总复习

（注：蓝色字为老师最后上课给的复习提纲，黑色字体为自己总结的各个知识点的扩充，最后一页紫色字体为老师上课说的话，排除的知识点，还有考题举例，可以先看最后的紫色字体，再回头复习前面知识点，有些空白的为没有总结的知识点，先复习着，后面补充好再发。）

————张楠鸽

一、数据结构

1、 红黑树、序统计树、区间树

1) 红黑树的性质、操作及时间
2) 红黑树的应用——序统计树、区间树的定义、构造
3) 数据结构的扩张步骤

考点总结：

红黑树的性质（P163）：

1、每个结点要么是红的要么是黑的。

2、根结点是黑的。

3、每个叶结点（叶结点即指树尾端 NIL 指针或 NULL 结点）都是黑的。

4、如果一个结点是红的，那么它的两个儿子都是黑的。

5、 对于任意结点而言，其到叶结点树尾端 NIL 指针的每条路径都包含相同数目的黑结点。

红黑树的操作：

（包括左旋、右旋 P165、查找、插入、删除，其中插入删除太复杂，不作要求）

红黑树的操作时间：

因为一棵由 n 个结点随机构造的二叉查找树的高度为 lgn，所以顺理成章，二叉查找树的一般操作的执行时间为 O(lgn)。但二叉查找树若退化成了一棵具有 n 个结点的线性链后，则这些操作最坏情况运行时间为 O(n)。

红黑树虽然本质上是一棵二叉查找树，但它在二叉查找树的基础上增加了着色和相关的性质使得红黑树相对平衡，从而保证了红黑树的查找、插入、删除的时间复杂度最坏为 O(log n)。

红黑树的应用：顺序统计树和区间树，实际都是对红黑树的扩张。

顺序统计树（P181）：

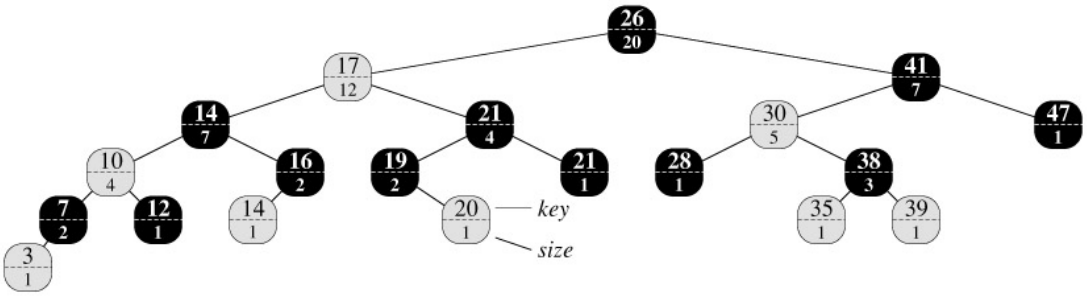
在包含 n 个元素的无序集合中，寻找第 i 个顺序统计量的时间复杂度为 O(n)。通过建立一种特定的结构，可以使得任意的顺序统计量都可以在 O(lgn)的时间内找到。这就是下面会提到的基于红黑树的顺序统计树。

相比于基础的数据结构，顺序统计树增加了一个域 size[x]。这个域包含以 x 为根的子树的节点数(包含 x 本身)。size 域满足等式：

$$size[x] = size[left[x]] + size[right[x]] + 1$$

再根据红黑树的排序特性，我们就可以 O(lgn)的时间内完成下面的操作。

顺序统计树如下图所示：



查找第 i 小的元素

实现 OS-SELECT(x, i)返回以 x 为根的子树中包含第 i 小关键字的节点的指针。根据排序树的性质，我们知道左子树的键值要小于根节点的键值，右节点的键值要大于根节点，这相当于静态的顺序统计量的 PARTITION 已经完成。同时我们知道左右子树的大小，我们就可以确定在哪个分支进行接下来的查找。

OS-SELECT(x, i) 整个过程如下：

```cpp
OS-SELECT(x, i)
r ← size[left[x]]+1
if i=r
   then return x
elseif i<r
   then return OS-SELECT(left[x], i)
else return OS-SELECT(left[x], i-r)
```

每次调用，必定会下降一层，故 OS-SELECT 的时间复杂度为 O(lgn)

确定一个元素的秩

这里的秩指的是节点 x 在线性序中的位置。根据排序树的性质，也就是节点 x 在中序遍历中的位置。利用中序遍历的特性就可以得到。

OS-RANK(T, x) 整个过程如下：

```cpp
OS-RANK(T, x)
r ← size[left[x]]+1
y ← x
while y ≠ root[T]
   do if y = right[p[y]]
      then r ← r + size[left[y]]+1
   y ← p[y]
```

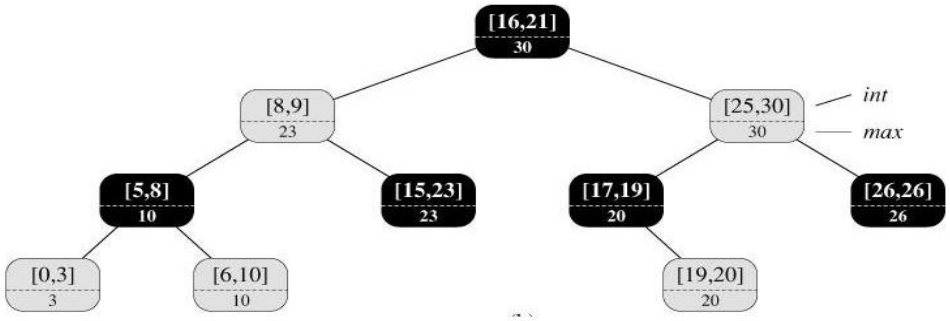
每次必然至少上升一层，故 OS-RANK 的时间复杂度为 O(lgn)

建立顺序统计树的时间为 O(nlgn)，那和一般的静态查找 O(n)相比，岂不是更加复杂？其实，这两种方法应用的场合不一样，如果只查找一次或几次，静态查找比较快速，如果多次查找（查找次数和 n 具有可比性），那么顺序统计树就体现出它的优点了。另外，顺序统计树还可以方便快速(O(lgn)时间内)的支持元素的插入和删除，这两点是静态顺序统计量方法无法比拟的。

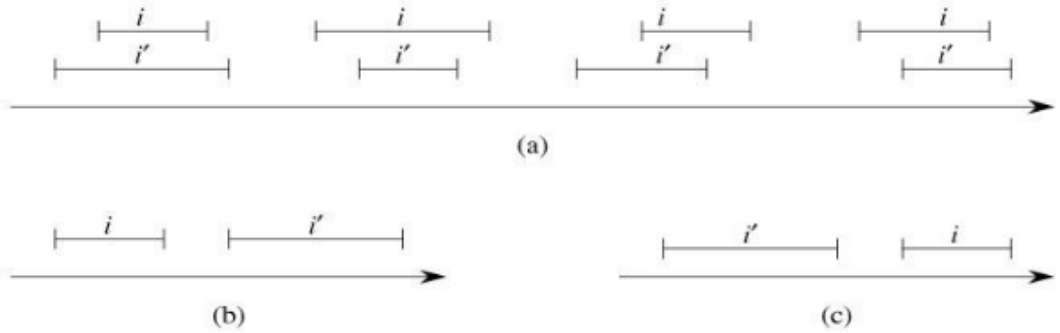
区间树：

区间树是在红黑树基础上进行扩展得到的支持以区间为元素的动态集合的操作，其中每个节点的关键值是区间的左端点。通过建立这种特定的结构，可是使区间的元素的查找和插入都可以在 O(lgn)的时间内完成。

