

# 数据结构复习提纲



张晓敏（第八、九、十三章）

鲍伟（第三、四、五、六、七、十二章）

徐卫霞（第九章）

杜泽林（第二章）

孙吉鹏（第十四章）

## 第二章 程序性能

### 一、概念

（一）程序性能——运行一个程序所需要的内存和时间。→时间复杂性&空间复杂性

（二）空间复杂性：指令空间+数据空间+环境栈空间

1、程序所需要的指令空间的数量取决于如下因素：

- （1）把程序编译成机器代码的编译器。
- （2）编译时实际采用的编译器选项。
- （3）目标计算机

2、不同类型的数据占用的内存也不相同。

3、每当一个函数被调用时，下面的数据将被保存在环境栈中：

- （1）返回地址
- （2）函数被调用时所有局部变量的值以及传值形式参数的值（仅对于递归函数而言）
- （3）所有引用参数及常量引用参数的定义

\*无法精确地分析一个程序所需要的空间。

\*可以确定程序中某些部分的空间需求，这些部分依赖于所解决实例的特征。

4、空间复杂性度量—— $S(P) = c + Sp$ （实例特征）

(1) 固定部分  $C$ ，独立于实例的特征。一般来说，这一部分包含指令空间（即代码空间）、简单变量及定长复合变量所占用空间、常量所占用空间等等。

(2) 可变部分  $S_p$ ，由以下部分构成：

复合变量所需的空间（这些变量的大小依赖于所解决的具体问题）

动态分配的空间（这种空间一般都依赖于实例的特征）

递归栈所需的空间（该空间也依赖于实例的特征）

局部变量及形式参数所需要的空间。

递归的深度（即嵌套递归调用的最大层次）。

(三) 时间复杂性——一个程序  $P$  所占用的时间  $T(P)$ =编译时间+运行时间。

1、估计运行时间的方法：P37 例 2-7（最大元素）、例 2-8（多项式求值）

(1) 找出一个或多个关键操作(对时间复杂性的影响最大)，确定这些关键操作所需要的执行时间

(2) 确定程序总的执行步数

2、执行步数——程序步

详见 P44-2.3.3 及其例题

(四) 渐近符号—— $O$ 、 $o$ 、 $\Omega$ 、 $\Theta$

1、 $f(n)=O(g(n))$  当且仅当存在正的常数  $c$  和  $n_0$ ，使得对于所有的  $n \geq n_0$ ，有  $f(n) \leq cg(n)$

(1) 最小上限

(2) 常用函数参见课本 P56~57

(3) 为了使语句  $f(n)=O(g(n))$  有实际意义，其中的  $g(n)$  应尽量地小。因此常见的是  $3n+3=O(n)$ ，而不是  $3n+3=O(n^2)$ ，尽管后者也是正确的。

(4) 有用的结论：

1. 如果  $f(n)=amnm+\dots+a_1n+a_0$  且  $am>0$ ，则  $f(n)=O(nm)$ 。

2. 加法规则： $T(n)=T_1(n)+T_2(n)=O(g_1(n))+O(g_2(n))=O(\max(g_1(n),g_2(n)))$ ——相加取其大

3. 乘法规则： $T(n)=T_1(n)*T_2(n)=O(g_1(n))*O(g_2(n))=O(g_1(n)*g_2(n))$

2、 $f(n)=\Omega(g(n))$  当且仅当存在正的常数  $c$  和  $n_0$ ，使得对于所有的  $n \geq n_0$ ，有  $f(n) \geq cg(n)$ 。

(1) 最大下限

(2) 有用的结论：如果  $f(n)=amnm+\dots+a_1n+a_0$  且  $am>0$ ，则  $f(n)=\Omega(nm)$ 。

3、 $f(n)=\Theta(g(n))$  当且仅当存在正常数  $c_1, c_2$  和某个  $n_0$ ，使得对于所有的  $n \geq n_0$ ，有  $c_1g(n) \leq f(n) \leq c_2g(n)$

(1) 有用的结论：如果  $f(n)=amnm+\dots+a_1n+a_0$  且  $am>0$ ，则  $f(n)=\Theta(nm)$

4、 $f(n)=o(g(n))$  当且仅当  $f(n)=O(g(n))$  且  $f(n) \neq \Omega(g(n))$

\*请仔细阅读 2.4 节的例题

## 二、经典算法

搜索算法——顺序搜索/折半搜索算法

排序算法——名次排序/选择排序/冒泡排序/插入排序算法

1、顺序搜索——遍历数据并逐一比较

2、折半搜索 P64 程序 2-30

主要思路：

搜索过程从  $x$  与数组  $[left:right]$  中间元素的比较开始。

如果  $x$  等于中间元素，则查找过程结束。

如果  $x$  小于中间元素，则仅需要查找数组的左半部分，所以  $right$  被修改为  $middle - 1$ 。

如果  $x$  大于中间元素，则仅需要在数组的右半部分进行查找， $left$  将被修改为  $middle +$

1。

- 3、名次排序——先计算每个元素的具体位置，再将其移动到相应位置
- 4、选择排序——每次寻找出[0,n]中的最大元并将其置于[n-1,每次循环 n--。
- 5、冒泡排序——每次冒泡依次将相邻元素的较大元向右移动，得到[0,n]的最大元，每次循环 n--。
- 6、插入排序

主要思路：

因为只有一个元素的数组是一个有序数组，所以可以从包含 n 个元素的数组的第一个元素开始。通过把第二个元素插入到这个单元数组中，可以得到一个大小为 2 的有序数组。插入第三个元素可以得到一个大小为 3 的有序数组。按照这种方法继续进行下去，最终将得到一个大小为 n 的有序数组。

## 第三章 数据描述

链表的最基本的操作实际上就是查找，插入和删除是基于查找之上的操作。

### 一、公式化描述：

所谓的公式化描述就是利用数组随机访问的特性组织数据，将数据连续的存放在数组中，利用数组的下标快速的访问。

$location(i) = i-1;$

优点：因为使用了数组，随意访问具有随意性，可以在常量时间内访问第 k 个元素。

缺点：数组的特点在于长度不可修改，所以对于扩大存储容量来说，公式化描述的线性表是很困难的；其次由于在删除和插入时保证数据的连续性，不得不移动元素，这是需要消耗时间的。

### 二、链表描述：

链表描述就是使用一个标记记录下每一个节点的下一个节点的地址，每次通过一个节点的 link 找到下一个节点。

对于链表要熟悉掌握查找，插入删除的操作代码。一般代码中都会涉及几种情况：

- 判断 k 是否合法 ( $k < 0$  或 1，或者是 k 是否超出上界)；
- 判断链表是否为空；
- 找到第 k 个元素进行操作；
- 操作时注意几种情况：

①是否在头结点操作；

②是否在尾结点操作；

③操作结束后链表的情况（是否为空等）；

在单链表的基础上扩充单向循环链表，双向链表，双向循环链表等，本质上的操作都和单链表类似，注意将每种情况都考虑到就好。

优点：由于两个物理节点之间在存储上并没有什么实质的联系，可以是不连续的存储，所以链表的最大的就是它的存储是没有限制的。只要有足够的空间链表就可以继续扩充。在插入和删除方面，因为只需要修改 link 指针的指向就可以实现，所以不需要大量的移动元素，相比于公式化描述减少了时间复杂度。

缺点：相比于公式化描述的链表而言，因为多了一个要保存的下一节点的地址，所以要付出更多的空间。其次由于不是连续存储，所以只能依靠某一节点寻找下一节点，这就导致了查询的时间的消耗。因为插入和删除实际上是基于查询的基础上的，所以这就导致了插入和删除的复杂度并没有想象中的那么快。

### 三、间接寻址

间接寻址的技术实际是想结合公式化描述和链表描述的优点。使用一个数组记录每一个

元素的地址，根据地址找到真正的元素，这一点和机组中的间接寻址是一样的。就是依据字典的目录查找具体的字。

优点：因为使用数组记录每一个地址，所以可以实现在单位的时间内查找元素。这是链表做不到的。相比较于公式化描述的链表，因为只要保存节点地址的指针而不用保存具体的元素，所以移动元素所耗费的时间比公式化要少。

缺点：凡是涉及使用数组的，都存在扩容的问题，这是一个通病。

#### 四、模拟指针

如果说间接寻址是对链表描述的线性表不能再单位时间内查询的优化，那么模拟指针就是对公式化描述的线性表需要频繁移动数据的优化。模拟指针相当于把链表描述的线性表给整合到一个数组中，节点的 **link** 保存的是下一个节点在数组中的位置（下标）。这样只要修改 **link** 的值就可以像链表一样不用移动元素。实际的操作还是链表的操作。

#### 五、应用：

- 1、箱子排序；
- 2、基数排序；
- 3、等价类；

等价类中关键的两个方法就是 **find** 和 **union**（查找某一元素的所在等价类，以及合并两个等价类）。

链表实现的等价类是将每一个等价类用一条链表来描述，链表的头就是该等价类的等价元，链表中的每个节点要保存的是 **E**（所属等价类），**size**（等价类大小实际使用时只要修改链表的第一个元素的 **size** 就可以了），**link**（下一节点的地址）这三个信息。两个等价类的合并实际就是两个链表的合并。在合并（**i**, **j**）时把较短的一条链中的节点所在的等价类 **E** 全部修改为 **j**，然后将 **size** 修改，最后是将第二条链的首部接到第一条链的二号位置，将第一条链的二号位置之后的链接到第二条链的尾部（这样做是有原因的，很多人可能想的直接把第二条链接在第一条链的后面，这样做就需要找到第一条链的最后一个位置，这是需要时间的，所以不如从头插入；而将一号链的二号位置及之后的节点接到第二条链是可以接受的，因为我们在修改第二条链的等价类时就已经找到了最后一个节点的位置。）

## 第四章 数组和矩阵

了解行主映射和列主映射

#### 一、数组：

课本中介绍的一维数组和二维数组类实际上就是我们平时学习的数组的一个扩充，这两个类主要是重载了一些运算符，使数组之间的运算变得比较简单（调用一下函数即可）。需要注意的是，在定义二维数组的时候，大量的运用了一维数组的方法来实现。

数组的下标是从 0 开始的，而且默认的是 0 行 0 列的数组。

#### 二、矩阵：

矩阵最重要的两个操作就是 **Store** 和 **Retrieve**，一个是如何存，一个是如何访问，这两个是一个相反的过程。对于这两个方法，本质上只要知道了矩阵中（**i**, **j**）行和列对于的存储关系就可以了。

这里需要注意的是矩阵的下标是从 1 开始的。

一般的矩阵： $d[(i-1)*cols+j-1]$

对角矩阵：（对角线映射）

- ①  $d[i-1]$   $i=j$
- ② 0  $i \neq j$

三对角矩阵：（对角线映射）

- ①  $d[i-2]$   $i-j = 1;$
- ②  $d[n+i-2]$   $i=j$
- ③  $d[2*n+i-2]$   $i-j=-1$
- ④  $0$   $|i-j|>1$

下三角矩阵：（行主映射）

- ①  $d[i*(i-1)/2+j-1]$   $i>=j$
- ②  $0$   $i<j$

上三角矩阵和对称矩阵都可以归纳到下三角矩阵中，就不具体解释了。

对于数组和矩阵还有一点需要说明，对于数组和矩阵的构造方法，如果类型  $T$  是内部数据类型（如 `int`, `char`）那么构造函数的复杂度是  $O(1)$ ，如果是自定义的类型，那么复杂度是（一维数组就是  $O(\text{size})$ ，二维数组就是  $O(\text{rows}*\text{cols})$ ），这是因为如果是自定义的类，在 `new` 的时候需要调用  $T$  的构造函数。

三、稀疏矩阵：

第四章中最难的就是稀疏矩阵，它有两种描述方式——数组描述和链表描述。

我们重点学习的是数组描述，对于稀疏矩阵的操作，掌握转置和两个稀疏矩阵的加即可。

转置：在转置的代码中有两个重要的中间量 `ColSize` 和 `RowNext`；`ColSize` 顾名思义就是转置前每一列中非 0 元素的个数。`RowNext` 则是记录每一行转置后所在的数组中的位置。

0	0	0	2	0	0	1	0
0	6	0	0	7	0	0	3
0	0	0	9	0	8	0	0
0	4	5	0	0	0	0	0

a[]	0	1	2	3	4	5	6	7	8
row	1	1	2	2	2	3	3	4	4
col	4	7	2	5	8	4	6	2	3
value	2	1	6	7	3	9	8	4	5

`ColSize`：（注意是转置后的列）

0	0	2	1	2	1	1	1	1
---	---	---	---	---	---	---	---	---

`RowNext`:(下标为 0 的是不用的)

	0	0	2	3	5	6	7	8
--	---	---	---	---	---	---	---	---

`RowNext[i] = ColSize[i-1]+RowNext[i-1];`

//第  $i$  列的第一个元素的位置 `RowNext[i]` 应该是  $i-1$  列的位置加上  $i-1$  行的非 0 元素的大小。

两个稀疏矩阵的相加一般步骤：

1、先比较两个矩阵的行列是否相等，不相等抛出异常；如果相等就创建一个新的稀疏矩阵 `c`；

2、两个游标 `ct`（遍历 `this` 矩阵的游标）和 `cb`（遍历 `b` 矩阵的游标）扫描，先比较两个游标所指的元素在矩阵中的位置先后（按照行主映射的位置），

①将先出现的元素追加到 `c` 中，然后移动对应的游标；

②如果位置相等，计算相加之后的值再追加到 `c` 中。

3、当其中一个数组遍历结束时，可以将另一个的元素逐个追加到 `c` 中；

注意：在比较 `ct` 和 `cb` 游标所指元素的位置先后时，使用了 `indt`（记录 `ct` 所指元素按行主映

射的位置)和 `indb` (记录 `cb` 所指元素按行主映射的位置) 两个中间量, 直接比较 `indt` 和 `indb` 的大小比较位置先后, 避免了直接比较行列的复杂代码。

## 第五章——堆栈

堆栈是一种 LIFO (后进先出) 的数据结构, 只允许从一端插入和删除。常见的操作有: `IsEmpty`, `IsFull`, `Add`, `Delete`, `Top`;

### 一、堆栈的实现

堆栈的实现由四种方法——使用公式化描述的堆栈 (派生类和自定义两种), 链表描述的堆栈 (派生类和自定义两种);

#### 1、公式化描述的派生类: 这个类是继承 `LinearList` 类实现的

优点: 大大的减少了代码的编写量, 调用 `LinearList` 中的成员函数即可。

程序的可靠性很强 (因为 `LinearList` 是确认没有问题的);

缺点: 每次插入和删除的时候都需要将整个数组遍历到最后, 时间开销大, 效率低。

#### 2、公式化描述的自定义类: 这个类里定义了一个 `int` 型变量 `top` 用来记录堆栈的栈顶, 插入元素时先判断栈是否满, 如果没有就给 `stack[++top]` 赋值, 删除元素时先判断栈是否为空, 如果不是 `x=stack[top--]` 即可。

判断栈满的方法: 比较 `top` 和 `MaxSize` 即可。

优点: 插入和删除的效率大大提高;

缺点: 数组的通病, 空间的利用率不高 (可以使用一个数组管理多个堆栈来弥补, 这样比较复杂不做讨论)。

#### 3、链表描述的堆栈: 实际上派生类和自定义类并没有太多的区别。对于链表, 从头部开始插入和删除的效率是最高的, 都是单位时间。所以在用链表实现堆栈时可以把头部作为堆栈的顶。

对于链表描述的堆栈判断栈满的操作是, 先尝试 (`try`) 创建一个节点, 如果可以创建则说明有空间, 然后删掉创建的节点 (因为我们并不打算使用它), 如果不能创建则抓住 (`catch`) `NoMem` 的异常, 返回 `true`。不够优雅。

优点: 插入和删除的操作 都是单位时间。

### 二、应用

#### 1.括号匹配:

对表达式的每个字符从左至右进行遍历;

遇到左括号就入栈;

遇到右括号就弹出栈顶匹配的左括号元素 (弹出的时候应该 `try……catch` 一下, 避免从空栈中删除元素, 这种情况就是没有匹配的左括号);

在处理完每个字符后, 在处理栈里没有处理的左括号 (这种情况是没有匹配的右括号), 将他们输出。

#### 2、汉诺塔:

解决汉诺塔问题最优雅的算法就是递归, 使用递归的复杂度是  $O(2^n)$  (2 的 n 次方); 课本上给出了一种使用堆栈解决汉诺塔的问题, 本质上还是使用了递归, 只是每次递归操作的栈都是同一组栈。



因为汉诺塔只能从柱子的一端放盘子，一端取盘子，所以和栈的操作是一样的。使用三个堆栈表示三个柱子。

### 3、车厢重排：

从前往后检查入轨上车厢的编号

如果刚好是要输出的编号（`p[i]==NowOut`）则输出；

处理完之后（`NowOut++`）循环检查缓冲轨上的元素是否有需要输出的（`MinH==NowOut`），直到不能输出为止；

如果不是要输出的编号，则将这节车厢放到缓冲轨上

如果不能放进缓冲轨上（`! Hold` 即放进去影响后面的排序）那么返回 `false`，表示这个序列没办法重排；

**Hold 函数：**车厢 `u` 被放在 `v` 号缓冲轨上，`v>u`，且 `v` 应该是所有满足条件中最小的那个编号。

### 4、离线等价类：

使用一组链表描述，每一个链表里记录与当前元素的等价元素。

添加关系（`i, j`）：在第 `i` 个链表中添加 `j` 元素，并且在第 `j` 个链表中添加 `i` 元素。

输出等价类：使用 `out` 数组记录元素是否输出，从 `0` 开始遍历数组，先判断第 `i` 个元素是否输出，如果没有则输出，然后将与之等价的未输出元素添加至堆栈中逐个处理。

### 5、迷宫老鼠：

基本思路：

为了避免迷宫的边缘（有两三种选择）和内部（四种选择）操作的不同，可以在原来迷宫的周围加上一层障碍物。

选择路劲的原则，事先规定好遍历的选择（`option`）：向右，向下，向左，向上。然后为这几个 `option` 设置一个偏移量的数组，为了后面操作的简单。设置了一个 `LastOption`（最后的选择），当 `option>LastOption` 时，表示没有可以选择的路可走。

避免走重复的路：对于每一个走过的单元，都会设置为 `1`（有障碍物），这样在回溯的时候就不至于从新走一遍。

遍历的原则：首先把迷宫的入口作为当前位置。

- 如果当前位置是迷宫出口，那么已经找到了一条路径，搜索工作结束。
- 如果当前位置不是迷宫出口
  - 则在当前位置上放置障碍物，以便阻止搜索过程又绕回到这个位置。
  - 然后检查相邻的位置中是否有空闲的(即没有障碍物)
    - 如果有，就移动到这个新的相邻位置上，然后从这个位置开始搜索通往出口的路径。
    - 如果没有，回溯，选择另一个相邻的空闲位置，并从它开始搜索通往出口的路径。

如果所有相邻的空闲位置都已经被探索过，并且未能找到路径，则表明在迷宫中不存在从入口到出口的路径。

## 第六章——队列的总结

队列是一种 FIFO（先进先出）的数据结构，从一端进，从另一端出。

### 一、实现

队列中使用 `front` 和 `rear` 两个游标分别标记队首和队尾。

### 1、数组实现（不循环）：

队列空的条件（ $\text{rear} < \text{front}$ ），队满的条件（ $\text{rear} == \text{MaxSize} - 1 \ \&\& \ \text{front} == 0$ ）。

缺点：因为在多次添加和删除后可能出现  $\text{first} > 0$ ， $\text{rear} = \text{MaxSize} - 1$  的情况（也就是数组的头部有空间，但不能继续插入），此时为了继续插入元素，必须移动元素到对首，这就消耗了时间。

### 2、数组实现（循环）：

队列空的条件（ $\text{front} == \text{rear}$ ），队满的条件（ $(\text{rear} + 1) \% \text{MaxSize} == \text{front}$ ）

优点：循环使用数组，使数组的利用率提高了。

注意：因为是循环使用，所以每次操作  $\text{rear}$  和  $\text{front}$  加一的时候总是要模  $\text{MaxSize}$ ，避免数组越界。

### 3、链表实现：

队列空的条件（ $\text{front} == \text{rear}$ ），队列满的条件（同栈的判断一样，尝试申请一个新节点。）

链表的实现的队列和链表本身的实现没有什么区别，只是多了一个  $\text{rear}$  指针指向最后一个元素，所以可以在单位时间内插入元素。

## 二、应用：

### 1、车厢重排

使用队列重排的实现和使用堆栈的重排是类似的，但需要注意区别

1）、Hold 函数：将  $u$  放入队尾为  $v$  号车厢的缓冲轨上， $v$  应该满足  $v < u$ ， $v$  是所有满足条件的最大值。如果有空缓冲轨则入空轨。

