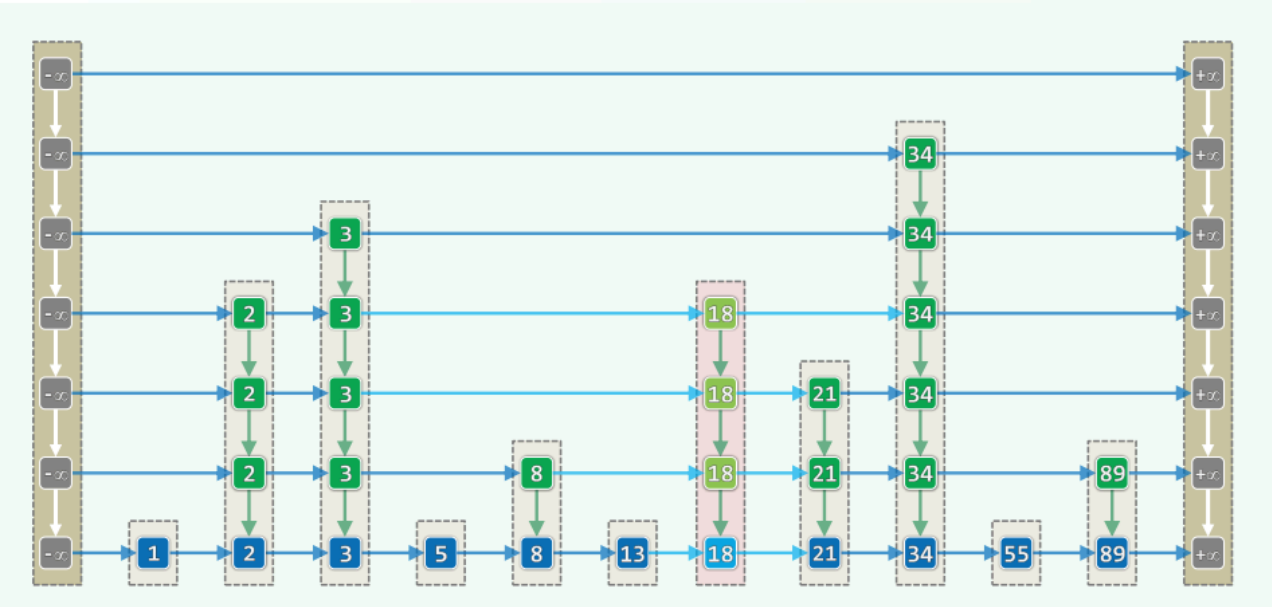
❖ 定理：$\mathbb{E}(Y) = (1-p)/p = (1-0.5)/0.5 = 1$ 次

❖ 因此：在同一层列表中连续跳转的期望时间成本 = 1次跳转 + 1次驻足 = 2

2. 插入与删除

时间复杂度：$O(logn)$

```
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
 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 ) {
    // 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，且已是最顶层，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？
```




```
template <typename K, typename V>
bool Skiplist<K, V>::remove( K k ) {
 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
 /* ... 1. 预备 ... */
 ListNodePosi< Quadlist<Entry<K, V>>* > qlist = last(); //从底层Quadlist开始
 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
     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？
```

10 图 Graph

$G = (V; E)$

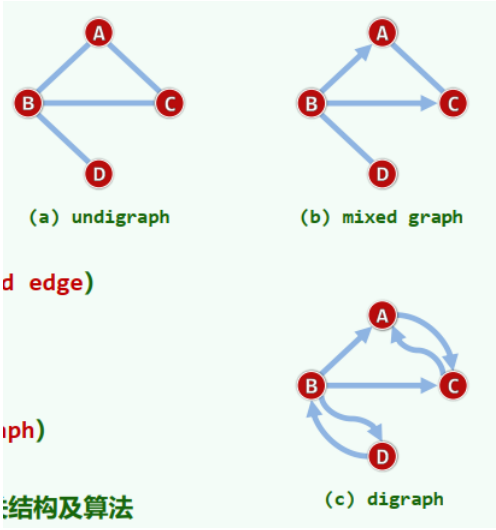
10.1 基本概念

节点数 $vertex$： $n = |V|$
边树 $edge|arc$： $e = |E|$

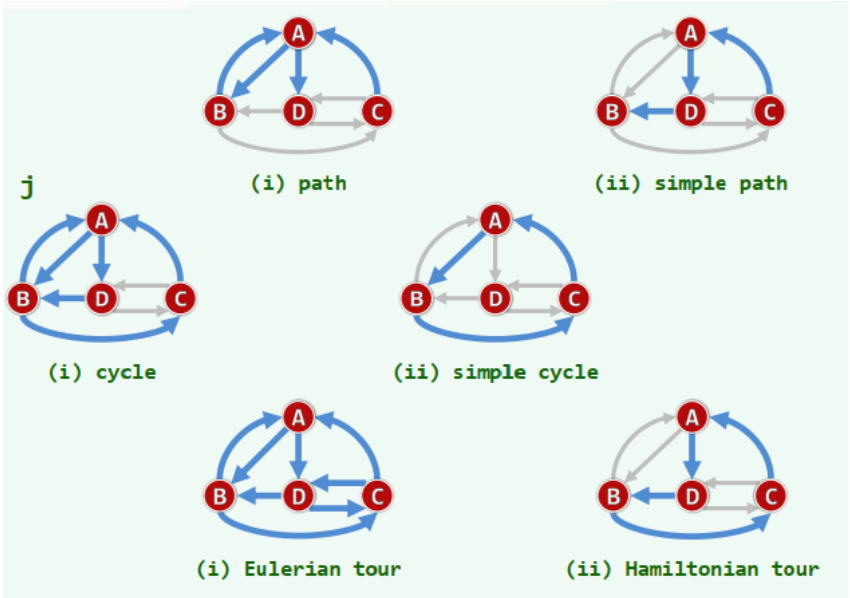
同一条边的两个顶点——彼此**邻接**（adjacency）
同一顶点自我邻接——构成**自环**（self-loop）
不含自环及重边——即为**简单图**（simple graph）
非简单图（non-simple）暂不讨论

顶点与其所属的边——彼此**关联**（incidence）
度（degree/valency）——与同一顶点关联的边数

若邻接顶点u和v的**次序**无所谓，则（u, v）为**无向边**（undirected edge）
所有边均无方向的图，即**无向图**（undigraph）
反之，**有向图**（digraph）中均为**有向边**（directed edge）。u，v分别称作边（u, v）的**尾**（tail）和**头**（head）
无向边、有向边并存的图，称作**混合图**（mixed graph）



路径 $\pi =< v_0, v_1, \ldots, v_k >$，**长度** $|\pi| = k$
简单路径 $v_i \neq v_j$ 除非 $i = j$
环/环路：$v_o = v_k$
有向无环图（DAG）
欧拉环路：$|\pi| = |E|$，各边恰好出现一次
哈密尔顿环路：$|\pi| = |V|$，各顶点恰好出现一次



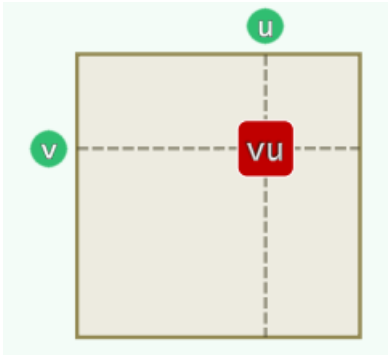
图$G=(V;E)$的子图$T=(V;F)$若是树，则为其**支撑树**（spanning tree）
同一图的支撑树，通常并不唯一
各边e均有对应的权值$wt(e)$，则为**带权网络**（weighted network）
同一网络的支撑树中，总权重最小者为**最小支撑树**（MST）

```
template <typename Tv, typename Te> class Graph {
private:
    void reset() { //所有顶点、边的辅助信息复位
        for ( Rank v = 0; v < n; v++ ) { //顶点
            status(v) = UNDISCOVERED; dTime(v) = fTime(v) = -1;
            parent(v) = -1; priority(v) = INT_MAX;
            for ( Rank u = 0; u < n; u++ ) //边
                if ( exists(v, u) ) type(v, u) = UNDETERMINED;
        } //for
    } //reset
public: int n, e; //顶点、边数目
    /* ... 顶点操作、边操作、图算法： 无论如何实现，接口必须统一 ... */
} //Graph
```

1. 邻接矩阵 adjacency matrix

记录**顶点**之间的**邻接**关系——对应：矩阵元素（图中可能存在的边）

> 既然只考察简单图，对角线统一设置为0
> 空间复杂度为$\Theta(n^2)$，与图中实际的边数无关

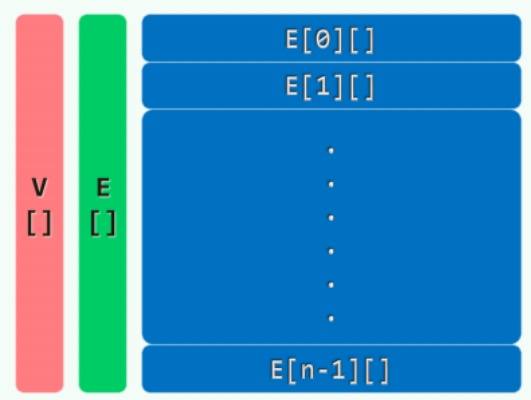



```
using VStatus = enum { UNDISCOVERED, DISCOVERED, VISITED };
template <typename Tv> struct Vertex { //不再严格封装
    Tv data; int inDegree, outDegree;
    VStatus status; //（如上三种）状态
    int dTime, fTime; //时间标签
    Rank parent; //在遍历树中的父节点
    int priority; //在遍历树中的优先级（最短通路、极短跨边等）
    Vertex( Tv const & d ) : data( d ), inDegree( 0 ), outDegree( 0 ), status( UNDISCOVERED
), dTime( -1 ), fTime( -1 ), parent( -1 ), priority( INT_MAX ) {}
};

using EType = enum { UNDETERMINED, TREE, CROSS, FORWARD, BACKWARD };
template <typename Te> struct Edge { //不再严格封装
    Te data; //数据
    int weight; //权重
    EType type; //在遍历树中所属的类型
```

```
    Edge( Te const & d, int w ) : data(d), weight(w), type(UNDETERMINED) {}
};
```

```
template <typename Tv, typename Te>
class GraphMatrix : public Graph<Tv, Te> {
private:
    Vector< Vertex<Tv> > V; //顶点集
    Vector< Vector< Edge<Te>* > > E; //边集
public: // 操作接口：  顶点相关、  边相关、  ...
    GraphMatrix() { n = e = 0; } //构造
    ~GraphMatrix() { //析构
        for ( Rank v = 0; v < n; v++ )
            for ( Rank u = 0; u < n; u++ )
                delete E[v][u]; //清除所有边记录
    }
};
```



静态操作

1. 顶点的读写

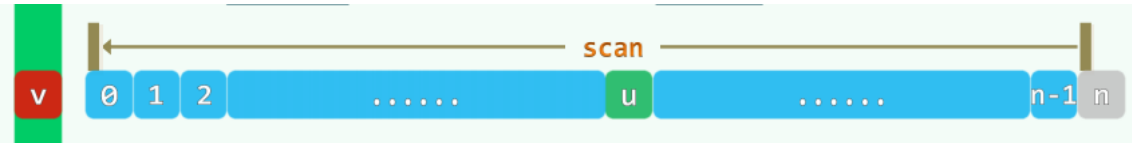
```
Tv & vertex(Rank v) { return V[v].data; } //数据
int inDegree(Rank v) { return V[v].inDegree; } //入度
int outDegree(Rank v) { return V[v].outDegree; } //出度
VStatus & status(Rank v) { return V[v].status; } //状态
int & dTime(Rank v) { return V[v].dTime; } //时间标签dTime
int & fTime(Rank v) { return V[v].fTime; } //时间标签fTime
Rank & parent(Rank v) { return V[v].parent; } //在遍历树中的父亲
int & priority(Rank v) { return V[v].priority; } //优先级数
```

2. 边的读写

```
bool exists( Rank v, Rank u ) { //判断边(v, u)是否存在（短路求值）
    return (v < n) && (u < n) && E[v][u] != NULL;
} //以下假定exists(v, u) = true
Te & edge( Rank v, Rank u ) { return E[v][u]->data; } //数值
EType & type( Rank v, Rank u ) { return E[v][u]->type; } //类型
int & weight( Rank v, Rank u ) { return E[v][u]->weight; } //权重
```

3. 邻点的枚举

```
Rank firstNbr( Rank v ) { return nextNbr( v, n ); } //假想哨兵
Rank nextNbr( Rank v, Rank u ) { //若已枚举至邻居u， 则转向下一邻居
    while ( -1 < u ) && !exists( v, --u ) ); //逆向顺序查找
return u;
} //O(n)——改用邻接表，可提高至O(1 + outDegree(v))
```



动态操作

1. 边的插入

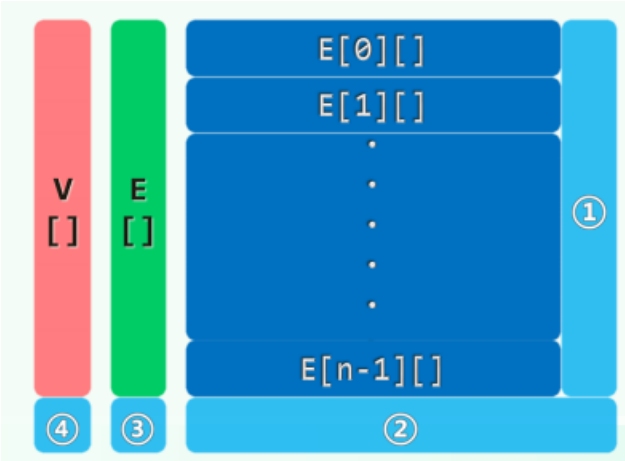
```
void insert( Te const & edge, int w, Rank v, Rank u ) {
    if ( exists(v, u) ) return; //忽略已有的边
    E[v][u] = new Edge<Te>( edge, w ); //创建新边（权重为w）
    e++; //更新边计数
    V[v].outDegree++; //更新顶点v的出度
    V[u].inDegree++; //更新顶点u的入度
}
```

2. 边的删除

```
Te remove( Rank v, Rank u ) { //删除（已确认存在的） 边(v, u)
    Te eBak = edge(v, u); //备份边(v, u)的信息
    delete E[v][u]; E[v][u] = NULL; //删除边(v, u)
    e--; //更新边计数
    V[v].outDegree--; //更新顶点v的出度
    V[u].inDegree--; //更新顶点u的入度
    return eBak; //返回被删除边的信息
}
```

3. 顶点插入

```
Rank insert( Tv const & vertex ) { //插入顶点，返回编号
    for ( Rank u = 0; u < n; u++ ) E[u].insert( NULL ); n++; //①
    E.insert( Vector< Edge<Te>* >( n, n, NULL ) ); //②③
    return V.insert( Vertex<Tv>( vertex ) ); //④
}
```



4. 顶点删除

```
Tv remove( Rank v ) { //删除顶点及其关联边，返回该顶点信息
    for ( Rank u = 0; u < n; u++ ) //删除所有出边
        if ( exists( v, u ) ) {
            delete E[v][u]; V[u].inDegree--; e--;
        }
    E.remove(v); n--; //删除第v行
    Tv vBak = vertex( v ); V.remove( v ); //备份之后，删除顶点v
    for ( Rank u = 0; u < n; u++ ) //删除所有入边及第v列
        if ( Edge<Te> * x = E[u].remove( v ) )
            { delete x; V[u].outDegree--; e--; }
    return vBak; //返回被删除顶点的信息
}
```

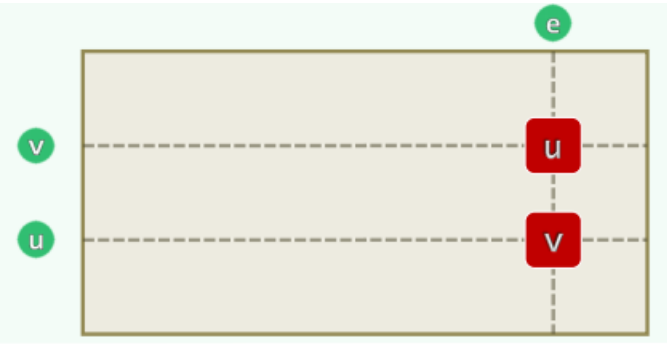
性能分析

- 直观，易于理解和实现
- 适用范围广泛，尤其适用于**稠密图**（dense graph）
- 判断两点之间是否存在联边：$O(1)$
  获取顶点的出/入度数：$O(1)$
  添加、删除边后更新度数：$O(1)$
- **扩展性**（scalability）：得益于Vector良好的控制策略，空间溢出等情况可被“透明的”处理
- 但是！
- $O(n^2)$空间，与边数无关
- **平面图**（planar graph）：**可嵌入**于平面的图，满足$V-E+F-C=1$
  平面图：$e \leq 3\times n-6=O(n)<<n^2$
  此时，空间利用率 = $\frac{1}{n}$
- **稀疏图**（sparse graph）：空间利用率同样很低，可采用压缩存储技术

2. 关联矩阵 incidence matrix

记录**顶点**与**边**之间的**关联**关系——对应：矩阵元素（每条边的两个节点）

> 空间复杂度为$\Theta(n\times e)=O(n^3)$
> 空间利用率$=\frac{2e}{ne}=\frac{2}{n}$
> 解决某些问题时十分有效



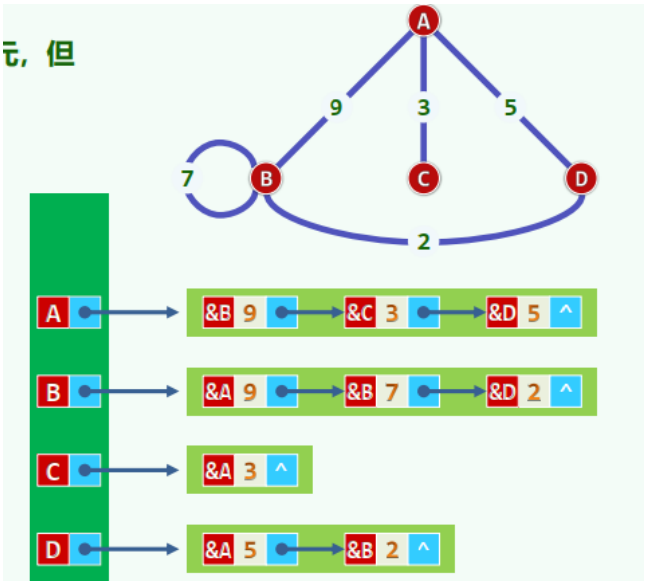
3. 邻接表

空间复杂度

- 有向图：$O(n+e)$
- 无向图：$O(n+2\times e)=O(n+e)$
  注意：无向弧被重复存储
- 适用于稀疏图
- 平面图 = $O(n+3\times n)=O(n)$，较值邻接矩阵，有极大改进

时间复杂度

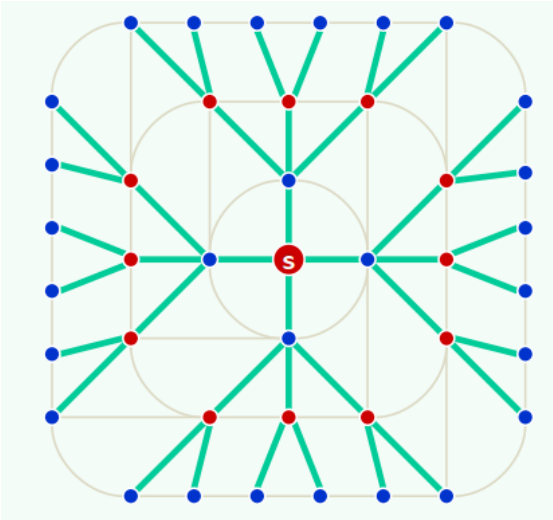
- 建立邻接表（递增式构造）：$O(n+e)$
- 枚举所有以顶点v为尾的弧：$O(1+deg(v))$
- 枚举（无向图中）顶点v的邻居：$O(1+deg(v))$
- 枚举所有以顶点v为头的弧：$O(n+e)$
  可改进至$O(1+deg(v))$（建立逆邻接表）
- 计算顶点v的出度/入度：
  增加度数记录域：$O(n)$附加空间
  增加/删除弧时更新度数：$O(1)$时间，总体$O(e)$时间
  每次查询：$O(1)$时间
- 给定顶点u和v，判断是否$<u,v>\in E$
  - 有向图：搜索u的邻接表，$O(deg(u))=O(e)$
  - 无向图：搜索u或v的邻接表，$O(max(deg(u),deg(v)))=O(e)$
  - “并行”搜索：$O(2\times min(deg(u),deg(v)))=O(e)$
  ——改进：散列（如果装填因子选取得当）



10.2 广度优先搜索 BFS

Breadth-First Search

> 相当于树的**层次遍历**
> 事实上，BFS也的确会构造出原图的一棵**支撑树**（BFS tree）




```
template <typename Tv, typename Te>
void Graph<Tv, Te>::BFS( Rank v, int & clock ) {
    Queue<Rank> Q;
    status(v) = DISCOVERED;
    Q.enqueue(v); //初始化
    while ( ! Q.empty() ) { //反复地
        Rank v = Q.dequeue(); dTime(v) = ++clock; //取出队首顶点v，并
        for ( Rank u = firstNbr(v); -1 < u; u = nextNbr(v, u) )
            if ( UNDISCOVERED == status(u) ) { //若u尚未被发现，则
                status(u) = DISCOVERED; Q.enqueue(u); //发现该顶点
                type(v, u) = TREE; parent(u) = v; //引入树边
            }
            else //若u已被发现（正在队列中），或者甚至已访问完毕（已出队列），则
                type(v, u) = CROSS; //将(v, u)归类于跨边
        status(v) = VISITED; //至此，当前顶点访问完毕
    }
}
```

1. 连通分量 + 可达分量

> 问题：给定**无向图**，找出其中任一顶点s所在的**连通图**
> 给定**有向图**，找出源自其中任一顶点s的**可达分量**

算法：从s出发做BFS，输出所有**被发现**的顶点，队列为空后立即终止

```
template <typename Tv, typename Te>
void Graph<Tv, Te>::bfs( Rank s ) { //s为起始顶点
    reset(); int clock = 0; Rank v = s; //初始化O(n+e)
    do //逐一检查所有顶点，一旦遇到尚未发现的顶点
```

```
        if ( UNDISCOVERED == status(v) ) //累计O(n)
        BFS( v, clock ); //即从该顶点出发启动一次BFS
    while ( s != (v = ((v + 1) % n)) ); //按序号访问，不漏不重
} //无论共有多少连通/可达分量，bfs均可遍历它们，而且自身累计仅需线性时间
```

2. 边分类

- (v, u)被标记为TREE时， v为DISCOVERED且u为UNDISCOVERED



- (v, u)被标记为CROSS时，v和u均为DISCOVERED或者v为DISCOVERED而u为VISITED

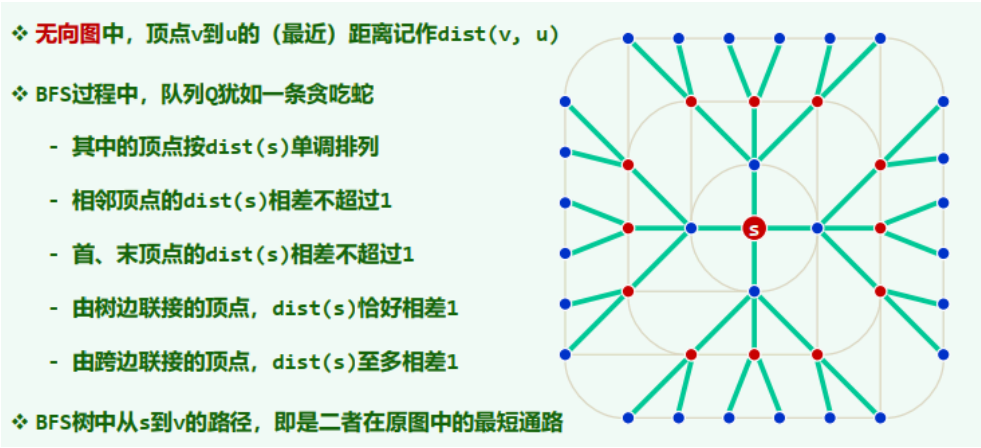


3. BFS树 / 森林

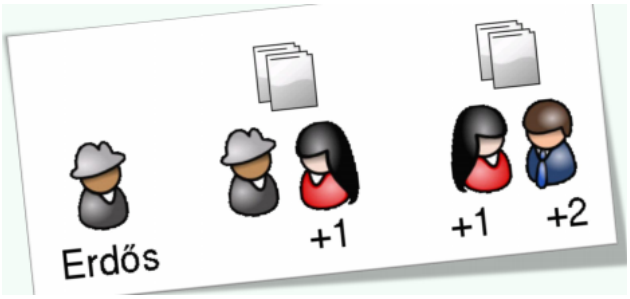


4. 最短路径

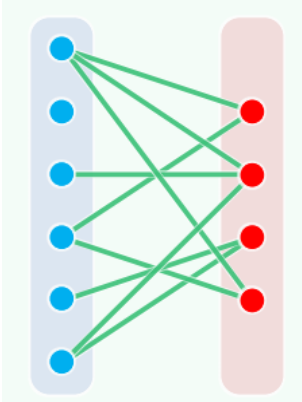
无向图中，顶点v到u的举例记作$dist(v, u)$
BFS树中从s到v的路径，即是二者在原图中的最短通路