相比于基础的数据结构，增加了一个 max[x]，即以 x 为根的子树中所有区间的断点的最大值。区间树如下图所示：



这里的区间查找的并不是精确查找，而是查找和给定区间重叠的元素，重叠的定义如下图所示：



图中(a)表示两个区间重叠的情况，其他的(b)、(c)表示没有重叠的情况。

区间查找

实现 INTERVAL-SEARCH(T, i)，返回一个和区间 i 重叠的区间，若无，则返回 nil[T]。基本思想是我们通过左子树的 max 进行划分：如果左子树的 max 值小于 low[i]，则左子树不存在这样的区间和 i 重叠，转到右子树；否则，转到右子树，因为左子树的端点小于右子树，若左子树无可能，右子树也必然不可能。

INTERVAL-SEARCH(T, i)整个过程如下：

```cpp
INTERVAL-SEARCH(T, i)
x ← root[T]
while x≠nil[T] and i does not overlap x
   do if left[x]≠nil[T] and max[left[x]]≥low[i]
         then x ← left[x]
         else x ← right[x]
return x
```

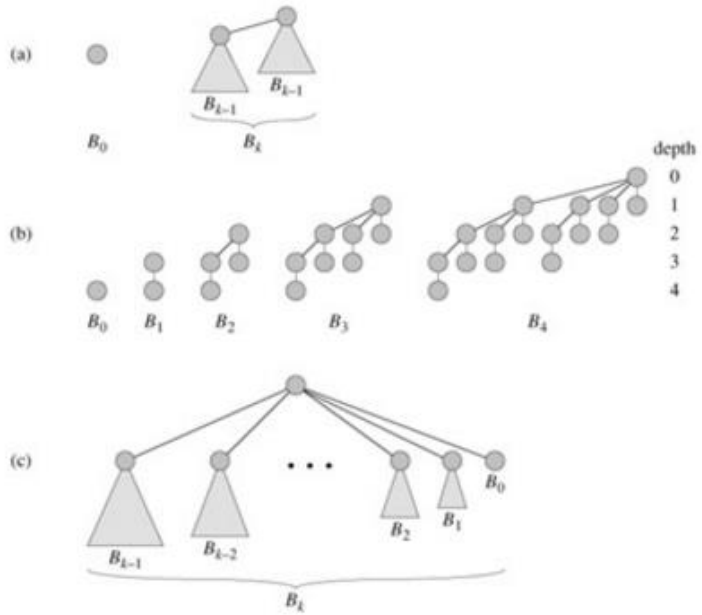
每次调用，必定会下降一层，故 INTERVAL-SEARCH 的时间复杂度为 O(lgn)。

数据结构的扩张步骤（P185）：
从特殊到一般,数据结构的扩张步骤可以归纳为:
1.选择一种基础数据结构
2.确定要在基础数据结构中添加哪些信息
3.验证可用基础数据结构上的基本修改操作来维护这些新添加的信息
4.根据需求设计新的操作

2、 二项堆

1) 二项树的定义、性质
2) 二项堆的定义
3) 根表的性质
4) 二项堆的操作、时间

考点总结：
二项树（P278）：
二项树是一种递归的定义:
1. 二项树 B[0]仅仅包含一个节点
2. B[k]是由两棵 B[k-1]二项树组成，其中一颗树是另外一颗树的子树。
下面是 B0 - B4 二项树：



显然二项树具有如下的性质：
1. 对于树 B[k]该树含有 2^k 个节点；
2. 树的高度是 k；
3. 在深度为 i 中含有 $\binom{k}{i}$节点，其中 i = 0， 1，2 … , k;

4.根的度数为 k，它大于任何其他结点的度数；并且如果根的子女从左到右编号为 k-1，k-2，…，0，子女 i 是子树子树 Bi 的根。

二项堆（P278）：
二项堆是由一组满足下面的二项树组成：
1. H 中的每个二项树遵循最小堆性质：结点的关键字大于或等于其父节点的关键字，我们说这种最小堆是有序的；
2. 对于任意的非负整数 k，在 H 中至多有一棵二项树的根具有度数 k；
另外定义：
1. 二项堆的度数定义成子女个数；
2. 定义二项堆的根表是二项树的根节点形成的链表；

根表的性质：二项堆中各二项树的根被组织成一个链表，称之为根表。根表是一个严格按根的度数递增排序的链表。

二项堆的操作、时间（P281）：
创建一个新二项堆
MAKE-BINOMIAL-HEAP 分配并返回一个对象 H，且 head[H]=NIL，运行时间为 O（1）

寻找最小关键字
遍历根表，找出根表中关键字最小的结点。

```
1  y ← NIL
2  x ← head[H]
3  min ← ∞
4  while x ≠ NIL
5     do if key[x] < min
6           then min ← key[x]
7                y ← x
8        x ← sibling[x]
9  return y
```

因为一个二项堆是最小堆有序的，故最小关键字必在根节点中，过程 BINOMIAL-HEAP-MINIMUM 检查所有的根，将当前最小者存于 min 中，而将指向当前最小者的指针存于 y 之中，最多要检查 $\lfloor \lg n \rfloor + 1$ 个根，故 BINOMIAL-HEAP-MINIMUM 的运行时间为 O(lgn)。

合并两个二项堆
合并包含三个函数：

BINOMIAL-LINK，连接操作，即将两棵根节点度数相同的二项树 Bk-1 连接成一棵 Bk。伪代码：

```
BINOMIAL-LINK(y, z)
1  p[y] ← z
2  sibling[y] ← child[z]
3  child[z] ← y
4  degree[z] ← degree[z] + 1
```

合并操作分为两个阶段：

第一阶段：执行 BINOMIAL-HEAP-MERGE，将两个堆 H1 和 H2 的根表合并成一个链表 H，它按度数排序成单调递增次序。MERGE 的时间复杂度 O(logn)。n 为 H1 和 H2 的结点总数。（对于每一个度数值，可能有两个根与其对应，所以第二阶段要把这些相同的根连起来）。

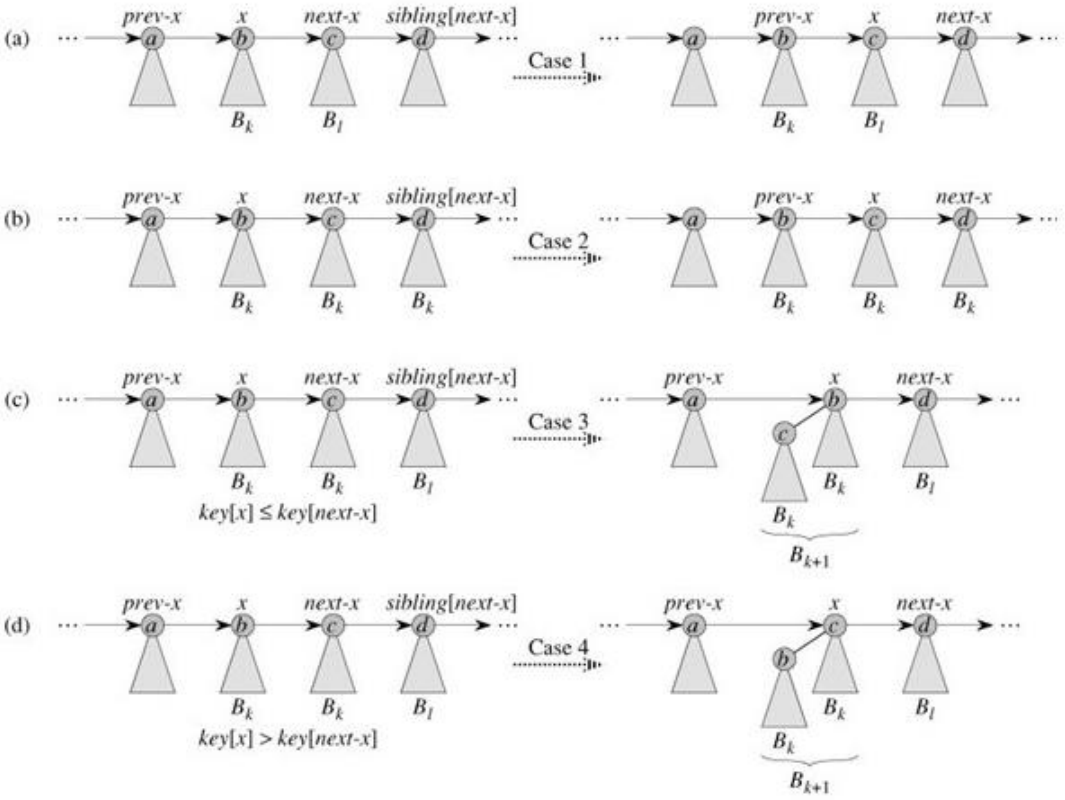
第二阶段：将相等度数的根连接起来，直到每个度数至多有一个根时为止。执行过程中，合并的堆 H 的根表中至多出现三个根具有相同的度数。（MERGE 后 H 中至多出现两个根具有相同的度数，但是将两个相同度数的根的二项树连接后，可能与后面的至多两棵二项树出现相同的度数的根，因此至多出现三个根具有相同的度数）