### 2、迷宫最短路径

同迷宫老鼠不同，迷宫老鼠是寻找的有效路径（即有路径即可），而迷宫最短路径顾名思义就是寻找最短的路径。

为了避免迷宫内部元素和边缘操作的不同，同样的在外围增加一圈障碍物。

路径选择原则：同迷宫老鼠（向右，向下，向左，向上），同样设置偏移量数组。

避免回到原点：从原点出发时将原点设置为 2，因为原点的周围四个方向的数据只可能是 1，所以在处理这些 1 时，由于原点非零，故不会循环处理。

搜索原则：（从  $a$  到  $b$  的路径）

先从  $a$  开始，将  $a$  所能到达的所有方格标记为 1，并添加到队列中；

然后将依次处理队列中的 1，将 1 所能到达的方格标记标记为 2，并添加到队列中；

然后依次处理队列中的元素，直到搜索到  $b$ ；或者队列为空，队列为空则表示没有这样的路径。

输出路径：从  $b$  开始寻找比  $b$  的标号小 1 的方格，然后以当前方格为基础寻找比当前方格小 1 的方格，重复这样的操作，直到到达  $a$  为止。

## 第七章——跳表和散列

注：这一部分比较难，大家理解概念就好。

### 一、字典的实现：

字典的实现实际上和链表本身的实现没有太多的区别，只不过是字典是有序的，插入和删除时都要保证字典的特性。相比较于链表本身，就是在从前往后查找的时候多了一个  $\text{key}$  值得判断。——P.219

### 二、跳表



由于链表描述的特性，我们没有办法像公式化描述一样二分搜索，但是如果在链表添加若干的指针（例如增加一个指向中间元素的指针），我们就可以利用二分搜索的思想减少比较的次数。将时间复杂度缩小到  $\log N$  的级别。

**i 级链元素：**元素在最高可以在 i 级链上，那么元素就是 i 级链元素。第 0 级链元素有 n 个，第 i 级链上的元素有  $n/(2^i)$  的 i 次方。

可以把跳表想象成是在最基础的链表上选出几个中间几个元素组成 1 级链表，然后从 1 级链表中选出中间的若干元素组成 2 级链表……如此重复，将这几个多级链表组合在一起查询就是跳表。

**如何插入元素：**先确定插入元素的级数 i（这是通过概率来模拟的），然后通过跳表的二分查询找到应该插入的位置，然后将元素插入，修改  $0 \sim i$  级的链表指针。这样就保证了跳表的结构，时间复杂度  $O(n)$ ——P.223

跳表的插入，查询和删除的时间复杂度为  $O(n + \text{MaxLevel})$ 。平均时间复杂度  $O(\log N)$ 。  
三、Hash 散列（线性开型寻址）

1、数组描述：哈希函数  $f(k) = k \% D$ ;

**如何插入：**插入关键字为 k 的元素时，先通过哈希函数计算出对应的桶的编号，查看  $f(k)$  对应的桶是否有元素，没有就直接插入该桶，如果有，则发生碰撞。

**如何处理碰撞：**一个有效的方法就是将发生碰撞的元素插入  $f(k)$  桶后的第一个空桶中。这里值得注意的是数组是循环使用的（同循环队列），所以最后一个桶的下一个就是第一号桶。

**如何查询：**先通过哈希函数找到  $f(k)$  桶，从  $f(k)$  桶及之后的桶里查看是否有关键字为 k 的元素，1) 找到对应元素，2) 找到一个空桶，3) 再次找到  $f(k)$  桶。对于后两种情况表示没有关键字为 k 的元素。

**如何删除：**删除的时候可能会带来某些元素无法被查询到，这个时候为了保持散列的特性，我们需要适当地移动元素来维持。也可以不移动元素而是采用布尔值标记桶的状态——P.231

2、链表描述：链表描述的散列中，数组存放的是每一个桶（链表）的头结点，使用链表处理碰撞显得相对容易。

**如何插入：**通过哈希函数找到  $f(k)$  个桶，然后将元素添加到链表中。由于每次插入都要进行一遍查询，所以有序的链表会更加有效。

**如何删除：**和链表的操作类似。只不过是第  $f(k)$  个桶的有序链表上进行操作。

**优化：**在每个链表的末尾增加一个无穷大的标记（也可以所有的链表共用一个无穷大标记。）这样相当于一个哨兵，减少了每次循环的判断次数。

## 第八章 树<sup>[0]</sup>

### +概念&理解

**树：**t 是一个非空的有限元素的集合，其中一个元素为**根**，余下的元素组成 t 的子树。

**根：**层次中最高层的元素。

**孩子：**根的下一级的节点。是其他元素构成的子树的根。

**叶子：**没有孩子节点的元素。

**级：**规定树根的级为 1，其孩子的级为 2，孩子的孩子为 3，以此类推。

**元素的度：**指其孩子的个数。

**树的度：**元素的度的最大值。

对比	
线性结构	树形结构
第一个数据元素（无前驱）	根结点（无前驱）
最后一个数据元素（无后继）	多个叶子节点（无后继）
其他元素（一前驱，一后继）	其他元素（一前驱，多后继）

#### +二叉树·基本概念&理解

**二叉树：**t 是有限个元素的集合（可以为空）。当二叉树非空时，其中有一个称为**根**的元素，余下的元素（如果存在）被组成 2 个二叉树，分别称为 t 的**左子树**和**右子树**。

**区别：**

二叉树可以为空，树不可为空；

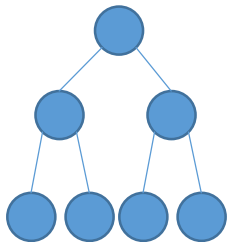
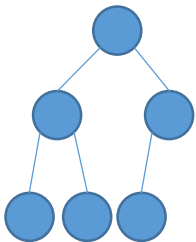
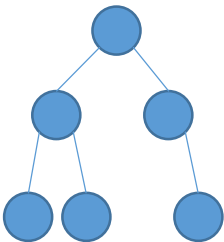
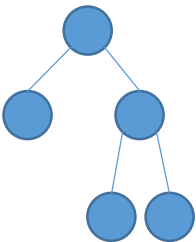
二叉树每个元素恰有 2 棵子树（子树可能为空），树可有若干子树；

二叉树子树间有序且能分为左子树与右子树，树的子树间无序。

**满二叉树：**高度为 h 的二叉树恰好有  $2^h - 1$  个元素时，这棵二叉树称为满二叉树。

**完全二叉树：**深度为 k，节点个数为 n 的二叉树为完全二叉树，当且仅当它与 k 层满二叉树前  $1 \sim n$  个节点所构成的二叉树结构相同。

即：前 k-1 层为满二叉树，其余元素连续排列于第 k 层左端。

			
满二叉树	完全二叉树	非完全二叉树	非完全二叉树

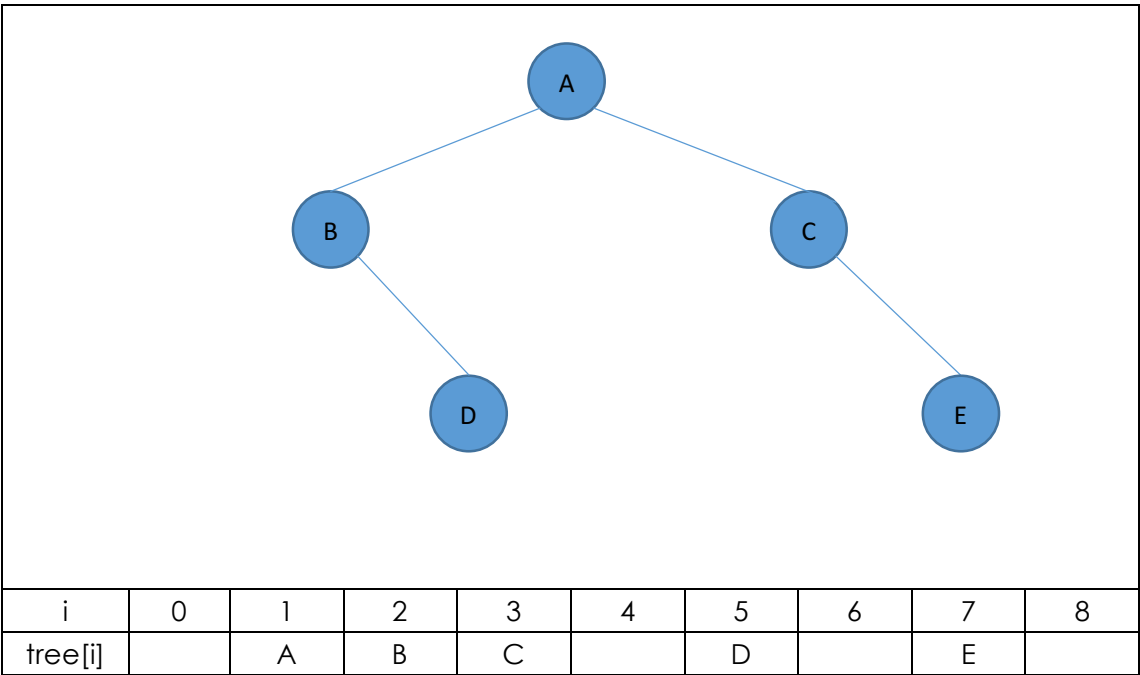
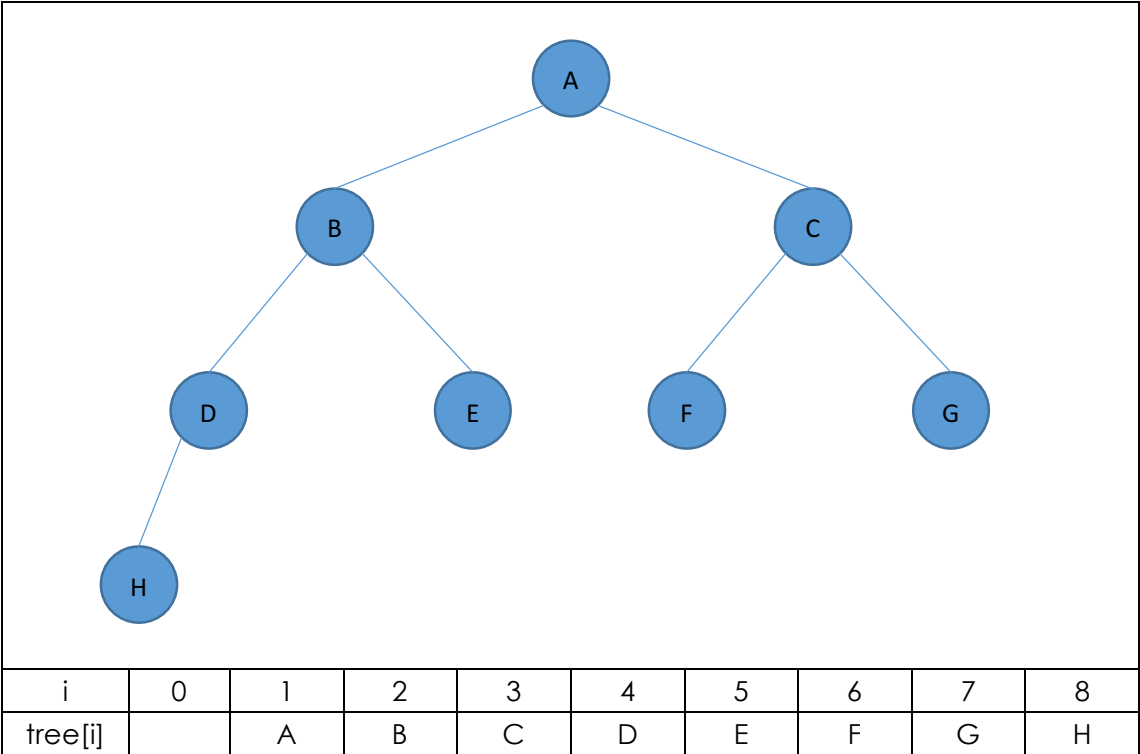
**二叉树特性：**

1. 包含  $n$  ( $n > 0$ ) 个元素的二叉树有  $n-1$  条边。
2. 高度为  $h$  ( $h \geq 0$ ) 的二叉树元素个数最少有  $h$  个，最多有  $2^h - 1$  个。
3. 元素为  $n$  ( $n \geq 0$ ) 个元素的二叉树高度最高为  $n$ ，最低为  $\lceil \log_2(n+1) \rceil$ 。
4. 假设完全二叉树一元素序号为  $i$  ( $1 \leq i \leq n$ )，则满足以下条件：
  - 4.1. 若  $i=1$ ，则该元素为根节点；若  $i>1$ ，则该元素的父亲节点为  $i/2$ ；
  - 4.2. 若  $2i > n$ ，则该元素无左儿子节点；否则该元素的左儿子为  $2i$ ；
  - 4.3. 若  $2i+1 > n$ ，则该元素无右儿子节点；否则该元素的右儿子为  $2i+1$ ；
5. 度为 0 的节点数=度为 2 的节点数+1。

#### +二叉树的存储

两种方法：**公式化描述**与**链表描述**。

公式化描述：利用特性 4，可以将二叉树元素存在数组中。



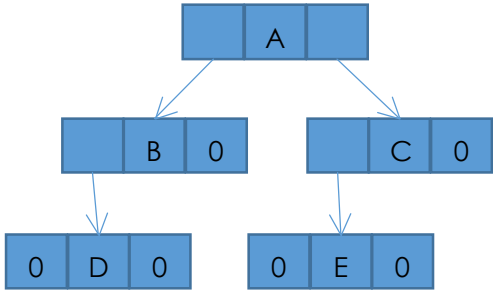
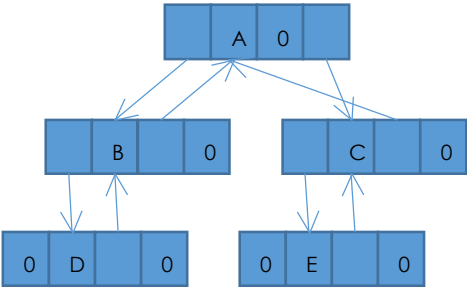
存储空间使用：n+1 到  $2^n$ （取决于缺少元素的数量），在树右斜时会最占用存储空间。  
因此公式化描述适合在缺少元素个数较少时使用。

+链表描述

- 每个元素由一个节点表示。
- \*三叉链表添加一个父亲节点。

然后树就变成了这样

Left Child	Data	Right Child	
Left Child	Data	Parent	Right Child

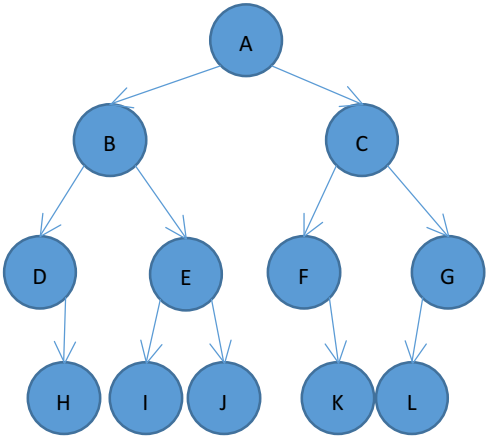
二叉链表				三叉链表				
								
No.	Data	Le.Ch.	Ri.Ch.	No.	Data	Par.	L.Ch.	R.Ch.
1	A	2	3	1	A	0	2	3
2	B	4	0	2	B	1	4	0
3	C	5	0	3	C	1	5	0
4	D	0	0	4	D	2	0	0
5	E	0	0	5	E	3	0	0

-代码(二叉链表)

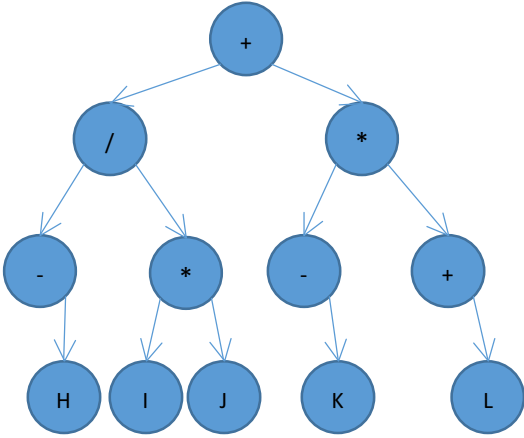
```
template class <T>
class BinaryTreeNode
{
public:
{
    BinaryTreeNode(){LeftChild=RightChild=NULL;}
    BinaryTreeNode(T& e){data=e;LeftChild=RightChild=NULL;}
    BinaryTreeNode(T& e,BinaryTreeNode* l,BinaryTreeNode* r)
    {data=e;LeftChild=l;RightChild=r;}
}
private:
{
    T data;
    BinaryTreeNode<T> *LeftChild,*RightChild;
}
}
```

+二叉树遍历

二叉树遍历分前序遍历、中序遍历、后序遍历、层序遍历。  
在二叉树操作中，通常是通过在遍历过程中访问节点时进行操作，例如复制、删除、输出等。

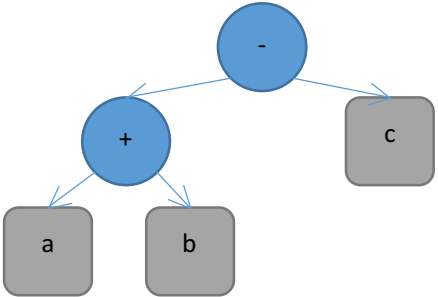
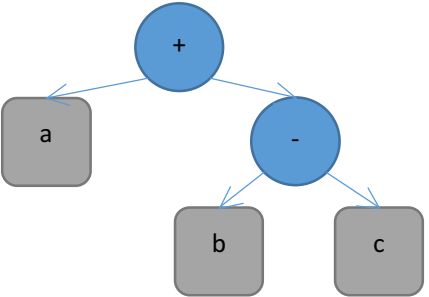
	前序遍历	ABDHEIJCFKGL
		访问根-左子树-右子树
	中序遍历	DHBIEJAFKCLG
		左子树-访问根-右子树
	后序遍历	HDIJEBKFLGCA
		左子树-右子树-访问根
	层序遍历	ABCDEFGHGIJKL
		以层级顺序输出

-前中后缀表达式

	前缀表达式	+/-H*IJ*-K+L
	中缀表达式	-H/I*J+-K*+L
	后缀表达式	H-IJ*/K-L+*+

容易看出来，中缀表达式很容易存在歧义，这时我们是可以通过套括号的方式将其歧义消除的。

例如：

a+b-c	
	
((a)+(b))-(c))	((a)+((b)-(c)))

-遍历的代码实现<sup>[1]</sup>

前序遍历	<pre> template class &lt;T&gt; void PreOrder(BinaryTreeNode&lt;T&gt; *t) {     if(t)     {         Visit(t);         PreOrder(t-&gt;LeftChild);         PreOrder(t-&gt;RightChild);     } } </pre>
中序遍历	<pre> template class &lt;T&gt; void InOrder(BinaryTreeNode&lt;T&gt; *t) {     if(t)     {         InOrder(t-&gt;LeftChild);         Visit(t);         InOrder(t-&gt;RightChild);     } } </pre>
后序遍历	<pre> template class &lt;T&gt; void PostOrder(BinaryTreeNode&lt;T&gt; *t) {     if(t)     {         PostOrder(t-&gt;LeftChild);         PostOrder(t-&gt;RightChild);         Visit(t);     } } </pre>
逐层遍历 <sup>[2]</sup>	<pre> template class &lt;T&gt; void LevelOrder(BinaryTreeNode&lt;T&gt; *t) {     LinkQueue&lt;BinaryTreeNode&lt;T&gt;*&gt; Q;     while(t)     {         Visit(t);         if(t-&gt;LeftChild)Q.Add(t-&gt;LeftChild);         if(t-&gt;RightChild)Q.Add(t-&gt;RightChild);         try{Q.delete(t);t=Q.top();}         catch(OutOfBounds){return;}     } } </pre>