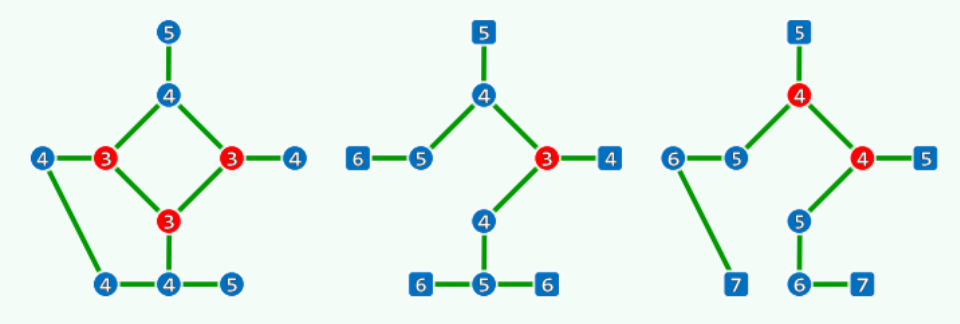
5. Erdös Number: collaborative distance



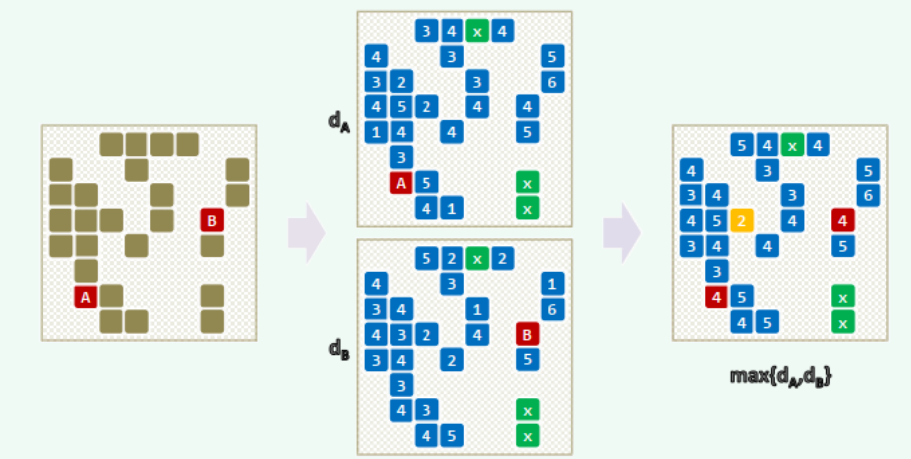
6. Bipartite Graph (Bigraph)



7. Eccentricity / Radius / Diameter / Center



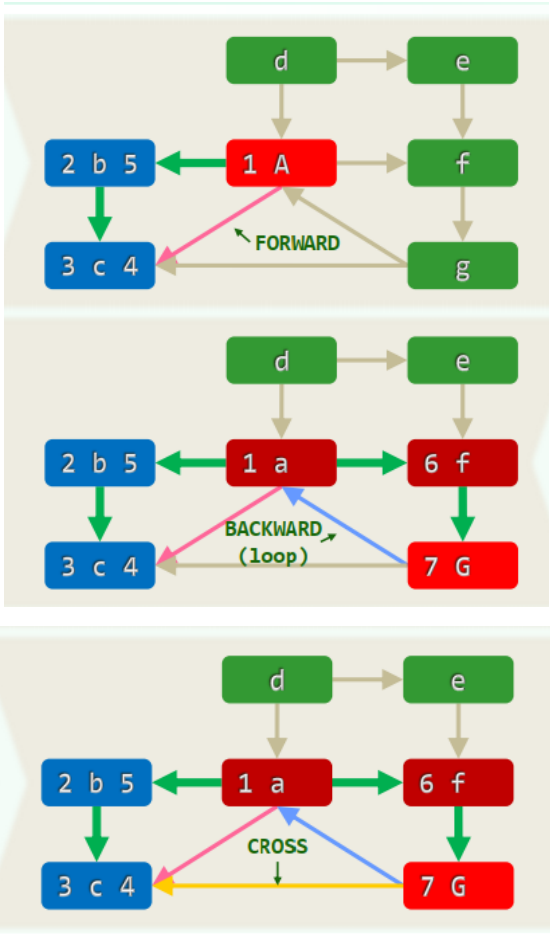
8. Knights of the Round Table



10.3 深度优先搜索 DFS

Depth-First Search

> 相当于树的**先序遍历**
> 事实上，DFS也的确会构造出原图的一棵支撑树（DFS tree）




```
template <typename Tv, typename Te>
void Graph<Tv, Te>::DFS( Rank v, int & clock ) {
    dTime(v) = ++clock; status(v) = DISCOVERED; //发现当前顶点v
    for ( Rank u = firstNbr(v); -1 < u; u = nextNbr(v, u) ) //v的邻居u
        switch ( status(u) ) {
            case UNDISCOVERED: //u尚未发现，意味着支撑树可在此拓展
                type(v, u) = TREE;
                parent(u) = v;
                DFS( u, clock );
                break; //递归
            case DISCOVERED: //u已被发现但尚未访问完毕，应属被后代指向的祖先
                type(v, u) = BACKWARD;
                break;
            default: //u已访问完毕（VISITED有向图），则视承袭关系分前向边或跨边
                type(v, u) = dTime(v) < dTime(u) ? FORWARD : CROSS;
                break;
        }
    status(v) = VISITED; fTime(v) = ++clock; //至此，当前顶点v方告访问完毕
}
```

1. 连通分量 + 可达分量

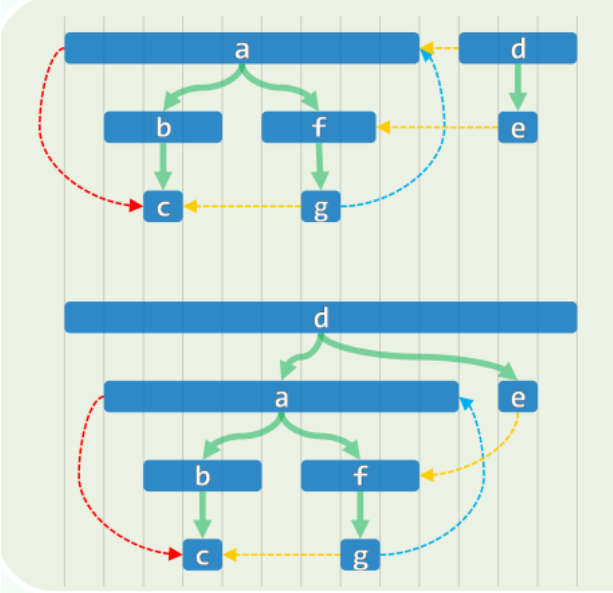
```
template <typename Tv, typename Te>
void Graph<Tv, Te>::dfs( Rank s ) { //s为起始顶点
    reset(); int clock = 0; Rank v = s; //初始化
    do //逐一检查所有顶点，一旦遇到尚未发现的顶点v
        if ( UNDISCOVERED == status(v) )
            DFS( v, clock ); //即从v出发启动一次DFS
    while ( s != (v = ((v + 1) % n)) ); //按序号访问，不漏不重
}
```

2. DFS树 / 森林

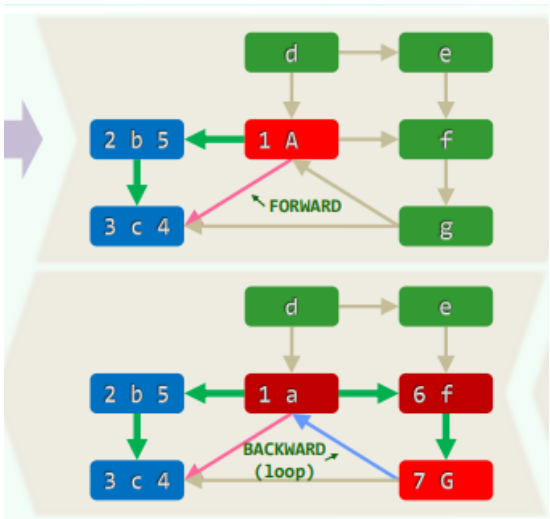
> 从顶点s出发的DFS
> 在**无向图**中将访问**与s连通**的所有顶点（connectivity）
> 在**有向图**中将访问**由s可达**的所有顶点（reachability）
>
> 所有DFS树构成**DFS森林**

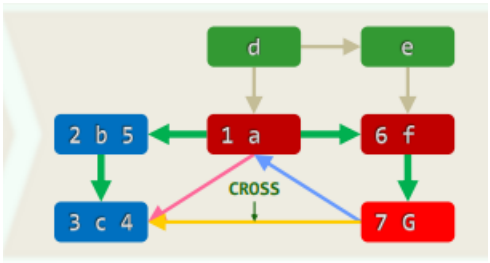
活跃期：$active[u] = (dTime[u], fTime[u])$
Lemma：给定有向图$G = (V, E)$及其任一DFS森林，则
u是v的后代 iff $active[u] \subseteq active[v]$
u与v无关 iff $active \cap active[v] = \emptyset$
仅凭`status[]`，`dtime[]`和`ftime[]`即可对各边分类...

边分类：









3. 遍历算法应用举例

连通图的支撑树（DFS/BFS Tree）	DFS/BFS
非连通图的支撑森林	DFS/BFS
连通性检测	DFS/BFS
无向图环路检测/二部图判定	DFS/BFS
有向图环路检测	DFS
顶点之间可达性检测/路径求解	DFS/BFS
顶点之间的最短距离	BFS
直径/半径/围长/中心	BFS
欧拉回路	DFS
拓扑排序	DFS
双连通分量、强连通分量分解	DFS

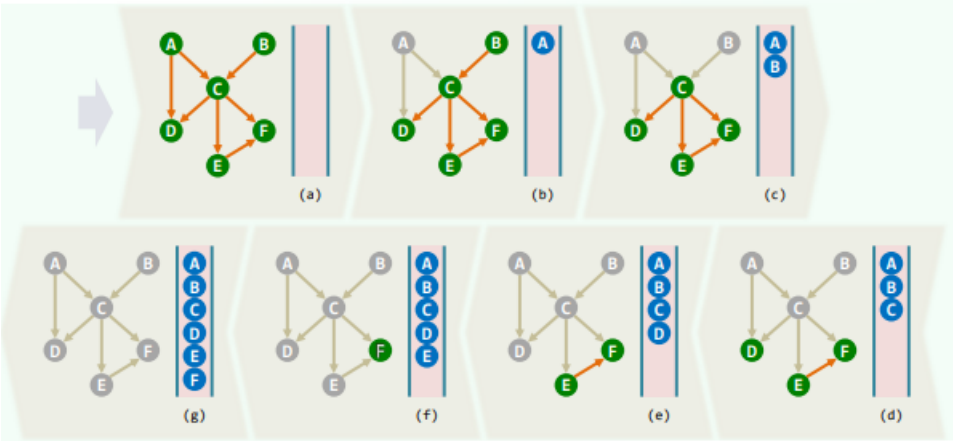
拓扑排序

有向无环图：Directed Acyclic Graph

> 任给有向图G（不一定是DAG），尝试将所有顶点排成一个**线性序列**，使其次序与原图**相容**（每一顶点都不会通过边指向**前驱**顶点）
>
> 若原图存在回路（即并非DAG），检查并报告。否则，给出一个相容的线性序列

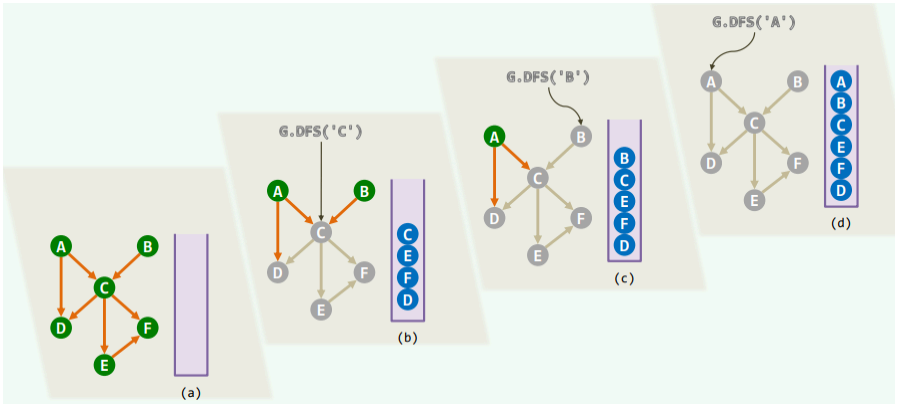
策略：**顺序**输出零**入**度顶点

```
//将所有入度为零的顶点存入栈S，取空队列Q：O(n)
while ( ! S.empty() ) { //O(n)
    Q.enqueue( v = S.pop() ); //栈顶v转入队列
    for each edge( v, u ) //v的邻接顶点u若入度仅为1
    if ( u.inDegree < 2 ) S.push( u ); //则入栈
    G = G \ { v }; //删除v及其关联边（邻接顶点入度减1）
} //总体O(n + e)
return |G| ? "NOT_A_DAG" : Q; //残留的G空，当且仅当原图可拓扑排序
```



零出度算法：**逆序**输出零**出**度顶点

> *（基于DFS，借助栈S）*
> 对图G做DFS，其间
> 每当有顶点被标记为VISITED，则将其压入S
> 一旦发现有**后向边**，则报告“NOT_A_DAG”并退出
> DFS结束后，顺序弹出S中的各个顶点
> 各节点按**ftime**逆序排列，即是拓扑排序
> 复杂度与DFS相当，也是$O(n+e)$




```
template <typename Tv, typename Te> //顶点类型、边类型
bool Graph<Tv, Te>::TSort( Rank v, int & clock, Stack<Tv>* S ) {
    dTime(v) = ++clock;
    status(v) = DISCOVERED; //发现顶点v
    for ( Rank u = firstNbr(v); u < INT_MAX; u = nextNbr(v, u) ) //考查v的每一邻居u
        switch ( status(u) ) { //并视u的状态分别处理
            case UNDISCOVERED:
                parent(u) = v; type(v, u) = TREE;
                if ( ! TSort(u, clock, S) ) //从顶点u处深入
                    return false; break;
            case DISCOVERED: //一旦发现后向边（非DAG）
                type(v, u) = BACKWARD; //则退出而不再深入
                return false;
            default: //VISITED (digraphs only)
                type(v, u) = dTime(v) < dTime(u) ? FORWARD : CROSS;
                break;
        }
    status(v) = VISITED;
    S->push( vertex(v) ); //顶点被标记为VISITED时入栈
    return true;
}
```

10.4 优先级搜索 PFS

不同顶点选取策略，取决于存放和提供顶点的**数据结构**——**Bag**
此类结构，为每个顶点v维护一个**优先级数**——`priority(v)`
每个顶点都有**初始**优先级数，并可能随算法的推进而**调整**
通常的习惯是，优先级数越**大/小**，优先级越**低/高**

```
template <typename Tv, typename Te>
template <typename PU> //优先级更新器（函数对象）
void Graph<Tv, Te>::PFS( Rank v, PU prioUpdater ) { //PU的策略
    priority(v) = 0; status(v) = VISITED; parent(v) = -1; //起点v加至PFS树中
    while (1) { //将下一顶点和边加至PFS树中
        // update
        for ( Rank u = firstNbr(v); -1 < u; u = nextNbr(v, u) )
            //对v的每一个邻居u
            prioUpdater( this, v, u ); //更新其优先级及其父亲
```

```
        // select
        for ( int shortest = INT_MAX, u = 0; u < n; u++ )
            if ( UNDISCOVERED == status(u) ) //从尚未加入遍历树的顶点中
                if ( shortest > priority(u) ) //选出下一个
                    { shortest = priority(u); v = u; } //优先级最高的顶点v
        // visit
        if ( VISITED == status(v) ) break; //直至所有顶点均已加入
        status(v) = VISITED; type( parent(v), v ) = TREE; //将v加入树
    } //while
} //如何推广至非连通图?
```

复杂度：

- **更新**树外顶点的优先级：$O(n+e)$
- **选出**新的优先级最高者：$O(n^2)$

> ❖ 执行时间主要消耗于内、外两重循环；其中两个内循环前、后并列
>
> ❖ 前一内循环的累计执行时间：若采用邻接矩阵，为O(n²)；若采用邻接表，为O(n+e)
>
> 后一循环中，优先级更新的次数呈算术级数变化{ n, n - 1, ..., 2, 1 }，累计为O(n²)
>
> 两项合计，为O(n²)
>
> ❖ 后面将会看到：若采用优先级队列，以上两项将分别是O(e*logn)和O(n*logn) //保持兴趣
>
> 两项合计，为O((e+n)*logn)
>
> ❖ 这是很大的改进——尽管对于稠密图而言，反而是倒退 //已有接近于O(e + n*logn)的算法
>
> ❖ 基于这个统一框架，如何解决具体的应用问题...

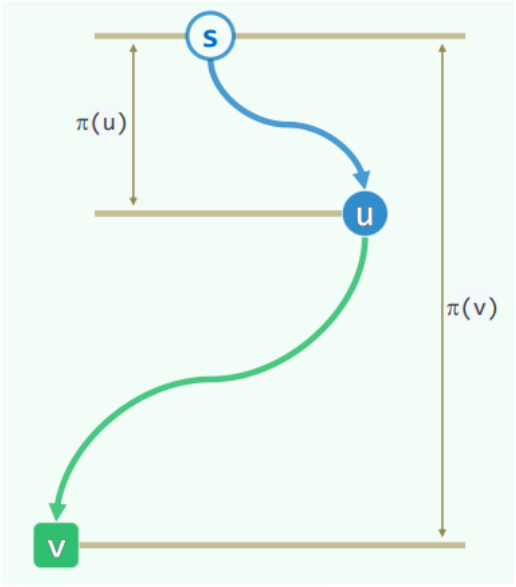
10.5 Dijkstra算法：最短路径

> Dijkstra：SSSP / Single-Source Shortest Path 给定顶点s，计算s到**其余**各个顶点的最短路径及长度
> Floyd-Warshall：APSP / All-Pairs Shortest Path 找出**每对**顶点u和v之间的最短路径及长度

1. 最短路径树

Lemma：任一最短路径的前缀，也是一条最短路径

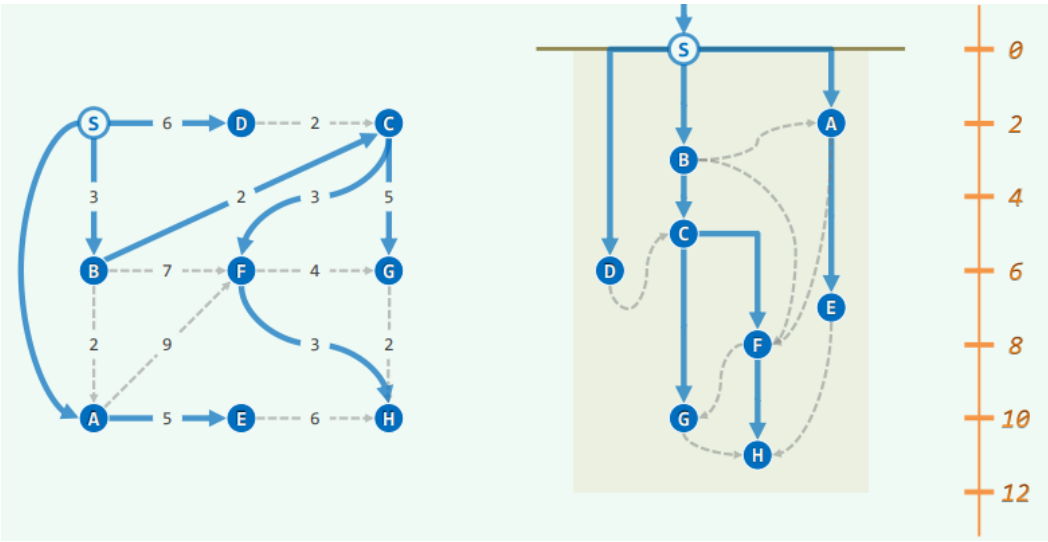
> $u \in \pi(v)\ only\ if\ \pi(u) \subseteq \pi(v)$



消除歧义：

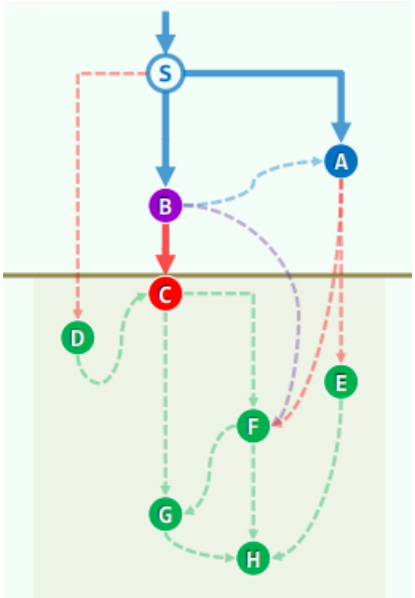
> 各边权重均为正，否则
> 有可能出现总权重**非正**的环路
> 以致最短路径无从定义
> 有**负权重的边**时，即便多有环路总权重皆为正，Dijkstra算法**依然可能**失效
> 任意两点之间，最短路径唯一
> 在不影响计算结果的前提下，总可通过适当扰动予以保证（*）

Shortest Path Tree：



2. 算法实现

> $\forall v \notin V_k$, let $priority(v) = ||s, v|| \leq \infty$
> 于是套用PFS框架，为将T_k扩充至T_{k+1}，只需
> 选出优先级最高的跨边e_k及其对应顶点u_k，并将其加入T_k
> 随后，更新V *or* V_{k+1}中所有顶点的优先级（数）
> 注意：优先级数随后可能改变（降低）的顶点，必与u_k**邻接**
> 因此，只需枚举u_k的每一邻接顶点v，并取
> $priority(v) = min(priority(v), priority(u_k) + ||u_k, v||)$
> 以上完全符合PFS的框架，唯一要做的工作无非是按照`prioUpdater()`规范，编写一个优先级（数）更新器...




```
//Prority Updater ~ DijkPU
g->pfs( 0, DijkPU<char, Rank>() ); //从顶点0出发，启动Dijkstra算法
template <typename Tv, typename Te> struct DijkPU {
    virtual void operator()( Graph<Tv, Te>* g, Rank v, Rank u ) {
        //对v的每个
        if ( UNDISCOVERED != g->status(u) ) return; //尚未被发现的邻居u，按
        if ( g->priority(u) > g->priority(v) + g->weight(v, u) ) { //Dijkstra
            g->priority(u) = g->priority(v) + g->weight(v, u); //策略
            g->parent(u) = v; //做松弛
        }
    }
};
```

10.6 Floyd-Warshall算法：最短路径

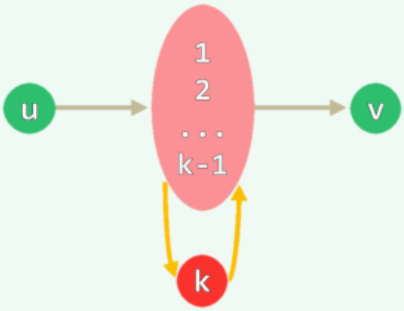
给定带权网络G，计算其中**所有点对**之间的最短距离
应用：确定G的中心点（center）
radius(G, s) = s的半径 = 所有顶点到s的最大距离
中心点 = 半径最小的顶点s
直觉：依次将个顶点作为源点，调用Dijkstra算法
时间 = $n \times O(n^2) = O(n^3)$
思路：图矩阵 ~ 最短路径矩阵
效率：$O(n^3)$，与执行n次Dijkstra相同
优点：形式简单、算法紧凑、便于实现；允许**负权边**（但不允许**负权环路**）

1. 思路

❖ u和v之间的最短路径可能是

- 不存在通路，或者
- 直接连接，或者
- 最短路径(u, x) + 最短路径(x, v)





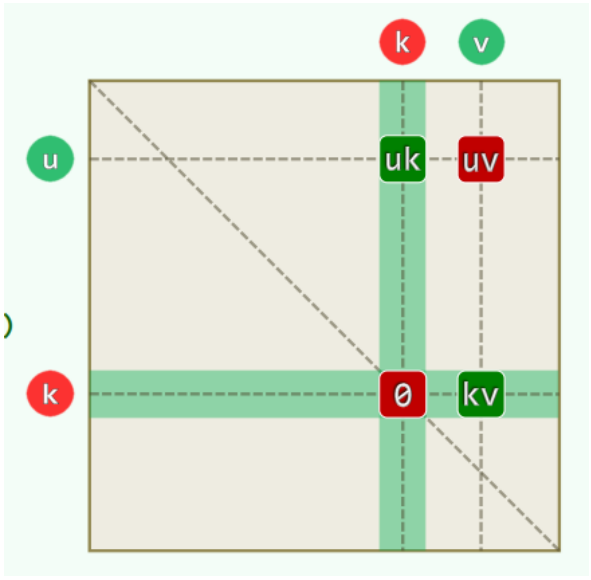
❖ 将所有顶点随意编号：1, 2, ..., n

定义： $d^k(u,v)$ = 中途只经过前k个顶点中转，从u通往v的最短路径长度

- $d^k(u,v) = w(u,v) \qquad (\text{if } k = 0)$
- $d^k(u,v) = \min\{\, d^{k-1}(u,v),\ d^{k-1}(u,k) + d^{k-1}(k,v) \,\} \qquad (\text{if } k \geq 1)$

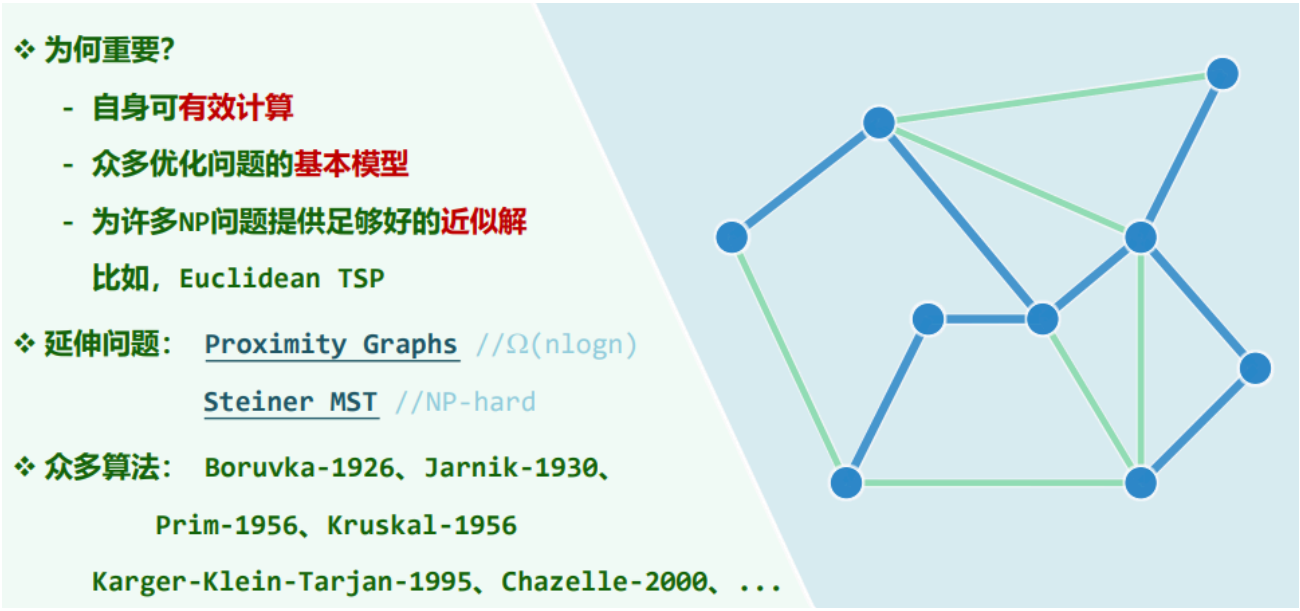
动态规划：

```
for k in range(0, n)
    for u in range(0, n)
        for v in range(0, n)
            A[u][v] = min(A[u][v], A[u][k] + A[k][v])
```