第二阶段根据当前遍历到的根表中的结点 x，分四种情况考虑:

Case1：degree[x] != degree[sibling[x]]。此时，不需要做任何变化，将指针向根表后移动即可。(下图示 a)
Case2：degree[x] == degree[sibling[x]] == degree[sibling[sibling[x]]]。此时，仍不做变化，将指针后移。(下图示 b)
Case3 & Case4：degree[x] = degree[sibling[x]] != degree[sibling[sibling[x]]] (下图示 c 和 d)

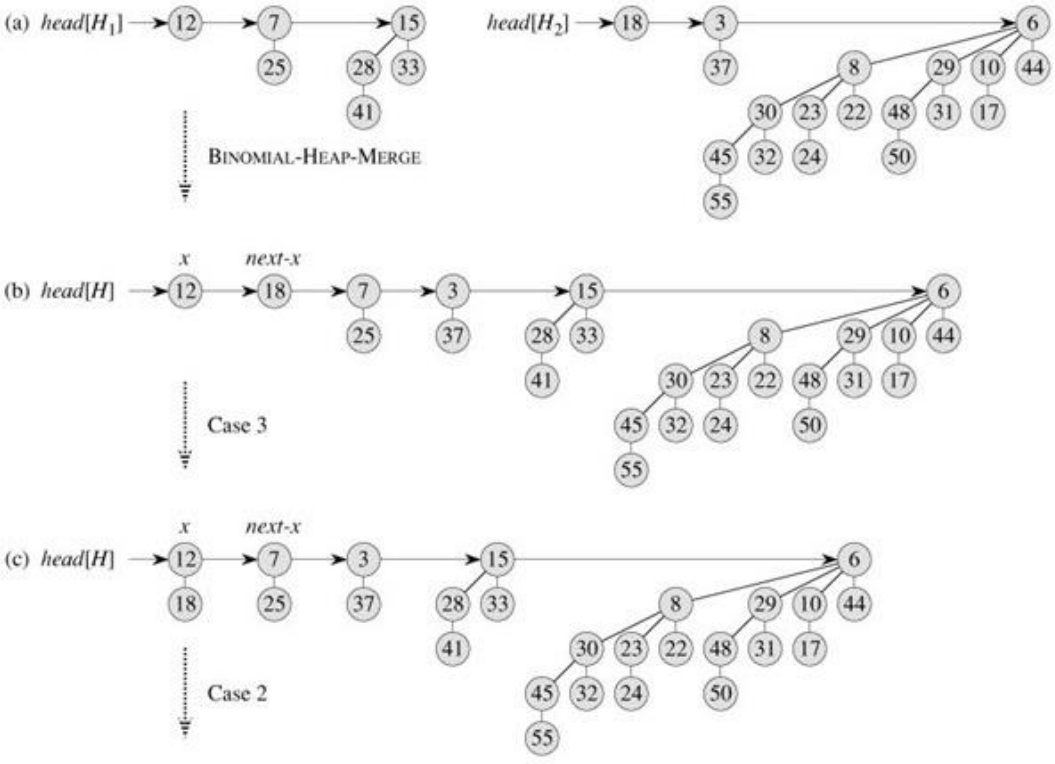
Case3：key[x] <= key[sibling[x]]。此时，将 sibling[x]连接到 x 上。
Case4：key[x] > key[sibling[x]]。此时，将 x 连接到 sibling[x]上。

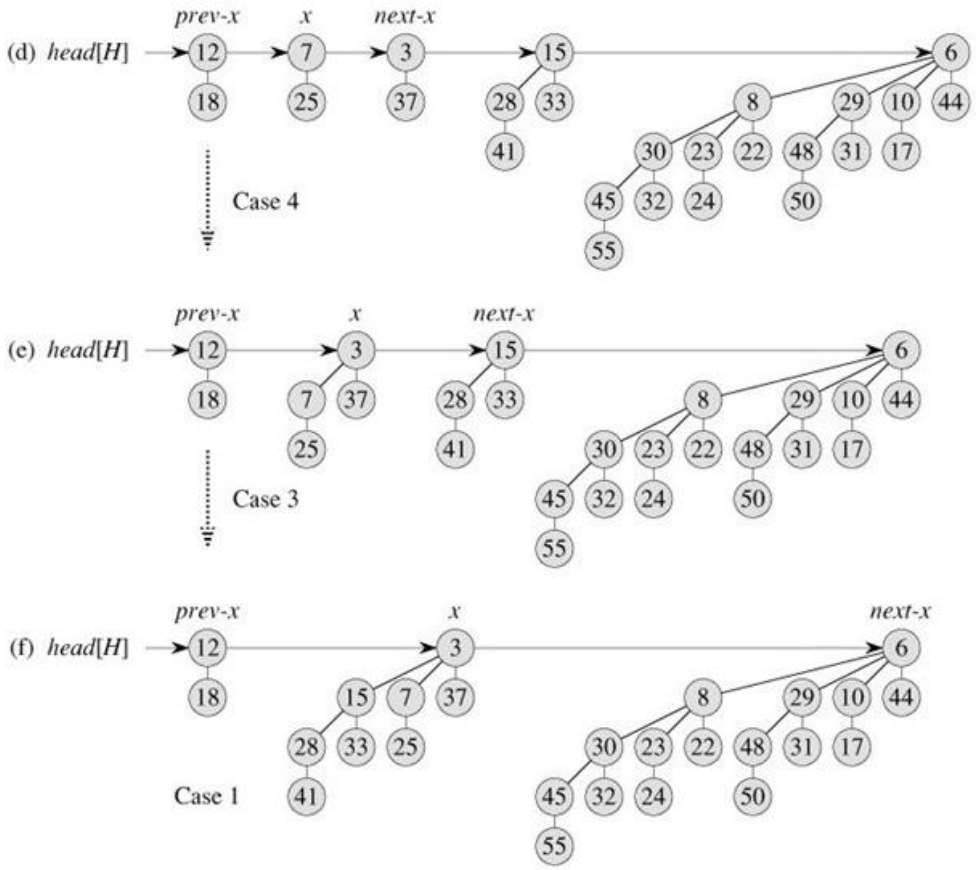
复杂度：O(logn)，　四个过程变化情况：



（注：合并的步骤和注意事项这里没有写的很详细，不方便理解，见书上 P281，过程仅帮助理解，可以画出合并的最终结果即可）

完整的一个合并Sample:





插入一个结点：
先构造一个只包含一个结点的二项堆，再将此二项堆与原二项堆合并，伪代码：

```
BINOMIAL-HEAP-INSERT(H, x)
1  H' ← MAKE-BINOMIAL-HEAP()
2  p[x] ← NIL
3  child[x] ← NIL
4  sibling[x] ← NIL
5  degree[x] ← 0
6  head[H'] ← x
7  H ← BINOMIAL-HEAP-UNION(H, H')
```

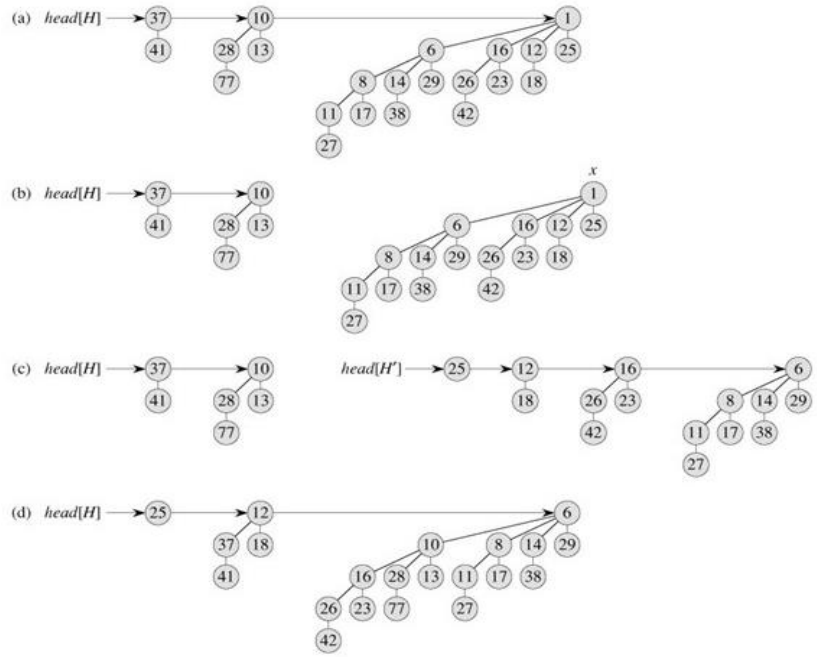
这个过程先在 O(1)时间内，构造一个只包含一个结点的二项堆，再在 O(lgn)时间内，将其与包含 n 个结点的二项堆合并。

抽取有最小关键字的结点（时间：$O(\lg n)$）

从根表中找到最小关键字的结点，将以该结点为根的整棵二项树从堆取出，删除取出的二项树的根，将其剩下的子女倒序排列，组成了一个新的二项堆，再与之前的二项堆合并。
伪代码：

```
BINOMIAL-HEAP-EXTRACT-MIN(H)
1  find the root x with the minimum key in the root list of H,
           and remove x from the root list of H
2  H' ← MAKE-BINOMIAL-HEAP()
3  reverse the order of the linked list of x's children,
           and set head[H'] to point to the head of the resulting list
4  H ← BINOMIAL-HEAP-UNION(H, H')
5  return x
```

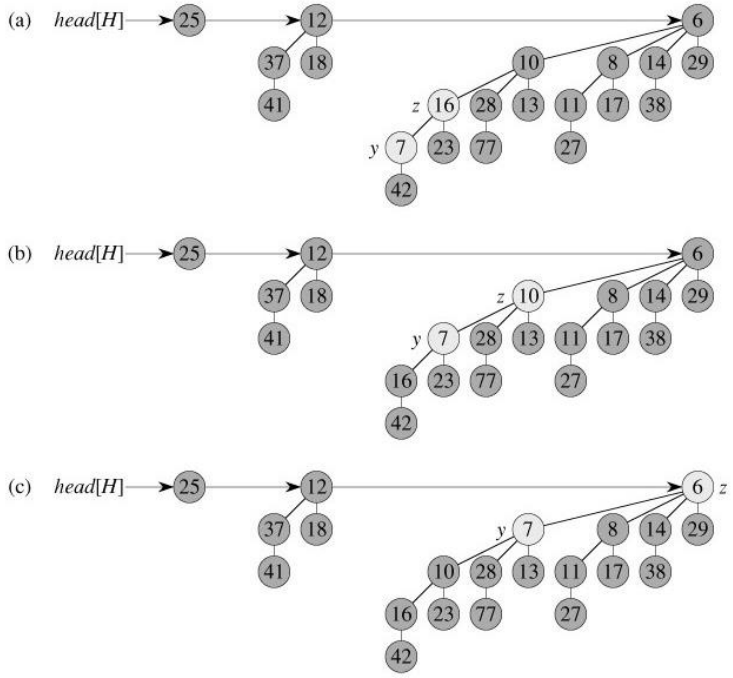
示例图：



减小关键字的值（时间：O(lgn)）：
伪代码：

```
BINOMIAL-HEAP-DECREASE-KEY(H, x, k)
1 if k > key[x]
2    then error "new key is greater than current key"
3 key[x] ← k
4 y ← x
5 z ← p[y]
6 while z ≠ NIL and key[y] < key[z]
7     do exchange key[y] ↔ key[z]
8        ▸ If y and z have satellite fields, exchange them, too.
9        y ← z
10        z ← p[y]
```

减小关键字的过程类似维护最小堆结构，key[y]与 y 的父结点 z 的关键字作比较。如果 y 为根或者 key[y] >= key[z],则该二项树已是最小堆有序。否则结点研究违反了最小堆有序，故将其关键字与其父节点 z 的关键字相交换，同时还要交换其他卫星数据。第 6 行的 while 循环这个过程。
示例图：



删除一个关键字，（时间 O(lgn)）：
删除的原理非常简单，把关键字减小，让它到达根节点，然后剔除最小值即可。

3、 Fib 堆

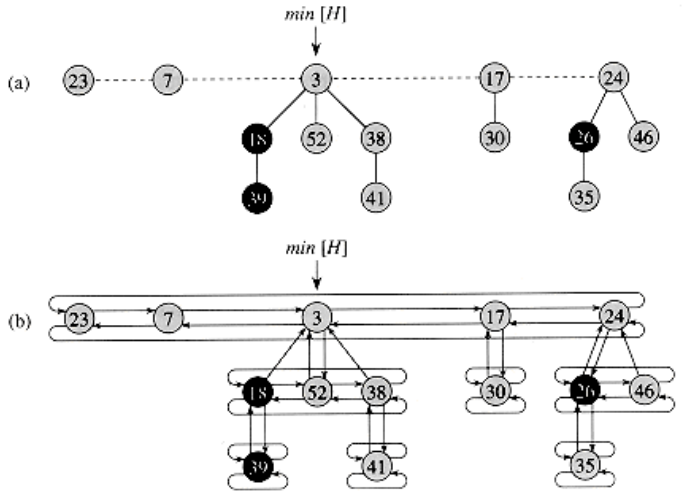
1) Fib 堆的定义

2) 根表的性质

3) Fib 堆的操作、时间

考点总结：
Fib 堆的定义：
斐波那契堆是一种松散的二项堆，与二项堆的主要区别在于构成斐波那契堆得树可以不是二项树，并且这些树的根排列是无序的（二项堆的根结点排序从左到右是按照结点个数排序的，不是按照根结点的大小）。
斐波那契堆得优势在于它对建堆、插入、抽取最小关键字、联合等操作能在 O(1)的时间内完成（不涉及删除元素的操作仅需要 O（1））。这是对二项堆效率的巨大改善。在 EXACT-MIN|, DELETE 的操作数目较小时，斐波那契堆是很理想的。对许多稠密图来说，每一次 DECREASE-KEY 调用的 O(1)加起来，就是对二叉或者二项堆的最坏情况的一种显著改善。用于解决最小生成树和寻找单源最短路径等问题的快速算法都要用到斐波那契堆。