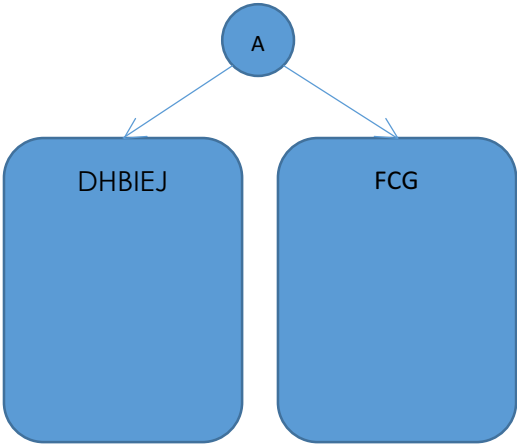
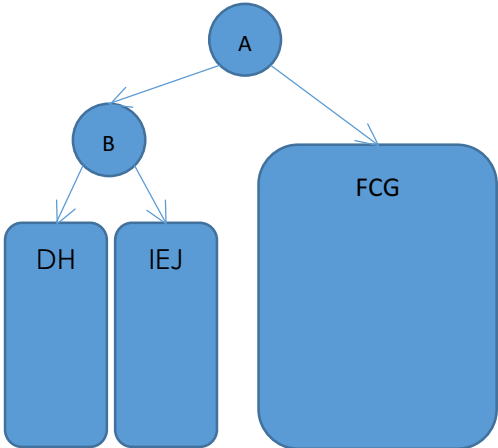
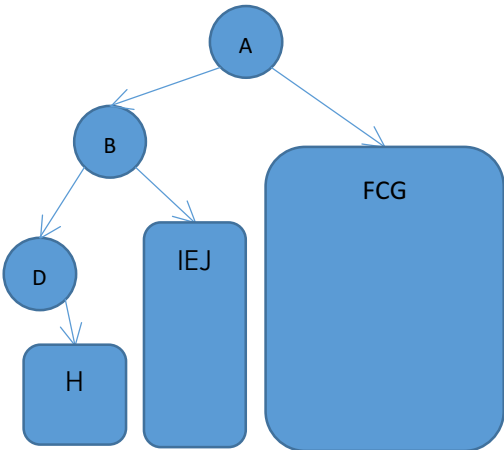


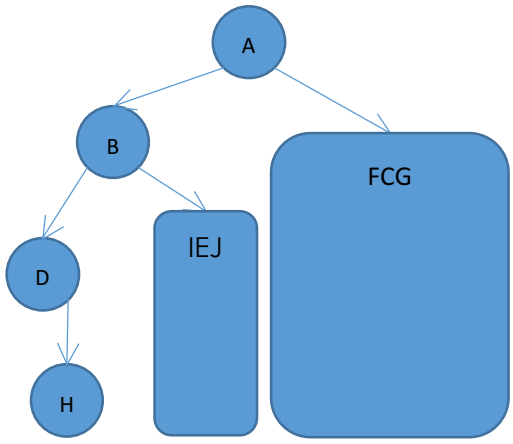
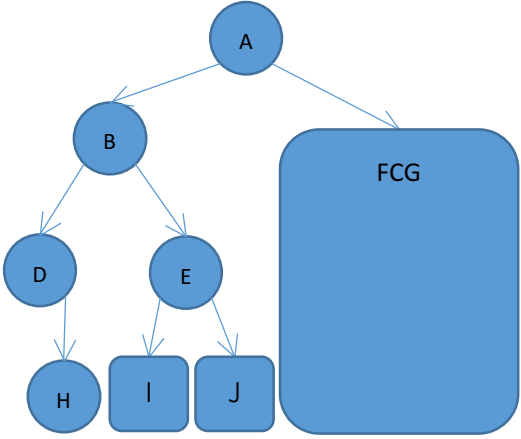
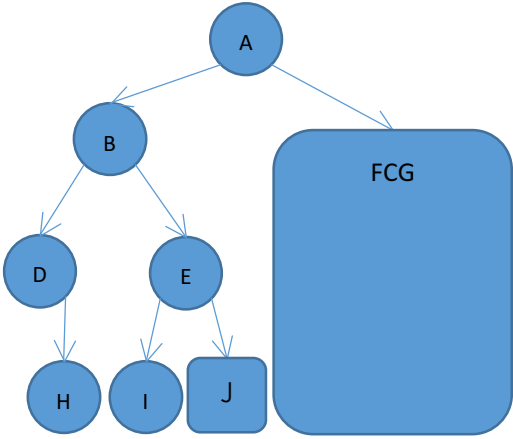
-复杂度计算

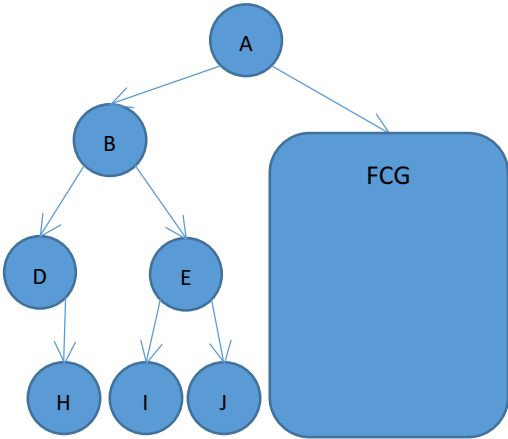
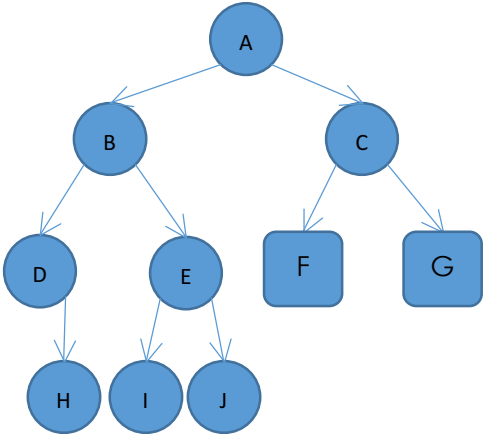
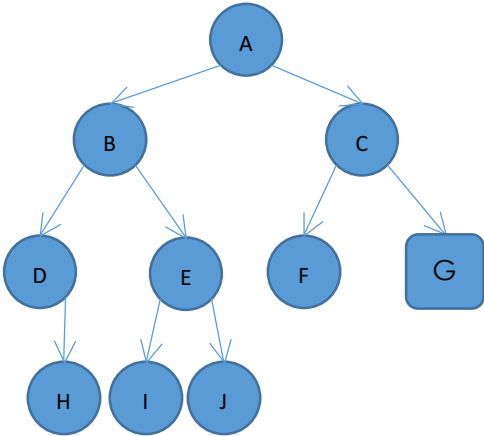
空间复杂度：O(n)

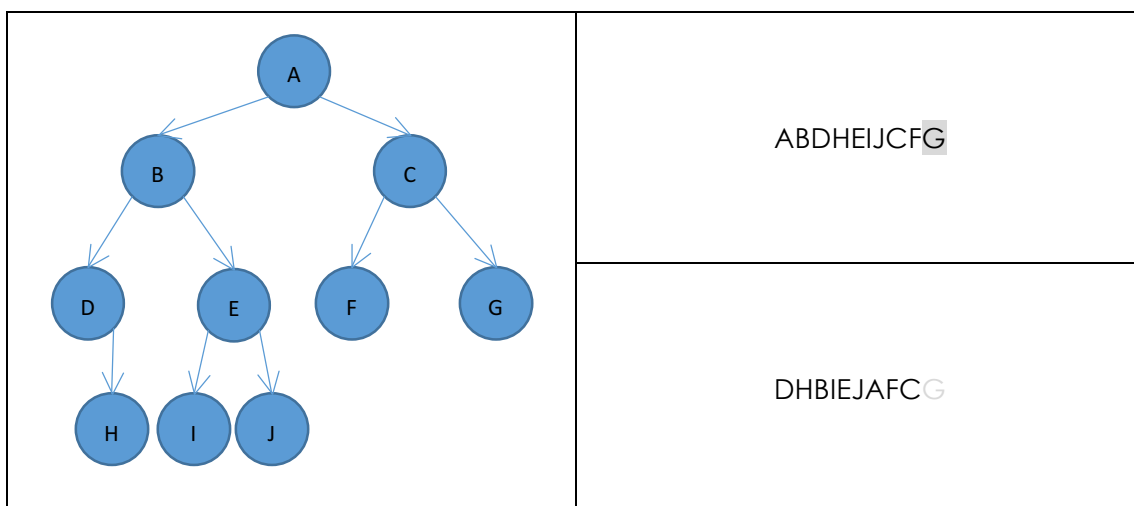
时间复杂度：O(n)

+根据前序遍历结果与中序遍历结果构造二叉树

前序遍历	中序遍历
ABDHEIJCFG	DHBIEJAFCG
	<div>ABDHEIJCFG</div> <div>DHBIEJFCG</div>
	<div>ABDHEIJCFG</div> <div>DHBIEJAFCG</div>
	<div>ABDHEIJCFG</div> <div>DHBIEJAFCG</div>

	ABDHEIJCFG
	DHBI EJAFCG
<p>图在下边→ ←</p> 	ABDHEIJCFG
	DHBI EJAFCG
	ABDHEIJCFG
	DHBEJ AFCG

	ABDHEIJCFG
	DHBIEJAF
<p>图也在下边← →</p> 	ABDHEIJCFG
	DHBIEJAF
	ABDHEIJCFG
	DHBIEJAF



### -抽象数据类 BinaryTree 的实现

操作如下：

Create(): 创建一个空的二叉树；

IsEmpty: 如果二叉树为空，则返回 true，否则返回 false

Root(x): 取根节点；如果操作失败，则返回 false，否则返回 true

MakeTree(root, left, right): 创建一个二叉树，root 作为根节点，left 作为左子树，right 作为右子树

BreakTree(root, left, right): 拆分二叉树

PreOrder: 前序遍历

InOrder: 中序遍历

PostOrder: 后序遍历

LevelOrder: 逐层遍历

### -应用

求二叉树高度思路：二叉树高度=max(左子树高度, 右子树高度)+1，依次为思路递归完成操作即可，访问节点操作改为求左右节点的最大值，然后+1。

统计节点思路：访问节点操作改为计数器+1。

附：抽象数据类 BinaryTree 实现代码[\[3\]](#)

```
template<class T>
class BinaryTree {
public :
    BinaryTree() {root=0;};
    ~BinaryTree() {};
    bool IsEmpty() const
    {return ((root) ? false : true);}
    bool Root(T& x) const;
    void MakeTree(const T& element,BinaryTree<T>&left,BinaryTree<T>& right);
    void BreakTree(T& element,BinaryTree<T>&left ,BinaryTree<T>& right);
    void PreOrder( void(*Visit) (BinaryTreeNode<T> *u) )
        {PreOrder(Visit,root);}
    void InOrder(void (*Visit) (BinaryTreeNode<T> *u))
        {InOrder(Visit,root);}
    void PostOrder(void(*Visit ) (BinaryTreeNode<T> *u));
        {Postorder (Visit, root); }
    void LevelOrder(void(*Visit) (BinaryTreeNode<T> *u));
private:
    BinaryTreeNode<T>*root; void PreOrder(void (*Visit) (BinaryTreeNode<T>
    *u),BinaryTreeNode<T> *t);
    void InOrder(void(*Visit) (BinaryTreeNode<T> *u), BinaryTreeNode<T> *t);
    void PostOrder(void(*Visit) (BinaryTreeNode<T> *u),BinaryTreeNode<T> *t);
}
```

```
template<class T>
bool BinaryTree<T>::Root(T& x) const
{
    if (root){
        x = root->data;
        return true;
    }
    else return false;
}
```

```
template<class T>
void BinaryTree < T > :: MakeTree(const T& element, BinaryTree<T>& left,
BinaryTree<T>& right)
{
    root = new BinaryTreeNode< T >(element, left.root, right.root);
    left.root = right.root = 0;
}
```

```
template<class T>
void BinaryTree < T >::BreakTree(T& element, BinaryTree<T>& left, BinaryTree<T>&
right)
```

```

{if (!root) throw BadInput();
element = root->data;
left.root = root->LeftChild;
right.root = root->RightChild;
delete root;
root = 0;
}

```

```

template<class T>
void
BinaryTree<T>::PreOrder(void(*Visit)(BinaryTreeNode<T>*u), BinaryTreeNode<T>*t)
{
    if (t) {
        Visit(t);
        PreOrder(Visit, t->LeftChild);
        PreOrder(Visit, t->RightChild);
    }
}

```

```

template<class T>
void BinaryTree<T>::LevelOrder(void(*Visit)(BinaryTreeNode<T>*u))
{
    LinkQueue<BinaryTreeNode<T>*> Q;
    BinaryTreeNode<T> *t;
    t=root;
    while(t){
        Visit(t);
        if (t->LeftChild) Q.Add(t->LeftChild);
        if (t->RightChild) Q.Add(t->RightChild);
        try{Q.Delete(t);}
        catch(OutOfBounds){return;}
    }
}

```

```

Public:
.....
int Height( ) const {return Height (root);}

private : //返回树*t 的高度
.....
int Height (BinaryTreeNode<T> *t) const;
.....
template <class T>
int  BinaryTree<T>::Height(BinaryTreeNode<T> *t) const
{
    if (!t) return 0;
}

```



```

int hl = Height(t->LeftChild);
int hr = Height(t->RightChild);
if (hl > hr)    return ++hl;
else    return ++hr;
}

```

### -信号放大器-问题描述

存在一树形结构流动着信号。从根节点开始，信号从节点向子节点流动。边上有父节点到子节点的信号衰减量。除根以外的节点允许设置信号放大器。

设计一个算法确定信号放大器的摆放位置，要求尽量少的使用信号放大器，且信号衰减不超过规定的容忍值。

### -思路

定义变量  $d[i]$  表示节点  $i$  与其父节点间的衰减量。

$D[i]$  表示从节点  $i$  到以  $i$  为根节点的子树的任一叶子的衰减量的最大值。

$D[i] = \max(D[j] + d[j])$ ,  $j$  为  $i$  的一个子节点

### -伪代码

```

D[i]=0;
for(j 遍历 i 的子节点)
{
    if(D[j]+d[j]>容忍值)
        j 放置放大器;
    else
        D[i]=max(D[i], D[j]+d[j]);
}

```

### -在线等价类

关于这部分的内容去我之前写的 贪心 部分看吧，在那最下边那一块儿有讲解的。

FIN.

Ref.

大体结构，采取例题参考孔兰菊老师的 PPT

内容参考《数据结构，算法与应用——C++语言描述》，Sartaj Sahni，王诗林等译，机械工业出版社

使用字体：微软雅黑(标题)，微软雅黑 Light，Century Gothic

P. s.

[0] 树:这一部分讲的是课本第 8 章到第 11 章的内容，所以会有很多字。

[1] 这里列举的代码为使用链表描述二叉树的情况。公式化描述很简单，改为访问数组下标，左儿子为访问  $2*i$ , 右儿子访问  $2*i+1$  即可。

[2] 因为层序遍历本质为广度优先搜索，广度优先搜索依赖于队列，所以层序遍历会涉及到队列的相关知识。

[3] 代码源自孔兰菊老师的 PPT。

——吉鹏智库 张晓敏@2016/12/31 署名-非商业性使用-相同方式共享

# 第九章 优先队列

+定义

**优先队列**：0 个或多个元素的集合，每个元素具有一个值和一个优先值。与普通的队列不同的是，优先队列是按照优先级进行出队操作的，而不是先入先出(FIFO-First In First Out)原则。不同元素允许具有相同优先级。

优先队列具有的操作有：查找一个元素，插入一个新元素，删除一个元素。

**最小/最大优先原则** (Min/Max Priority Queue)：查找或删除元素时查找或删除最小/最大优先级的元素。

描述优先队列的方式有：线性表，堆，左高树。

+线性表描述优先队列

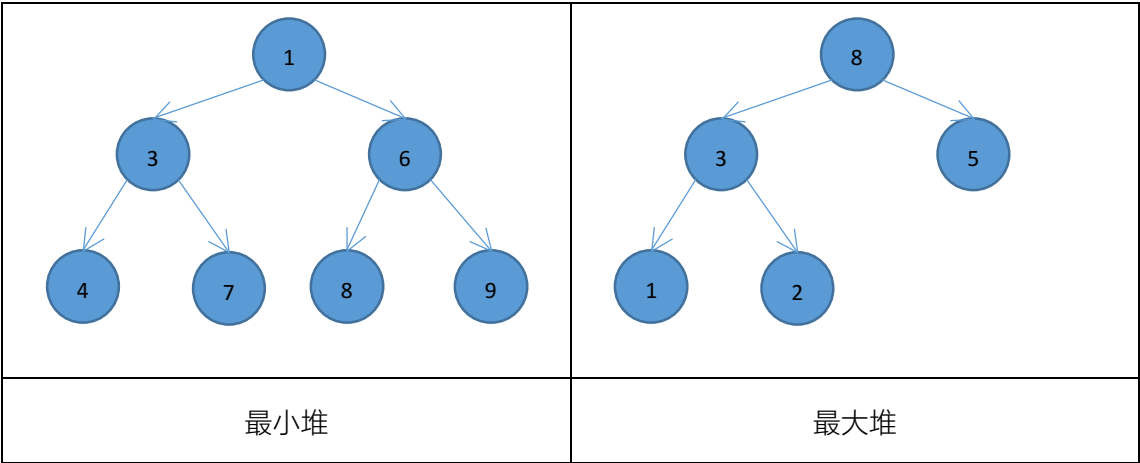
分无序线性表与有序线性表两种情况，公式化描述与链表描述两种方式。

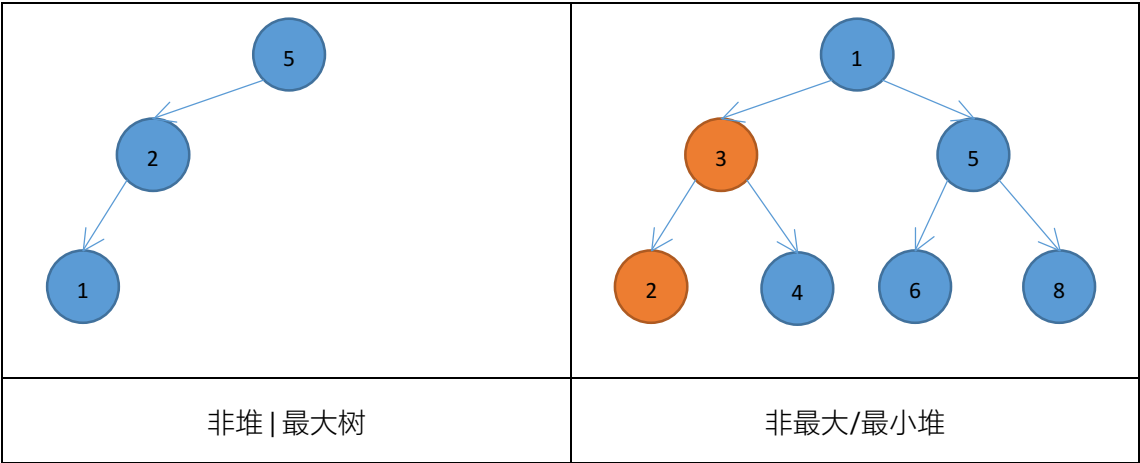
时间复杂度		无序线性表	有序线性表
公式化描述	插入元素	$O(1)$ //插入于尾	$O(n)$
	删除元素	$O(n)$ //查找优先级	$O(1)$
链表描述	插入元素	$O(1)$ //插入于首	$O(n)$
	删除元素	$O(n)$	$O(1)$

+堆描述优先队列 • 定义

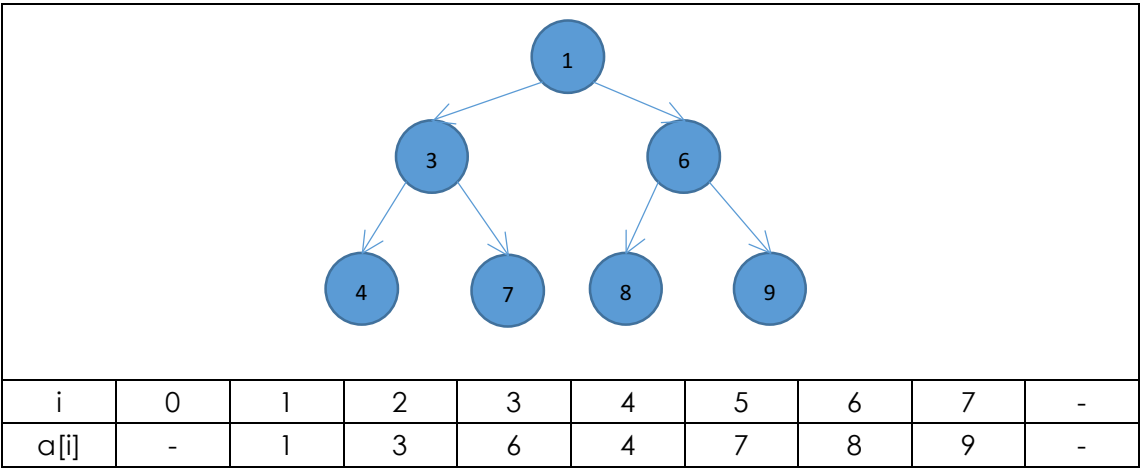
**最大/小树**：每个节点的值都大于/小于其子节点(如果存在)的树。子节点个数可以大于2。