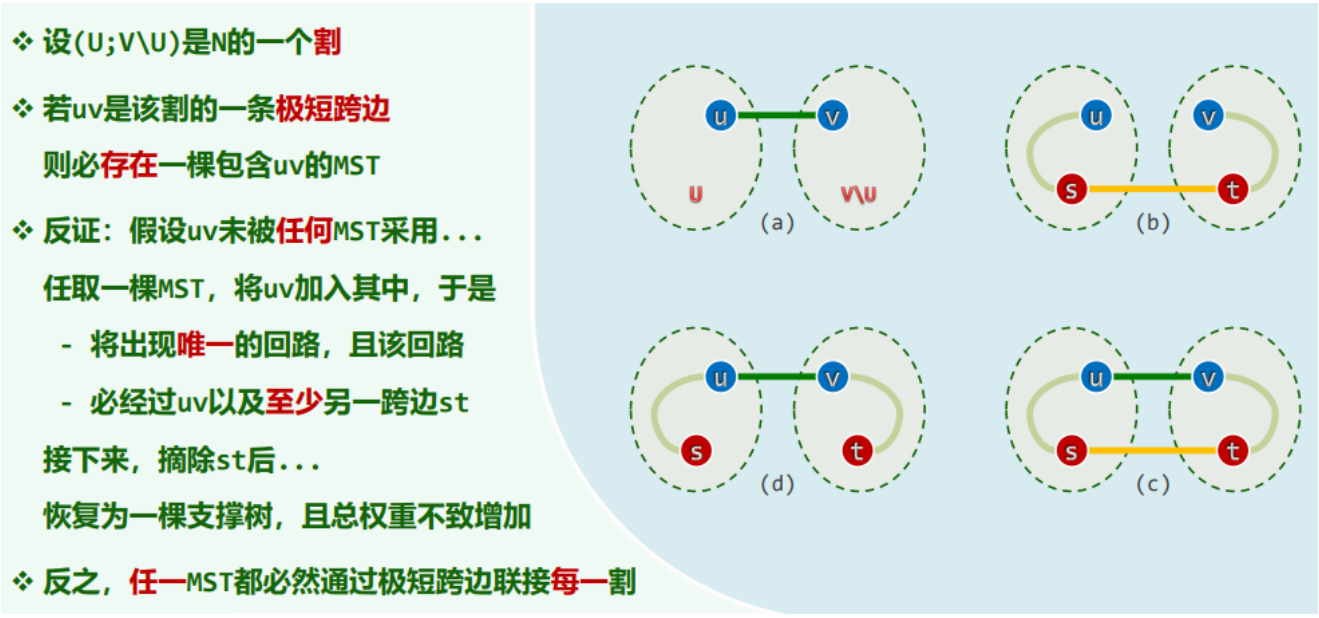
10.7 Prim算法：最小支撑树 MST

> 连通网络$N = (V; E)$的子图$T = (V; F)$
> **支撑 / spanning** = 覆盖N中所有顶点
> **树 / tree** = 连通且无环，$|F| = |V| - 1$
> 同一网络的支撑树，未必唯一
> **minimum = optimal**: 总权重$w(T) = \sum_{e \in F} w(e)$达到最小



1. Prim算法：极短跨边

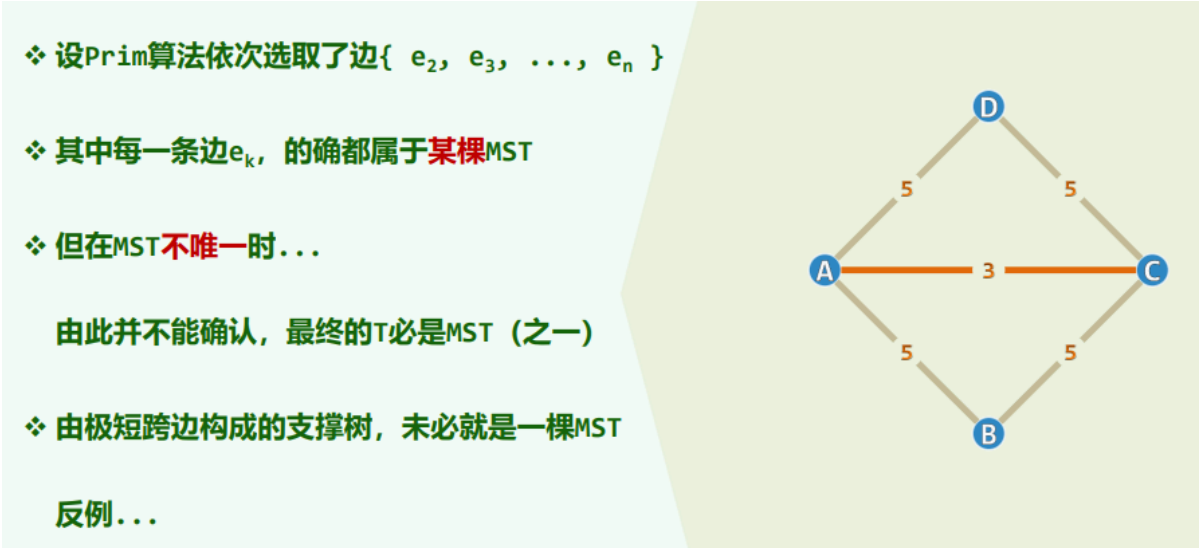
> 任何环路C上的**最长**边f，都不会被MST采用
> 否则，移除f之后，MST将**分裂**为两棵树，将其视作一个**割**，则C上必有该割的另一**跨边**e，既然$|e| < |f|$，那么只要用e**替换**f，就可以的到一棵**总权重更小**的支撑树
>
> 若uv是某一割的一条**极短跨边**，则必存在一棵包含uv的MST
> **任一**MST都必然通过极短跨边连接**每一**割



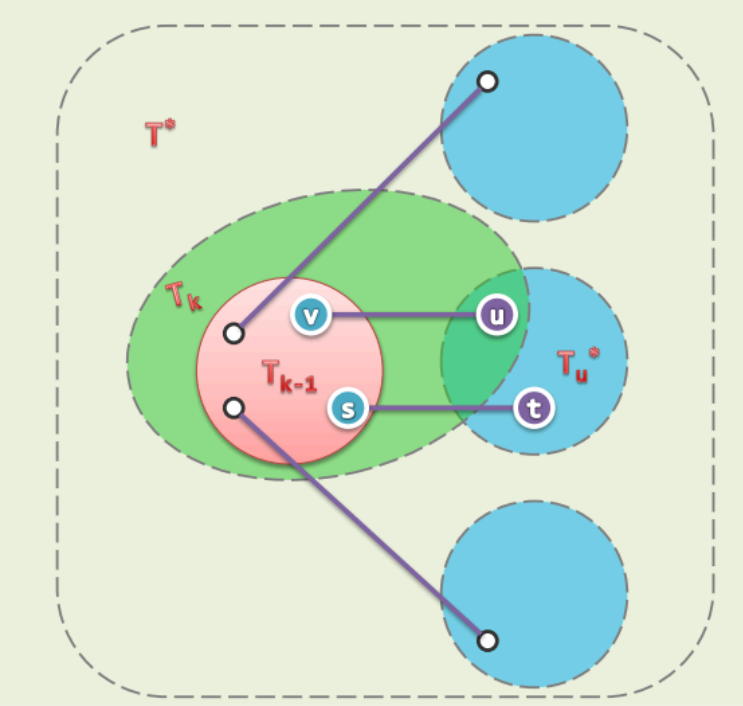
> 递增式构造：$T_{k+1} = (V_{k+1}; E_{k+1}) = (V_k) \cup \{v_{k+1}\}; E_k \cup \{v_{k+1}u\})$，其中$u \in V_k$

注意：当MST不唯一时，由极短跨边构成的支撑树，未必就是一棵MST

> 反例：

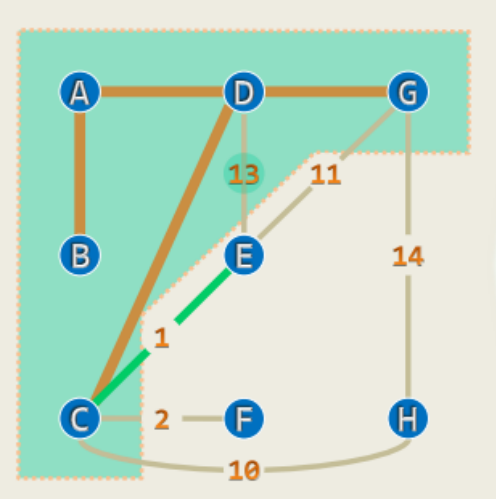


可行的证明：
在不增加总权重的前提下，可以将任一MST**转换**为T，每一T_k都是某棵MST的**子树**



2. Prim算法：实现

于是套用PFS框架，为将T_k扩充至T_{k+1}，只需
选出优先级最高的跨边e_k及其对应顶点u_k，并将其加入T_k
随后，更新$V\backslash V_{k+1}$中所有顶点的优先级（数）
注意：优先级数随后可能改变（降低）的顶点，必与u_k**邻接**
因此，只需枚举u_k的每一邻接顶点v，并取
$priority(v) = min(priority(v), ||u_k, v||)$
以上完全符合PFS的框架，唯一要做的工作无非是按照`prioUpdater()`规范，编写一个优先级（数）更新器...




```
// Priority Updater ~ PrimPU
g->pfs( 0, PrimPU<char, Rank>() ); //从顶点0出发，启动Prim算法
template <typename Tv, typename Te> struct PrimPU { //Prim算法的顶点优先级更新器
    virtual void operator()( Graph<Tv, Te>* g, Rank v, Rank u ) {
        if ( UNDISCOVERED != g->status(u) ) return; //对v的每个尚未被发现的邻居u，按
        if ( g->priority(u) > g->weight(v, u) ) { //Prim
            g->priority(u) = g->weight(v, u); //策略
            g->parent(u) = v; //做松弛
        }
    }
};
```

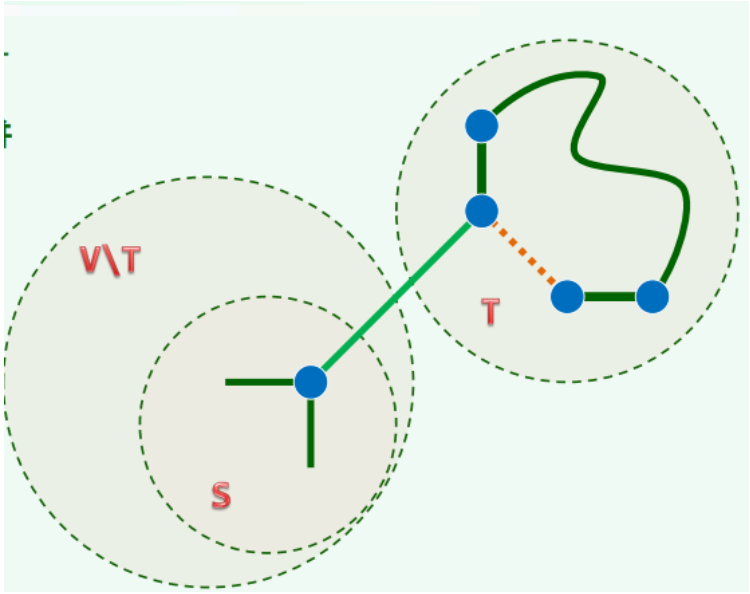
10.8 Kruskal算法

> 贪心原则：
> 根据代价，从小到大依次尝试各边
> 只要“安全”就加入该边
> *（每步局部最优 =全局最优）*

1. 正确性

定理：Kruskal引入的每条边都属于**某棵**MST
设：边$e = (u, v)$的引入导致树T和S的合并
若：将$(T; V|T)$视作原网络N的割，则e当属于该割的一条**跨边**
在确定引入e之前，该割的所有跨边都经Kruskal考察，且只可能因不短于e而被淘汰
故：e应当是该割的一条**极短跨边**

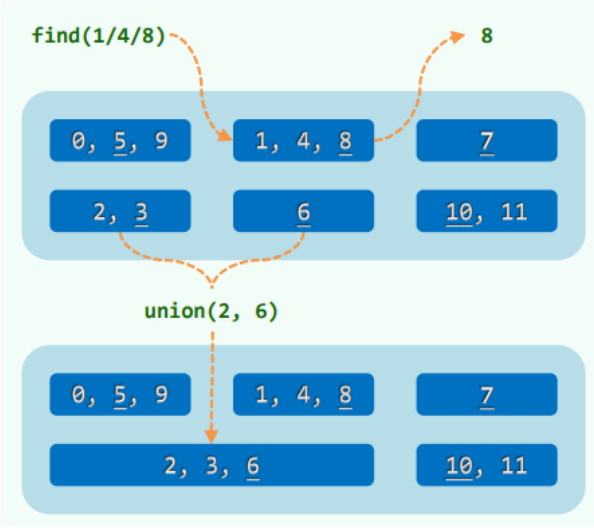
> Kruskal算法过程中不断生长的森林，总是**某棵**MST的**子图**



2. 并查集

并查集 - OI Wiki (oi-wiki.org)

> Union-Find问题：
> 给定一组互不相交的等价类
> 各由其中一个成员作为代表
>
> `Find(x)`：找到元素x所属等价类
> `Union(x, y)`：合并x和y所属等价类
> `Singleton`：初始时各包含一个元素
>
> Kruskal = Union-Find



- **查询**：沿着树向上移动，直至找到根节点
- **路径压缩**：查询过程中经过的每个元素都属于该集合，将其直接镰刀根节点以加快后续查询
- **合并**：将一棵树的根节点连到另一棵树的根节点
  启发式合并：选择哪棵树的根节点作为新树的根节点会影响未来操作的复杂度，我们可以将节点较少或深度较小的树连到另一棵，以免发生退化
- **删除**：将删除的叶子节点的父亲设为自己。（制作副本）
- **移动**：通过以副本作为父亲，保证要移动的元素都是叶子。

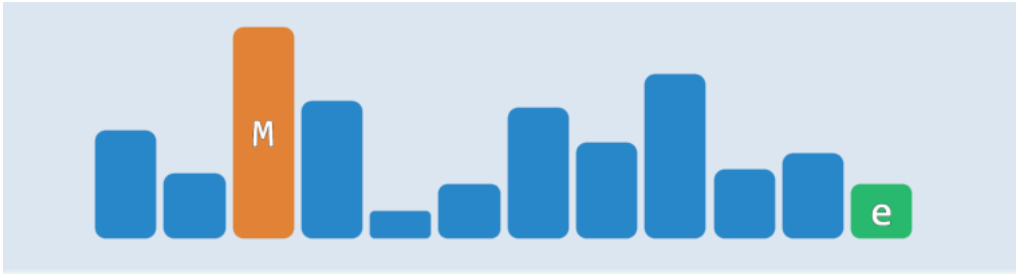
11 优先级队列

```
template <typename T> struct PQ { // priority queue
    virtual void insert(T) = 0;
    virtual T getMax() = 0;
    virtual T delMax() = 0;
};
```

> 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也都是PQ的特例——插入和删除的**位置受限**

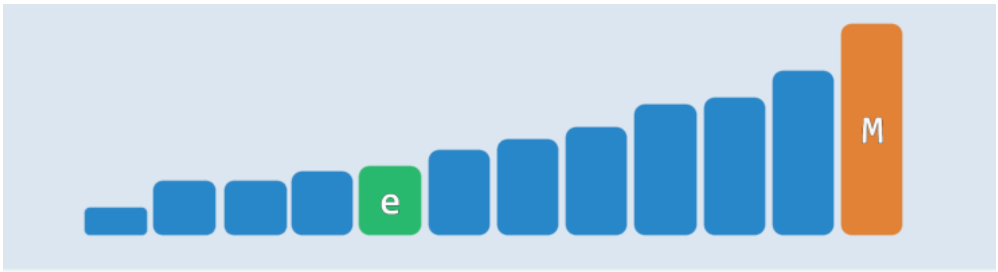
引入

1. Vector



getMax()	delMax()	insert()
traverse() Θ(n)	remove(traverse()) Θ(n) + O(n) = Θ(n)	insertAsLast(e) O(1)

Sorted Vector:



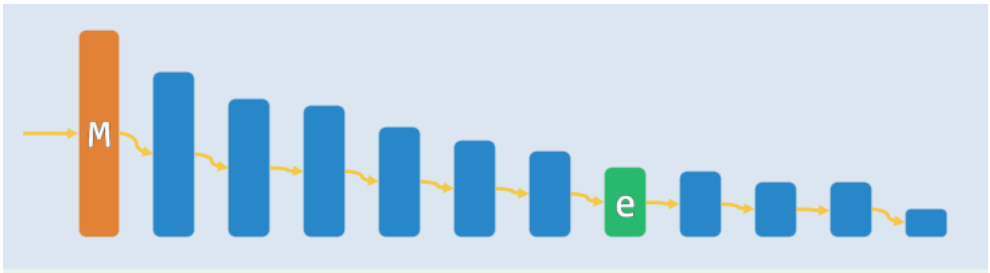
getMax()	delMax()	insert()
[n - 1] O(1)	remove(n - 1) O(1)	insert(1 + search(e), e) O(logn) + O(n) = O(n)

2. List



getMax()	delMax()	insert()
traverse() $\Theta(n)$	remove(traverse()) $\Theta(n) + O(1) = \Theta(n)$	insertAsFirst(e) $O(1)$

Sorted List:



getMax()	delMax()	insert()
first() $O(1)$	remove(first()) $O(1)$	insertA(search(e), e) $O(n) + O(1) = O(n)$

3. BBST

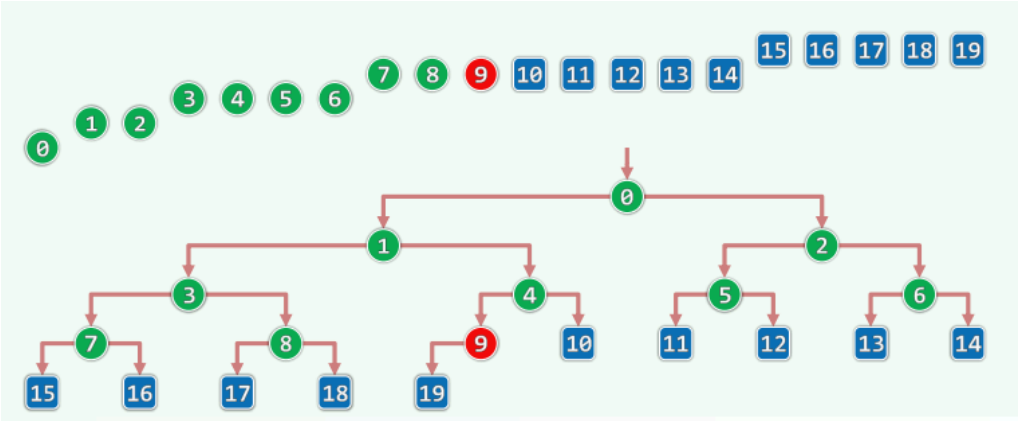
AVL、Splay、Red-black：三个接口均只需$O(logn)$时间

> 但是BBST的功能**远远超出了**PQ的需求：
> 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

11.1 完全二叉堆

结构性：逻辑元素、物理节点依**层次遍历**次序彼此对应
逻辑上等同于**完全二叉树**
物理上直接借助**向量**实现
内部节点的最大秩 = $\left[\frac{n-1}{2}\right]_{下整} = \left[\frac{n-3}{2}\right]_{上整}$

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



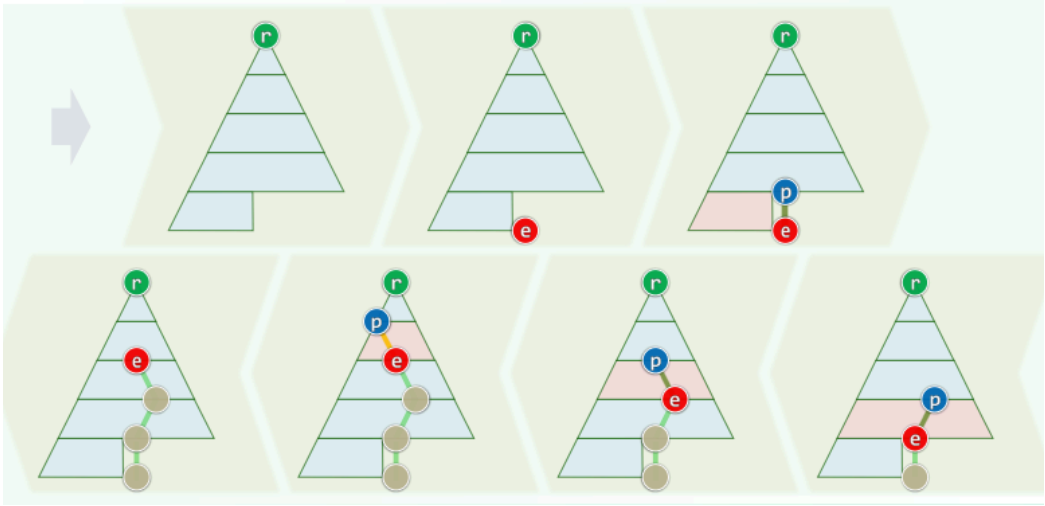
PQ_ComplHeap = PQ + Vector

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

堆序性：只要$0 < i$，必然满足$H[i] \leq H[Parent(i)]$，故$H[i]$即是全局最大

1. 插入

算法：逐层上滤



```
template <typename T>
void PQ_ComplHeap<T>::insert( T e ) {
    Vector<T>::insert( e );
    percolateUp( _elem, _size - 1 );
```

```
} //此insert()非彼

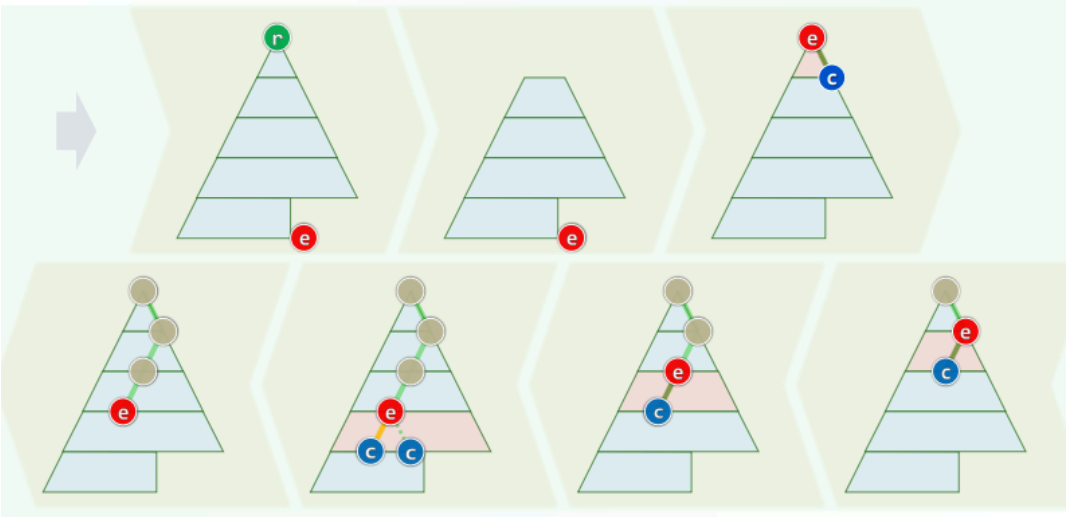
template <typename T>
Rank percolateUp( T* A, Rank i ) { //0 <= i < _size
    while ( 0 < i ) { //在抵达堆顶之前，反复地
        Rank j = Parent( i ); //考查[i]之父亲[j]
        if ( lt( A[i], A[j] ) ) break; //一旦父子顺序，上滤旋即完成；否则
        swap( A[i], A[j] ); i = j; //父子换位，并继续考查上一层
    } //while
    return i; //返回上滤最终抵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

效率：

> e在上滤过程中，只可能与**祖先**们交换
> 完全树必平衡，e的祖先不超过$O(logn)$个，因此时间复杂度在$O(logn)$内
> 然而，就**数学期望**而言，实际效率往往远远**更高**

2. 删除

算法：割肉补疮 + 逐层下滤




```
template <typename T> T PQ_ComplHeap<T>::delMax() { //删除
    T maxElem = _elem[0];
    _elem[0] = _elem[ --_size ]; //摘除堆顶，代之以末词条
    percolateDown( _elem, _size, 0 ); //对新堆顶实施下滤
    return maxElem; //返回此前备份的最大词条
}
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）
}
```

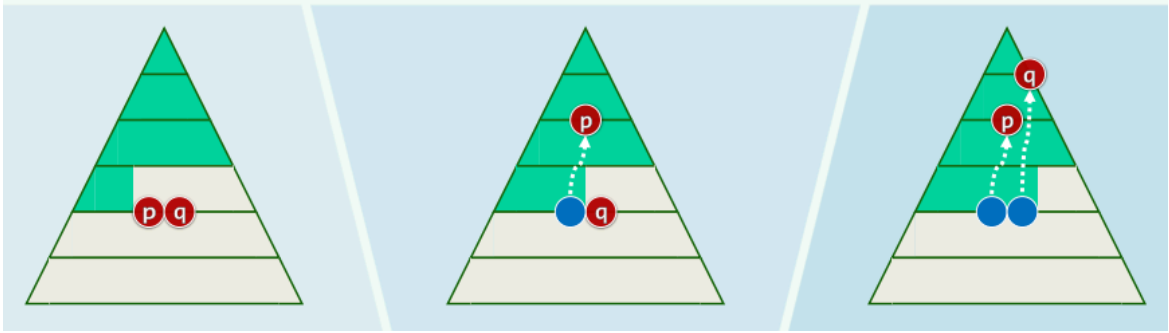
效率：

> e在每一高度至多交换一次，累计交换不超过$O(logn)$次
> 通过下滤，可在$O(logn)$时间内
> 删除堆顶节点，并整体重新调整为堆
> 然而，就**数学期望**而言，实际效率往往远远**更高**

3. 批量建堆

蛮力算法：