对于一个给定的斐波那契堆 H，可以通过指向包含最小关键字的树根的指针 min[H]来访问，这个结点被称为斐波那契堆中的最小结点。如果一个斐波那契堆 H 是空的，则 min[H] = NIL. 在一个斐波那契堆中，所有树的根都通过 left 和 right 指针链接成一个环形的双链表，称为堆的根表。于是，指针 min[H]就指向根表中具有最小关键字的结点
Fib 堆如图所示：



Fib 堆根表的性质：

根排列是无序的。所有树的根都通过 left 和 right 指针链接成一个环形的双链表。

Fib 堆的操作、时间：

创建一个新的 fib 堆：

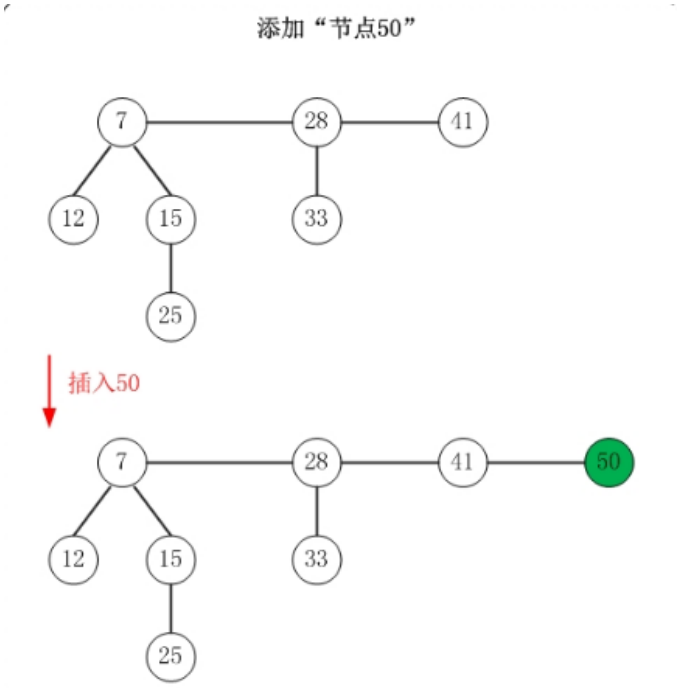
创建一个空的斐波那契堆，过程 MAKE-FIB-HEAP 分配并返回一个斐波那契堆对象 H，且 n[H]=0, min[H]=NIL；此时 H 中还没有树。因为 t[H]=0, min[H]=0，该空斐波那契堆的势Φ(H)=0.因此，MAKE-FIB-HEAP 的平摊代价就等于其 O(1)的实际代价。

势函数：Φ(H)=t(H)+2m(H).其中 t(H)表示 H 的根表中树的棵数，m(H)表示 H 中有标记结点的个数。

插入一个结点：

插入一个节点到堆中，直接将该节点插入到"根链表的 min 节点"之前即可；若被插入节点比"min 节点"小，则更新"min 节点"为被插入节点。

```
1  degree[x] ← 0
2  p[x] ← NIL
3  child[x] ← NIL
4  left[x] ← x
5  right[x] ← x
6  mark[x] ← FALSE
7  concatenate the root list containing x with root list H
8  if min[H] = NIL or key[x] < key[min[H]]
9     then min[H] ← x
10 n[H] ← n[H] + 1
```



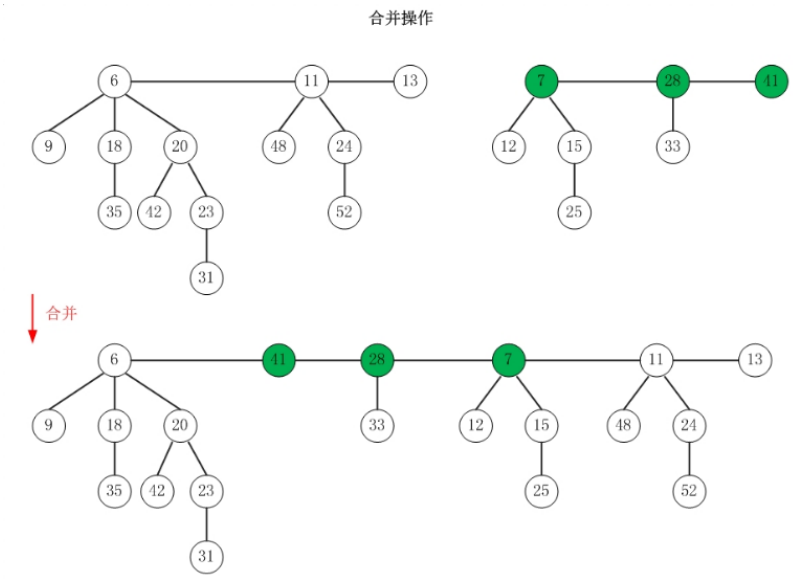
斐波那契堆的根链表是"双向链表", 这里将 min 节点看作双向联表的表头(后文也是如此)。在插入节点时，每次都是"将节点插入到 min 节点之前(即插入到双链表末尾)"。此外，对于根链表中最小堆都只有一个节点的情况，插入操作就很演化成双向链表的插入操作。
平摊代价：设 H 为输入的斐波那契堆，H’ 为插入结果，则：t(H’)= t(H) +1, m(H’)= m(H)，势的增加为((t(H)+1)+2m(H))- (t(H)+2m(H))=1,实际代价为 O(1),平摊代价为 O(1) +1=O(1)

寻找最小结点：
由于存在 min[H]这个指针，故可以在 O(1)内找到最小结点。

合并两个斐波那契堆：
不同于二项堆, 这个操作在斐波那契堆里非常简单。仅仅简单地将 H1 和 H2 的两根表串联，然后确定一个新的最小结点。

```
FIB-HEAP-UNION(H1, H2)

1   H ← MAKE-FIB-HEAP()
2   min[H] ← min[H1]
3   concatenate the root list of H2 with the root list of H
4   if (min[H1] = NIL) or (min[H2] ≠ NIL and min[H2] < min[H1])
5   then min[H] ← min[H2]
6   n[H] ← n[H1] + n[H2]
7   free the objects H1 and H2
8   return H
```



势的改变为

$\Phi(H)-(\Phi(H_1)+\Phi(H_2))=(t(H)+2m(H))-((t(H_1)+2m(H_1))+(t(H_2)+2m(H_2)))=0$，这是因为 $t(H)=t(H_1)+t(H_2)$，且 $m(H)=m(H_1)+m(H_2)$。所以，FIB-HEAP-UNION 的平摊代价与其 $O(1)$ 的实际代价相等。

抽取最小结点：

减小一个关键字

删除一个结点

最大度数的界

二、算法设计方法

1、分治法

1) 基本步骤
2) 适用条件
3) 相关算法

分治法：对于一个规模为 n 的问题，若该问题可以容易地解决（比如说规模 n 较小）则直接解决，否则将其分解为 k 个规模较小的子问题，这些子问题互相独立且与原问题形式相同，递归地解这些子问题，然后将各子问题的解合并得到原问题的解。