**最大/小堆**：最大/小的完全二叉树。





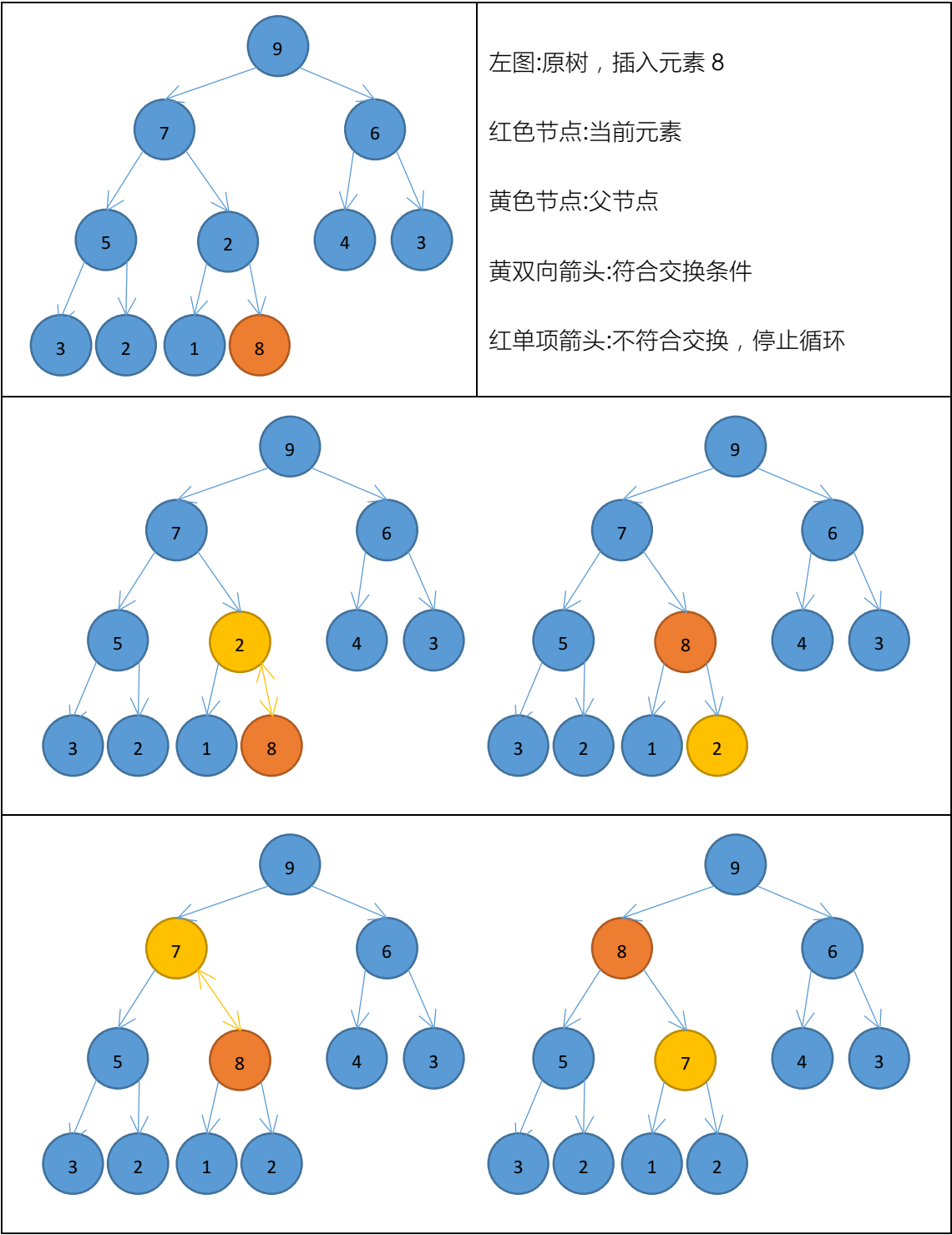
因为是完全二叉树，因此可以使用一维数组描述堆（公式化描述）。

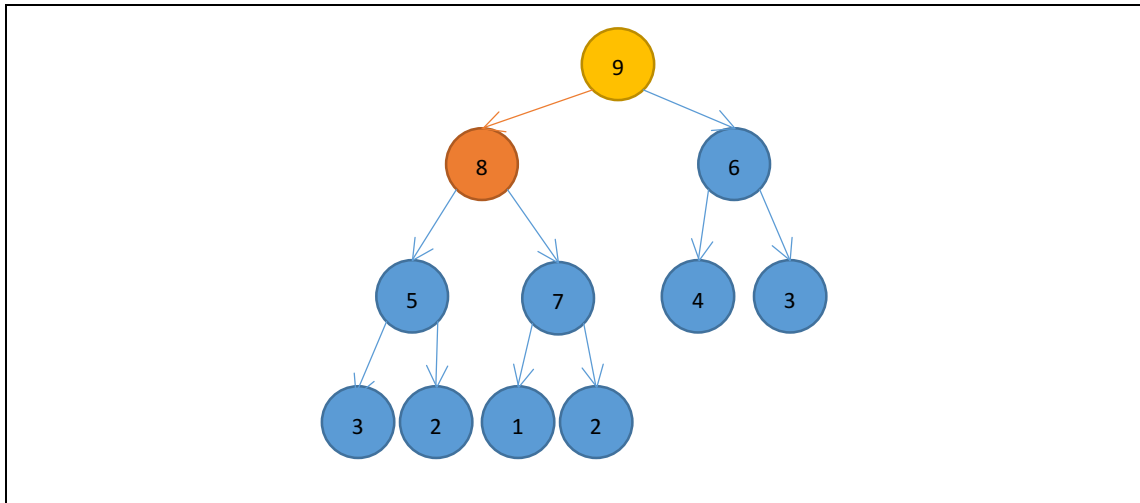


**+最大堆相关操作·插入·伪代码**

```
n++;  
a[n]=新元素;  
int i=n;  
while(i>1&&a[i/2]<a[i]){  
    swap(a[i/2],a[i]);  
    i/=2;  
}
```

**+示例**





### -时间复杂度

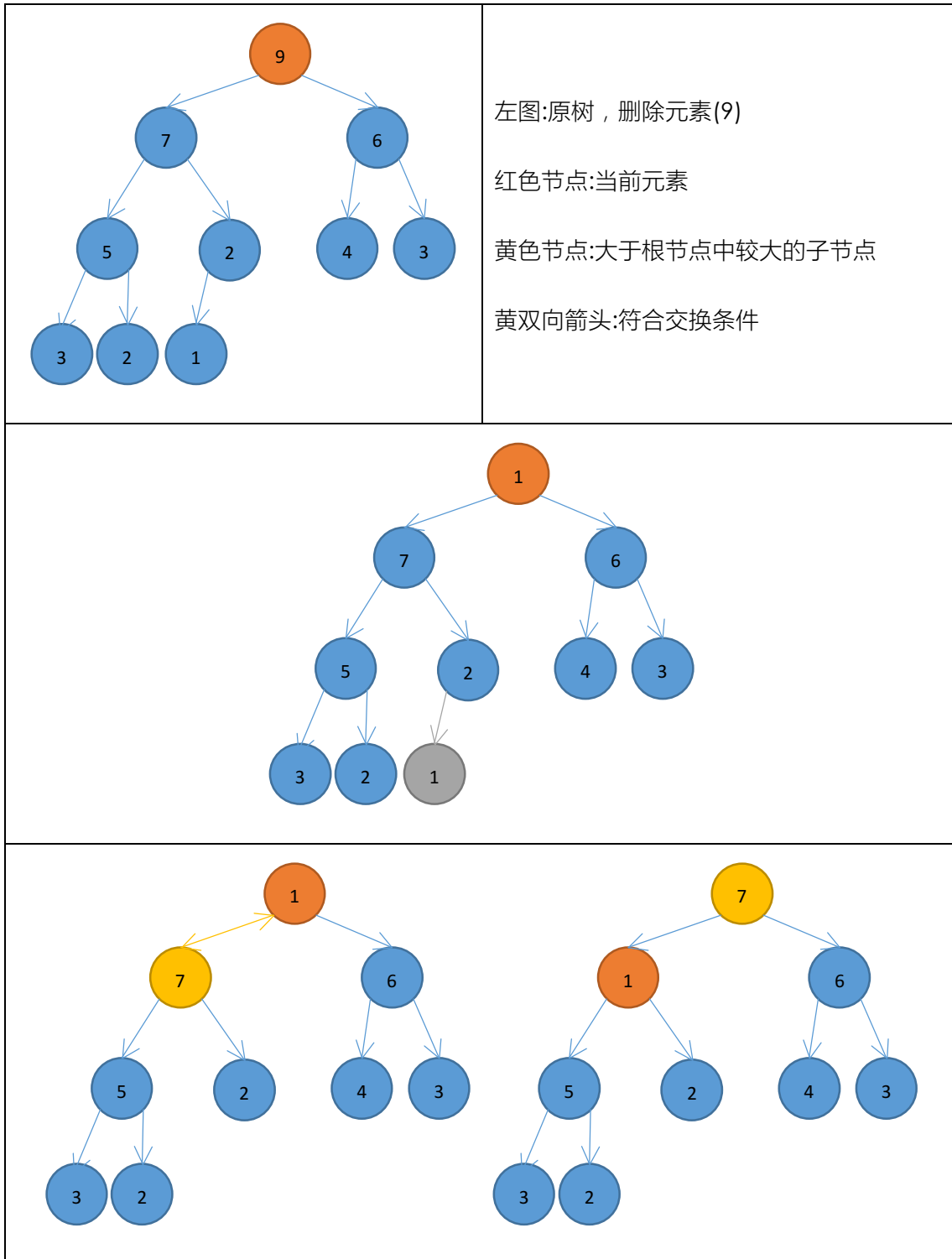
$$O(\text{height}) = O(\log n)$$

### +删除·伪代码 根本就是正常代码嘛

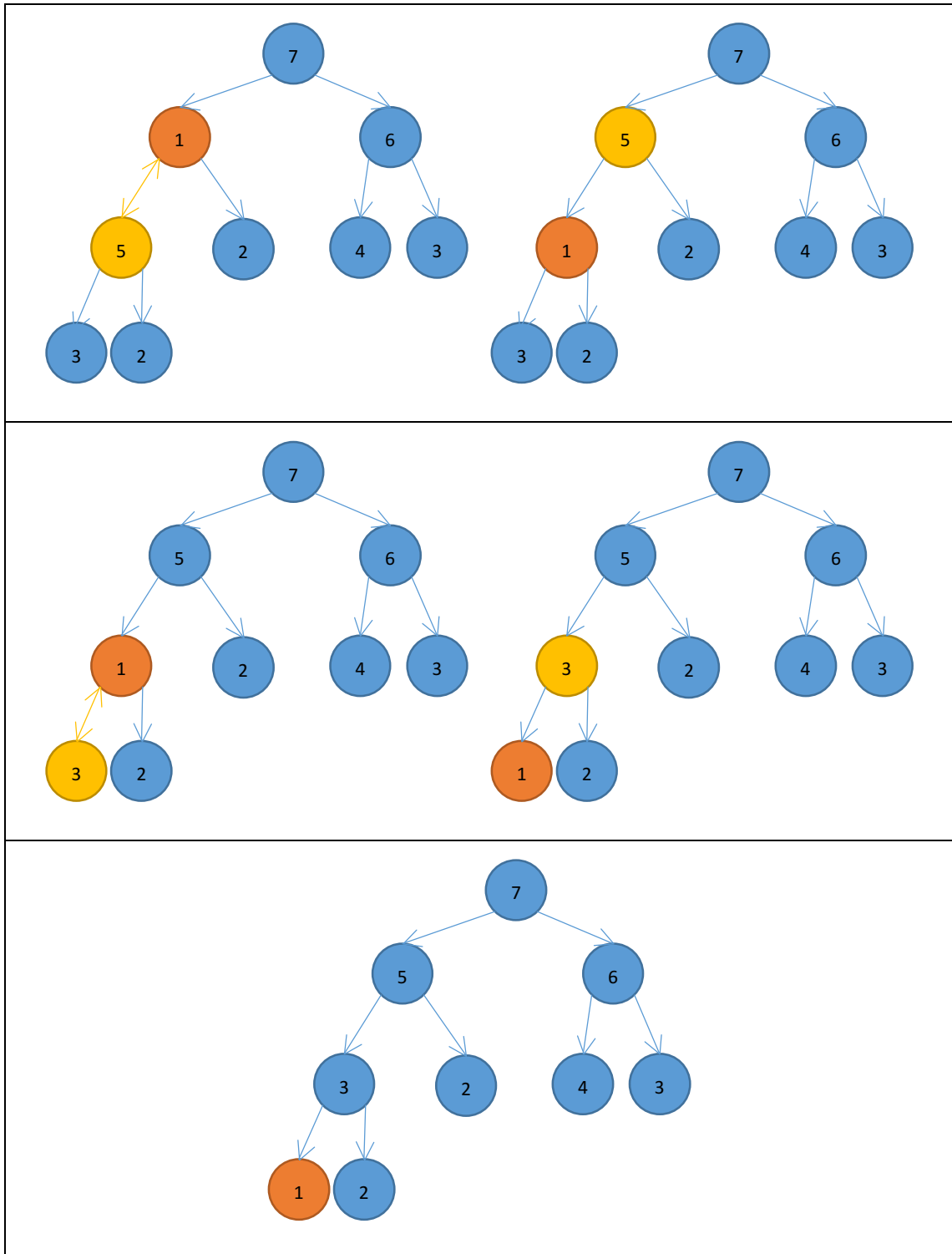
```

a[1]=a[n];
n--;
int i=1;
while(a[i*2]>a[i] || a[i*2+1]>a[i])
    if(a[i*2]>a[i*2+1])
    {
        Swap(a[i*2],a[i]);
        i=i*2;
    }
    else
    {
        Swap(a[i*2+1],a[i]);
        i=i*2+1;
    }
  
```

### +示例







### -时间复杂度

与插入的时间复杂度相同，为  $O(\log n)$

### +初始化堆

两种方法：

1. 向空树逐渐添加元素从而做到初始化： $O(n \log n)$
2. 必要时对树进行调整： $O(n)$

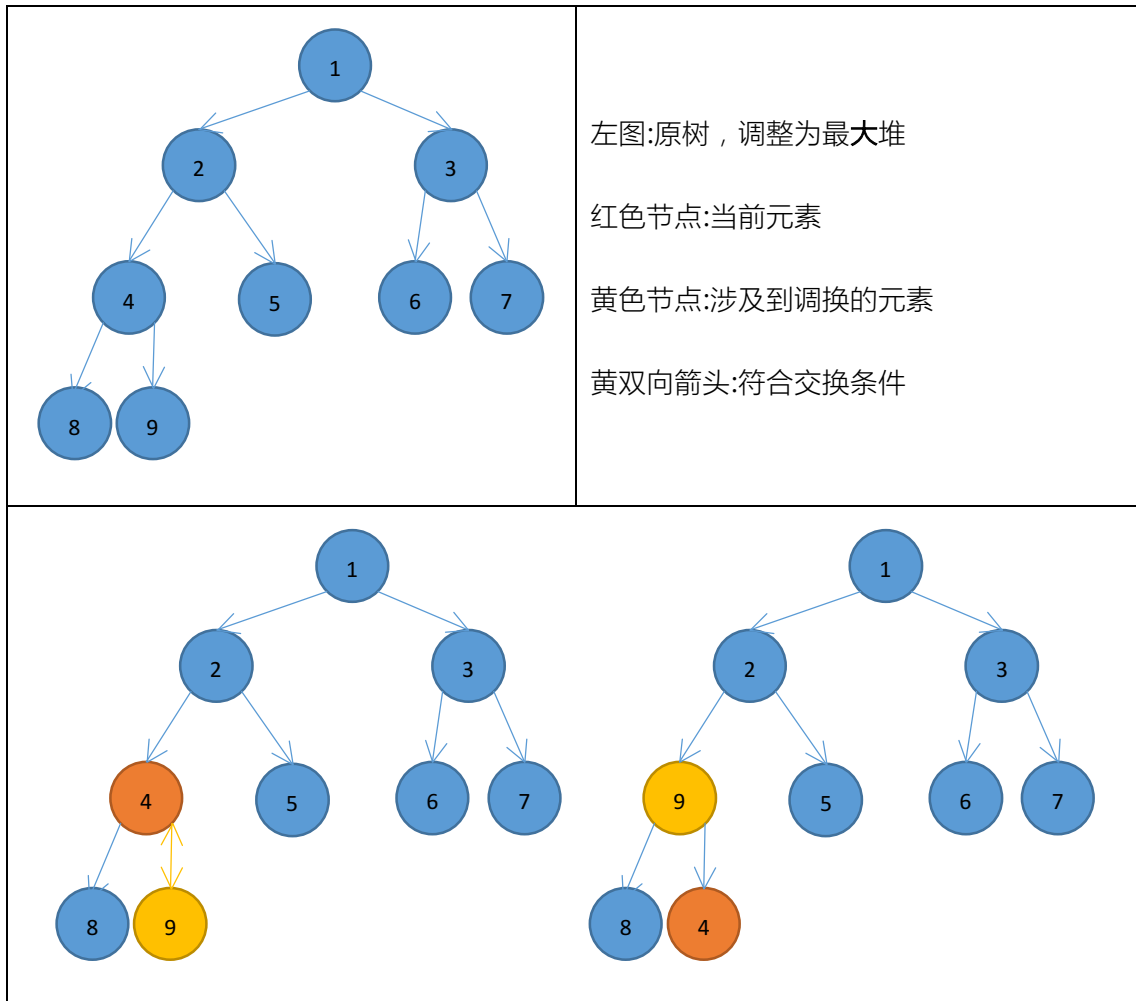
### -方法 2 思路

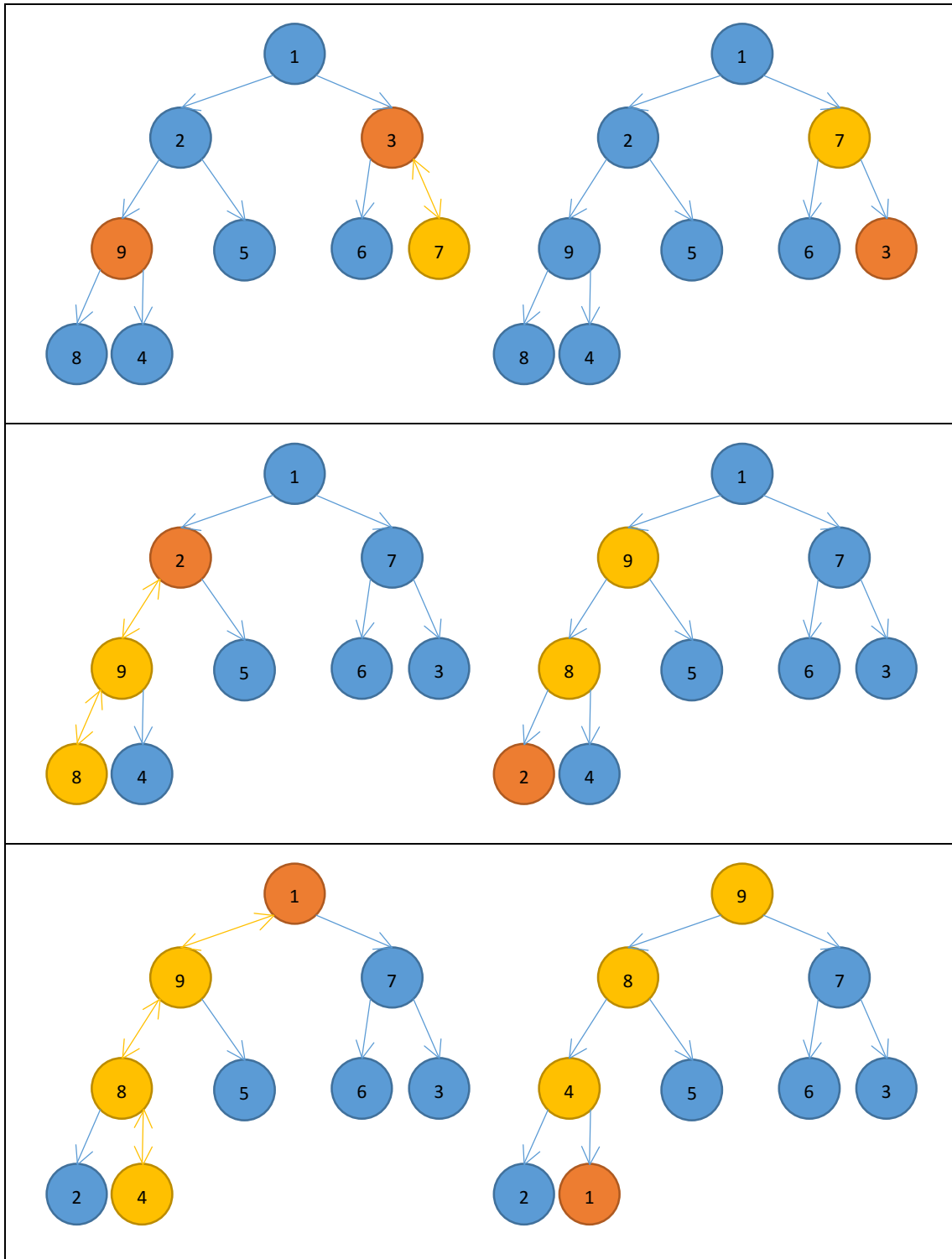
从  $n/2$  个元素开始(从这个元素向前的元素都有子节点)倒序处理到第一个元素, 对其中的第  $i$  个元素时, 执行以下操作:

如果以第  $i$  个元素为根节点的子树已经是最大堆, 则不处理;

否则, 调整其子树使其成为堆。

### -示例





-时间复杂度

$O(n)$ <sup>[2]</sup>

## +左高树 • 概念

~~wee~~ 当时学这玩意儿来着??