```
// 自上而下的上滤
PQ_ComplHeap( T* A, Rank n )
    { copyFrom( A, 0, n ); heapify( _elem, n ); }
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
}
```

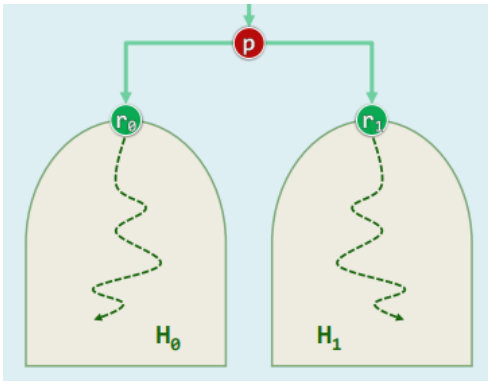


> Worst Case = $O(nlogn)$

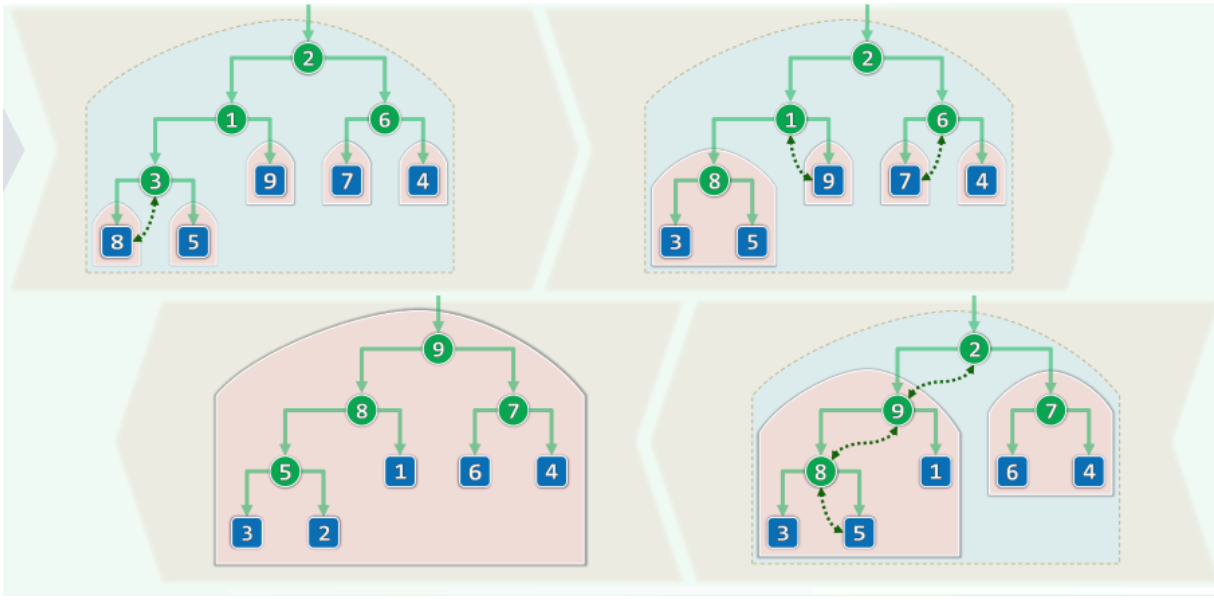
自下而上的下滤：

> 任意给定堆H_0和H_1，以及节点P
> 未得到堆$H_0 \cup \{p\} \cup H_1$，只需将r_0和r_1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; 0 <= i; i-- ) //依次（从最后一个内部节点）
    percolateDown( A, n, i ); //下滤各内部节点
} //可理解为子堆的逐层合并，堆序性最终必然在全局恢复
```



Example：



效率：

> 每个内部节点所需的调整时间，正比于其**高度**而非**深度**
> 不失一般性，考察满树：$n = 2^{h+1} - 1$
> 所有节点的**高度**总和：
>
> $$S(n) = \sum_{k=1}^{h} k \cdot 2^{h-k} = \sum_{k=1}^{h} \sum_{i=1}^{k} 2^{h-k} = \sum_{i=1}^{h} \sum_{k=i}^{h} 2^{h-k}$$
> $$= \sum_{i=1}^{h} \sum_{k=0}^{h-i} 2^{k} = \sum_{i=1}^{h} \{2^{h-i+1} - 1\} = \sum_{i=1}^{h} 2^{h-i+1} - h$$
> $$= \sum_{i=1}^{h} 2^{i} - h = 2^{h+1} - 2 - h = \mathcal{O}(n)$$
>
> **算法优化的本质**：堆结构“深度”和“高度”的不确定性。采用高度累计减少复杂度。

课后思考：

课后

❖ insert()：最坏情况下效率为O(logn)，平均情况呢？

❖ heapify()：构造次序颠倒后，为什么复杂度会有实质降低？

这一算法在哪些场合不适用？

❖ 扩充接口：decrease(i, delta) //任一元素_elem[i]的数值减小delta

increase(i, delta) //任一元素_elem[i]的数值增加delta

remove(i) //删除任一元素_elem[i]

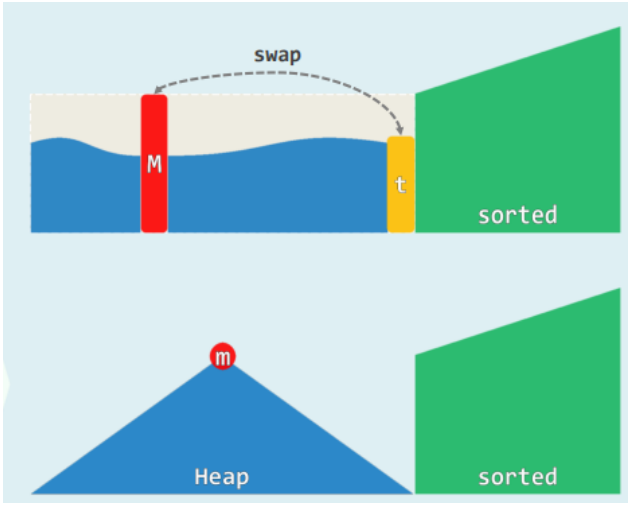
❖ 借助完全堆，在$\mathcal{O}(n \log n)$时间内构造Huffman树

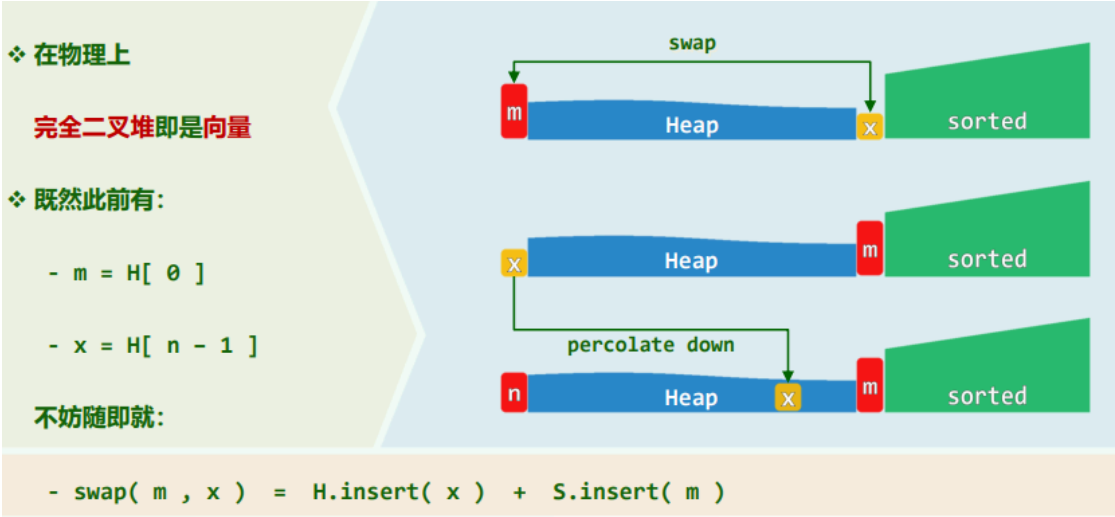
❖ 大顶堆的delMin()操作，能否也在 $\mathcal{O}(\log n)$ 时间内完成？

难道，为此需要同时维护一个小顶堆？

11.2 堆排序

> 对比选择排序`SelectionSort()`：
> 初始化：`heapify()`，$O(n)$
> 迭代：`delMax()`，$O(logn)$
> 不变性：$H \leq S$
>
> $O(n) + n \times O(logn) = O(nlogn)$
> ——J. Williams, 1964

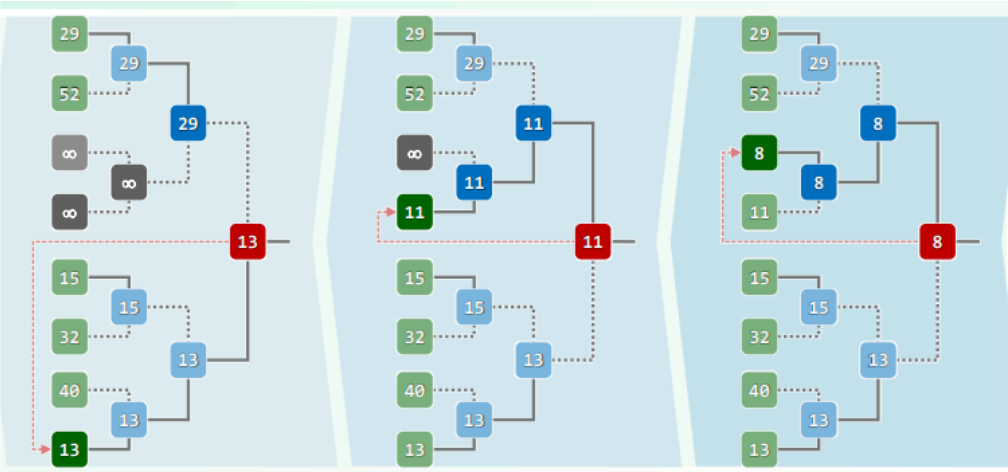





```
template <typename T> //对向量区间[lo, hi)做就地堆排序
void Vector<T>::heapSort( Rank lo, Rank hi ) {
    T* A = _elem + lo; Rank n = hi - lo;
    heapify( A , n ); //待排序区间建堆，O(n)
    while ( 0 < --n ) //反复地摘除最大元并归入已排序的后缀，直至堆空
        { swap( A[0], A[n] ); percolateDown( A, n, 0 ); } //堆顶与末元素对换后下滤
}
```

1. 锦标赛树：胜者树

> **完全二叉树**
> 叶节点：待排序元素（选手）
> 内部节点：孩子中的胜者
> 树根总是全局**冠军**：类似于堆顶
> 内部结点各对应于一场比赛的胜者——重复存储
>
> `create()`：$O(n)$
> `remove()`：$O(logn)$
> `insert()`：$O(logn)$
>
> `Tournamentsort()`
> CREATE a tounament tree for the input list while there are active leaves
> RMOVE the root
> RETRACE the root down to its leaf
> DEACTIVATE the leaf
> REPLAY along the path back to the root



效率：

空间：$O(\text{节点数}) = O(\text{叶节点数}) = O(n)$

构造：仅需$O(n)$时间

更新：每次只需要重排上一优先者的**祖先**

为此，只需从其所在叶节点除法，逐层上溯直到树根

如此，为确定各轮优胜者，总共所需时间仅$O(logn)$

时间：$n\text{轮} \times O(logn) = O(nlogn)$

选取：从n个元素中找出最小的k个

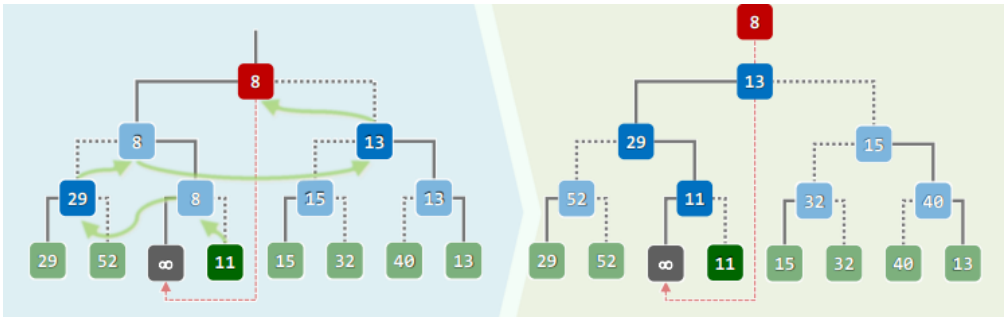
初始化：$O(n)$

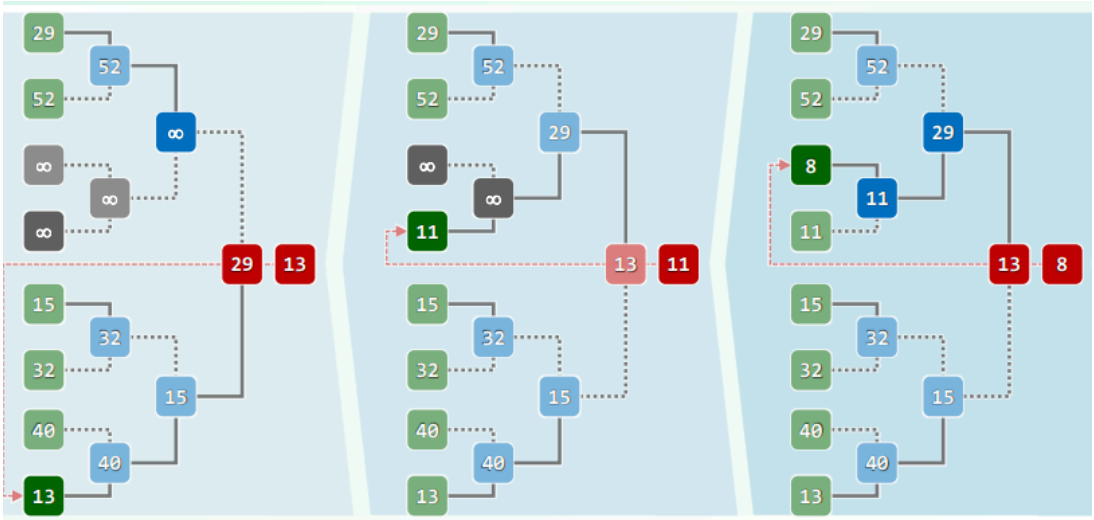
迭代k步：$O(klogn)$，与**小顶堆**旗鼓相当？

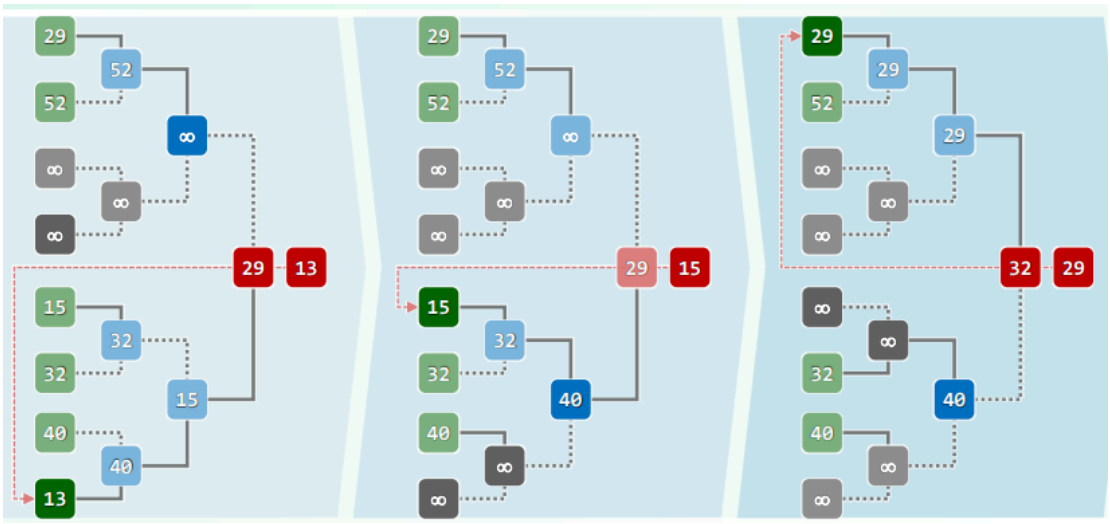
> **渐进**意义上确实如此，但就**常系数**而言，区别不小
> `Floyd`算法，`delMax()`中的`percolateDown()`在每一层需做2次比较，累计大致$2logn$次

2. 锦标赛树：败者树

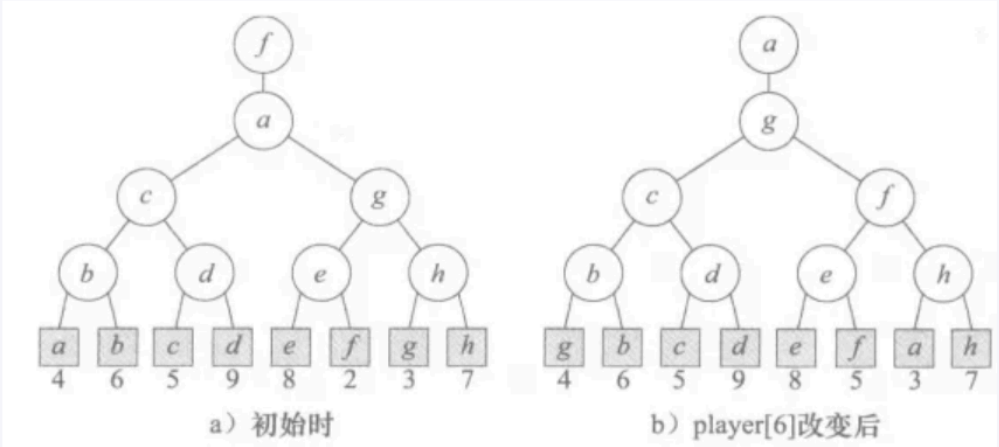
> 内部节点记录对应比赛的**败者**
> 增设根的“父节点”，记录冠军
> ——这样的算法设计是因为胜者树的REPLAY涉及“找**兄弟**比赛”的过程（时间上就有消耗）
> 而采用败者树REPLAY时每次的比赛只需要找**父亲**即可




29
52
52
∞
∞
∞
29
13
15
32
32
15
40
40
13
29
52
52
29
∞
∞
11
13
11
15
32
32
15
40
40
13
29
52
52
29
8
11
11
13
8
15
32
32
15
40
40
13

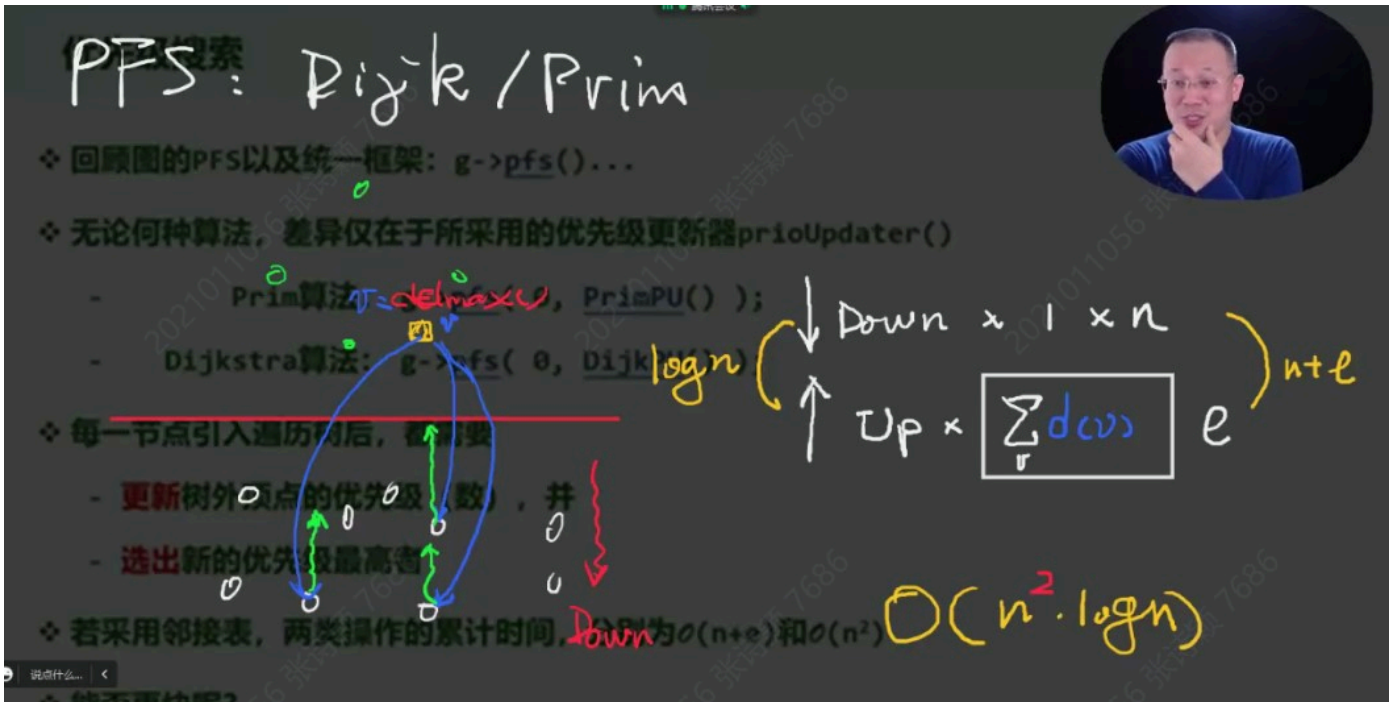

29
52
52
∞
∞
∞
29
13
15
32
32
15
40
40
13
29
52
52
∞
∞
∞
29
15
15
32
32
40
40
∞
∞
29
29
52
52
29
∞
∞
∞
32
29
∞
∞
32
40
40
∞
∞

- 如果每个内部节点记录的是在该节点比赛的输者而不是赢者，那么当赢者player[i]改变后，在从该节点到根的路径上，重新确定每一场比赛的选手所需要的操作量就可以减少。最终的赢者科技路在tree[0]中
- 下图a与图b相对应的输者树，它有8名选手。当赢者f的关键字编程5时，我们移动带它的父节点tree[6]进行比赛，比赛的选手是player[tree[6]]和player[6]
- 也就是说，为确定选手f'=player[6]的对手，只需简单地查看tree[6]即可，而在赢者树中，还需要查看tree[6]的其它子节点
- 在tree[6]的比赛完成后，输者e被记录在此节点，f'继续在tree[3]比赛，对手是前一场的输者g，而g就记录在tree[3]中
- 这次的输者是fg'，它被记录于tree[3]
- 赢者g则继续在tree[1]比赛，对手是上一场比赛的输者a，而a就记录在tree[1]。这次的输者是g，它被记录在tree[1]。新的输者树如下图b所示



11.3 多叉堆

> 背景：优先级搜索（例如Prim和Dijkstra算法）中，若采用邻接表，**更新**优先级和**选出**优先级最高者的累计时间分别为$O(n+e)$和$O(n^2)$。能否更快?



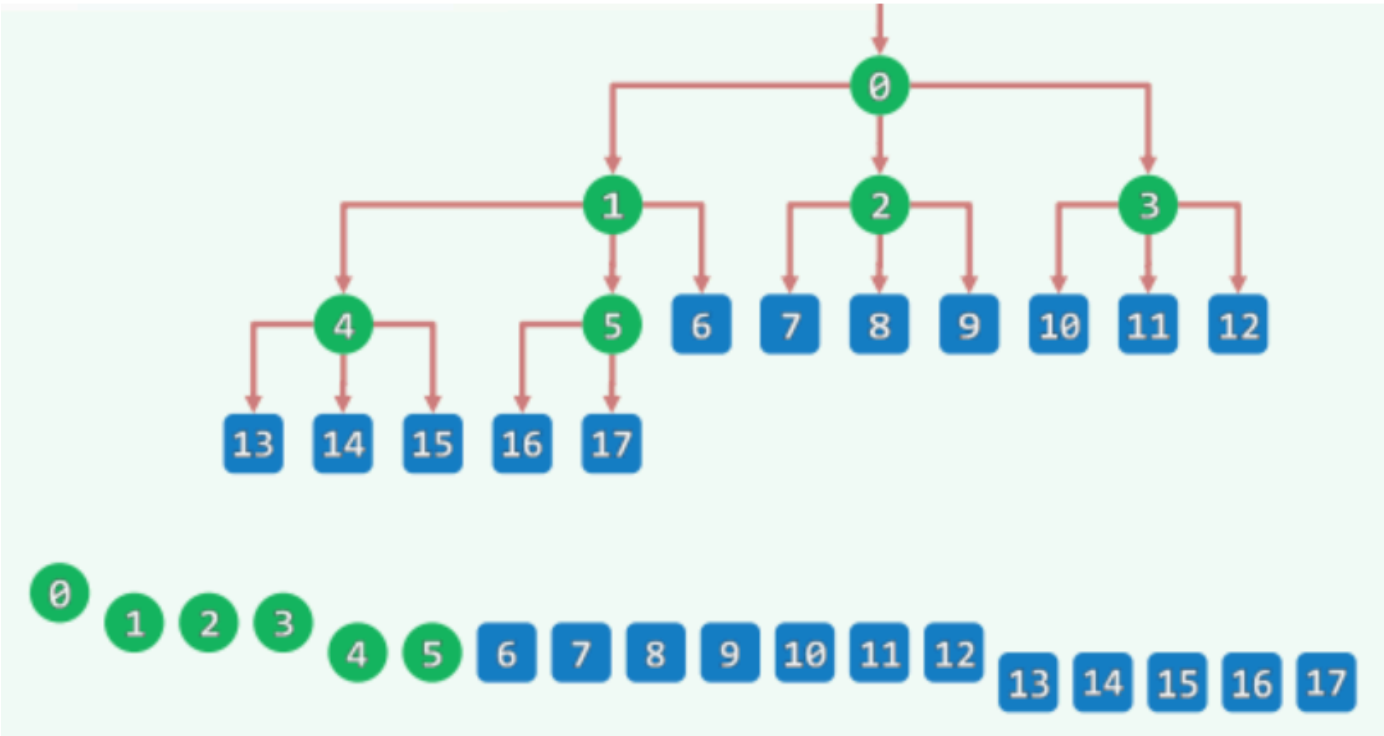
优先级队列对pfs的改进

> PFS中的各顶点可以组织为**优先级队列**
> 为此需要使用PQ接口：
> `heapify()`：由n个顶点创建初始PQ，总计$O(n)$
> `delMax()`：取优先级最高 / 极短跨边`(u, w)`，总计$O(nlogn)$
> `increase()`：更新所有关联顶点到u的距离，提高优先级，总计$O(elogn)$
> 总体运行时间 = $O((n+e)logn)$
>
> 对于**稀疏图**，处理效率很高
> 对于**稠密图**，反而不如常规实现的版本

多叉堆多优先级队列的改进

> 给予**Vector**实现，父子节点的**秩**可以简明地互相换算
> $parent(k) = [(k-1)/d]$
> $child(k,i) = k \cdot d + i, 0 < i \leq d$
> `heapify()`：$O(n)$，不可能再快了
> `delMax()`：$O(logn)$，实质就是`perculateDown()`，已是极限
> `increase()`：$O(logn)$，实质就是`perculateUp()`，还有改进空间...