基本步骤：

step1 分解：将原问题分解为若干个规模较小，相互独立，与原问题形式相同的子问题；

step2 解决：若子问题规模较小而容易被解决则直接解，否则递归地解各个子问题

step3 合并：将各个子问题的解合并为原问题的解。

适用条件：

1) 该问题的规模缩小到一定的程度就可以容易地解决

2) 该问题可以分解为若干个规模较小的相同问题，即该问题具有最优子结构性质。

3) 利用该问题分解出的子问题的解可以合并为该问题的解；

4) 该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子子问题。

相关算法：
二路归并排序、快速排序

2、 动态规划

1) 基本步骤
2) 基本要素
3) 相关算法

动态规划：每次决策依赖于当前状态，又随即引起状态的转移。一个决策序列就是在变化的状态中产生出来的，所以，这种多阶段最优化决策解决问题的过程就称为动态规划。

基本思想与分治法类似，也是将待求解的问题分解为若干个子问题（阶段），按顺序求解子阶段，前一子问题的解，为后一子问题的求解提供了有用的信息。在求解任一子问题时，列出各种可能的局部解，通过决策保留那些有可能达到最优的局部解，丢弃其他局部解。依次解决各子问题，最后一个子问题就是初始问题的解。

由于动态规划解决的问题多数有重叠子问题这个特点，为减少重复计算，对每一个子问题只解一次，将其不同阶段的不同状态保存在一个二维数组中。

与分治法最大的差别是：适合于用动态规划法求解的问题，经分解后得到的子问题往往不是互相独立的（即下一个子阶段的求解是建立在上一个子阶段的解的基础上，进行进一步的求解）。
动态规划的关键点：
1、最优化原理，也就是最有子结构性质。这指的是一个最优化策略具有这样的性质，无论过去状态和决策如何，对前面的决策所形成的状态而言，余下的决策必须构成最优策略，简单来说就是一个最优化策略的子策略总是最优的，如果一个问题满足最优化原理，就称其有最优子结构性质。
2、无后效性，指的是某个状态下的决策的收益，只与状态和决策相关，与达到该状态的方式无关。
3、子问题的重叠性，动态规划将原来指数级的暴力搜索算法改进到了具有多项式时间复杂度的算法，其中的关键在于解决了荣誉，重复计算的问题，这是动态规划算法的根本目的。
4、总体来说，动态规划算法就是一系列以空间换取时间的算法。

基本步骤：

（1）描述最优解的结构。

（2）递归的定义最优解的值。

（3）按自底向上方式计算出最优值

（4）由计算出的结果，构造一个最优解

基本要素：

动态规划最重要的就是确定动态规划三要素：

（1）问题的阶段

（2）每个阶段的状态

（3）从前一个阶段转化到后一个阶段之间的递推关系。

相关算法：
0-1 背包问题

3、 贪心法

1) 基本思想
2) 基本要素
3) 相关算法
4) 理论基础——胚的定义及相关概念
5) 三种方法的相同及不同点

贪心算法 :贪心算法是指，在对问题求解时，总是做出在当前看来是最好的选择。也就是说，不从整体最优上加以考虑，他所做出的仅是在某种意义上的局部最优解。

贪心算法没有固定的算法框架，算法设计的关键是贪心策略的选择。必须注意的是，贪心算法不是对所有问题都能得到整体最优解，选择的贪心策略必须具备无后效性，即某个状态以后的过程不会影响以前的状态，只与当前状态有关。
所以对所采用的贪心策略一定要仔细分析其是否满足无后效性。
贪心算法适用的问题

贪心策略适用的前提是：局部最优策略能导致产生全局最优解。

实际上，贪心算法适用的情况很少。一般，对一个问题分析是否适用于贪心算法，可以先选择该问题下的几个实际数据进行分析，就可做出判断。

与动态规划算法相同的是，都具有最优子结构性质（相同点）；不同的是，动态规划算法通常以自底向上的方式解各子问题，而贪心算法则通常以自顶向下的方式进行，以迭代的方式作出相继的贪心选择，每作一次贪心选择就将所求问题简化为规模更小的子问题（不同点）

基本思想：

1.建立数学模型来描述问题。
2.把求解的问题分成若干个子问题。
3.对每一子问题求解，得到子问题的局部最优解。

4.把子问题的解局部最优解合成原来解问题的一个解。

基本要素：
贪心选择性质和最优子结构性质。

相关算法：
部分背包问题

理论基础——胚的定义及相关概念：
1、胚（matroids)
def：一个胚应是满足下述条件的有序对 M=(S,I)
①S 是有穷非空集
②I 是 S 的一个非空独立子集族，使得若 B∈I 且
A B，则 A∈I。满足此性质的 I 具有遗传性，即 B 独立，B 的子集亦独立，其中 Φ∈I。
③若 A∈I，B∈I 且|A|<|B|，则存在一个元素 x∈B-A，使得 A∪{x} ∈I，则称 M 具有交换性。

例如：由无向图 G=(V,E)可定义 $M_G=(S_G,I_G)$，其中
①S_G=E 为 G 中的边集
②I_G 为 S_G 的无回路边集族，即若 A∈E 则 A∈I_G⇔A 中无回路，边集 A 独立⇔子图 G_A=(V,A)是一森林。
定理 16.5
若 G 是一无向图，则 $M_G=(S_G,I_G)$是一个胚。
2、最大独立子集
给定一个胚 M=(S,I)，对于 I 中一个独立子集 A∈I，若 S 中有一个元素 x 不属于 A，使得将 x 加入 A 后仍保持 A 的独立性，即 A∪{x}∈I，则称 x 为 A 的一个可扩张元素（extension）。
当胚中的一个独立子集 A 没有可扩张元素时，称 A 是一个最大独立子集。
定理 16.6 胚中所有最大独立子集大小相等
proof：设 A 和 B 是 M 的二个最大独立子集，且|A|<|B|
由胚的交换性可知，此时存在一个元素 x∈B-A，使得 A∪{x}∈I，这与 A 是最大独立子集矛盾。
同理可证|B|<|A|时，所以|A|=|B|。
3. 加权胚
若对 M=(S,I)中的 S 给定一个权函数 w，使得对任意的 x∈S，有 w(x)>0，则称 M 为加权胚。
S 的子集 A 的权可定义为：
w(A)=∑w(x)，x∈A
4. 最优子集
使 w(A)权值达到最大的独立子集 A 称为最优子集。
例：求连通网络 G=(V,E)的最小生成树问题。
定义加权胚 $M_G=(S_G,I_G)$的权为 w’ 为：
w’ (e)=W_0-w(e)>0，其中 e∈E，W_0 为大于 G 中任一边权值的常数
令 A 是 G 的生成树，则 w’ (A)=(|V|-1)W_0-w(A)
若 w’ (A)最大则 w(A)最小，所以 A 是 G 的 MST。

三种方法的相同及不同点：

动态规划与分治法最大的差别是：适合于用动态规划法求解的问题，经分解后得到的子问题往往不是互相独立的（即下一个子阶段的求解是建立在上一个子阶段的解的基础上，进行进一步的求解）。

贪心算法与动态规划算法相同的是，都具有最优子结构性质（相同点）；不同的是，动态规划算法通常以自底向上的方式解各子问题，而贪心算法则通常以自顶向下的方式进行，以迭代的方式作出相继的贪心选择，每作一次贪心选择就将所求问题简化为规模更小的子问题（不同点）

三、算法分析

1、渐近符号

1) 定义

2) 三个符号间的关系

3) 一些已知量间的渐进关系

渐近符号：

渐近记号包括：

(1)Θ（西塔）：紧确界。 相当于"="

(2)O （大欧）：上界。 相当于"<="

(3)o（小欧）：非紧的上界。 相当于"<"

(4)Ω（大欧米伽）：下界。 相当于">="

(5)ω（小欧米伽）：非紧的下界。 相当于">"

1Θ 符号(渐近符号)

def: $\Theta(g(n)) = \{ f(n)$:存在大于 0 的常数 C_1、C_2 和 n_0，使得对于所有的 $n \geq n_0$，都有 $0 \leq C_1 g(n) \leq f(n) \leq C_2 g(n)\}$