左高树的时间空间效率都很高，使用链表结构。

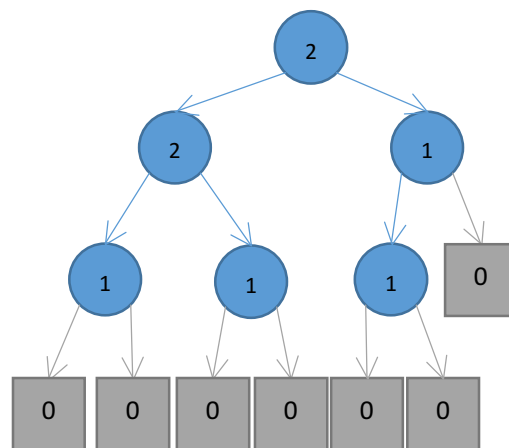
它适合实现优先队列的某些应用，尤其是关于合并两个优先队列或者多个长度不同的队列的操作。它分为**高度优先左高树** (HBLT-Height-biased leftist tree) 与**重量优先左高树** (WBLT-Weight-biased leftist tree)。

外部节点：代替空子树的节点。

内部节点：具有非空子树的节点。

扩充节点：增加外部节点的二叉树。

## -高度优先左高树 • 定义



定义  $s()$  表示该节点到达子树的所有路径中最短的一条路径的长度。

外部节点的  $s()$  值为 0，内部节点  $s() = \min(s(L), s(R)) + 1$

对于一棵树的任何一个内部节点，均符合  $s(L) \geq s(R)$ ，则该树为高度优先左高树。

## -性质

若  $x$  为 HBLT 的一内部节点，则：

1. 以  $x$  为根的子树的节点数目至少  $2^{s(x)} - 1$
2. 若以  $x$  为根的子树有  $m$  个节点，则  $s(x)$  最多为  $\log_2(m+1)$
3. 通过最右路径（即，此路径是从  $x$  开始沿右孩子移动）从  $x$  到达外部节点的路径长度为  $s(x)$ 。

## -重量优先左高树

定义  $w(x)$  为  $x$  为根的子树的节点数。

对于一棵树的任何一个内部节点，均符合  $w(L) \geq w(R)$ ，则该树为重量优先左高树。

最大/小 HBLT/WBLT 即为符合最大/最小树规则的 HBLT/WBLT。

## +最大 HBLT 操作 • 合并<sup>[3]</sup>

合并的实现利用递归思想，假设  $A, B$  为需要合并的最大 HBLT,  $A$  的左子树为  $A_1$ , 右子树为  $A_r$ 。假设  $\text{Data}[A] > \text{Data}[B]$ 。

特殊条件：若其中一个为空，则直接返回另一棵树。

否则进行操作：

将 A 的根作为新树的根（假设为 R）。

将 Ar 与 B 进行合并（规则同本规则），假设合并得到的结果为树 C。

查看  $s[A1]$  与  $s[C]$  的大小关系，较大的作为根 R 的左子树，较小的作为根 R 的右子树。

则这棵以 R 为根，A1 与 C 为左右子树的树为合并后的最大 HBLT。

### -初始化最大 HBLT

A. 可以向一棵空最大 HBLT 中逐渐与每一个元素进行合并。

时间复杂度： $O(n \log n)$

B. 可以将每个元素构成一棵最大 HBLT 树，然后加入 FIFO 队列。从队列首取出 2 个最大 HBLT 合并之，并将其加入队尾，直至队内只剩一棵 HBLT 树。

时间复杂度： $O(n)$

### +堆的应用 • 堆排序

*其实提出来堆这个概念就很容易想到它可以用来排序了...*

通过对数据进行初始化最小堆/最大堆操作，然后依次取出&删除第一个元素即可完成排序。

初始化复杂度： $O(n)$

取出总复杂度： $O(n \log n)$

堆排序特别适合需要实时维护的排序，比如一言不合就加入数据啥的这种。

### +霍夫曼编码 • 定义

一种文本压缩算法。利用扩充二叉树实行的可变长编码。

外部节点表示被编码字符，从根到外部节点的路径表示编码。

一般规定向左儿子移动时取 0，向右儿子移动时取 1。

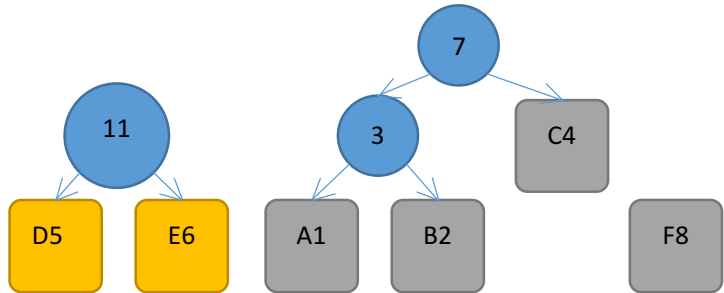
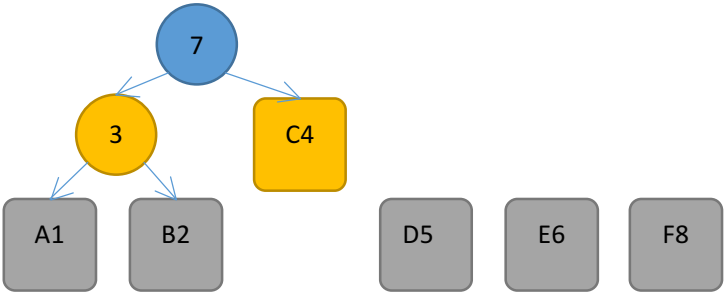
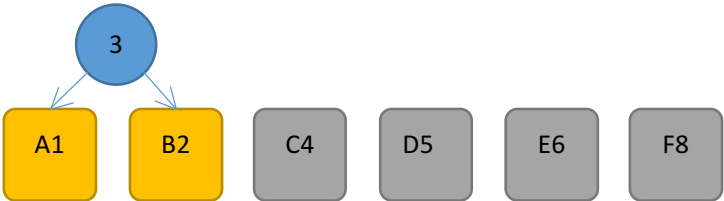
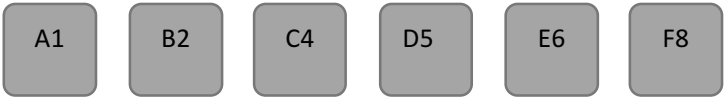
按照霍夫曼编码得到的编码具有最小加权外部路径长度的二叉树。

### -构造霍夫曼树

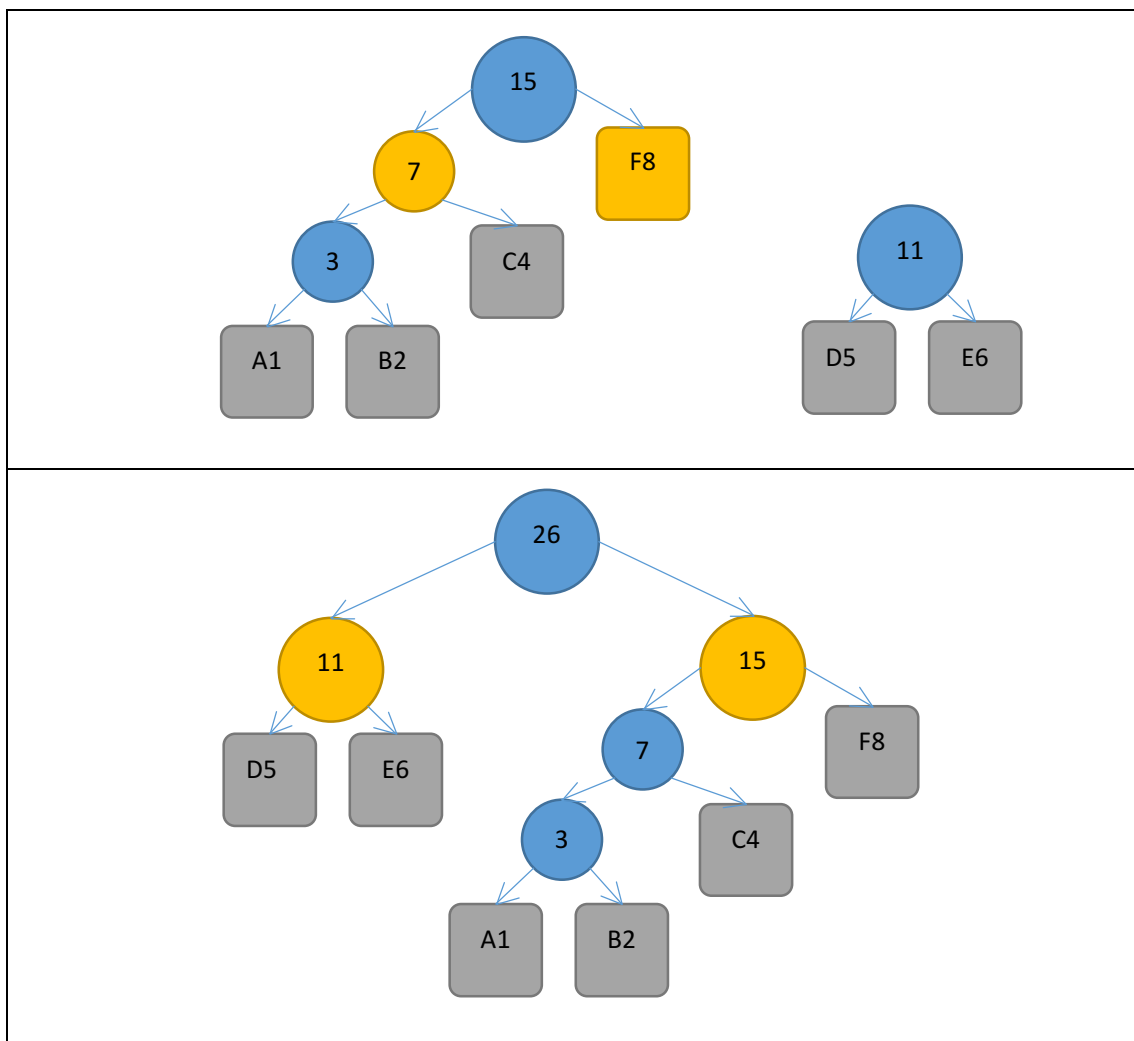
初始化：初始化二叉树集合，每棵二叉树只有一个外部节点各自表示一个不同的字符。

重复进行操作：从二叉树集合中选出权重最小的 2 棵二叉树，合并为一棵新的二叉树（新建一个根节点，将两棵二叉树分别作为根节点的左右儿子），新树权重为两子树之和。将新树加入集合。

结束条件：集合里只剩一棵树。







FIN.

Ref.

大体结构，采取例题参考孔兰菊老师的 PPT

内容参考《数据结构，算法与应用——C++语言描述》，Sartaj Sahni，王诗林等译，机械工业出版社

使用字体:微软雅黑(标题)，微软雅黑 Light，Century Gothic

P. s.

[1]方便起见，时间复杂度的  $O(\log_2 n)$  统一简写为  $O(\log n)$ ，省略的底数一般均为 2。

[2]计算过程：

堆的高度 =  $h$ ；在  $j$  层节点的数目  $\leq 2^{j-1}$ 。以  $j$  层节点为根的子树的高度 =  $h-j+1$

调整（或重构）以  $j$  层节点为根的子树：  $O(h-j+1)$ 。

调整（或重构）所有以  $j$  层节点为根的子树  $\leq 2^{j-1}(h-j+1) = t(j)$ 。

总的时间：  $t(1) + t(2) + \dots + t(h-1) = O(2^h) = O(n)$ 。

[3]最大 HBLT 插入元素可视为原树与一个节点组成的树进行合并。

最大 HBLT 删除元素可视为该树的左右子树进行合并。

初始化最大 HBLT 可视为将  $n$  个元素插入到最大 HBLT 中。

因此省略插入、删除操作。

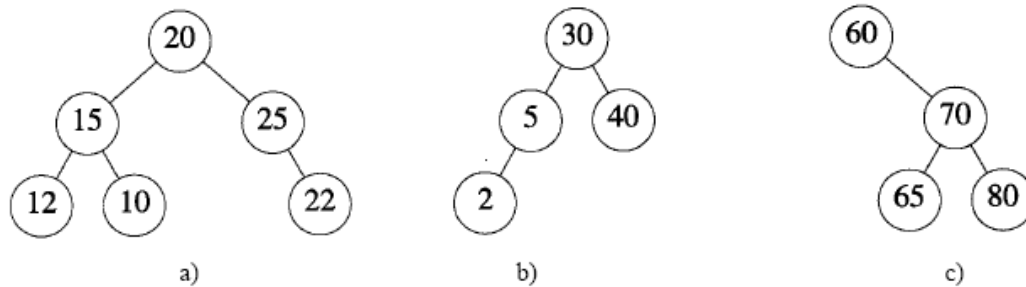
## 第十一章 搜索树

二叉搜索树：

特征（定义）：二叉搜索树是一棵可能为空的二叉树，一棵非空的二叉搜索树满足以下特征：

- 1) 每个元素有一个关键值，所有的关键值都是唯一的。
- 2) 根节点左子树的关键值（如果有的话）小于根节点的关键值。
- 3) 根节点右子树的关键值（如果有的话）大于根节点的关键值。
- 4) 根节点的左右子树也都是二叉搜索树

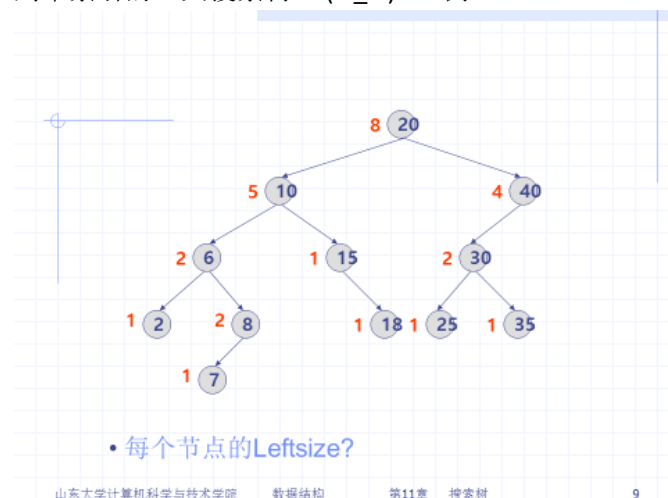
例：



高度优先左高树是左子树比右子树大，注意区别

PS：a）不是二叉搜索树

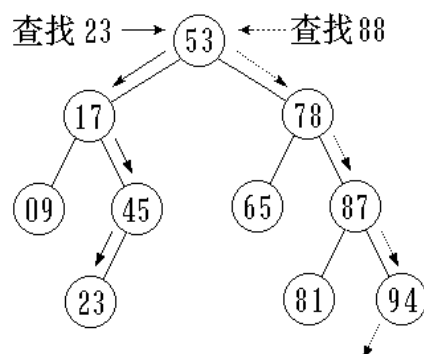
在二叉搜索树的基础上给每个节点加上索引（每个节点的左子树的节点数目 leftSize），得到带索引的二叉搜索树  $O(n \log n)$ ，例：



山东大学计算机科学与技术学院    数据结构    第11章    搜索树    9

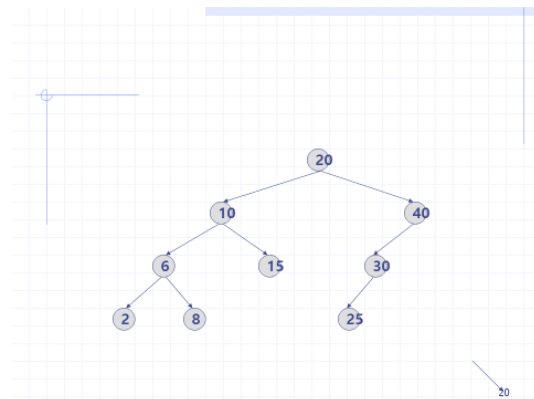
二叉搜索树的操作：

搜索：



过程：搜索 23 时，从根节点开始比较， $23 < 53$ ，再从 53 的左子树搜索，左子树的根节点为 27，小于 23，再找 17 的右子树，根节点 45 大于 23，找左子树，左子树的根为 23，搜索到了 23。

## 插入



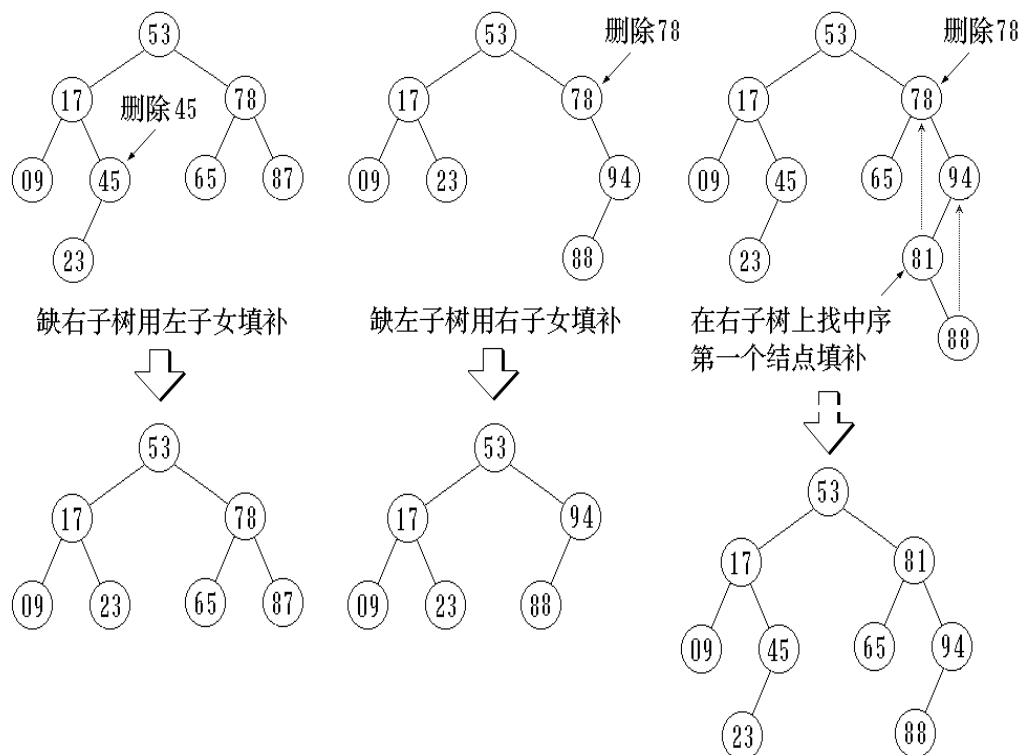
插入 35，先搜索树中是否存在 35，若存在，则不再插入。如不存在：从根节点开始查找 35 的位置，原理与搜索相同，找到 35 的位置是节点 30 的右子树，插入。

## 删除

考虑包含被删除元素的节点  $p$  的三种情况：

- 1)  $p$  是树叶：直接删除即可。
- 2)  $p$  只有一个非空子树：将  $p$  的子树替代  $p$  原来的位置
- 3)  $p$  有两个非空子树：在  $p$  的左子树找最大值或右子树的最小值替代  $p$  的位置。

汇总：



PS：带索引的二叉搜索树的搜索：

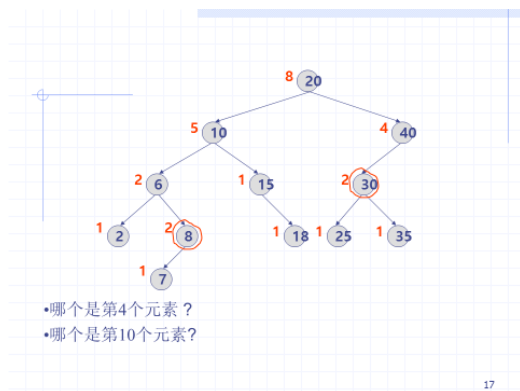
若要搜索指定索引号  $k$  的节点：则从根节点开始搜索，假设搜索到的当前根节点为  $x$

如果  $k=x.LeftSize$ , 第  $k$  个元素是  $x$

如果  $k < x.LeftSize$ , 第  $k$  个元素是  $x$  的左子树的第  $k$  个元素.