将**二叉堆**改成**多叉堆**（d-heap）：
堆高度降至$O(log_d n)$
上滤 / `perculateUp()`成本降至$O(log_d n)$
下滤 / `perculateDown()`成本增至（只要$d > 4$）$d \cdot log_d n = \frac{lnn}{lnd} d$



则此时PFS时间复杂度为$n \cdot dlog_d n + e \cdot log_d n = (n \cdot d + e) \cdot log_d n$
当$d \approx \frac{e}{n} + 2$时，总体性能达到最优$O(e \cdot log_{(\frac{e}{n}+2)} n)$

对于**稀疏图**保持高效：$e \cdot log_{\frac{e}{n}+2} n \approx n \cdot log_{\frac{n}{n}+2} n = O(nlogn)$
对于**稠密图**改进较大：$e \cdot log_{\frac{e}{n}+2} n \approx n^2 \cdot log_{\frac{n^2}{n}+2} n \approx n^2 = O(e)$
对于一般的图，会**自适应**地实现最优

11.4 左式堆

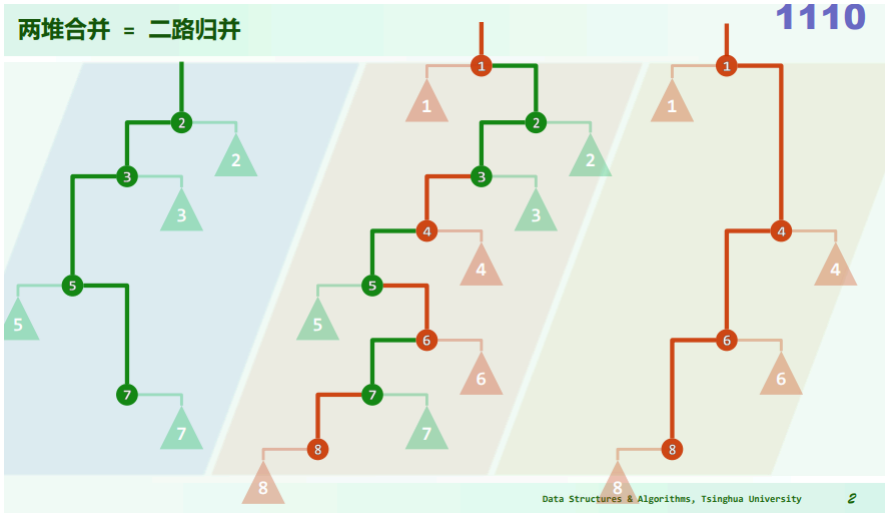
1. 堆合并

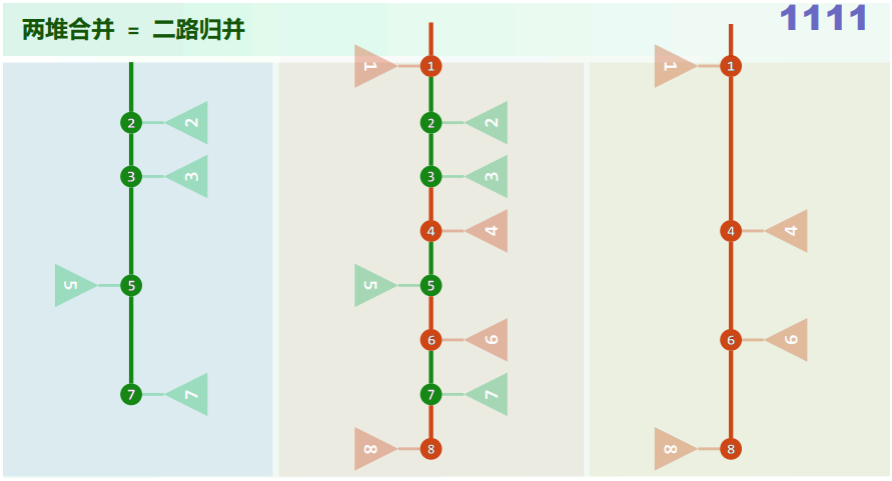
$H = merge(A, B)$：将堆A和B合二为一

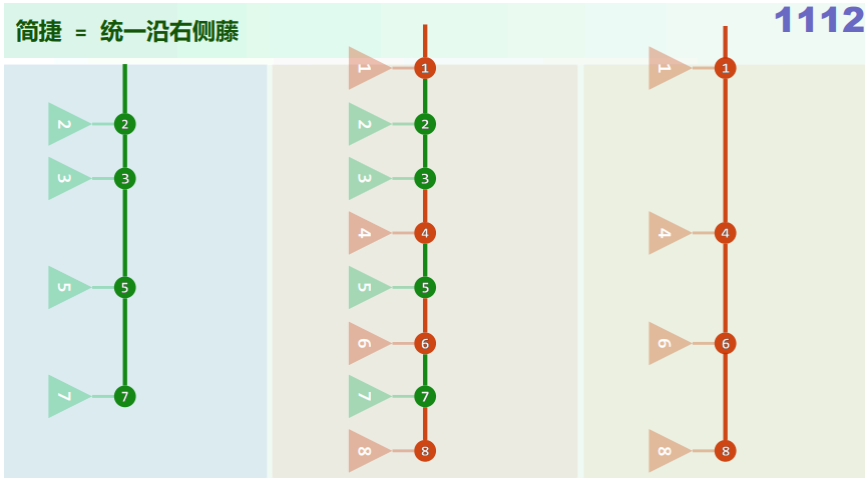
方法一：`A.insert(B.delMax())`
$O(m \times (logm + logm(n+m))) = O(m(log(n+m)))$
方法二：`union(A, B).heapify(n + m)`
$O(m+n)$

二堆合并 = 二路归并

> 放弃“完全性”（completeness），保留“堆序性”

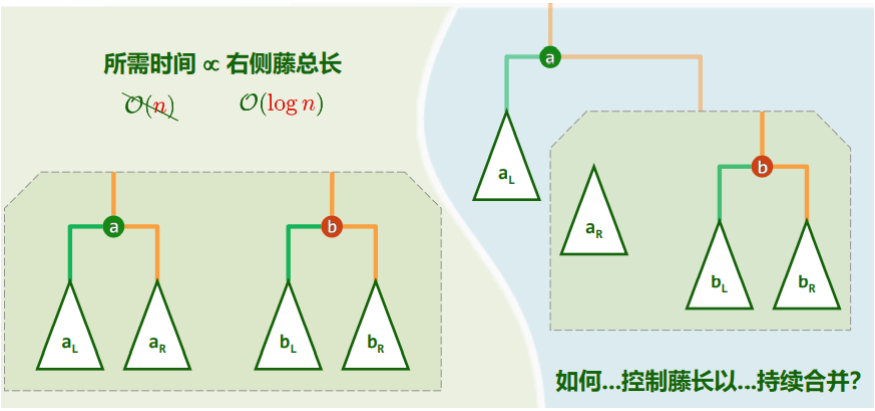






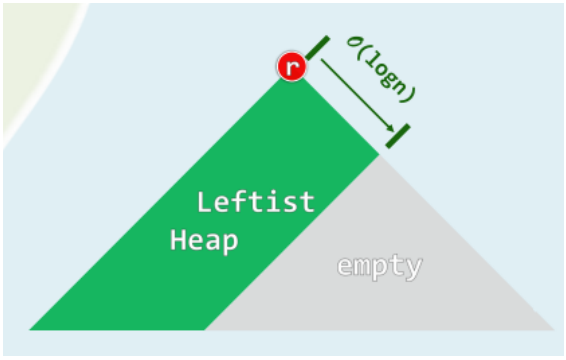
递归实现：所需时间正比于**右侧藤总长**$O(logn)$

可以保证藤长不超过$logn$...但是一次合并之后藤长变为$2logn$...



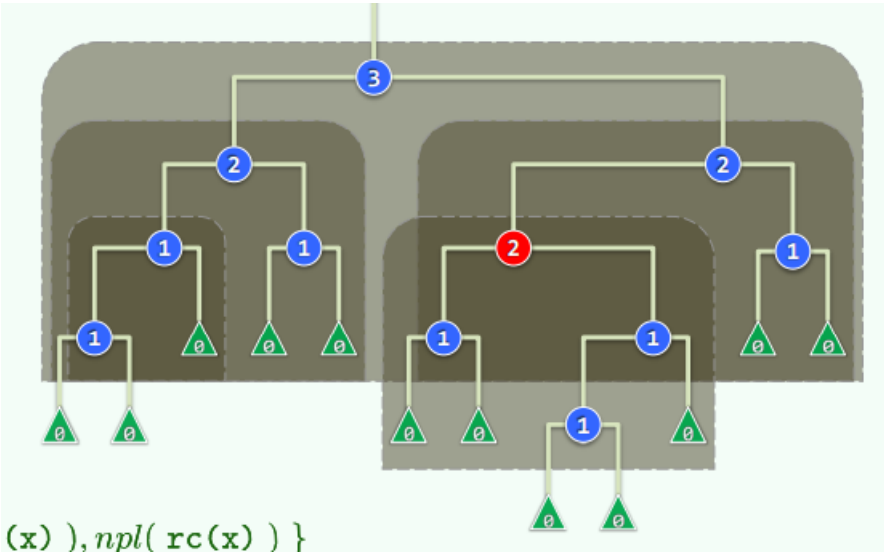
2. NPL与控制藤长

> C. A. Crane, 1972:
> 保持**堆序性**，附加**新条件**，使得在堆合并过程中，只涉及**少量**节点：$O(logn)$
>
> 新条件 = **单侧倾斜**:
> **节点分布**偏向**左**侧
> **合并操作**只涉及**右**侧



空节点路径长度

> 引入所有的外部节点
> 消除**一度**节点，转为**真**二叉树
>
> **Null Path Length**
> $npl(NULL) = 0$
> $npl(x) = 1 + min\{npl(lc(x)), npl(rc(x))\}$
>
> 验证：$npl(x) = x$到外部节点的最近距离 $=$ 以x为根的最大满子树的高度



左式堆

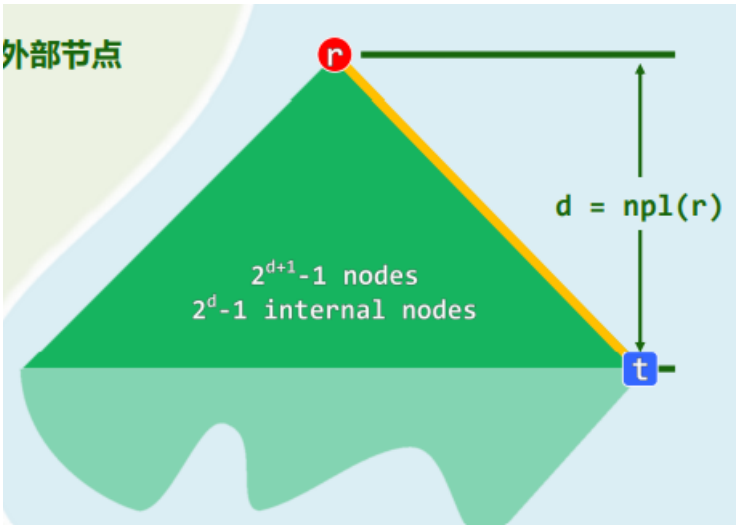
对任何内节点x，都有：$npl(lc(x)) \geq npl(rc(x))$
推论：$npl(x) = 1 + npl(rc(x))$

左倾性与堆序性，相容而不矛盾
左式堆的子堆必然也是左式堆
左式堆倾向于**更多**节点分布于左侧分支

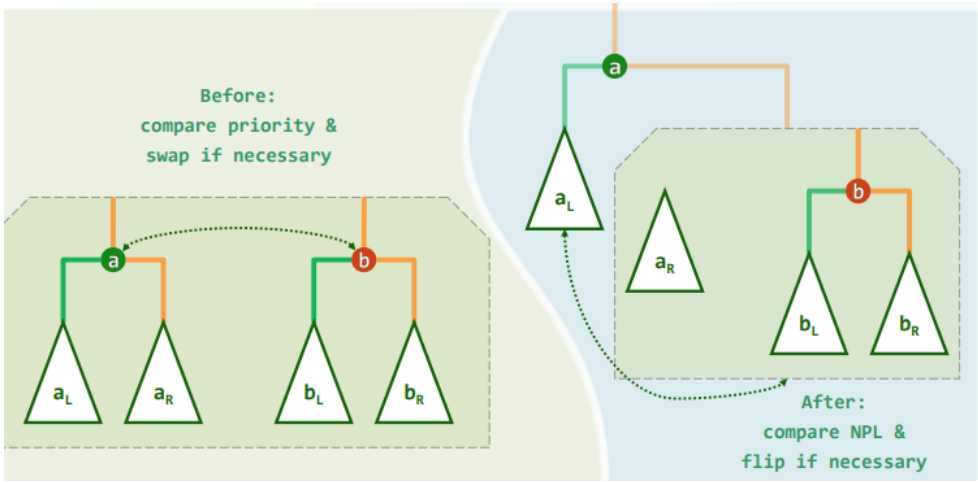
右侧链

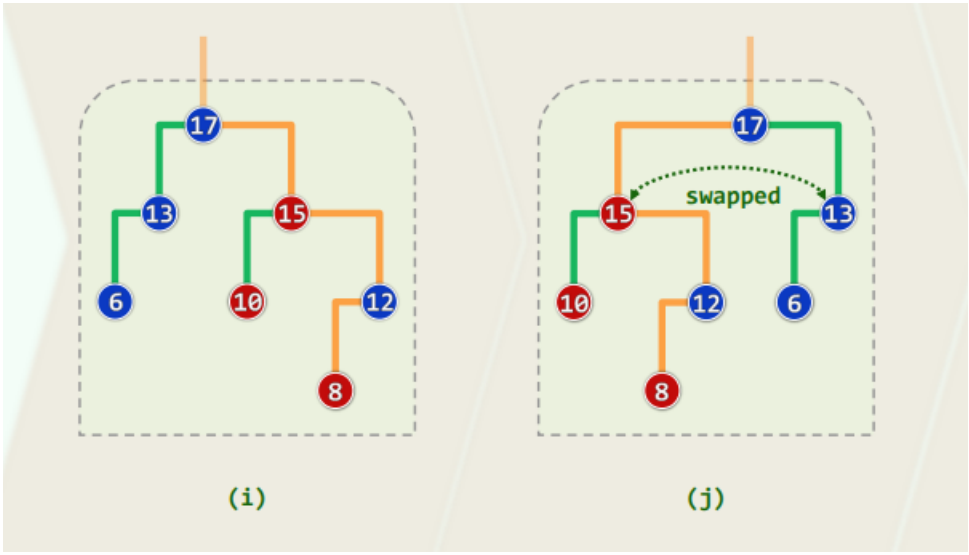
$rChain(x)$：从节点x出发，一直沿**右分支**前进
特别地，$rChain(r)$的终点，即全堆中**最浅**的外部节点
$npl(r) = |rChain(r)| = d$
存在一棵以r为根，高度为d的**满子树**

右侧链长为d的左式堆，**至少**包含
$2^d - 1$个内部节点
$2^{d+1} - 1$个节点
反之，包含n个节点的左式堆，右侧链长度$d \leq [log_2(n+1)] - 1 = O(logn)$



3. 合并算法






```
template <typename T> //基于二叉树，以左式堆形式实现的优先级队列
class PQ_LeftHeap : public PQ<T>, public BinTree<T> {
public:
    T getMax() { return _root->data; }
    void insert(T); T delMax(); //均基于统一的合并操作实现...
    PQ_LeftHeap( PQ_LeftHeap & A, PQ_LeftHeap & B ) {
        _root = merge(A._root, B._root); _size = A._size + B._size;
        A._root = B._root = NULL; A._size = B._size = 0;
    }
};

template <typename T>
BinNodePosi<T> merge(BinNodePosi<T> a, BinNodePosi<T> b) {
    if ( !a ) return b; if ( !b ) return a; //递归基
    if ( lt( a->data, b->data ) ) swap( a, b ); //确保a>=b
    ( a->rc = merge( a->rc, b ) )->parent = a; //将a的右子堆，与b合并
    if ( ! a->lc || a->lc->npl < a->rc->npl ) //若有必要
        swap( a->lc, a->rc ); //交换a的左、右子堆，以确保左子堆的npl不小
    a->npl = a->rc ? 1 + a->rc->npl : 1; //更新a的npl
    return a; //返回合并后的堆顶
}
```

```
//迭代实现
template <typename T>
BinNodePosi<T> merge( BinNodePosi<T> a, BinNodePosi<T> b ) {
    if ( !a ) return b; if ( !b ) return a; //退化情况
    if ( lt( a->data, b->data ) ) swap( a, b ); //确保a>=b
    for ( ; a->rc; a = a->rc ) //沿右侧链做二路归并，直至堆a->rc先于b变空
        if ( lt( a->rc->data, b->data ) ) {
            b->parent = a; swap( a->rc, b);
        } //接入b
    (a->rc = b)->parent = a; //直接接入b的残余部分（必然非空）
    for ( ; a; b = a, a = a->parent ) { //从a出发沿右侧链逐层回溯（b == a->rc）
        if ( !a->lc || a->lc->npl < a->rc->npl )
            swap( a->lc, a->rc ); //确保npl合法
        a->npl = a->rc ? a->rc->npl + 1 : 1; //更新npl
    }
    return b; //返回合并后的堆顶
}
```

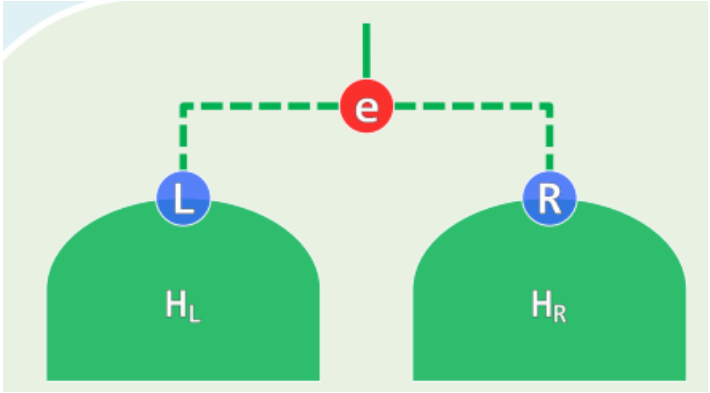
4. 插入与删除

- **插入**

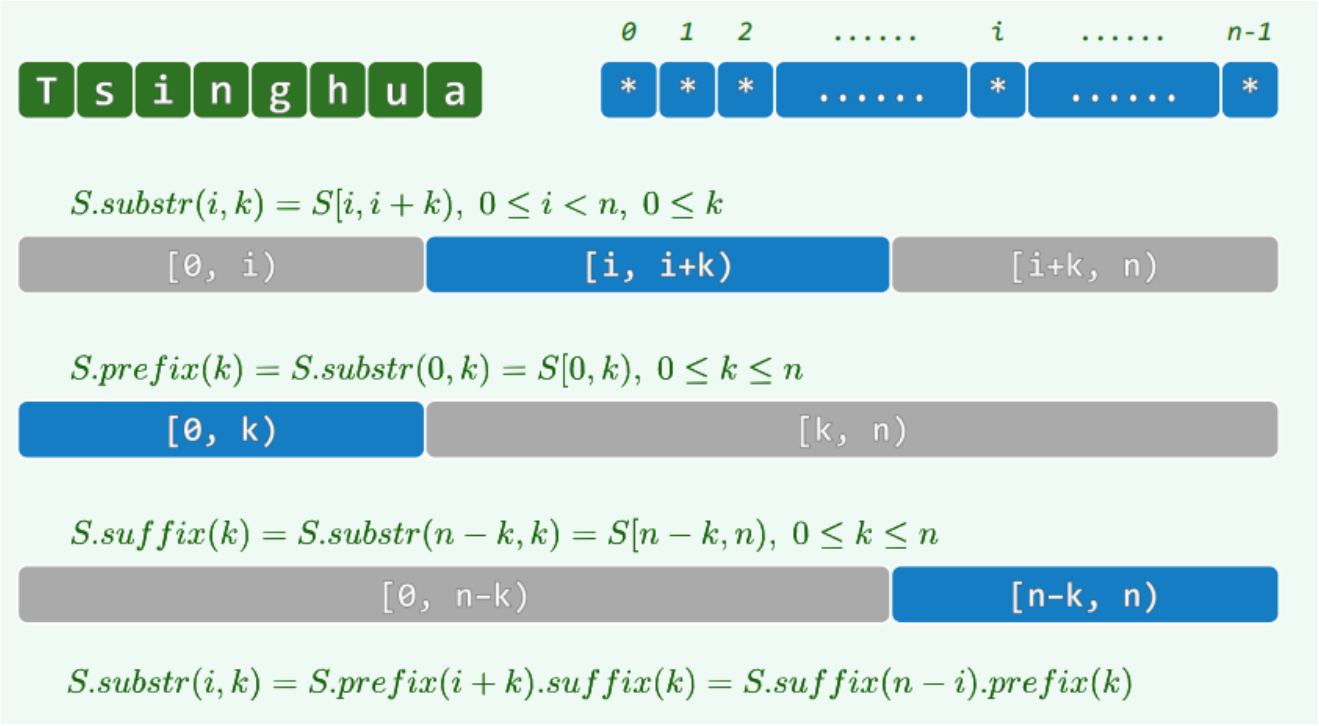
```
template <typename T>
void PQ_LeftHeap<T>::insert( T e ) { //O(logn)
    _root = merge( _root, new BinNode<T>( e, NULL ) );
    _size++;
}
```

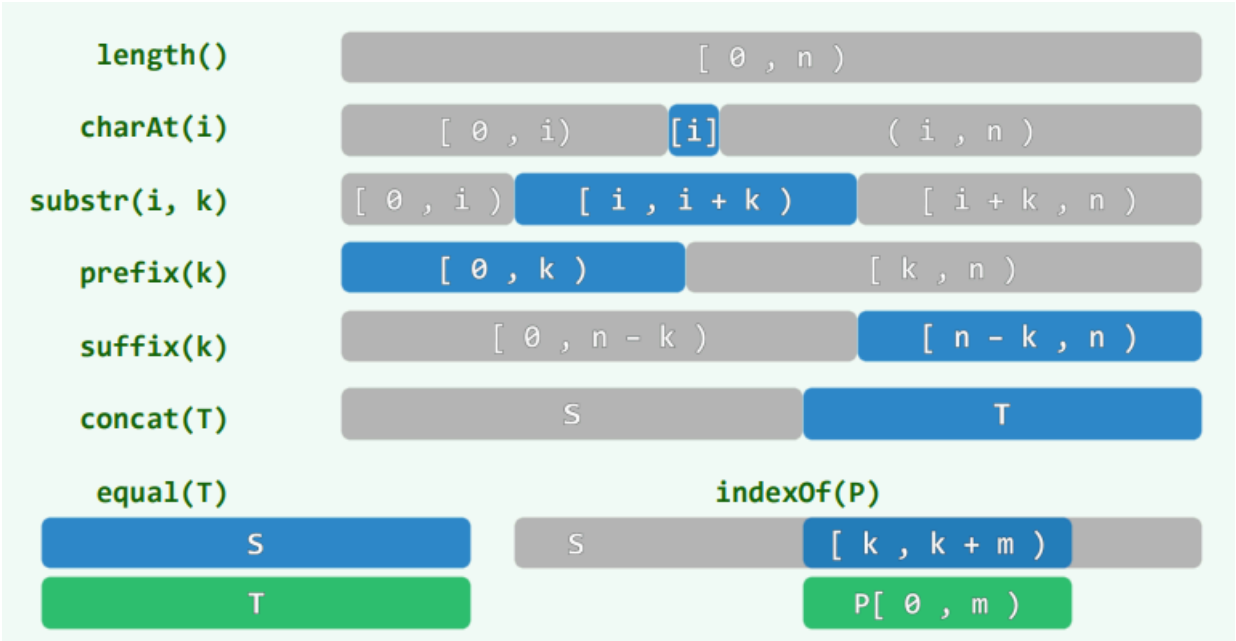
- **删除**

```
template <typename T>
T PQ_LeftHeap<T>::delMax() { //O(logn)
    BinNodePosi<T> lHeap = _root->lc; if (lHeap) lHeap->parent = NULL;
    BinNodePosi<T> rHeap = _root->rc; if (rHeap) rHeap->parent = NULL;
    T e = _root->data;
    delete _root; _size--;
    _root = merge( lHeap, rHeap );
    return e;
}
```