记为：$f(n) = \Theta(g(n))$。

2. O 符号（渐近上界）

def: $O(g(n))=\{f(n)$:存在大于 0 的常数 C、n_0，使得对于所有的 $n \geq n_0$ 时，都有 $0 \leq f(n) \leq Cg(n)\}$

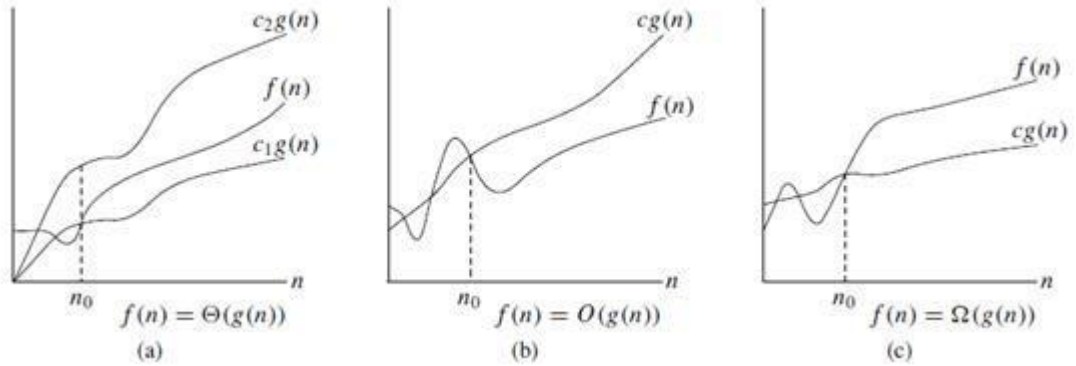
记为：$f(n)=O(g(n))$

特别提示，大 O 既可表示紧上界也可表示松上界。

3. 2.3 Ω 符号(渐进下界)

def: Ω(g(n))={f(n):存在正常数 C 和 n_0，使得对于所有 的 $n \geq n_0$，都有 $0 \leq Cg(n) \leq f(n)$}

记为：f(n) = Ω(g(n))



Θ,O,Ω 符号之间存在关系如下：

定理 3.1（P29）

对于任意的函数 f(n)和 g(n)，当且仅当 f(n)=O(g(n))且 f(n)=Ω(g(n))时，f(n)=Θ(g(n))。

性质：

函数间的比较

实数的许多关系属性可以应用于渐近比较。下面假设 $f(n)$和 $g(n)$是渐近正值函数。

传递性：

$f(n) = \Theta(g(n))$ 和 $g(n) = \Theta(h(n))$ 蕴含 $f(n) = \Theta(h(n))$

$f(n) = O(g(n))$ 和 $g(n) = O(h(n))$ 蕴含 $f(n) = O(h(n))$

$f(n) = \Omega(g(n))$ 和 $g(n) = \Omega(h(n))$ 蕴含 $f(n) = \Omega(h(n))$

$f(n) = o(g(n))$ 和 $g(n) = o(h(n))$ 蕴含 $f(n) = o(h(n))$

$f(n) = \omega(g(n))$ 和 $g(n) = \omega(h(n))$ 蕴含 $f(n) = \omega(h(n))$

自反性：

$$f(n) = \Theta(f(n)) \quad f(n) = O(f(n)) \quad f(n) = \Omega(f(n))$$

对称性：

$$f(n) = \Theta(g(n)) \text{ 当且仅当 } g(n) = \Theta(f(n))$$

一些已知量间的渐进关系：

2、 传统分析方法

1) 基本思想

2) 最好、最坏和平均情况下的时间复杂度分析

3、 递归算法的分析方法

1) T(n)的一般式

2) 替换法

3) 递归树方法

4) Master 方法

递归算法的分析方法（第 2 次课的 ppt）

T(n)的一般式：

$$T(n) = \begin{cases} \Theta(1) & n \le C \\ aT(\frac{n}{b}) + D(n) + C(n) & n > C \end{cases}$$

其中：Θ(1)表示常数时间

a为分解后的子问题个数

$1/b$为分解后子问题与原问题相比的规模

$D(n)$表示分解子问题所花费的时间

$C(n)$表示合并子问题解所花费的时间

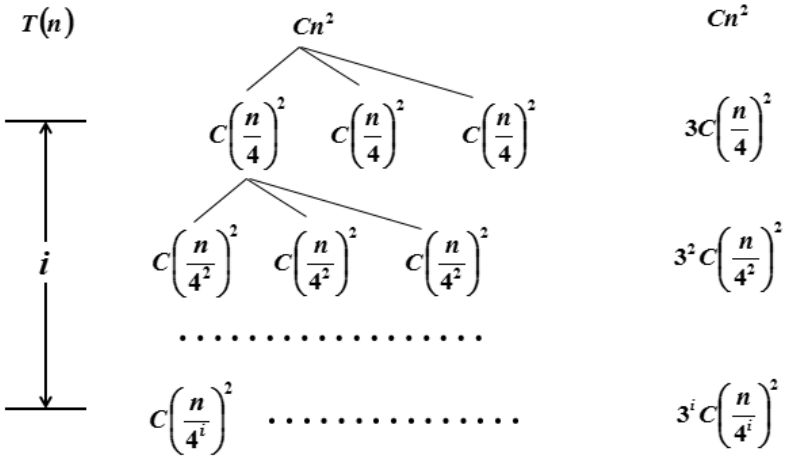
替换法：

思想：首先对 T(n)的解做一个猜测，然后用归纳法证明所做猜测是否正确。

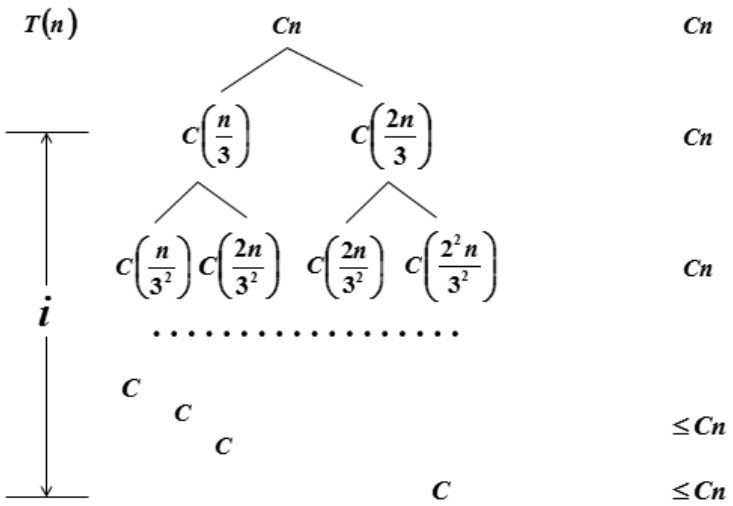
这种方法适用于问题的形式相对简单或比较熟悉，或者问题的解较易猜测的情况。

递归树方法：

例 3：$T(n)=3T(n/4)+cn^2$



例 4：$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + cn$



Master 方法：

条件：如果 T(n)具有如下递归形式：

T(n)=aT(n/b)+f(n)，其中 a≥1，b>1 且为常数，f(n)为渐进函数

定理 4.1 Master 定理

对于非负递归函数 T(n)=aT(n/b)+f(n)，其中 a ≥1，b>1 且为常数，f(n)为渐进函数，则 T(n)有如下三种渐进界限：

1) if $f(n) = O\left(n^{\log_b a-\varepsilon}\right)$, 常数 $\varepsilon > 0$，则$T(n) = \theta\left(n^{\log_b a}\right)$

2) if $f(n) = \theta\left(n^{\log_b a}\right)$, 则$T(n) = \theta\left(n^{\log_b a} \times \lg n\right)$

3) if $f(n) = \Omega\left(n^{\log_b a+\varepsilon}\right)$, 常数 $\varepsilon > 0$且∃常数$c < 1$, n足够大时，