如果  $k > x.LeftSize$ , 第  $k$  个元素是  $x$  的右子树的第  $(k-x.LeftSize)$  个元素.

例：



若要找第四个节点：从根节点开始， $8 > 4$ ，进入左子树搜索， $5 > 4$ ，再次进入左子树， $2 < 4$ ，进入右子树搜索，注意此时要找的索引号变为  $(4-2)$ ，右子树的根节点索引为 2，找到~，第四个节点即为关键值为 8 的节点。

AVL 树：

特征：

空二叉树是 AVL 树。

如果  $T$  是一棵非空的二叉树， $T_L$  和  $T_R$  分别是其左子树右子树，当  $T$  满足以下条件时， $T$  是一棵 AVL 树： $T_L$  和  $T_R$  是 AVL 树， $|h_L - h_R| \leq 1$ ， $h_L$  和  $h_R$  分别是左子树和右子树的高度。

PS: AVL 树的每个节点都有一个平衡因子 bf: 该节点的左子树的高度-右子树的高度。由 AVL 树的特点，平衡因子  $|bf| \leq 1$ ;

**AVL 搜索树**（平衡二叉搜索树/平衡二叉排序树）：既是二叉搜索树，也是 AVL 树。

**带索引的 AVL 搜索树**既是带索引的二叉搜索树，也是 AVL 树。

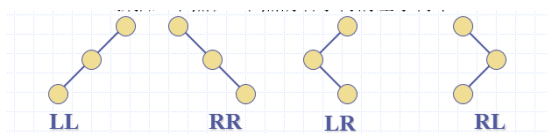
## AVL 树的操作：

**插入：**如果用二叉搜索树的插入方法将元素插入到 AVL 搜索树中，得到的树可能不符合左右子树高度差不超过 1 的限制，不再是 AVL 树(可能不再是平衡树)。

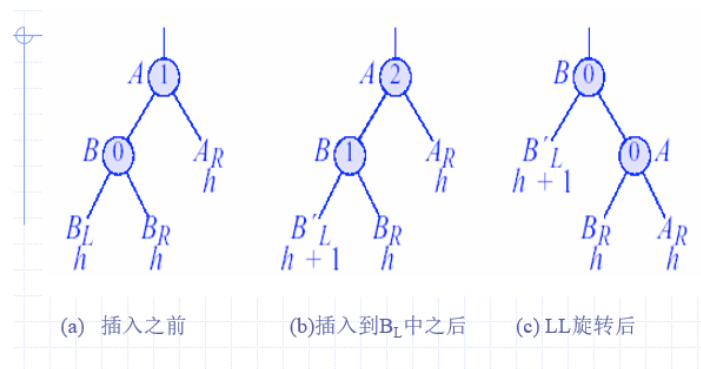
在插入后, A 的平衡因子是 -2 或 2，当节点 A 已经被确定时，A 的不平衡性：

1. **LL**:新插入节点在 A 节点的左子树的左子树中
2. **RR**:新插入节点在 A 节点的右子树的右子树中
3. **LR**:新插入节点在 A 节点的左子树的右子树中
4. **RL**:新插入节点在 A 节点的右子树的左子树中

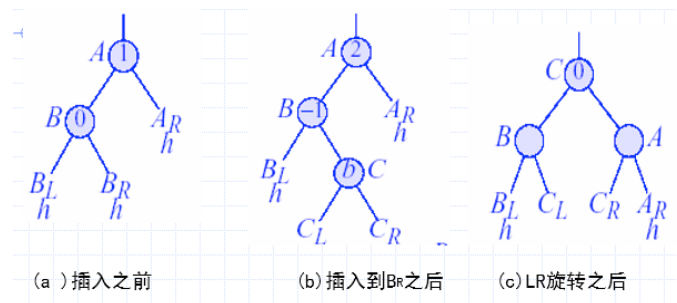
图解：



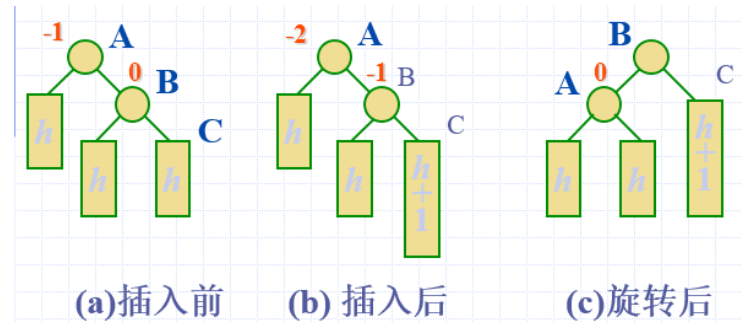
LL 不平衡：



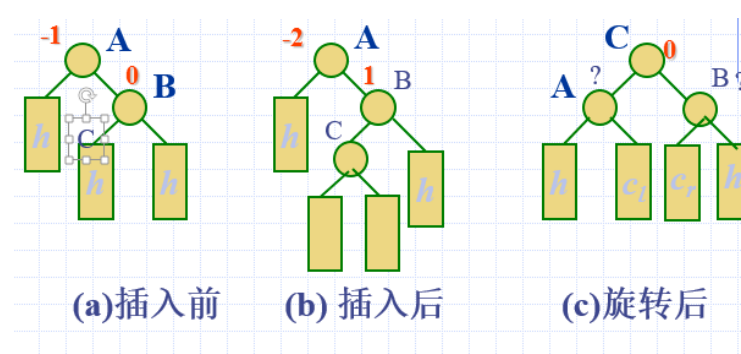
LR 不平衡:



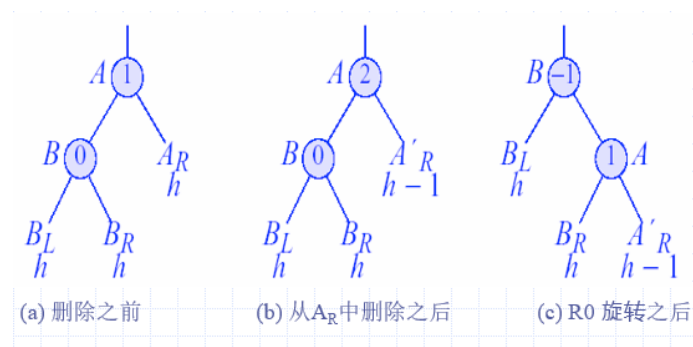
RR 不平衡:



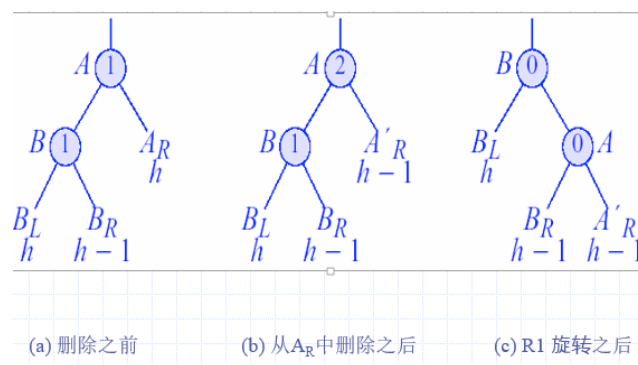
RL 不平衡:



**删除:** 删除可能导致某些节点的平衡因子变为-2 或 2，不再是 AVL 树情况一:



情况二:

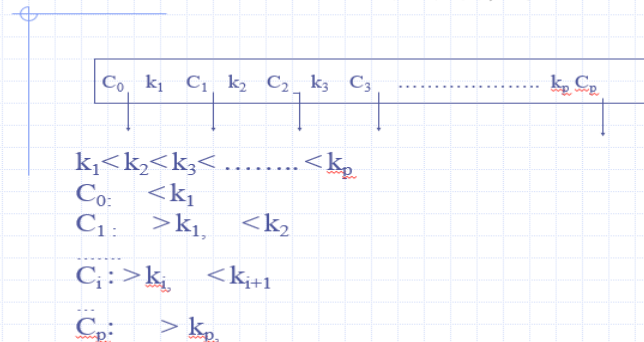


M 叉搜索树:

特征: m 叉搜索树可以是一棵空树, 如果非空, 它必须满足以下特征:

- 1) 在相应的扩充搜索树中(用外部节点替换零指针), 每个内部节点最多可以有  $m$  个子节点及  $1 \sim m-1$  个元素(外部节点不含元素和子女)。
- 2) 每个含  $p$  个元素的节点, 有  $p+1$  个子节点

3). 考察含  $p$  个元素的任意节点。设  $k_1, k_2, k_3, \dots, k_p$  是这些元素的关键值。这些元素顺序排列, 即:



M 叉搜索树的搜索:

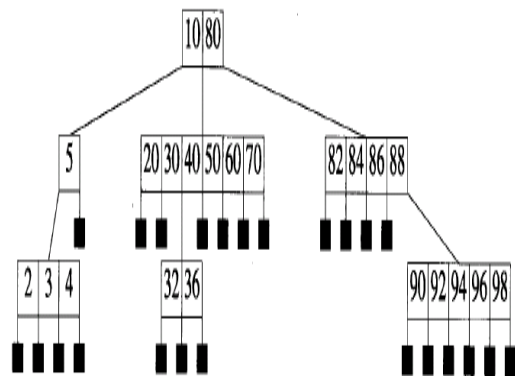


图11-24 七叉树

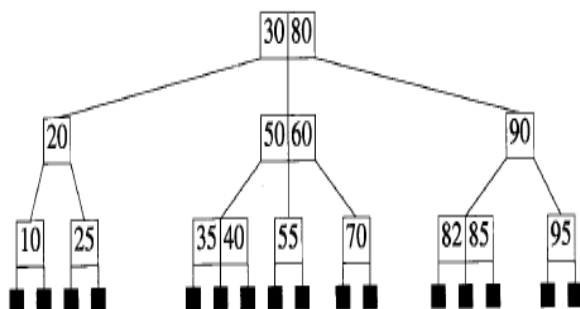
M 序 B-树:

特征: m 序 B-树(B-Trees of Order m) 是一棵 m 叉搜索树, 如果 B-树非空, 那么相应的扩充树满足下列特征:

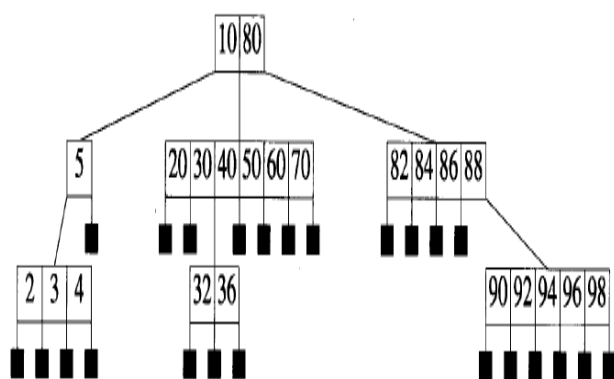
- 1) 根节点至少有 2 个孩子。
- 2) 除了根节点以外, 所有内部节点至少有  $\lceil m/2 \rceil$  个孩子。

3)所有外部节点位于同一层上。

例：3 序 B-树：三序 B-树的内部节点既可以有 2 个也可以有 3 个孩子，因此也把三序 B-树称作 2-3 树，如下图：



M 叉搜索树的搜索：



若要搜索 32，从根节点开始，比较 32 和 10,80 的大小关系，故延中间的指针往下找，再比较 32，与 40 所在节点的所有关键值的大小，延 30 与 40 之间的指针往下找，从而找到 32

图11-24 七叉树

M 叉搜索树的删除：以上图为例，若：

- 1) 删除类似 20 这样的节点：直接删除该节点即可。
- 2) 删除类似 5 这样的节点，需要从它的子女中找一个，左子树的最大值或右子树的最小值皆可。

M 叉搜索树的插入：以图 11-24 为例：

- 1) 若插入 31，类似于搜索的步骤，在 32 和 36 所在节点处搜索失败，插入该节点中即可。
- 2) 若插入 65，由于 60,70 所在节点已满。故 65 放入新节点中，作为该节点的第六个孩子

M 叉搜索树的高度：

B-树的高度：

定理 11-3：

设 T 是一棵高度为 h 的 m 序 B-树， $d = \lceil m/2 \rceil$ ，且 n 是 T 中的元素个数，则：

$$(a) \quad 2d^{h-1} \leq n \leq m^h - 1$$

$$(b) \quad \log_m(n+1) \leq h \leq \log_d((n+1)/2) + 1$$

**PS：**实际上，B-树的序取决于磁盘块的大小和单个元素的大小。

B-树的搜索：同 m 叉搜索树。

**B-树的插入：**

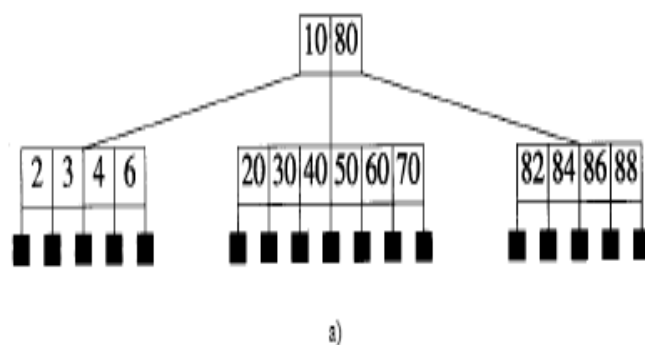
首先要检查具有相同关键值的元素是否存在，如果找到了这个元素，那么插入失败，因为不允许重复值存在。

当搜索不成功时，便可以将元素插入到搜索路径中所遇到的最后一个内部节点处：

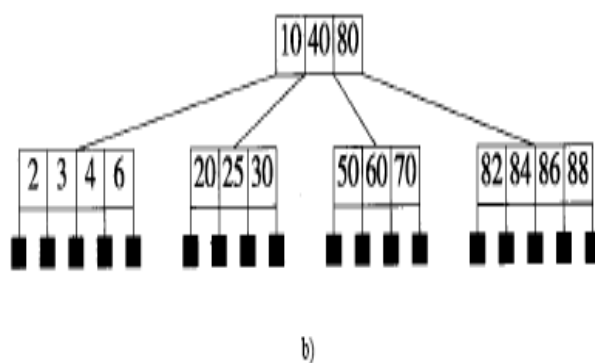
若该节点已经饱和，当新元素需要插入到饱和节点中时，饱和节点需要被分开。

例：插入 25：

插入前



插入后：



插入 25，而应该插入的节点已经饱和，所以要将饱和的节点分开，并且保证分开后的两个节点都满足 m 序 B-树的条件。

PS:

需要从磁盘中得到根节点及其中间孩子

写回分开的两个节点

写回修改后的根节点

磁盘访问次数一共是 5 次

### B-树的删除：

删除分为两种情况：

1. 被删除元素位于其孩子均为外部节点的节点中(即元素在树叶中)。
2. 被删除元素在非树叶节点中。既可以用左相邻子树中的最大元素，也可以用右相邻子树中的最小元素来替换被删除元素，这样 2 就转化为 1
  - A) 从一个包含多于最少数目元素 (如果树叶同时是根节点，那么最少数目元素是 1，如果不是根节点，则为  $\lceil m/2 \rceil - 1$ ) 的树叶中删除一个元素，只需要将修改后的节点写回。
  - B) 当被删除元素在一个非根节点中且该节点中的元素数量为最小值时
 

如果它的最相邻的兄弟(最相邻的左或右兄弟)有多于最少数目元素，用其最相邻的左或右兄弟中的元素来替换它。

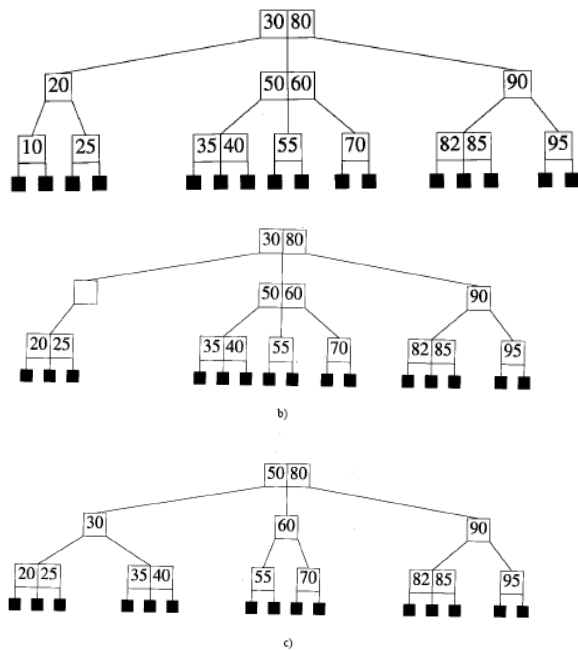
PS: 为保持低次数的磁盘访问，只检查最相邻兄弟之中的一个。

- C) 当被删除元素在一个非根节点中且该节点中的元素数量为最小值时，
 

若它的最相邻的兄弟(最相邻的左或右兄弟)有最少数目元素，将两个兄弟与父节点中介于两个兄弟之间的元素合并成一个节点。

对于 C 情况，给出例子：删除图中的 25：





要删除 25, 因为 25 所在节点元素数为最少数目, 而且它的兄弟也为最少数目, 故与父节点合并, 得到图 2, 原来父节点元素少于最少数目, 而兄弟节点可以借, 故通过父节点的父节点 (此处为根节点) 将兄弟节点的元素借给父节点。得到图 3

PS: 磁盘访问次数是 3 (到达包含被删除元素的树叶) + 2 (读第二和三层的最相邻右兄弟节点) + 4 (将第一, 二和三层的 4 个修改后的节点写回磁盘) 因此总的磁盘访问次数是 9 次

PS: 最坏情况下, 这种过程会一直回溯到根节点。当根节点缺少一个元素时, 它变成空节点, 将被抛弃, 树的高度减 1。

补充:

## 磁盘访问的总次数

□ 对于高度为  $h$  的 B-树的删除操作的最坏情况:

- 当合并发生在  $h, h-1, \dots, 3$  层,
- 2 层时, 从最相邻兄弟中获取一个元素。

□ 最坏情况下磁盘访问次数是  $3h$  :

- 找到包含被删除元素的节点:  $h$  次读访问
- 获取第 2 至  $h$  层的最相邻兄弟:  $h-1$  次读访问
- 在第 3 至  $h$  层的合并:  $h-2$  次写访问
- 对修改过的根节点和第 2 层的两个节点: 3 次写访问。

## 第十二章 图

### 一、定义及相关的概念

1、图: 图是一组顶点和边的集合, 图  $G = (E, V)$ ,  $E$  表示边的集合,  $V$  表示顶点的集合。

2、无向图和有向图: 无向图中每一条边是没有方向的,  $(i, j)$  表示  $i$  到  $j$  有边,  $j$  到  $i$  同样也是有边的。而有向图中仅表示  $i$  到  $j$  的边。

3、邻接和关联: 邻接是指顶点和顶点间的关系,  $(i, j)$  表示  $i$  邻接至  $j$ ,  $j$  邻接于  $i$ 。

关联是指顶点和边的关系， $(i, j)$  是关联于  $i$  的边，关联至  $j$  的边。

4、网络：网络是图的更一般形式，在每条边上加上权重就得到网络。有向图和无向图可以看成是权值相同的网络。

5、连通图：连通图中的任一对顶点之间都是有路径可到达的。

6、图的生成树：包含图中的所有顶点的子图并且是树的成为图  $G$  的生成树。（生成树是对于连通图而言的，不是连通图则没有办法形成生成树）。

7、 $n$  个顶点， $e$  条边的图的特性：所有度的和是边的两倍：

$$\sum_{i=1}^n d_i = 2e;$$

## 二、图的实现

### 1、邻接矩阵：

使用  $n \times n$  的二维数组来描述一个包含  $n$  个节点的图。

$A(i, j) = \begin{cases} 1 & \text{如果 } (i, j) \in E \text{ 或 } (j, i) \in E. \\ 0 & \text{其它} \end{cases}$  （无向图）