12 串





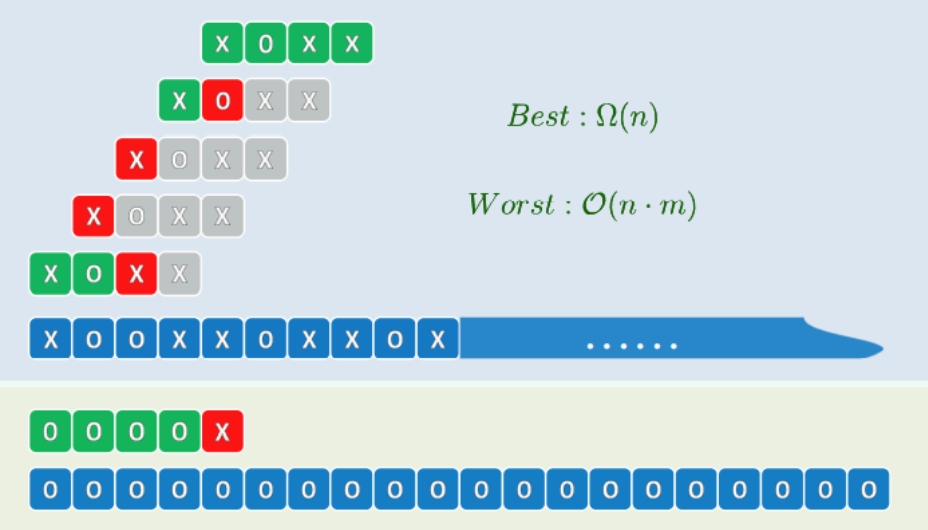
12.1 模式匹配

Text: 串，长度记为n
Pattern: 模式子串，长度记为m
记字符集为$\sum$，则字符集的个数$s = |\sum|$

蛮力算法:

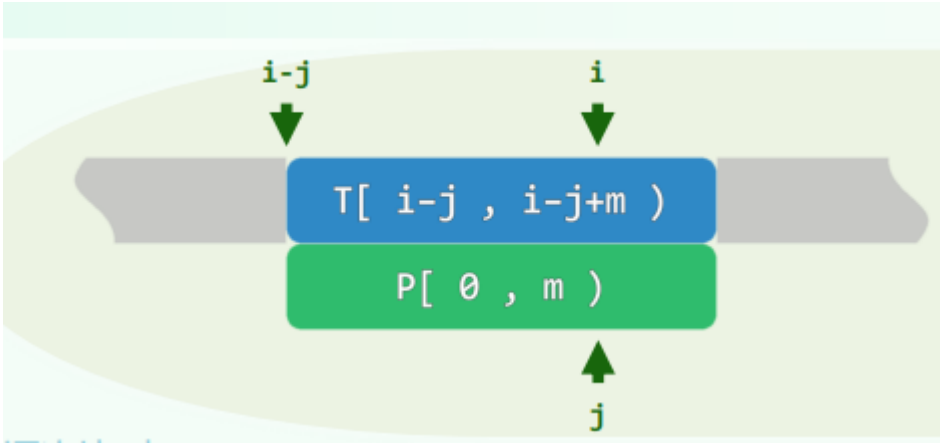
Best Case: $\Omega(n)$ （绝大概率情况下都是Best Case）

Worst Case: $O(n \cdot m)$（概率比较低）



Version 1:

```
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
        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
}
```



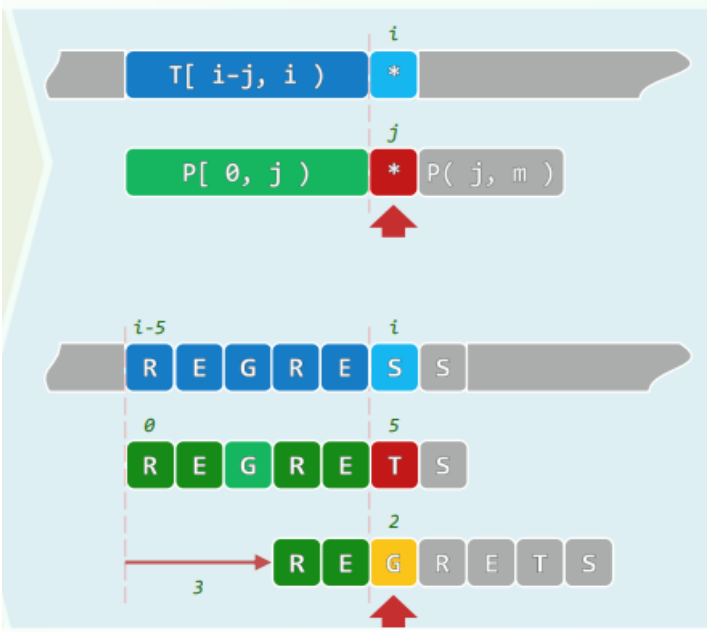
Version 2:

```
int match( char * P, char * T ) {
    size_t n = strlen(T), i = 0;
    size_t m = strlen(P), j;
    for ( i = 0; i < n - m + 1; i ++ ) { //T[i]与P[0]对齐后
        for ( j = 0; j < m; j ++ ) //逐次比对
            if ( T[i + j] != P[j] ) break; //失配，转下一对齐位置
        if ( m <= j ) break; //完全匹配（Python：可以写得更简洁、优美）
}
```

```
    return i; //最终的对齐位置：藉此足以判断匹配结果
}
```

1. KMP算法

> 在任意时刻，都有$T[i-j,i) == P[0,j)$
> 此时，我们已经掌握了$T[i-j,i)$的**所有信息**
> 因此，一旦失败，就应该已知哪些位置**值得/不必**对齐，则在下一轮比对中，$T[i-j',i)$可**直接接收**，而不必再做比对



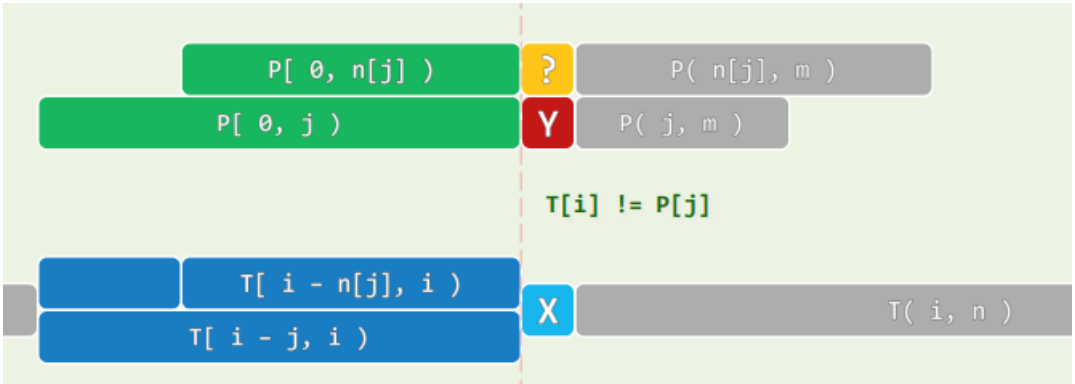
> 由此，i将**永远不必**回退
> 比对成功，则与j同步前进一个字符
> 否则，j更新为某一更小的**t**，比继续比对

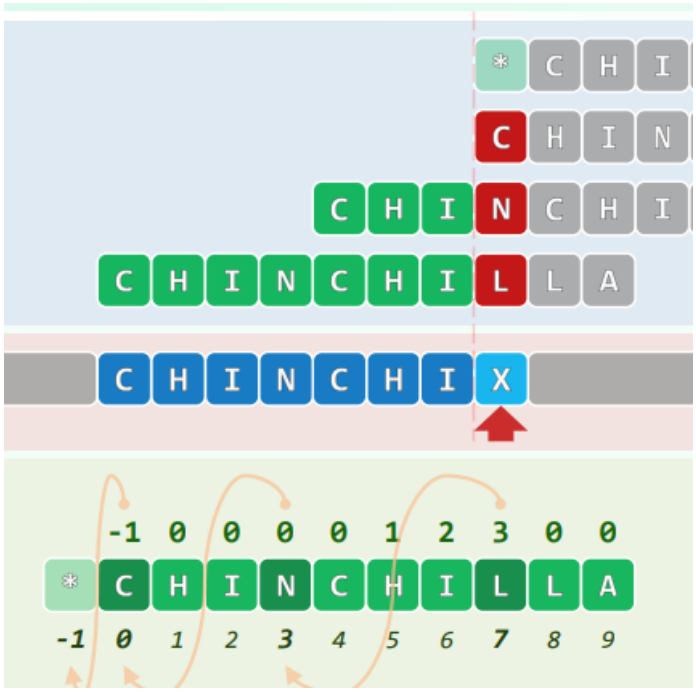
> 优化 = P可快速右移 + 避免重复比对

> 提问：为确定**t**，需花费多少时间和空间？更重要的，可否在**事先**就确定？

查询表

> **t：不仅可以事先确定，而且仅根据$P[0,j) = T[i-j,i)$即可确定**
> 视失败的位置j，无非**m种**情况
> 构造查询表$next[0,m)$，做好预案
> 一旦在$P[j]$处失配，只需将j替换为$next[j]$，继续与$T[i]$比对

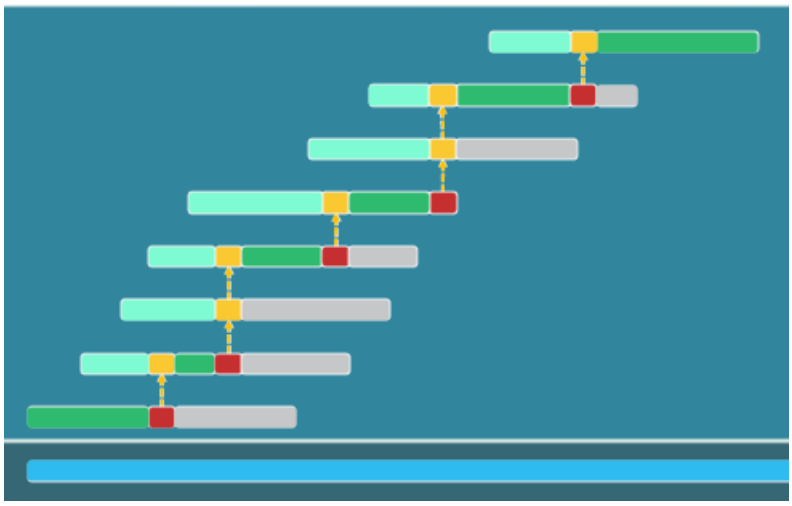




串首补齐一个**通配符**（`*`）

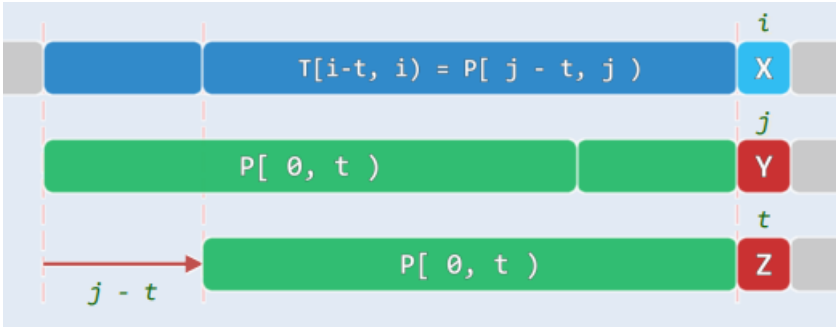
```
int match( char * P, char * T ) {
    int * next = buildNext(P);
    int n = (int) strlen(T), i = 0;
    int m = (int) strlen(P), j = 0;
    while ( j < m && i < n )
        if ( 0 > j || T[i] == P[j] ) {
            i++; j++;
        }
        else j = next[j];
    delete [] next;
    return i -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

快速右移，绝不回退：

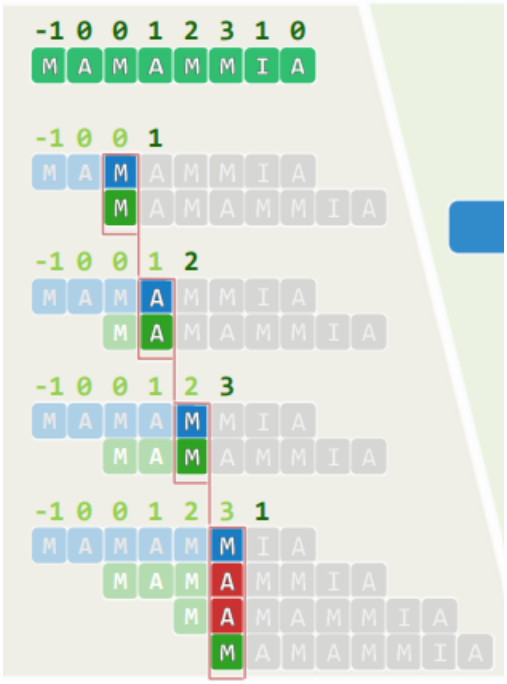


next表

> $\forall j \geq 1, N(P,j) = \{0 \leq t < j | P[0,t) = P[j-t,j)\}$ P在t处**自匹配**
> $next[j] = max\{N(P,j)\}$ 长度最大，位移最小，不致日后**回溯**



传递链：动态规划