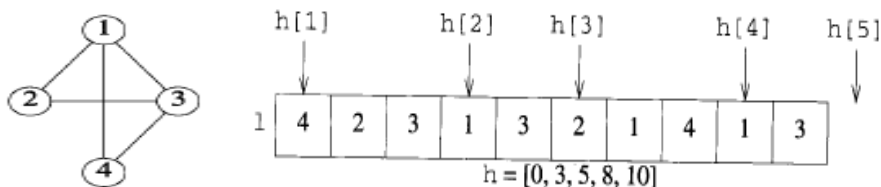
$A(i, j) = \begin{cases} 1 & \text{如果 } (i, j) \in E. \\ 0 & \text{其它} \end{cases}$  （有向图）

插入和删除时只需要修改  $(i, j)$  位置的值即可（无向图中还要修改  $(j, i)$  位置的值），时间复杂度为  $\Theta(1)$ 。

在计算  $i$  顶点的入度时，只要计算第  $i$  列的非零元素个数即可，计算出度时只要计算第  $i$  行非零元素的数目即可，复杂度都是  $\Theta(n)$ ；

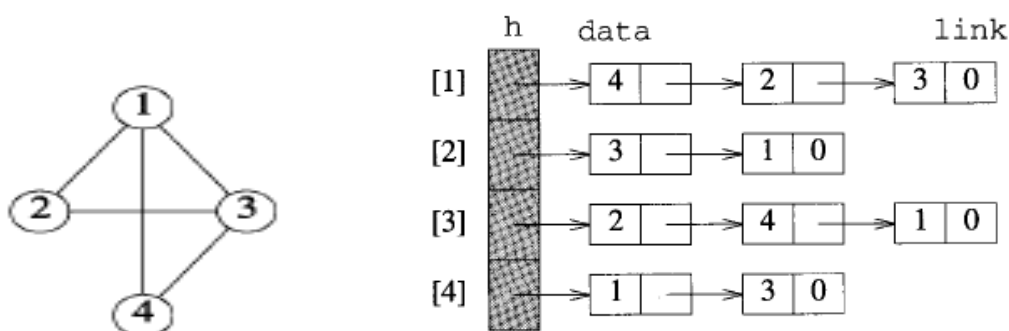
### 2、邻接压缩表：

使用两个数组（ $h$  和  $l$ ）标记邻接关系， $h$  数组  $h[i]$  中记录与  $i$  顶点邻接的顶点在  $l$  数组中的首位置。 $l$  中则记录邻接的顶点。如图所示



### 3、邻接链表：

一个指针数组，保存邻接顶点的链表的头结点。



在插入边时，直接在对应链表的头部插入，时间复杂度是  $\Theta(1)$ ，而删除元素时，需要在链表中查找对应元素，所以最坏是对应链表长度的复杂度。

在计算出度时，只要计算对应链表的长度即可，而计算入度时则需要将每个链表都搜索一遍。

### 4、耗费矩阵和耗费邻接表：

在邻接矩阵的基础上在  $(i, j)$  中存入耗费值即可，使用邻接链表的时候就是在每个节点上多添加一个记录耗费的标记。

注意：在图的实现中定义了一个私有变量 **NoEdge**，不同的图中，**NoEdge** 的值可能是不一样的，例如加权图中，我们一般使用无穷大给 **NoEdge** 赋值，而在一般的有向图和无向图中，我们可以使用 0 给 **NoEdge** 赋值。

### 三、遍历：

#### 1、图遍历器：

在图的各种遍历中都必须使用到图遍历器，这样避免了每次都要从头开始遍历所带来的时间开销。

使用 **pos** 数组记录每一行遍历的位置，对应有四个方法

**Begin(i)**: 返回与  $i$  节点邻接的第一个顶点

**NextVertex(i)**: 返回  $i$  节点邻接的下一个节点

**InitializePos()**: 初始化 **pos** 数组；

**DeactivatePos()**: 取消 **pos** 中保存的每一行的遍历信息。

#### 2、宽度优先搜索（BFS）

宽度优先的想法和迷宫最短路径很相似，都是使用队列来处理。具体的操作如下：

先访问开始顶点，将其加入队列中并且标记为已访问；

从队列中删除元素，

访问与这一元素邻接的所有未标记顶点，将他们加入队列并标记为已访问。

重复上面两步的操作直到队列为空。

伪代码参考书本 P.395，更有助于理解。

#### 3、深度优先搜索（DFS）

深度优先的想法和迷宫老鼠 的想法很相似，都是使用堆栈来处理，具体的操作如下：

先访问开始顶点，将其加入堆栈中，并标记为已访问；

从堆栈中删除元素，

访问与这一元素邻接的所有未标记顶点，将他们加入堆栈并标记为已访问。

重复上面两部的操作，直到堆栈为空。

伪代码参考书本 P.398。

例题：简述深度优先和宽度优先中使用栈和队列的作用。

队列作为一种先进先出的数据结构模式可以保证在宽度优先遍历时先进队的节点的子节点可以优先处理，保证了只有一层节点都处理完之后这一层的子节点才会顺序处理，实现了宽度优先。堆栈作为一种后进先出的数据结构保证了深度优先算法在处理一个分支后可以回溯到其刚进栈的上一节点继续处理，实现了深度优先的思想。

### 四、应用

#### 1、寻找路径

从刚刚的深度优先中不难看出，如果  $v$  和  $w$  之间有路劲，那么深度优先搜索时，这两点都是可达的，我们可以总结出一个寻找路径的而递归方法。

**FindPath** ( $v, w$ , length, path, reach) {

    将  $v$  标记为可达，将  $v$  加入 path 中；

$u$  是和  $v$  邻接的第一个节点；

    while ( $u$ ) {

```

    u 加入 path 中, length++;
    如果 u 就是 w, 那么久找到了 v 到 w 的路径, 否则继续递归
    然后寻找 u 到 w 的路径 FindPath (u, w, length, path, reach)
    如果找不到 length--, u 变为 v 的下一个节点继续。
    }
}

```

伪代码仅供参考, 具体代码详见课本 P.399

## 2、图的连通

显然, 无论我们使用深度优先还是宽度优先, 都可以找到开始节点所能到达的所有顶点, 这些顶点都是连通的, 我们只要把这些顶点都标记出来, 然后看看是不是所有的顶点都是可达的, 如果所有顶点都可达, 则可以知道图是连通的。

详见课本 P.401

## 第十三章 贪婪<sup>[0]</sup>算法

### +概念&理解: 最优化问题

贪婪算法, 顾名思义, 其实就是每次取最优解, 然后持续下去得到最终结果。这是一种解决**最优化问题**的思路 (当然, 也有其他的思路求解最优化问题)。

不是所有问题都可以贪心解决, 但可能需要以此为基础作更深一步的操作。例如最短路径问题<sup>[1]</sup>、0/1 背包问题<sup>[2]</sup>等等。

最优化问题会存在一组**限制条件**与一个**优化函数**, 至于它们是什么例题会提及。这些问题可能会有多个符合限制条件的结果, 这是**可行解**。而如果一个解能使得优化函数得到最优值, 那么它是**最优解**。

### -举例

#### 装载问题:

一艘船的最大载货重量为  $c$ , 现在有一些质量各不相同的货物需要被装载。假设第  $i$  件货物的重量为  $w_i$ , 求问如何装载更多的货物。

~~(这问题好简单啊...)~~

下面将用这个例子说明上面的四个概念。

为了简化说明, 引入一个变量  $x_i$  表示第  $i$  个货物是否被取 (被取值为 1, 不被取值为 0), 显然  $w_i x_i$  就是这一个货物在船内实际占用的重量了。

**限制条件** (由它对可行解作出限制):  $\sum w_i x_i < c$  ( $\sum$ 为求和, 下略)

**优化函数** (由它规定哪组可行解最优):  $\sum x_i$

**可行解**: 满足  $\sum w_i x_i < c$  的每一组  $x_1, x_2, \dots, x_i, \dots$

**最优解**: 在可行解中, 使得  $\sum x_i$  最小的那一组  $x_i$

### +概念&理解: 贪婪算法思想

**贪婪算法**: 对于问题的每一步, 按照**贪婪准则**, 做出看起来最优的决策。

**贪婪准则**: 做出这个最优决策的依据。

-举例：装载问题

标准的贪婪算法能够得到最优解的问题。

在这里我们采取的**贪婪准则**为**每次选择  $w_i$  最小的货物**，直到其不能够再装载任何的货物。

+单源最短路径问题描述

对于一个带权有向图，给定一个节点  $s$ ，求出它到图中任意路径的最短长度。

约定：所有边权  $\geq 0$ ，不存在负边权。

-Dijkstra 算法思路

贪婪算法，每次按照如下规则选择较短路径：

预处理：加入  $s$  到  $s$  的路径，长度为 0，不存在边。*注意边和路径意义不同，下略*

对于每次操作，考虑在已选择的路径中加入一条最短的路径。然后在产生的所有新路径中选择一条最短的路径。

-构造示例

解释：

节点： 红色:当前选中路径终点

灰色:曾被选中路径终点

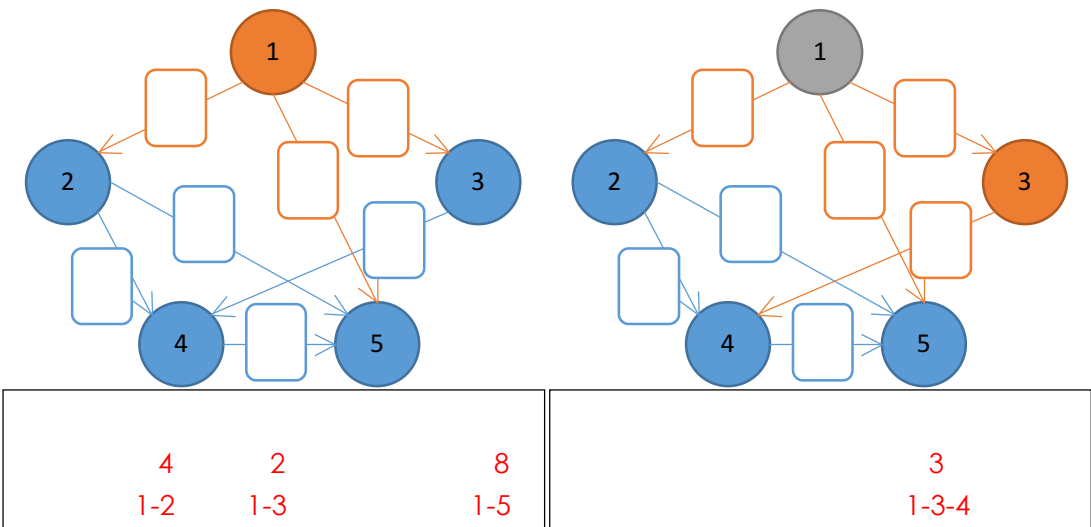
路径： 红色:被选用路径

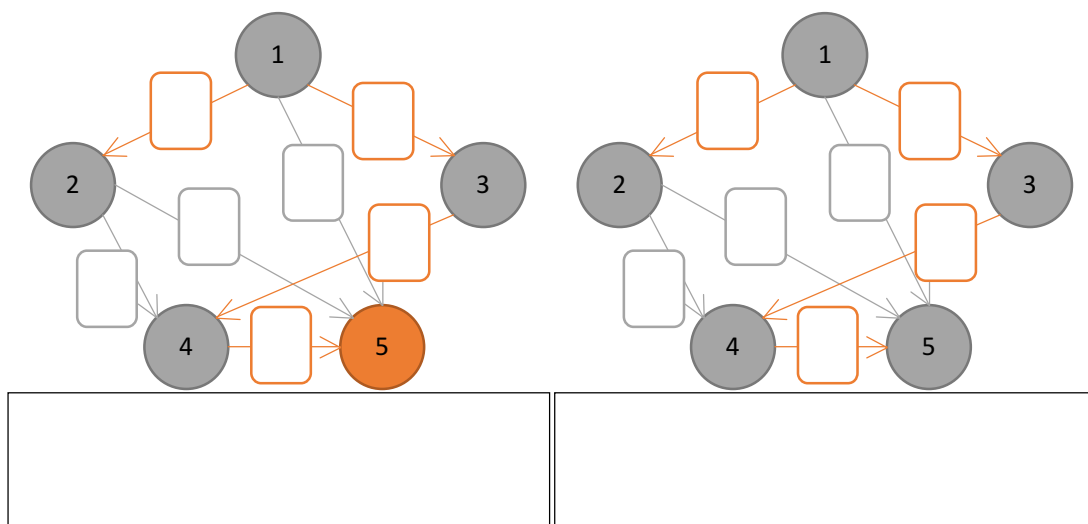
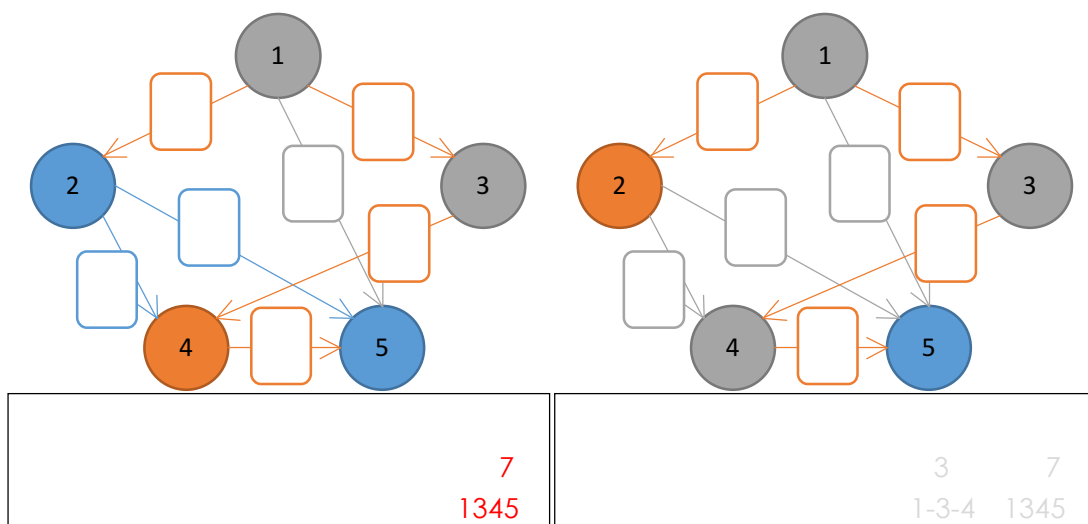
灰色:被弃用路径(不够优被抛弃)

数值： 红色:成功更新

灰色:尝试更新但更新失败(不够优被抛弃)

加粗:添加的路径的起点(即为红色节点)





### -伪代码

#### 数值定义：

$a[x][y]$ : 从  $x$  到  $y$  的路径（即为邻接矩阵存储的图）

$d[i]$ : 从  $s$  到  $i$  的最短距离

$s[i]$ : 记录  $i$  的**前继节点**（用于递归求最短距离的路径）

$L$ : 查找未选择的最短路径（链表或最小堆）

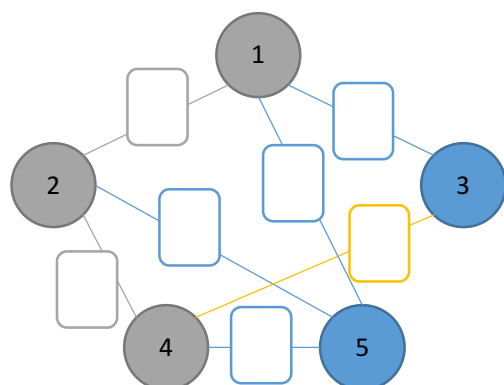
```
for(i from 1 to n)
  if(a[s][i]存在)
  {
    s[i]=s;
    d[i]=a[s][i];
    顶点 i 加入 L;
  }
```

<pre>else     d[i]=inf;//inf 为一个超级无敌螺旋大的数<sup>[3]</sup> while(L 非空) {     从 L 中选择顶点 i 满足 d[i]最小;抛出顶点 i;     for(j from 1 to n)         if (a[i][j]存在&amp;&amp; d[j]&gt;d[i]+a[i][j])         {             s[j]=i;             d[j]=d[i]+a[i][j];             顶点 j 加入 L;         } }</pre>
时间复杂度: $O(n^2)$

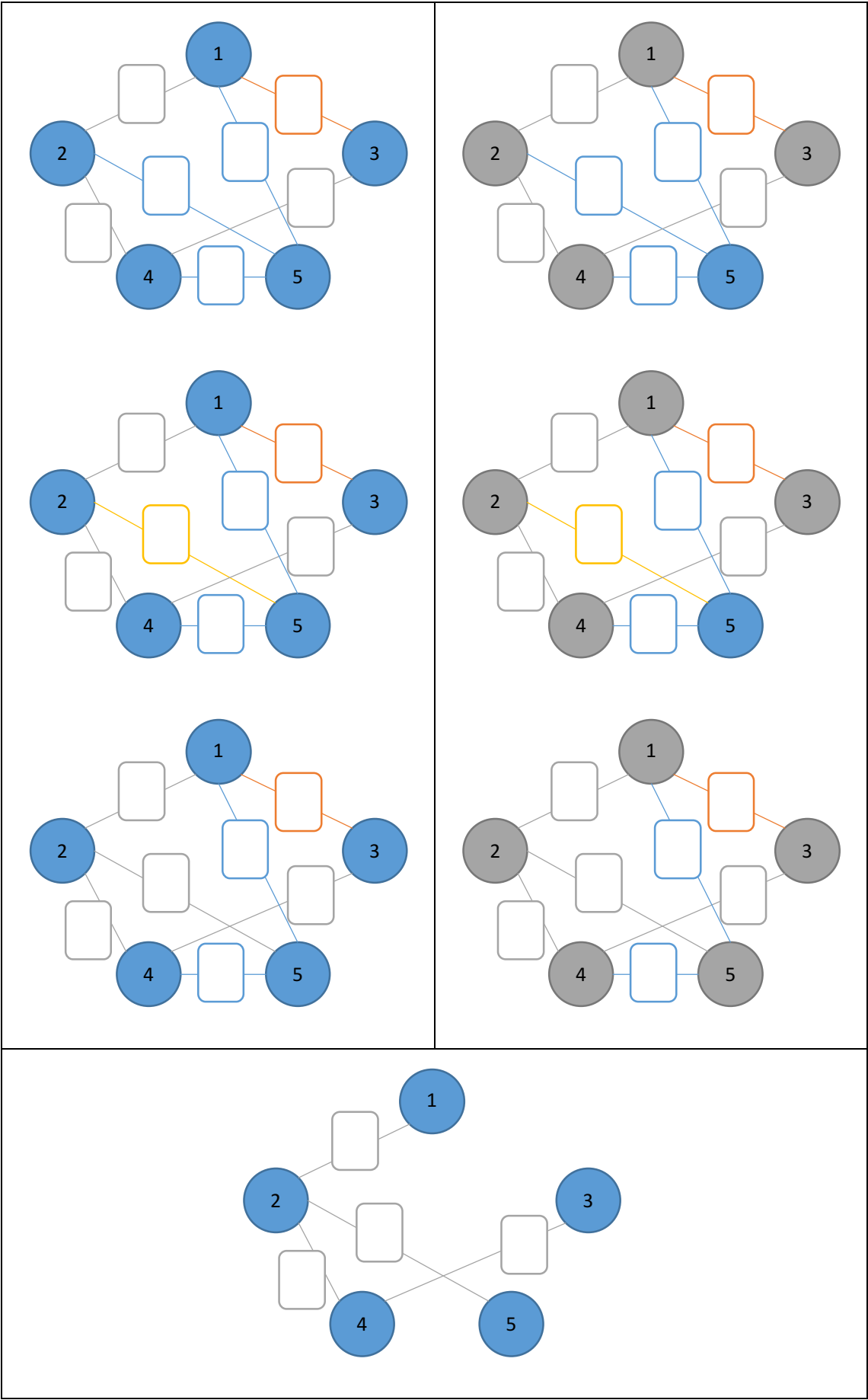
+最小生成树 · 问题描述
提供一加权无向图或强连通有向图，从中构成一个图，使其： 1. 能够从任意顶点出发都可系统地到达任意顶点//限制条件 2. 边权和尽量小//优化函数

-Kruskal 算法
对于每次决策，选择不会形成环路的最小耗费边加入已选择的边中。
-Prim 算法
从一个节点开始。
对于每次决策，选择一条边(u, v)和一个顶点，使得：
1. 加入边后的新集合仍然为一棵树；
2. (u, v)中的 u, v 有且仅有一个顶点已存在于集合中。
终止条件:集合中含有 n-1 个节点时终止。
-Sollin 算法 <sup>[4]</sup>
没啥存在感的算法... 略
-举例

示例图	
Kruskal 算法	Prim 算法







## -伪代码

Kruskal 算法	Prim 算法
E:提供的图      T:所选边的集合;	Tv(in Prim)所选顶点的集合
<pre> while(E 非空&amp;&amp; T &lt;n-1) {     寻找 E 中坠小的边 (u, v);     从 E 中删除边 (u, v);     if((u, v)不会在在加入 T 后成环)         将 (u, v) 加入 T; } if(T==n-1)     T 是最小生成树; else     E 不互联, 不能生成最小生成树 </pre>	<pre> 初始化 Tv={1}; while(E 非空&amp;&amp; T &lt;n-1) {     找出最小边权的边 (u, v)     其中 u 在 Tv 中, v 不在 Tv 中;     if((u, v)不存在)         break;      (u, v) 加入 T 中;     v 加入 Tv 中;     从 E 中删除边 (u, v); } if( T ==n-1)     T 是最小生成树; else     E 不互联, 不是最小生成树; </pre>

## -数据结构选择&amp;复杂度分析

Kruskal 算法	Prim 算法
<p>E:使用最小生成树实现 初始化:<math>O(n)</math> 每次查找与删除边:<math>O(\log n)</math></p> <p>T:使用数组 t, 包含元素 u, v, weight 判断是否有 n-1 条边:<math>O(n)</math> 判断是否成环:等会儿详细探讨 加边:在后面加一个即可:<math>O(n)</math></p>	<p>详细操作复杂度基本同 Kruskal 总复杂度 <math>O(n^2)</math></p>

## -关于 Kruskal 算法提到的判断是否成环问题

检查边 (u, v) 加入 T 后是否会成环, 可转化为 u、v 是否同属于同一个子图。

进而可延伸到[在线等价类<sup>\[6\]</sup>](#)问题, 后面将对在线等价类进行讲解。

这里我们取在线等价类问题中提及的 union 操作与 find 操作, 其中 union(find(i), find(j)) 可以做到将两个顶点所在的集合进行合并, 而 find 可以找出节点 i 所在集合的根节点, 从而判断两个顶点是否在同一集合内。

-成环问题涉及复杂度<sup>[6]</sup>

Initialize: $O(n)$

Union:略大于  $O(n+e)$

考虑边:利用最小堆,  $O(e \log e)$

Kruskal 总递进复杂度: $O(n+e \log e)$

## -在线等价类

上面提到了, 在线等价类是用于解决“两个元素是否在同一类中”与“合并两个类为同一个类”的问题的。

首先，我们需要给节点编号  $1 \sim n$  作为索引以便处理。

然后，建立一个一维数组 `parent[]`，以记录某节点的父亲节点。在这里规定根节点的 `parent` 为 0。

然后，初始化每个节点的 `parent` 为 0（表示每个节点都是一个集合）。

*我一个人就是一支军队。*

```
void Initialize(int n)
{
    parent = new int[n+1];
    for(int e=1;e<=n;e++)
        parent[e]=0;
}
```

在执行 `union(i, j)` 操作的时候，即为将根为 `i` 和 `j` 的两棵树进行合并。

*我们现在，都是士兵了。*

```
void Union(int i, int j)
{
    parent[j]=i;
}
```

在执行 `find(i)` 操作的时候，返回这个节点的根节点。

*我看到你们了。*

```
int Find(int e)
{
    while(parent[e]!=0)
        e=parent[e];
    return e;
}
```

### -优化

很容易看出来，如果在某棵树过高的话，`find` 操作的效率会有很严重的退化。所以我们引入两个规则与一个方法，并利用之对这棵树进行优化。

**重量规则：**在 `union` 操作中，将两棵子树中**节点数较多**的子树的根作为新树的根节点。

**高度规则：**在 `union` 操作中，将两棵子树中**高度较高**的子树的根作为新树的根节点。

**路径压缩：**缩短元素 `e` 到达根节点的路径。方法有三种：**紧凑路径法**、**路径分割法**、**路径对折**。课本中的样例使用的是**紧凑路径法**。

下面将依据**重量规则**对 `union` 操作进行优化，并利用**路径压缩**过程优化 `find` 操作。

### -union 操作优化

定义一个新的一维 `bool` 数组 `root[]`，以确定节点是否为根节点。

**根节点**的 `parent` 不再初始化为 0 即用于判断是否为根节点，而是用于计数该子树的节点个数。其他节点的 `parent` 仍指向父亲节点。

```
void Initialize(int n)
{
    root=new bool[n+1];
    parent=new int[n+1];
    for (int e=1;e<=n;e++) {
```

```

        parent[e]=1;
        root[e]=true;
    }
}

void Union(int i,int j)
{
    if(parent[i]<parent[j])
    {
        parent[j]+=parent[i];//根节点 parent 继续用于计数 胜者为王
        root[i]=false;//节点数较少的根节点不再是根节点 败者为寇
        parent[i]=j;//并将其 parent 指向新的根节点
    }
    else
    {
        parent[i]+=parent[j];
        root[j]=false;
        parent[j]=i;
    }
}

```

#### -find 操作优化

**紧凑路径法思路：**在 find 向上查找到根节点后，将所有沿途经过的节点的 parent 全部改为根节点。

容易看出来，在第一次 find 这个节点的时候是比较费时的，但此后的 find 会越来越省时间。

```

int find(int e)
{
    int j=e;
    while(!root[j])
        j=parent[j];
    int f=e;
    while(e!=j)
    {
        f=parent[e];//照顾好我七舅姥爷!!
        parent[e]=j;
        e=f;
    }
    return j;
}

```

#### +拓扑排序问题·问题描述

有一项大工程被拆分成了若干个任务，这些任务存在一定的先后顺序，所有小任务完成意味着大工程的完成。

为方便起见，将这个工程抽象为一个有向图，有向图的顶点表示任务，边  $(i, j)$  表示任务  $i$  须早于任务  $j$  进行。

请给出一个任务序列，使其满足这个先后关系。

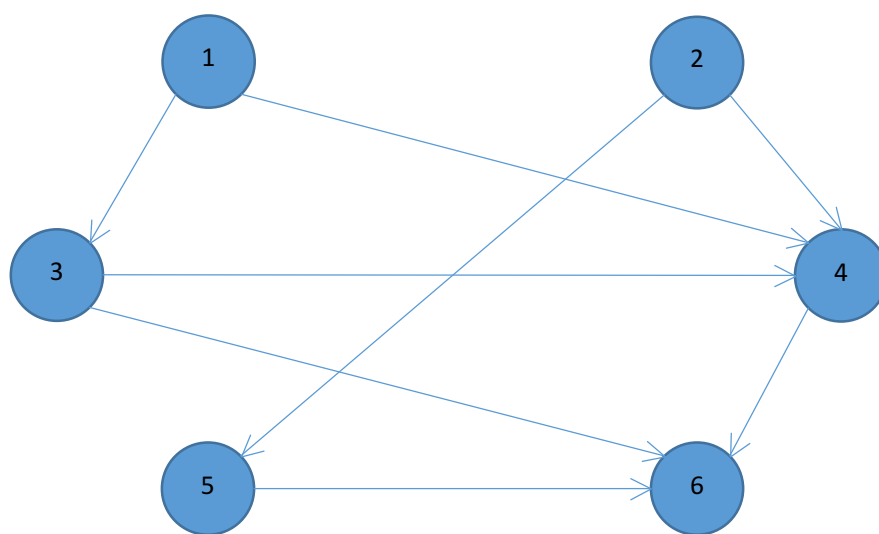
#### -概念&理解

**定点活动网络** (AOV - Activity on Vertex Network) 即为上面提到的图。（定义：表示任务的集合以及任务的先后顺序的图）

**拓扑序列**：一个顶点序列，满足：图中的每一条边  $(i, j)$ ，在序列中  $i$  在  $j$  前。

**拓扑排序**：根据 AOV 图建立拓扑序列的过程。

#### -举例



如图，有以下(但不限于以下的)拓扑序列

123456      132456      215346      251346      .....

#### -构造方法

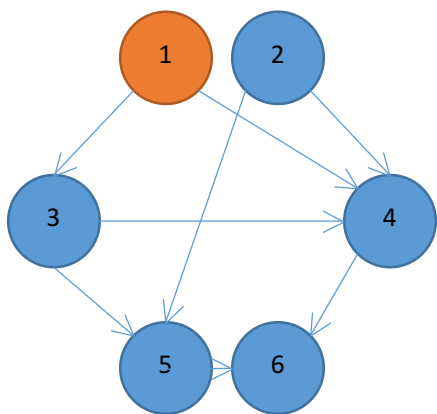
建立一个空序列  $V$ ，对于每次决策，从剩余的节点中向  $V$  中加入一个节点  $w$ ，其中  $w$  满足不存在任何边  $(v, w)$ ， $v$  为未出现在序列  $V$  中的节点。

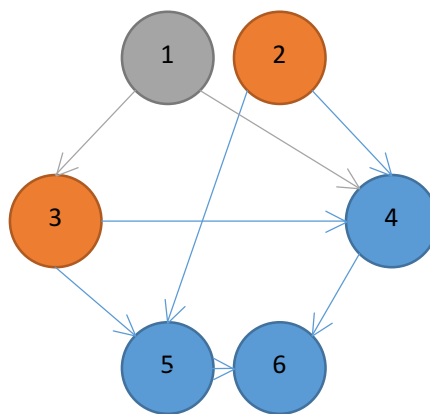
#### -另一种理解方法

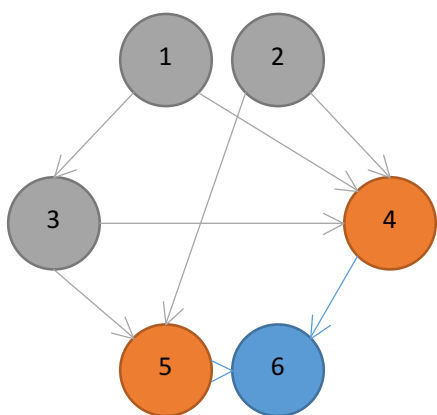
建立一个空序列  $V$ ，对于每次决策，从剩余的节点中向  $V$  中加入一个节点  $w$ ，其中  $w$  的入度为 0。在加入节点  $w$  后，将所有图中与  $w$  相关的边隐藏。隐藏的边不参与计算入度。

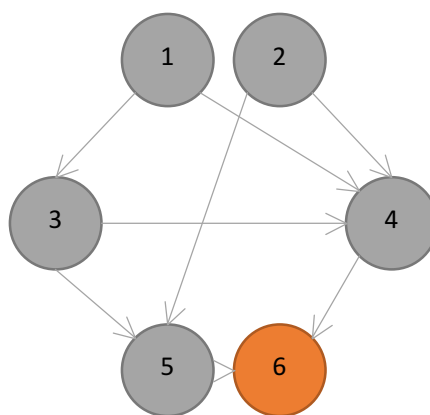
**入度**：指向这个节点的边的条数。

#### -构造示例










图中，红色节点表示在剩余的节点中符合条件的  $w$ 。

灰色的节点表示已被选过的节点，灰色的边表示边  $(v, w)$  中， $v$  已出现在序列  $V$  中了（或者第二种理解中所谓的“被隐藏的边”）。

推荐链接：<https://visualgo.net/dfsbfsc> 左下选择 *Topological sort*

怎么玩就自己摸索下吧挺好玩儿的

-伪代码（代码参见课本/课件）

```

while(true)
{
    寻找节点  $w$ ，其中  $w$  入度为 0;
    if(找不到  $w$ )
        break;
    将  $w$  加入到  $V$  的末尾;
    for(遍历所有从  $w$  出发的边  $(w, x)$ )
        节点  $x$  的入度-1;
}
  
```

```

}
if(V 中的节点数量少于 n 个)报错:图不存在拓扑序列;
else 输出序列 V;

```

### -时间复杂度

邻接矩阵存储图: $O(n^2)$

邻接链表存储图: $O(n+e)$

FIN.

Ref.

大体结构, 采取例题参考孔兰菊老师的 PPT

内容参考《数据结构, 算法与应用——C++语言描述》, Sartaj Sahni, 王诗林等译, 机械工业出版社

使用字体: 微软雅黑 (标题), 微软雅黑 Light, Century Gothic

P. s.

[0] 个人还是更喜欢管这算法叫贪心算法... 文中出现贪婪或者贪心这两种算法指的是同一个东西, 不要在意那些细节。

[1] 最短路问题: 给出一个图, 找出一条从节点  $s$  到  $t$  的路径, 使得这条路径的权值和在所有路径中最小。

[2] 0/1 背包问题: 背包的最大装载重量为  $c$ , 有多个物品可以装入。对于第  $i$  个货物, 它的重量为  $w_i$ , 价值为  $v_i$ 。求问如何装载物品使得价值总和最大。

[3] 初始化为  $inf$ :  $inf$  不是系统常量, 需要自己定义;  $inf$  的建议取值为不使得  $d[i]+a[i][j]$  溢出。此外课件使用的是 0, 这样的话需要在此后的判断条件中加入  $d[i]$  是否为 0。

[4] Sollin 算法: 思路: 起始条件为每个节点为一棵树从而组成森林。每次由每棵树均选取满足以下条件的一条边: 1. 这条边有且恰有一个节点在这棵树中; 2. 它的权值是所有可选择的边中最小的。多棵树可以同时选同一条边。然后丢弃重复边与构成环路的边。在只剩一棵树或没有可选边的时候停止。详见中文版课本 P430。

[5] 在线等价类问题: 见课本例题 3. 8. 3 (中文版课本 P117), 8. 10. 2 (中文版课本 P268)。下面提供的解释为 8. 10. 2, 即利用树形结构完成的在线等价类的方法。

[6] 复杂度计算: 详见中文版课本 P427 (程序 13-6 之前)。

——吉鹏智库 张晓敏@2016/12/29 署名-非商业性使用-相同方式共享

## 第十四章 分而治之算法

从实例中体会这种将大问题转化为小问题最终合并的算法思想。因为小问题的处理是跟原问题相似的, 所以分治都可以自然写成递归。

下面看两个典型例子

### 1. 归并排序 (Merge Sort)

算法思想:

将  $n$  个元素按非递增顺序排列。

若  $n$  为 1，算法终止；

否则，将这一元素集合分割成两个或更多个子集合，对每一个子集合分别排序，然后将排好序的子集合归并为一个集合。

#### 特例：

选择排序（遍历依次选择最大的与队尾元素交换）、插入排序（将一个元素看做已经有序，不断向后遍历有序插入前面有序元素）、冒泡排序（依次将最大元素冒到队尾）分别是分治算法的三个特例，它们把待排序集合不平均地划分。

#### 伪码：

分治的伪代码如下，思想就是不断递归处理规模更小的子问题，当子问题成为基问题后逐步返回得到原问题的解。

```
template<class T>
void sort( T E, int n)
{ //对 E 中的 n 个元素进行排序， k 为全局变量
if (n>=k) {
i = n/k;
j = n-i;
令 A 包含 E 中的前 i 个元素
令 B 包含 E 中余下的 j 个元素
sort(A,i);
sort(B,j);
merge(A,B,E,i,j); //把 A 和 B 合并到 E
}
else 使用插入排序算法对 E 进行排序
}
```

当全局变量  $K = 2$  时所花费的时间最少，为  $n \log n$ ，此法又叫二路归并排序。

此归并排序程序在 P446 程序 14-3——14-5

#### 重点解读代码：

1. 注意的是 MergePass 中

```
while (i<=n-2*s) {
// 归并两个大小为 s 的相邻段
Merge(x, y, i, i+s-1, i+2*s-1); //将 x[i:i+s-1] x[i+s:i+2s-1]合并到 y[i:i+2s-1]数组。
i=i+2*s;
}
//结束后 i>n-2*s,也就是说剩下不足 2s 个元素没办法将两个 s 合并到一起成 2s
if (i+s<n) Merge(x, y, i, i+s-1, n-1); // 剩下不足 2s 但是大于 s 个元素，直接合并到一起即使最后数组段不是 2s 个元素
else for (int j = i; j <= n-1; j++) // 把最后一段复制到 y，因为最后一段长度小于 2*s 的一半，肯定能保持有序
y[j] = x[j];
```

2. 关于 Merge 函数实现思路就是链表联系中那个 Merge 思路，头元素挨个比下去。

```
void Merge(T c[], T d[], int l, int m, int r)
{ //把 c[l:m]和 c[m+1:r]归并到 d[l:r].
```



```

int i=l, // 第一段的游标
j=m+1, // 第二段的游标
k=l; // 结果的游标
//只要在段中存在 i 和 j, 则不断进行归并
while ((i<=m)&&(j<=r))
if (c[i]<=c[j]) d[k++]=c[i++];
else d[k++]=c[j++];
// 考虑余下的部分
if (i>m) for (int q=j;q<=r;q++)
d[k++]=c[q];
else for (int q=i;q<=m;q++)
d[k++]=c[q];
}

```

**归并排序是稳定排序**，在 1 个或 2 个元素时，1 个元素不会交换，2 个元素如果大小相等也没有人故意交换，这不会破坏稳定性。那么，在短的有序序列合并的过程中我们可以保证如果两个当前元素相等时，我们把处在前面的序列的元素保存在结果序列的前面，这样就保证了稳定性。所以，归并排序是稳定的排序算法。**自然归并排序 P447**

## 2. 快速排序 (Quik Sort)

**排序伪码：**

```

//使用快速排序方法对 a[0:n-1]排序
从 a[0:n-1]中选择一个元素作为 middle, 该元素为支点
把余下的元素分割为两段 left 和 right, 使得 left 中的元素都小于等于支点, 而 right 中的元素都大于等于支点
递归地使用快速排序方法对 left 进行排序
递归地使用快速排序方法对 right 进行排序
所得结果为 left+middle+right

```

**快速排序实现思想：**

选定一个支点在最左边，保证从左向右直到某个点所有的数都要比这个支点小，从右向左直到某个点都要比这个支点大，只要保证这个点的值小于支点的值，最后交换这个点和支点就可以保证实现支点左边比支点小，支点右边比这个点大的效果。这个点一开始是不确定的，是通过不断循环，从左到右碰到一个大于支点的点 A 就和从右往左碰到一个小于支点的点 B 交换，直到从左向右的点 A 的下标要大于从右向左点 B 的下标，说明点 A 选在了点 B 右侧，这时 A 大于支点，B 小于支点，且在 A 左边的点都小于支点，在 B 右边的点都大于支点。时机成熟！只需将 B 和支点一换就可以完成目标，因为 A 大于 B，且仍满足在 B 的右侧。

**代码：**

```

void quickSort(T a[], int l, int r)
{//排序 a[l:r], a[r+1]有大关键值
if (l>=r) return;
int i=l, // 从左至右的游标
j=r+1; // 从右到左的游标
T pivot=a[l];

```

//把左侧 $\geq \text{pivot}$  的元素与右侧 $\leq \text{pivot}$  的元素进行交换

```
while(true){
    do{ i=i+1;
    }while(a[i]<pivot); //在左侧寻找 $\geq \text{pivot}$  的元素
    do{ j=j-1;
    }while(a[j]>pivot); //在右侧寻找 $\leq \text{pivot}$  的元素
    if(i>=j)break;//未发现交换对象
    Swap(a[i],a[j]);
}
//设置 pivot
a[l]=a[j];
a[j]=pivot;
quickSort(a,l,j-1);//对左段排序
quickSort(a,j+1,r);//对右段排序
}
```

快速排序不是稳定排序，万一支点就是和第二个值相同，相对位置肯定要换。

**复杂度：nlogn**

**中值快速排序**

了解，比快速排序更好，不必使用  $a[l]$  做为支点，而是取  $\{a[l], a[(l+r)/2], a[r]\}$  中大小居中的那个元素作为支点。

**选择第 k 名元素**

$j-l+1=k$ :

$j-l+1>k$ : 左面部分第 k 小的元素

$j-l+1<k$ : 右面部分第  $(k-(j-l+1)=k-j+l-1)$  小的元素

看支点左边几个元素，多了就从左边选支点，以老支点为右界，重新快排；

少了就选右边

正好有  $k-1$  就正好是这个支点

1