```
next[j + 1] = next[j] + 1 iff P[j] = P[next[j]]
```

```
int * buildNext( char * P ) {
    size_t m = strlen(P), j = 0;
    int *N = new int[m];
    int t = N[0] = -1;
    while ( j < m - 1 )
        ( 0 > t || P[j] == P[t] ) ? N[ ++j ] = ++t : t = N[t];
    return N;
}
```

分摊分析



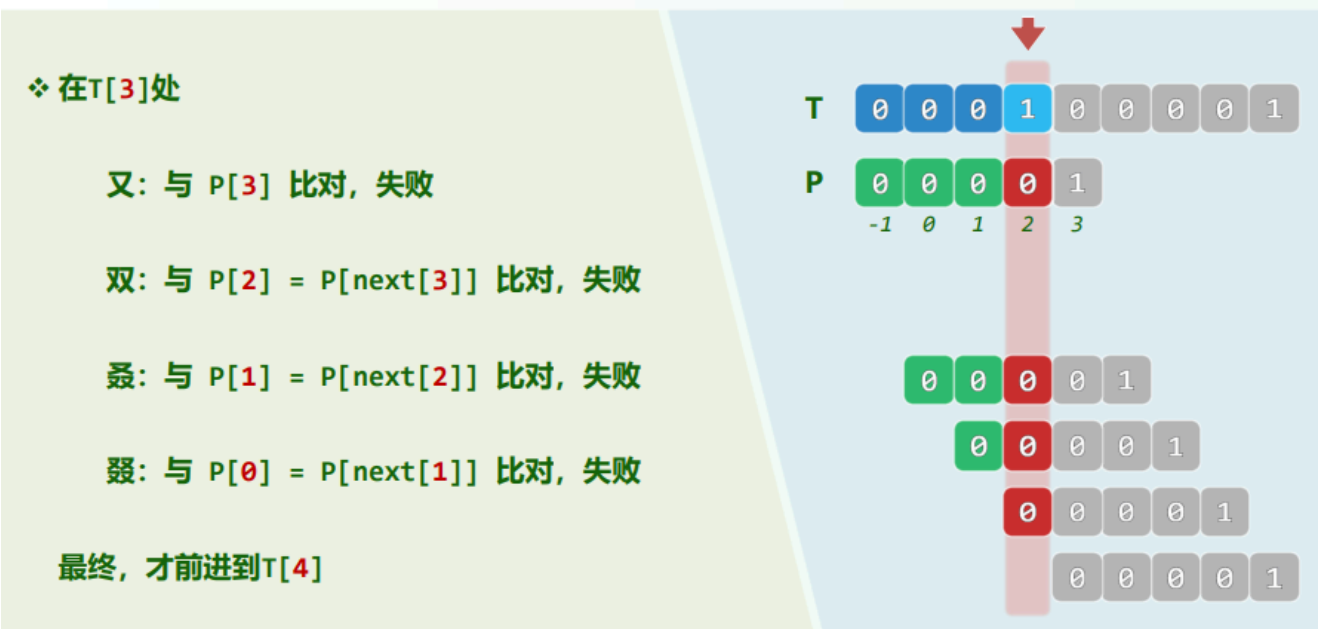
> 尽管如此，“黄色块”不会一直出现，其总的分摊复杂度并不高

while循环前，令计步器`k = 2 * i - j`（初始有k = 0），代表迭代步数的上界
则算法结束时：$k = 2 * i - j \leq 2(n-1) - (-1) = 2n - 1$

因此，总体时间复杂度为$O(n+m)$

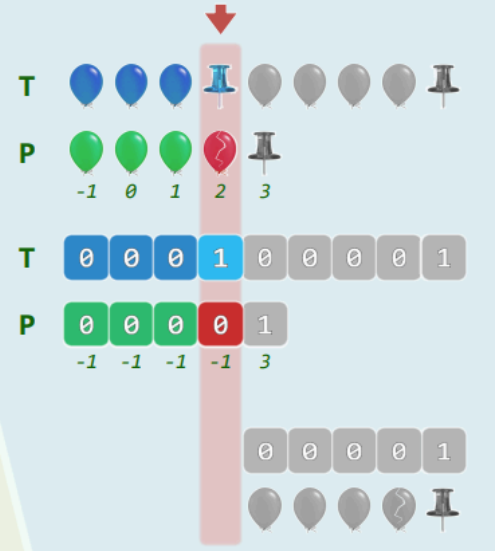
再分析

> 反例

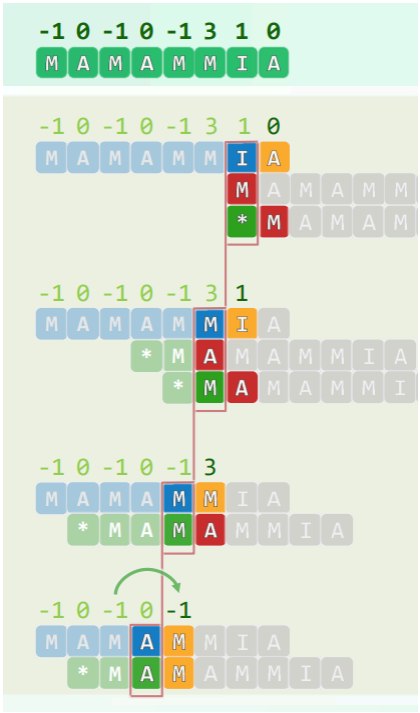


改进

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



渐近复杂度仍然是线性的。



小结：

充分利用**以往的**比对所提供的信息：模式串快速右移，文本串无需回退

经验 ~ 以往**成功**的比对：$T[i-j,i)$**是**什么

教训 ~ 以往**失败**的比对：$T[i]$**不是**什么

特别适用于**顺序**存储介质

单词匹配概率越**大**（字符集越**小**），优势越**明显**（比如二进制串）

否则，可能与蛮力算法的性能相差无几

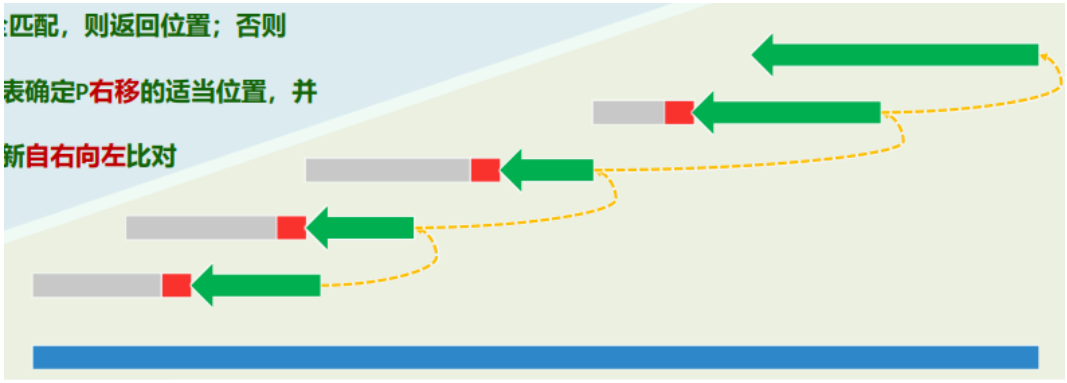
2. BM算法

BC策略：以终为始

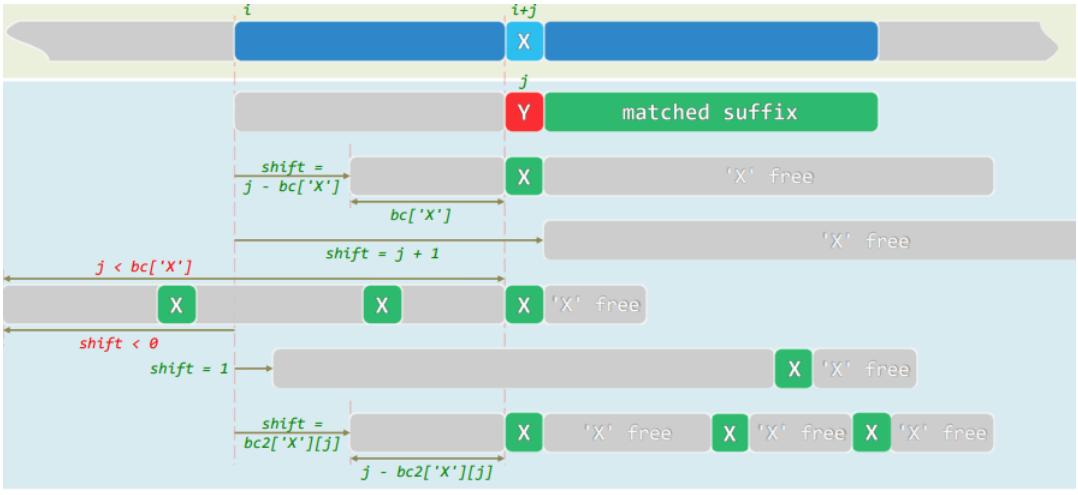


自左向右移动P，自右向左比对P和T

> Boyer + Moore, 1977: A Fast String Searching Algorithm
> 预处理：根据模式串P，预先构造`gs[]`表和`bc[]`表
> 迭代：**自右向左**依次比对字符，找到极大的匹配**后缀**
> 若完全匹配，则返回位置；
> 否则，查表确定P**右移**的适当位置，并重新**自右向左**比对

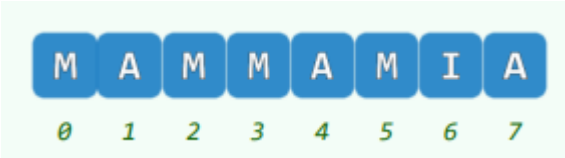


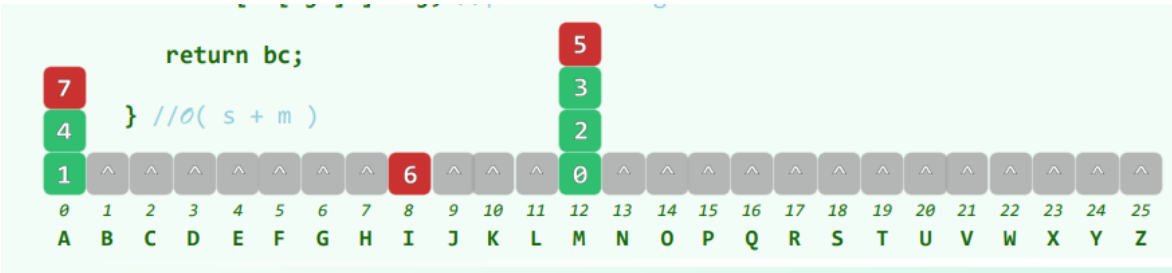
坏字符



构造`bc[]`：画家算法

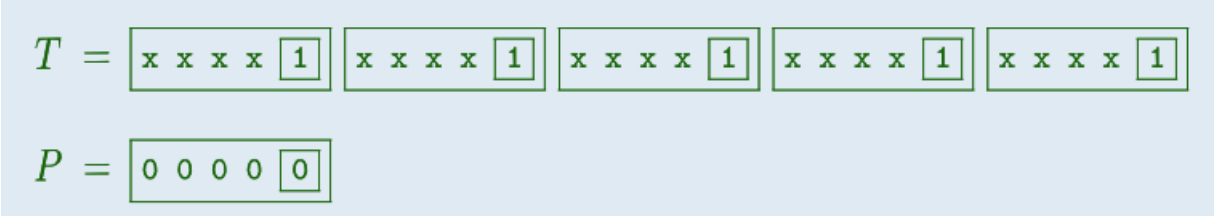
```
int* buildBC(char* P) {
    int* bc = new int[256];
    for (size_t j = 0; j < 256; ++j) bc[j] = -1;
    for (size_t m = strlen(P), j = 0; j < m; ++j)
        bc[P[j]] = j; // painter's algorithm
    return bc;
} // O(s + m)
```



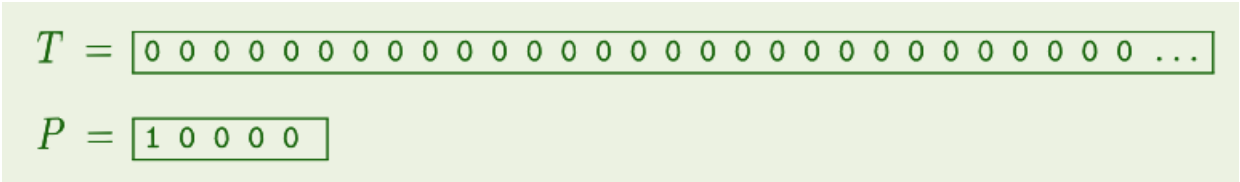


性能分析：

- 最好情况：$O(\frac{n}{m})$
  一般地，只要P不含$T[i+j]$，即可直接移动**m个**字符，仅需**单次**比较，即可排除**m个**对齐位置
  单词匹配概率**越小**，性能优势**越明显**（大字母表：ASCII, UniCode）
  P**越长**，这类移动的效果**越明显**

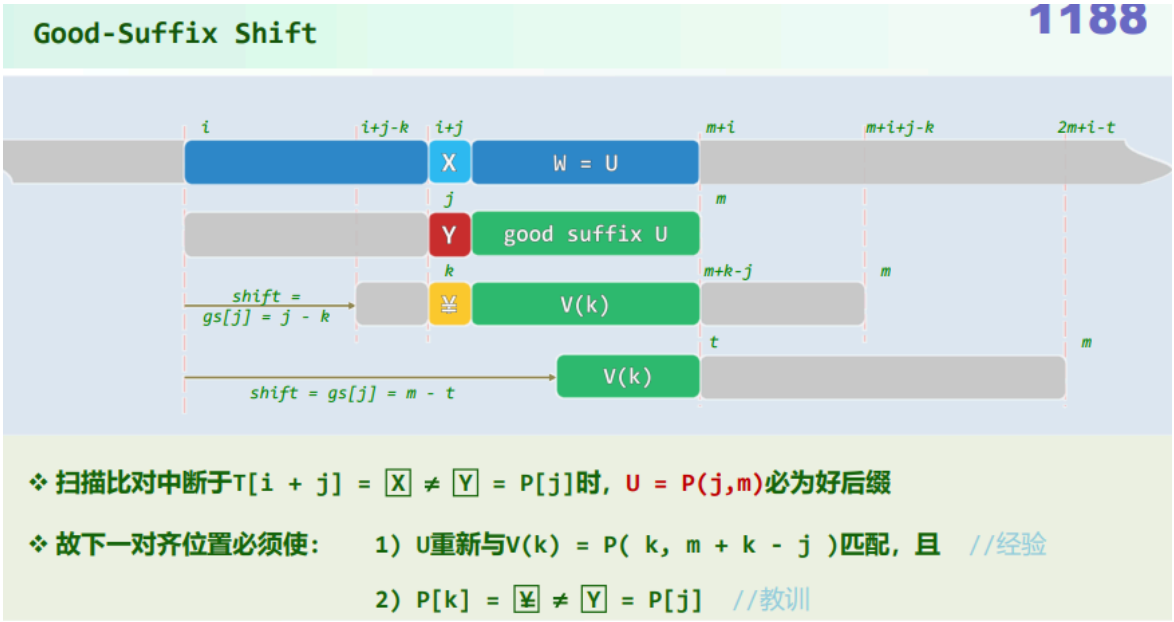


- 最差情况：$O(nm)$
  退化为蛮力算法：

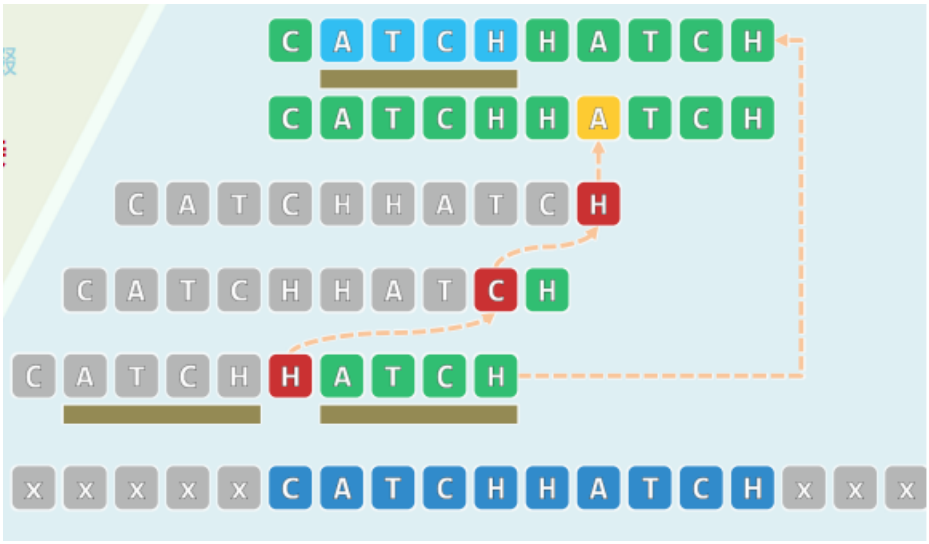


  每轮迭代，都要扫过**整个**P之后，方能确定右移**一个**字符
  此时，须经**m次**比较，方能排除**单个**对齐位置
  单词匹配概率**越大**的场合，性能**越接近于**蛮力算法（小字母表Bitmap + DNA）
  反思：借助以上`bc[]`表，仅仅利用了**失配**比对提供的信息（**教训**）
  类比：可否仿照KMP，同时利用起**匹配**比对提供的信息（**经验**）？

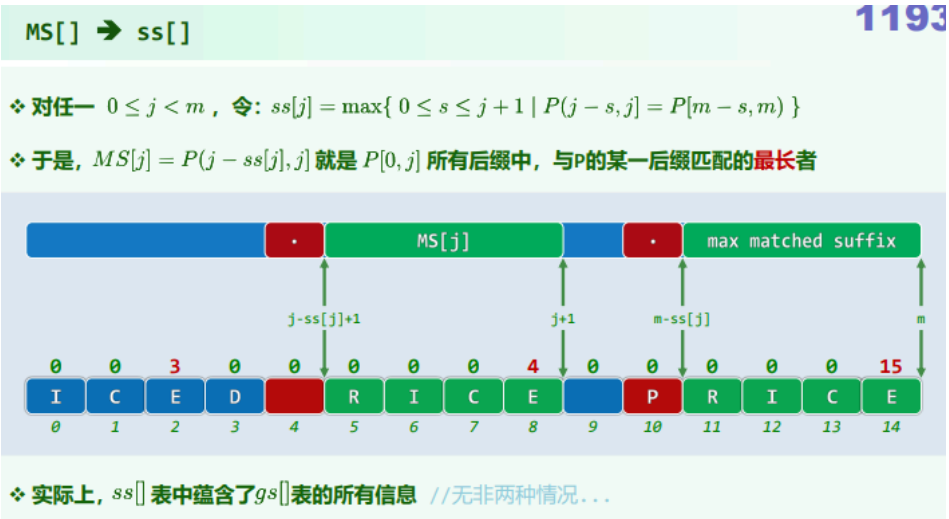
GS策略：好后缀

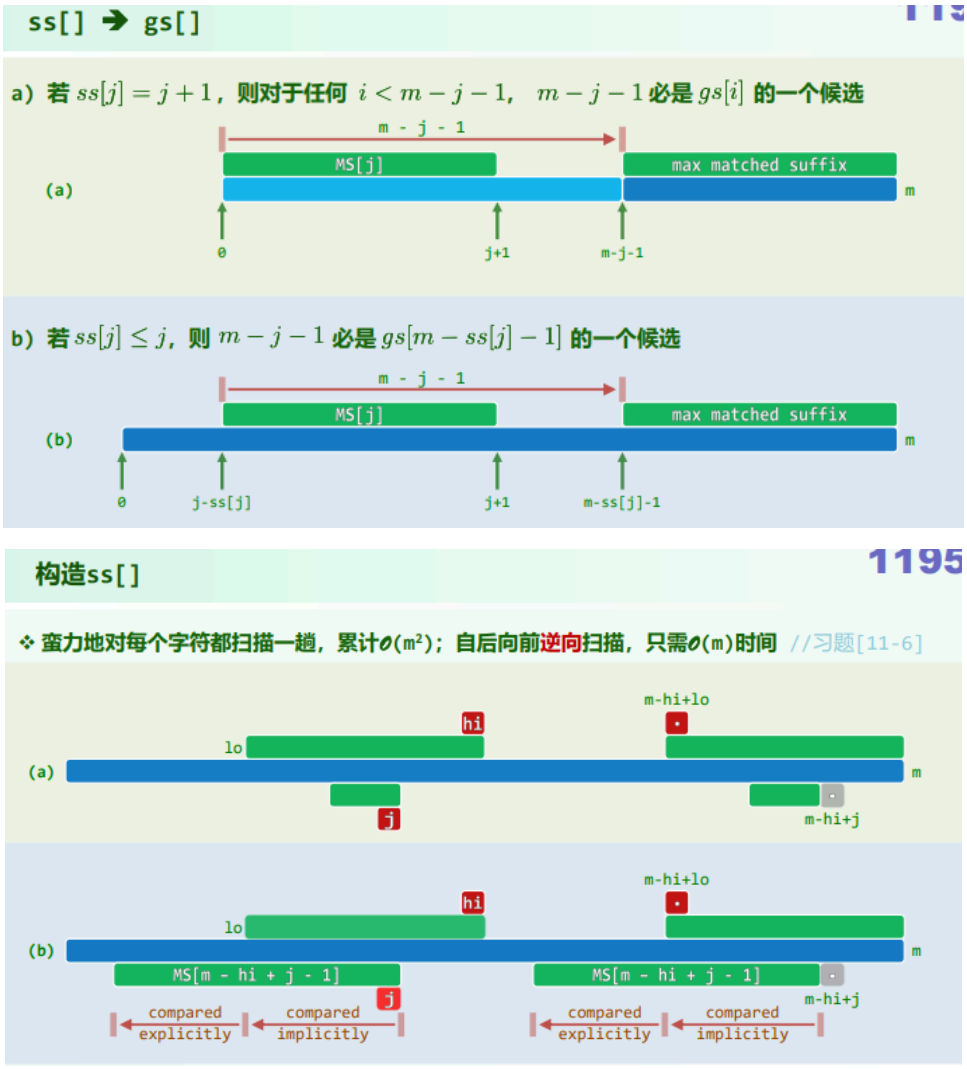


[EG]:

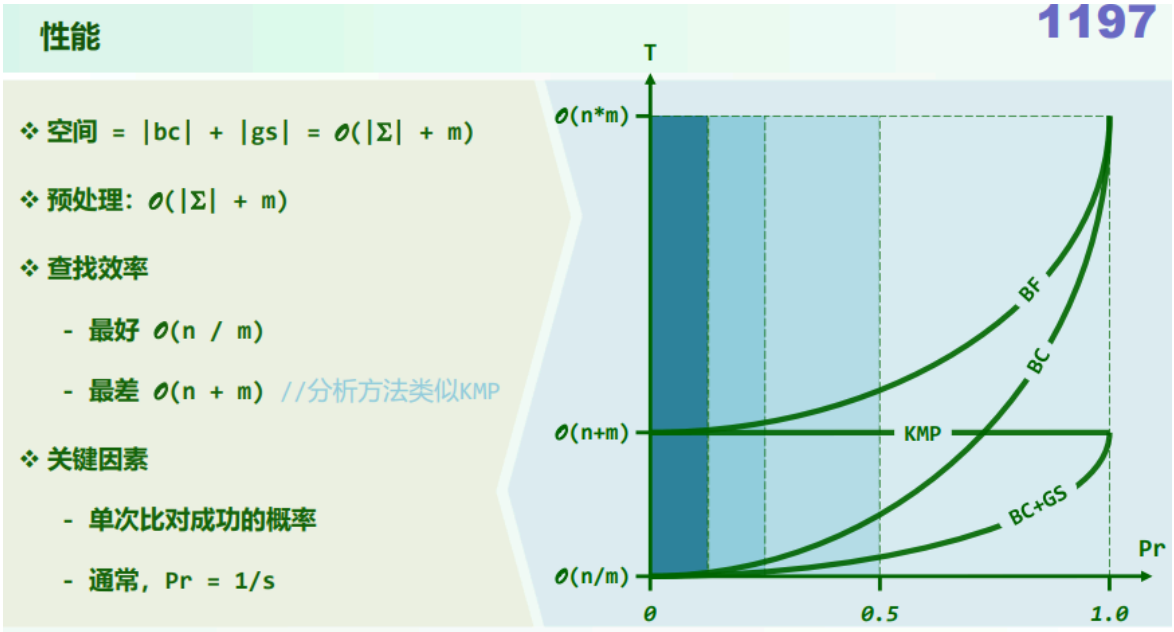


构造gs[]表





BC+GS综合性能



3. Karp-Rabin算法：串即是数

凡物皆数：Godel Numbering

逻辑系统的符号、表达式、公式、命题、定理、公理等，均可表示为**自然数**

每个有限维的自然数**向量**（包括字符串），都唯一对应于某个**自然数**

素数序列：$p(k)=$第k个素数$=2,3,5,7,11,13,17,19...$

$$\langle a_1, a_2, \ldots, a_n \rangle \rightleftharpoons p(1)^{1+a_1} \times p(2)^{1+a_2} \times \cdots \times p(n)^{1+a_n}$$

$$\langle \boxed{3} \quad \boxed{1} \quad \boxed{4} \quad \boxed{1} \quad \boxed{5} \quad \boxed{9} \quad \boxed{2} \quad \boxed{6} \rangle$$

$$2^{\boxed{4}} \times 3^{\boxed{2}} \times 5^{\boxed{5}} \times 7^{\boxed{2}} \times 11^{\boxed{6}} \times 13^{\boxed{10}} \times 17^{\boxed{3}} \times 19^{\boxed{7}}$$

$godel = 2^{1+7} \times 3^{1+15} \times 5^{1+4} \times 7^{1+5} \times 11^{1+12} = 139869560310664817087943919200000$

若果真如RAM模型所假设的**字长无限**，则只需**一个**寄存器即可...

凡物皆数：Cantor Numbering

$cantor_2(i,j) = [(i+j)^2+3\cdot i+j]/2$

$cantor_2(2,3) = [(2+3)^2+3\cdot 2+3]/2 = 17$

$cantor_2(3,2) = [(3+2)^2+3\cdot 3+2]/2 = 18$





$cantor_{n+1}(a_1, \ldots; a_{n-1}, a_n, a_{n+1}) =$

$cantor_n(a_1, \ldots; a_{n-1}, cantor_2(a_n, a_{n+1}))$

❖ 长度有限的字符串，都可视作d=1+|Σ|进制的自然数

"decade" = $453145_{(10)}$

//d = 1 + ('i'-'a') = 10

❖ 长度无限的字符串，都可视作[0,1)内的d进制小数

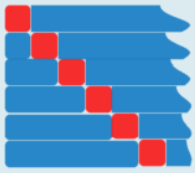
"bgahbhahbhdei..." = 0.2718281828459...

"fade" = 0.6145 = 0.6144999... = "faddiii..."

❖ Cantor's Diagonal: $\aleph_0 < \aleph_1$







十进制串，可直接视作自然数：指纹（fingerprint），等效于多项式法

$P = 82818 \qquad T = 27182818284590452353602874713527$

一般地，随意对字符编号$0,1,2,\ldots,d-1$ `//设d = |Σ|`

于是，每个字符串都对应于一个 d 进制自然数 //尽管不是单射

$CAT = 2019_{(26)} = 1371_{(10)}$ `//Σ = { A, B, C, ..., Z }`

$ABBA = 0110_{(26)} = 702_{(10)}$

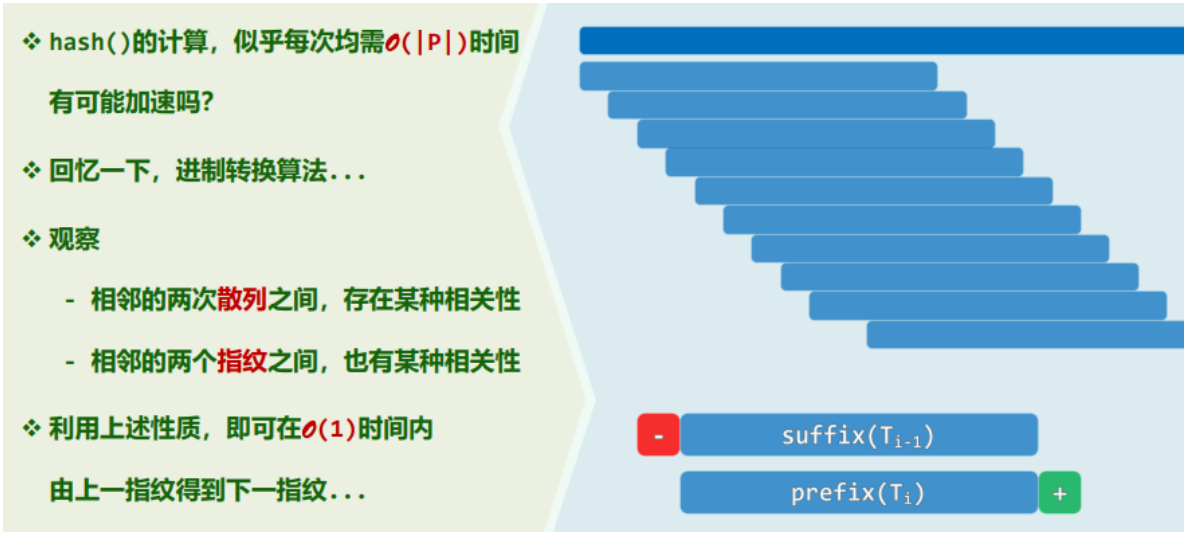
P在T中出现**仅当**T中某一子串与P**相等**

散列

基本构思：通过对比经压缩之后的**指纹**，确定匹配位置
关键技巧：通过**散列**，将指纹压缩至存储器支持的范



注意：通过`hash()`筛选后，还需经过**严格的**比对，方可最终确定是否匹配
适当选取散列函数，可以极大**降低**冲突的概率



字宽

$power_a(n) = a^n$

❖ 平凡实现:

```
pow = 1; //O(1)
while ( 0 < n ) //O(n)
  { pow *= a; n--; } //O(1+1)
```

❖ $T(n) = 1 + 2n = \mathcal{O}(n)$

线性? 伪线性!

❖ 所谓输入规模，准确地应定义为

用以描述输入所需的空间规模

❖ 对于此类数值计算

即是n的二进制位数，亦即n的打印宽度

$r = \lceil \log_2(n+1) \rceil = \mathcal{O}(\log n)$

$T(r) = \mathcal{O}(2^r)$ //指数复杂度!

a^{98765}

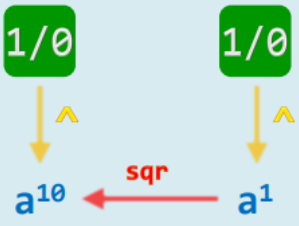
$$a^{9\cdot10^4 + 8\cdot10^3 + 7\cdot10^2 + 6\cdot10^1 + 5\cdot10^0}$$



$$\left(a^{10^4}\right)^9 \cdot \left(a^{10^3}\right)^8 \cdot \left(a^{10^2}\right)^7 \cdot \left(a^{10^1}\right)^6 \cdot \left(a^{10^0}\right)^5$$

从 O(n) 到 O(r = logn)

```
int power( int a, int n ) {
   int pow = 1, p = a; //O(1 + 1)
   while (0 < n) { //O(logn)
      if (n & 1) //O(1)
         pow *= p; //O(1)
      n >>= 1; //O(1)
      p *= p; //O(1)
   }
   return pow; //O(1)
}
```



❖ 输入规模 $= r = \lceil \log_2(n+1) \rceil$

❖ 运行时间 $= T(r) = 1 + 1 + 4r + 1 = \mathcal{O}(r)$

❖ 如此，“实现”了从指数到线性的改进

> $O(logn)$只对n比较小的时候成立

再分析

❖ 根据以上算法，“可以”在 $\mathcal{O}(\log n)$ 时间内计算出 $power(n) = a^n$

❖ 然而，a^n 的二进制展开宽度为 $\Theta(n)$

这意味着，即便是直接打印 a^n，也至少需要 $\Omega(n)$ 时间……哪里错了？

❖ 类似的悖论对 $fib(n)$ 也存在……

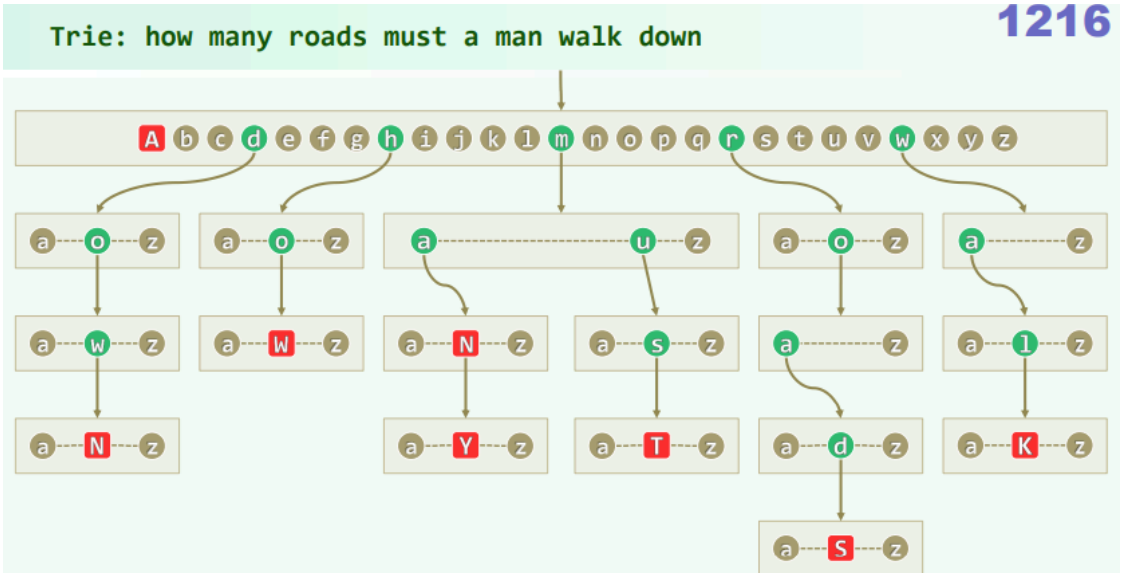
❖ 令：$\mathcal{A} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} fib(0) & fib(1) \\ fib(1) & fib(2) \end{bmatrix}$，则：$\mathcal{A}^n = \begin{bmatrix} fib(n-1) & fib(n) \\ fib(n) & fib(n+1) \end{bmatrix}$

❖ 因此参照上述power()算法，似乎也“可以”在 $\mathcal{O}(\log n)$ 时间内计算出 $fib(n)$

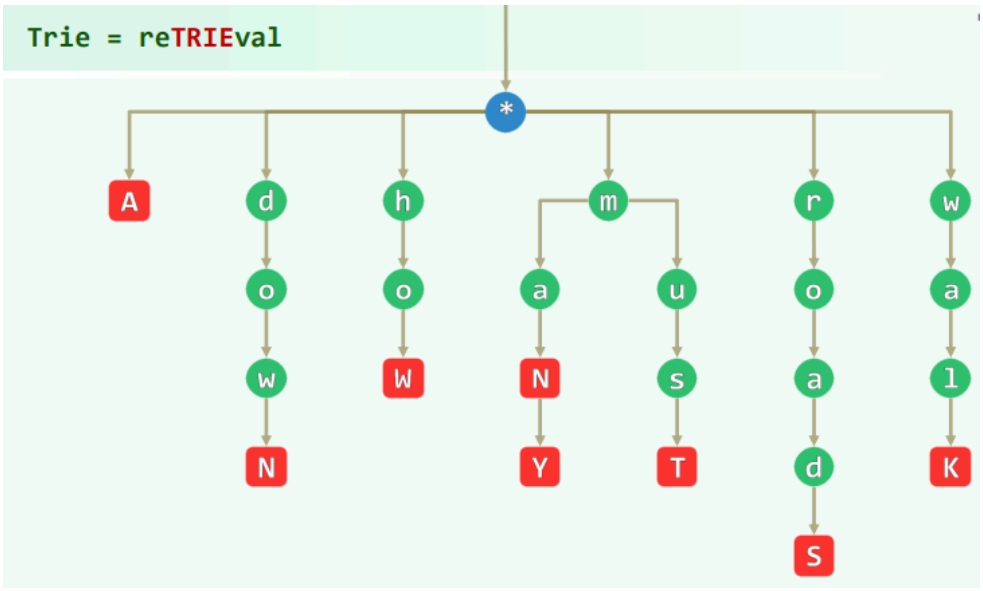
❖ RAM模型：　常数代价准则 (uniform cost criterion)

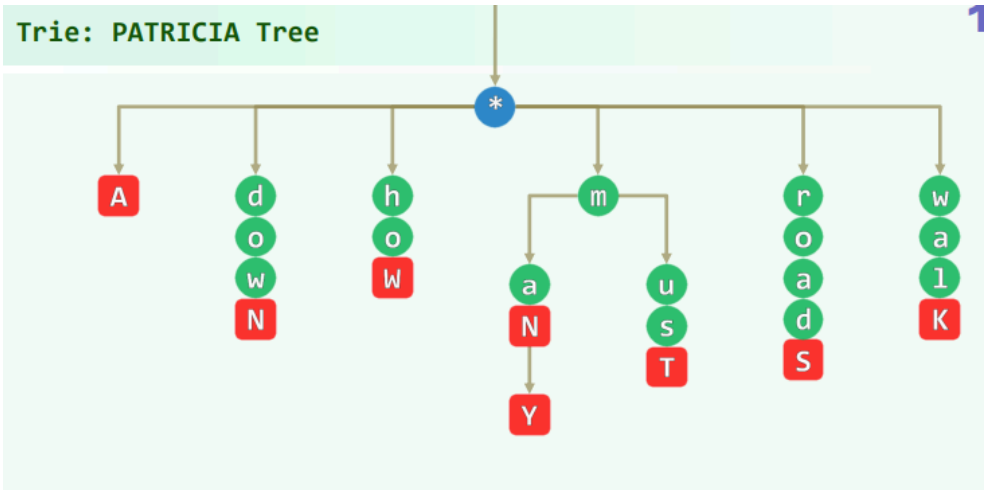
对数代价准则 (logarithmic cost criterion)

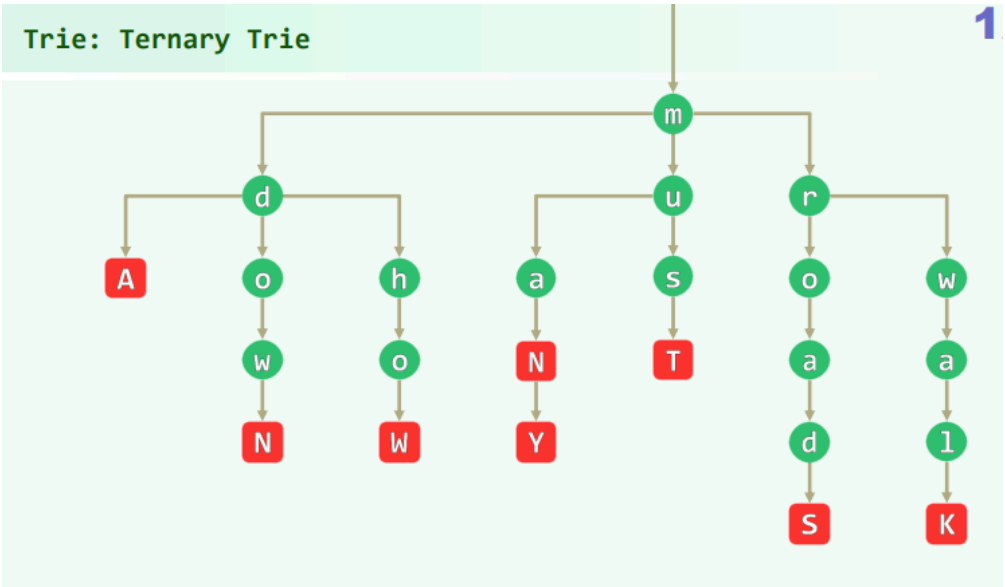
12.2 键树 Trie



> 其中，红色的节点表示单词的结束节点（不一定是叶子结点）


Trie = reTRIEval
*
A
d
o
w
N
h
o
W
m
a
N
Y
u
s
T
r
o
a
d
S
w
a
l
K


Trie: PATRICIA Tree
*
A
d
o
w
N
h
o
W
m
a
N
Y
u
s
T
r
o
a
d
S
w
a
l
K


Trie: Ternary Trie
m
d
A
o
w
N
h
o
W
u
a
N
Y
s
T
r
o
a
d
S
w
a
l
K

13 排序

13.1 快速排序

> C. A. R. Hoare：分而治之
> **pivot**：左侧/右侧的元素，均不比它更大/更小
> 以轴点为界，自然**划分**：$max([0, mi)) \leq min((mi, hi))$
> 前缀、后缀各自递归排序之后，原序列自然有序：$sorted(S) = sorted(S_L) + sorted(S_R)$
>
> mergesort难点在于**合**，quicksort在于**分**
> 通过**培养**轴点来实现上述划分

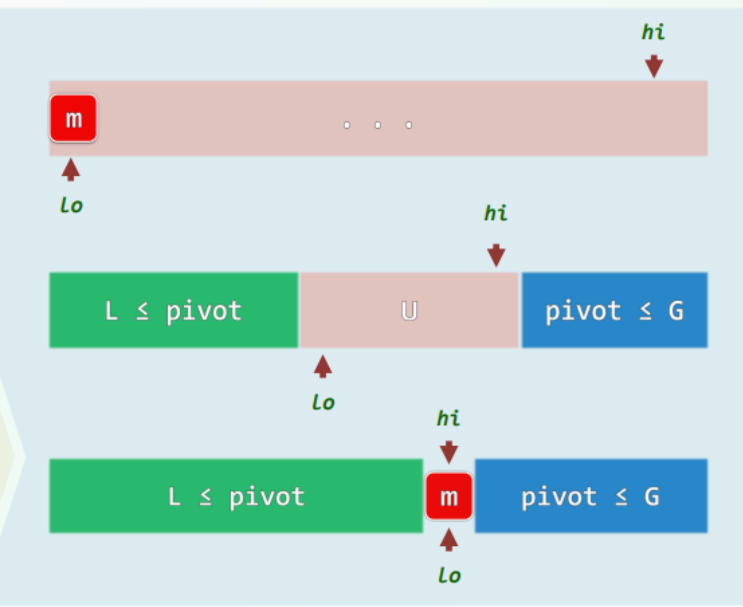


> 快速排序就是将所有元素**逐个转换**为轴点的过程

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

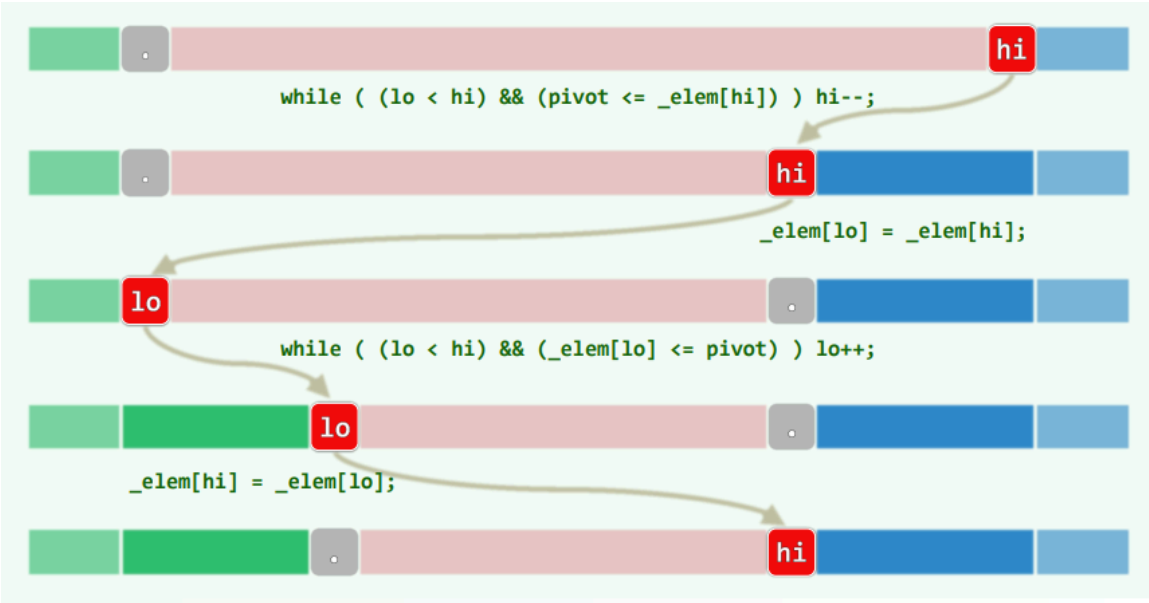
快速划分：LUG版

任选一个**候选者**（如`[0]`），**交替**地**向内**移动`lo`和`hi`
逐个检查当前元素：若更小/大，则转移归入L/G
当$lo = hi$时，只需将候选者**嵌入**与L、G之间，即成轴点
各元素最多移动一次（候选者两次）：累计时间$O(n)$，$O(1)$辅助空间

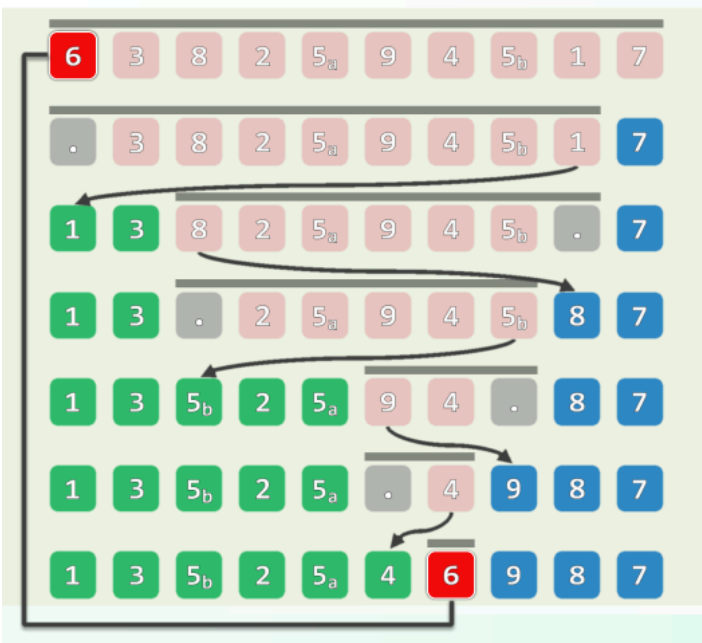



```
template <typename T>
Rank Vector<T>::partition( Rank lo, Rank hi ) { //[lo, hi)
    swap(_elem[lo], _elem[lo + rand() % (hi - lo)]); //随机交换
    T pivot = _elem[lo]; //经以上交换，等效于随机选取候选轴点
    while (lo < hi) { //从两端交替地向中间扫描，彼此靠拢
        do {hi--;}
            while (lo < hi && pivot <= _elem[hi]); //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;
}
```

不变性 + 单调性：$L \le pivot \le G$，`[lo]`和`[hi]`交替空闲

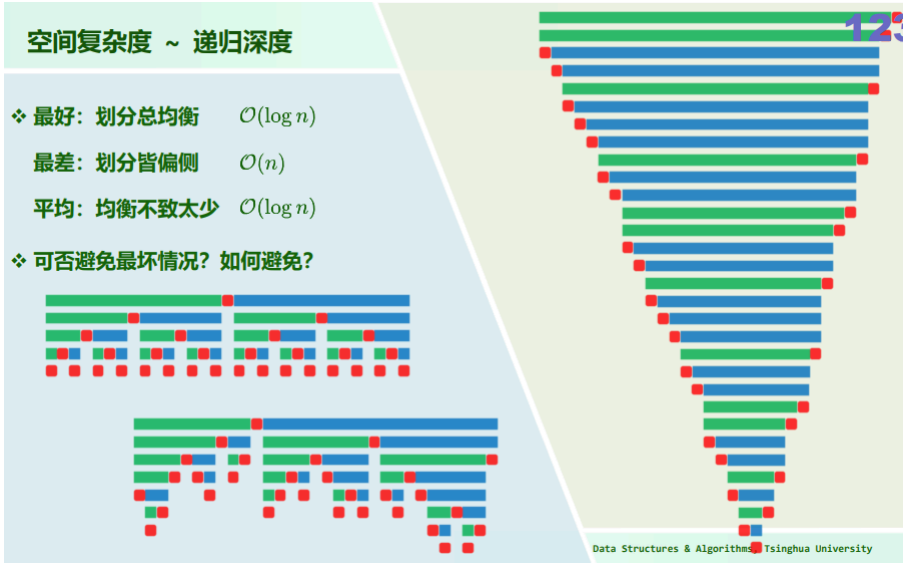


复杂度分析：
线性时间：尽管`lo`，`hi`交替移动，累计移动距离不过$O(n)$
in-pace：只需$O(1)$附加空间
unstable：`lo/hi`的移动方向相反，左/右侧的大/小重读元素可能前/后**颠倒**



2. 迭代、贪心与随机

空间复杂度 ~ 递归深度
最好：划分总是均衡 $O(logn)$
最差：划分总是偏侧 $O(n)$
平均：均衡不致太少 $O(logn)$



非递归 + 贪心

保证递归深度可能超过$O(logn)$，但辅助栈空间一定不超过$O(logn)$

```
#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); //等效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)
}
```

时间性能 + 随机

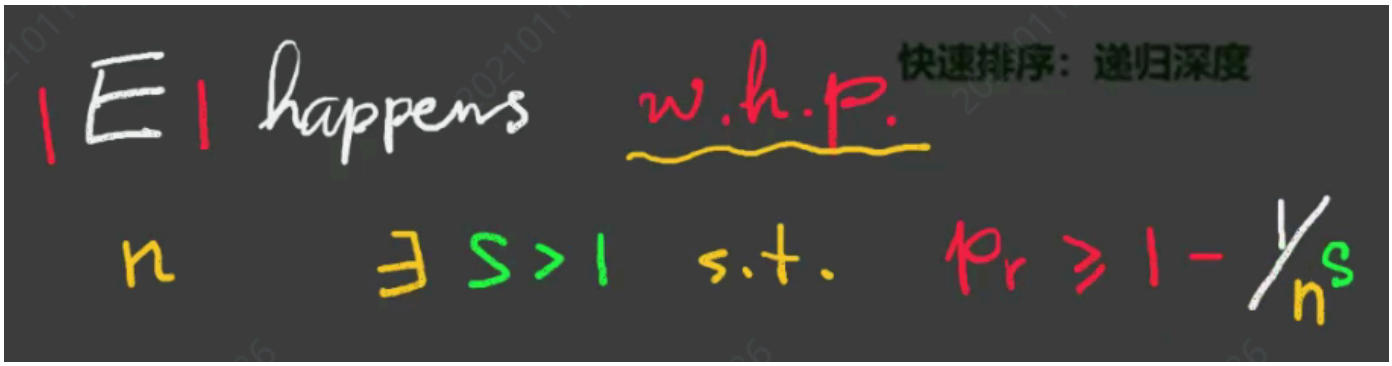
- 最好情况：每次划分都（接近）**平均**，轴点总是（接近）**中央**——到达下界
  $T(n) = 2 \cdot T(\frac{n-1}{2}) + O(n) = O(nlogn)$
- 最坏情况：每次划分都**极不均衡**（比如，轴点总是最小/大元素）——与起泡排序相当
  $T(n) = T(n-1) + T(0) + O(n) = O(n^2)$
- 采用**随机选取**（Randomization），**三者取中**（Sampling）之类的策略
  只能**降低**最坏情况的概率，而无法**杜绝**——既然如此，为何还称作**快速**排序?

3. 递归深度

> 最坏情况（$\Omega(n)$递归深度）概率极低
> 平均情况（$O(logn)$递归深度）概率极高
>
> 
>

除非**过于**侧片的pivot，都会**有效**地缩短递归深度
准居中：pivot落在宽度为$\lambda \cdot n$的居中区间（λ也是这种情况出现的概率）
每一递归路径上，**至多**出现$log_{\frac{2}{1+\lambda}} n$个**准居中**的pivots

期望深度
每递归一层，都有$\lambda(1-\lambda)$的概率**准居中**（**准偏侧**）
深入$\frac{1}{\lambda} \cdot log_{\frac{2}{1+\lambda}} n$层后，即可**期望**出现$log_{\frac{2}{1+\lambda}} n$次**准居中**，且有**极高的**概率出现
相反情况的概率$< (1-\lambda)^{(\frac{1}{\lambda}-1) \cdot log_{\frac{2}{1+\lambda}} n} = n^{(\frac{1}{\lambda-1}) \cdot log_{\frac{2}{1+\lambda}} (1-\lambda)}$，且随着$\lambda$增加而**下降**
至少有$1 - n^{2 \cdot log_{\frac{3}{2}} (\frac{2}{3})} = 1 - n^{-2}$的概率，使得递归深度不超过$\frac{1}{\lambda} \cdot log_{\frac{2}{1+\lambda}} n = 3 \cdot log_{\frac{3}{2}} n \approx 5.129 logn$



4. 比较次数

move的次数一定不超过compare的次数

期望的比较次数为$T(n)$：$T(1), T(2) = 1, \dots$
可以证明：$T(n) = O(nlogn)$



$$T(n) = (n-1) + \frac{1}{n} \cdot \sum_{k=0}^{n-1} [T(k) + T(n-k-1)] = (n-1) + \frac{2}{n} \cdot \sum_{k=0}^{n-1} T(k)$$

$$n \cdot T(n) = n \cdot (n-1) + 2 \times \sum_{k=0}^{n-1} T(k)$$
$$(n-1) \cdot T(n-1) = (n-1) \cdot (n-2) + 2 \times \sum_{k=0}^{n-2} T(k)$$

$$n \cdot T(n) - (n-1) \cdot T(n-1) = 2 \cdot (n-1) + 2 \times T(n-1)$$
$$n \cdot T(n) - (n+1) \cdot T(n-1) = 2 \cdot (n-1)$$



$$\frac{T(n)}{n+1} - \frac{T(n-1)}{n} = \frac{4}{n+1} - \frac{2}{n}$$

$$\frac{T(n)}{n+1} = \frac{T(n)}{n+1} - \frac{T(1)}{2} = 4 \cdot \sum_{k=2}^{n} \frac{1}{k+1} - 2 \cdot \sum_{k=2}^{n} \frac{1}{k} = 2 \cdot \sum_{k=1}^{n+1} \frac{1}{k} + \frac{2}{n+1} - 4 \approx 2 \cdot \ln n$$

$$T(n) \approx 2 \cdot n \cdot \ln n = (2 \cdot \ln 2) \cdot n \log n \approx 1.386 \cdot n \log n$$

后向分析：

设经排序后得到的**输出**序列为：$a_0, a_1, a_2, \dots a_i, \dots, a_j, \dots, a_{n-1}$
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**次数为$T(n) = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} Pr(i,j)$，亦即每一对$< a_i, a_j >$在排序过程中接收比较的**概率**综合

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$Pr(i,j)$无关

❖ 实际上，$\langle a_i, a_j \rangle$ 接受比较，当且仅当

在 $\{ a_i, a_{i+1}, a_{i+2}, \ldots, a_{j-2}, a_{j-1}, a_j \}$ 中，a_i 或 a_j 率先被确认

$$T(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} Pr(i,j) = \sum_{j=1}^{n-1} \sum_{i=0}^{j-1} Pr(i,j) = \sum_{j=1}^{n-1} \sum_{d=1}^{j} \frac{2}{d+1} \approx \sum_{j=1}^{n-1} 2 \cdot (\ln j - 1) \leq 2 \cdot n \cdot \ln n$$

对比：

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



快读划分：DUP版

> 大量甚至全部元素**重复**时，轴点位置总是接近于`lo`，子序列的划分极度失衡，二分递归退化为线性递归，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

LUG'版本：勤于拓展、懒于交换

```
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
    hi--; 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
        while (lo < hi && pivot <= _elem[hi]) hi--; //向左拓展G
        if (lo < hi) _elem[lo++] = _elem[hi]; //凡小于轴点者，皆归入L
        while (lo < hi && _elem[lo] <= pivot) lo++; //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
    _elem[lo] = pivot; return lo; //候选轴点归位；返回其秩
}
```

DUP'版本

```
template <typename T>
Rank Vector<T>::partition( Rank lo, Rank hi ) { //[lo, hi)
```

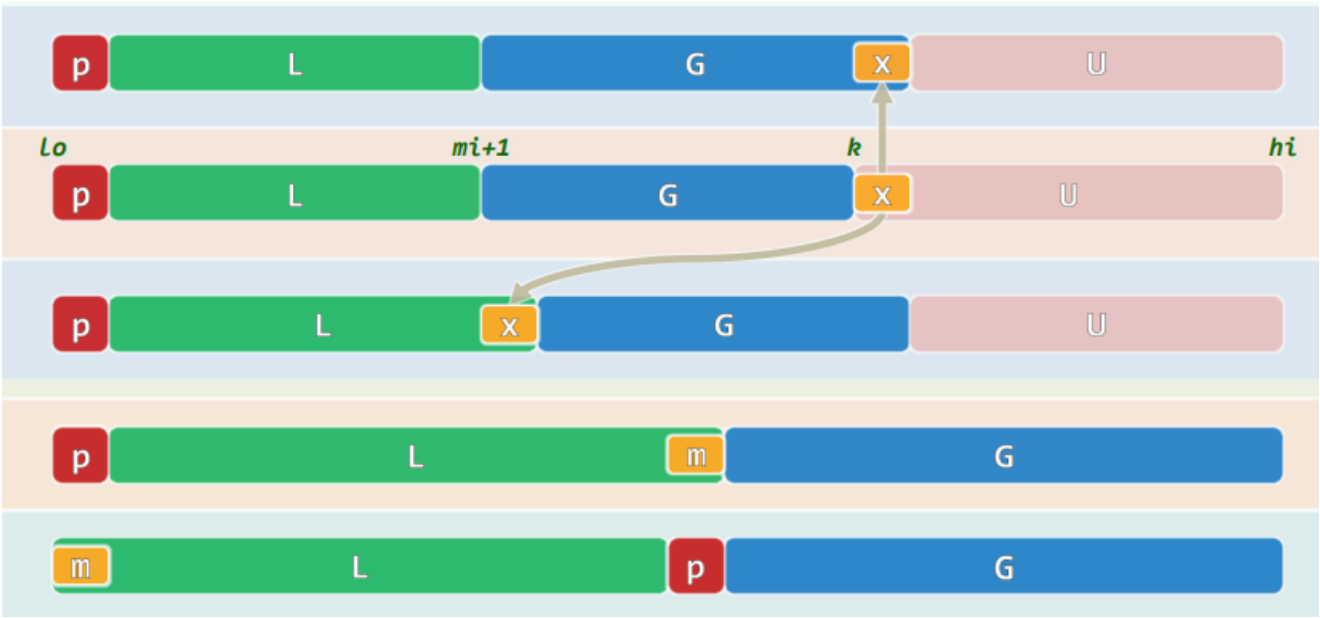
```
    swap( _elem[lo], _elem[lo + rand() % (hi - lo)] ); //随机交换
    hi--; T pivot = _elem[lo]; //经以上交换，等效于随机选取候选轴点
    while (lo < hi) { //从两端交替地向中间扫描，彼此靠拢
        while (lo < hi && pivot < _elem[hi]) hi--; //向左拓展G
        if (lo < hi) _elem[lo++] = _elem[hi]; //凡不大于轴点者，皆归入L
        while (lo < hi && _elem[lo] < pivot) lo++; //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
    _elem[lo] = pivot; return lo; //候选轴点归位；返回其秩
}
```

DUP版本：懒于拓展、勤于交换

```
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
    hi--; 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
        while (lo < hi)
            if (pivot < _elem[hi]) hi--; //向左拓展G，直至遇到不大于轴点者
            else { _elem[lo++] = _elem[hi]; break; } //将其归入L
        while ( lo < hi )
            if (_elem[lo] < pivot) lo++; //向右拓展L，直至遇到不小于轴点者
            else { _elem[hi--] = _elem[lo]; break; } //将其归入G
    } //assert: lo == hi
    _elem[ lo ] = pivot; return lo; //候选轴点归位；返回其秩
}
```

快速化分：LGU版

不变性：$S = [lo, hi) = [lo] + (lo, mi] + (mi, k) + [k, hi) = pivot + L + G + U$
$L < pivot \leq G$



单调性：$[k]$不小于轴点？直接G拓展：G滚动后移，L拓展
$pivot \leq S[k]?k++ : swap(S[++mi], S[k++])$

```
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]; int 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，L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
}
```

13.2 快速选择 QuickSelect

```
template <typename T>
void quickSelect( Vector<T> & A, Rank k ) {
    for ( Rank lo = 0, hi = A.size() - 1; 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)
            while ( i < j && pivot <= A[j] ) j--; A[i] = A[j];
            while ( i < j && A[i] <= pivot ) i++; A[j] = A[i];
        } //assert: quit with i == j
        A[i] = pivot;
        if ( k <= i ) hi = i - 1;
        if ( i <= k ) lo = i + 1;
    } //A[k] is now a pivot
}
```

期望性能

1. 记期望的比较次数为$T(n)$
   $T(1)=0, T(2)=1, \dots$
2. 可以证明：$T(n)=O(n)$
   $T(n)=(n-1)+\frac{1}{n}\times\sum_{k=0}^{n-1} max\{T(k),T(n-k-1)\}\le(n-1)+\frac{2}{n}\times\sum_{k=\frac{n}{2}}^{n-1}T(k)$
3. 事实上，不难验证：$T(n)<4\cdot n$
   $T(n)\le(n-1)+\frac{2}{n}\times\sum_{k=\frac{n}{2}}^{n-1}4k\le(n-1)+3n<4n$

13.3 线性选择 LinearSelect

```
linearSelect( A, n, k )
```

1274

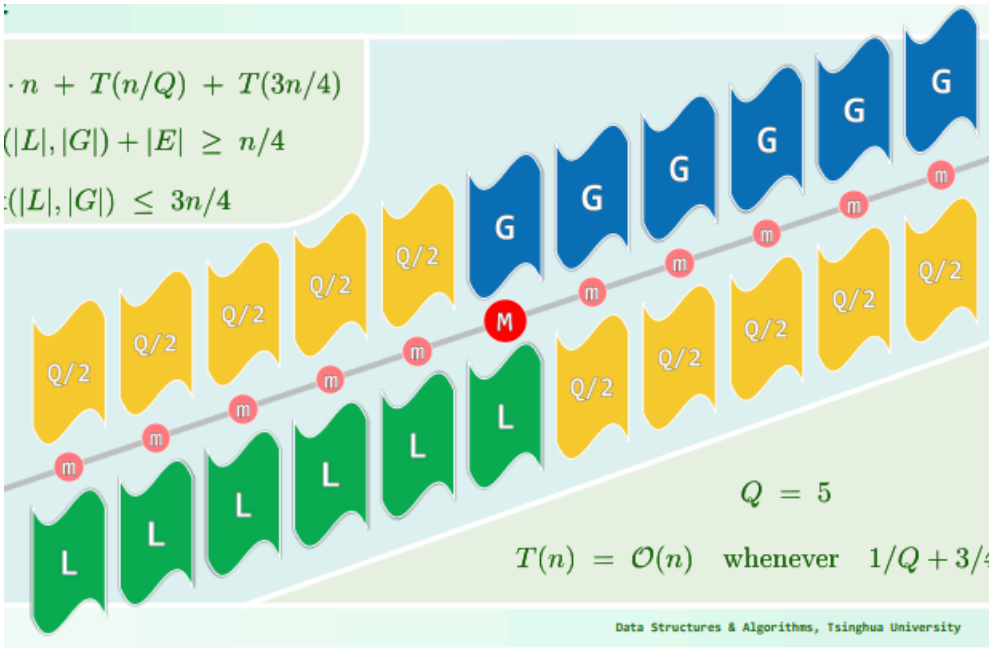
```
Let Q be a small constant
0. if ( n = |A| < Q ) return trivialSelect( A, n, k )
1. else divide A evenly into n/Q subsequences (each of size Q)
2. Sort each subsequence and determine n/Q medians  //e.g. by insertionsort
3. Call linearSelect() to find M, median of the medians  //by recursion
4. Let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



复杂度

$T(n) = c \cdot n + T(n/Q) + T(3n/4)$
$min(|L|, |G|) + |E| \geq \frac{n}{4}$
$max(|L|, |G|) \leq \frac{3n}{4}$

$Q = 5,\ T(n) = O(n)$ whenever $\frac{1}{Q} + \frac{3}{4} < 1$



❖ 将linearSelect()算法的运行时间记作T(n)

❖ 第0步：$O(1) = O(QlogQ)$ //递归基：序列长度|A| ≤ Q

❖ 第1步：$O(n)$ //子序列划分

❖ 第2步：$O(n) = Q^2 \times n/Q$ //子序列各自排序，并找到中位数

❖ 第3步：$T(n/Q)$ //从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：$T(3n/4)$ //为什么...