有$af\left(\frac{n}{b}\right) \le cf(n)$, 则$T(n) = \theta\left(f(n)\right)$

直观上看，*master*定理就是比较二个函数$f(n)$和$n^{\log_b a}$的渐进大小,即：

if $n^{\log_b a} > f(n)$ *then* $T(n)$适用条件1）

if $n^{\log_b a} = f(n)$ *then* $T(n)$适用条件2）

if $n^{\log_b a} < f(n)$ *then* $T(n)$适用条件3）

例5: $T(n) = 9T\left(\frac{n}{3}\right) + n$

例6 ： $T(n) = T\left(\frac{2n}{3}\right) + 1$

例7 ： $T(n) = 3T\left(\frac{n}{4}\right) + n \lg n$

例8 ： $T(n) = 2T\left(\frac{n}{2}\right) + n \lg n$

4、平摊分析方法

1) 基本思想
2) 合计法
3) 记账法
4) 势函数方法
5) 动态表分析

平摊分析（第 9 次课的 ppt）：
基本思想：
平摊分析是指在某种数据结构上完成一系列操作，在最坏情况下所需的平均时间。
平摊分析与传统分析方法的主要差别为：
①平摊分析时间与传统分析方法的平均情况下时间不同，它是最坏情况下的平均时间
②平摊分析不涉及概率分析
③平摊分析中时间函数 T(n)，其中 n 指的是操作数，而不是输入的规模

合计法：
合计法的思想为：把 n 个不同或相同的操作合在一起加以分析计算得到总时间的方法。即 n 个操作序列在最坏情况下的总时间，记为 T(n)，其中 n 为操作数。
由此得到在最坏情况下每个操作的平均时间为 T(n)/n。

1. 栈操作的平摊分析(不同类操作)

数据结构：栈 S，初始为空

操作：		实际代价
push(S,x)：	对象 x 进栈	O(1)
pop(S)：	从 S 中弹出一个元素	O(1)
multipop(S,k)：	从 S 中连续弹出 k 个元素	O(min{|S|,k})

2. 二进制计数器(同类操作)

数据结构：数组 A[0..k-1]，即是一 k 为二进制计数器，计　　数器初始值为 0
操作：加 1
算法：Increment(A)

```
i←0
while i<length[A] and A[i]=1
do      A[i]←0
    i←i+1
if i<length[A]   then A[i]←1
```

记账法：
记帐法思想：先对每个不同的操作核定一个不同的费用（时间），然后计算 n 次操作总的费用。
这种事先的操作费用核定可能与实际费用不符，存在二种情况：
超额收费：核定费用>实际费用，此时差额存放在该对象的身上
收费不足：核定费用<实际费用，此时差额由该对象身上的存款支付
费用的核定(平摊核定)是否正确需检查 n 次操作核定总费用是否大于 n 次操作实际总费用

若令C_i为第i次操作的实际费用

$\hat{C}_i$ 为第i次操作的平摊费用

那么要求：

$$\sum_{i=1}^{n} \hat{C}_i \geq \sum_{i=1}^{n} C_i \quad \text{（对任意的n）}$$

或改写为：

$$\sum_{i=1}^{n} (\hat{C}_i - C_i) \geq 0 \quad \text{（对任意的n）}$$

势函数方法：

势能法(potential method)

思想：通过定义一个势函数完成对每个操作的平摊核定

令 n 个操作的数据结构为 D，数据结构的初态为 D_0

C_i：表示第 i 次操作的实际成本

$C_i^{\wedge}$：表示第 i 次操作的平摊成本

D_i：表示在数据结构状态为 D_{i-1} 上完成第 i 次操作后的数据结构状态，op_i

即：　D_{i-1}=====>D_i，i=1,2,…,n

Φ：势函数

Φ(D_i)：表示将 Di 映射为一实数，这个实数值称为势能

用势函数 Φ 表示第 i 次操作的核定费用 $C_i^{\wedge}$定义如下：

$C_i^{\wedge}$= C_i+Φ(D_i)-Φ(D_{i-1})= C_i+Φ_i-Φ_{i-1}

其中 Φ_i-Φ_{i-1}称为势差，势差的不同结果类似与记帐法：

Φ_i-Φ_{i-1}>0：超额收费

Φ_i-Φ_{i-1}<0：收费不足

如同记帐法需对操作的平摊核定进行验证，为保证势函数的正确性，也需验证平摊总费用是否是实际总费用的上界，即：

$$\sum_{i=1}^{n} \hat{C}_i = \sum_{i=1}^{n} \left(C_i + \Phi_i - \Phi_{i-1}\right) = \sum_{i=1}^{n} C_i + \Phi_n - \Phi_0$$

∵ 要求$\sum_{i=1}^{n} \hat{C}_i \geq \sum_{i=1}^{n} C_i$

∴ 对任意的n，要求$\Phi_n - \Phi_0 \geq 0$

动态表分析：

表扩张(table expansion)：发生在插入一个元素 x 时，此时表满没有空间存放 x，所以将引起表扩张。表扩张的步骤为：

①申请一块更大表

②拷贝原表到新表中

③释放原表

④在新表中插入 x
表收缩（table contraction)：发生在删除一个元素 x 后，表中元素个数小于某个设定值时，引起表收缩。表收缩步骤为：
①删除元素 x
②申请一块更小的表
③拷贝原表到新表中
④释放原表
定义一个装填因子(load factor)α(T)描述动态表当前的状态：
①非空表：α(T)=num[T]/size[T]　　0≤α≤1
num[T]：表中元素个数，size[T]：表的大小
②空表：size[T]=0，此时定义 α(φ)=1
注意空表指的是表的大小为 0 的情况

5、 算法正确性分析
1) 循环不变式
2) Cut and paste 方法
3) 归纳法
4) 贪心选择性质

算法正确性分析：
循环不变式(loop Invariant)方法:先确定算法的循环不变式，然后检查此循环不变式在算法执行期间是否满足以下三个条件：
①初始状态(Initialization):循环前循环不变式为真
②保持状态(Maintance):如果本次循环前循环不变式为真，则本次循环后依然为真
③ 终止状态(Termination):退出循环时，循环不变式为真

Cut and paste 方法：

四、最大流
1、 流、流网络定义
2、 Ford-Fulkerson 方法
3、 剩余网络
4、 增广路径
5、 截
6、 相关算法及时间

最大流（第 13 次课 ppt）：
流、流网络定义：
流：在图 G 中，流是一实函数 f：V×V→R，并满足以下三个条件：
容量约束(capacity constraint)：

对于所有的 u,v∈V，f(u,v)≤c(u,v)

反号对称(skew symmetry):

对于所有的 u,v∈V，f(u,v)= -f(v,u)

流网络 G=(V,E)为一有向图，图中每条边(u,v)∈E 均为非负容量 c(u,v)≥0。如果(u,v)不属于 E，则 c(u,v)=0。

在流网络中有两个特殊的顶点称为源(source)和汇(sink),分别用 s 和 t 表示。

某个顶点的总净流定义为：

净流 = 流出总正流 − 流入总正流

除了源和汇外，顶点的净流值等于 0，所以流守恒可表述为 “流进 = 流出”。

Ford-Fulkerson 方法：

- Ford-Fulkerson-Method(G,s,t)
- 初始化流 f=0
- while 存在增广路径 P
- do 沿增广路径 P 增加流 f
- return f
- Ford-fulkerson 方法需解决以下几个问题：
- ①二个流相加是否仍是一个流及如何构造流？
- ②如何沿增广路径增加流？
- ③在流网络中没有了增广路径是否得到了最大流？

剩余网络：

- 所谓剩余网络就是在流网络中除去某个流后，所有剩余容量(residual capacity)组成的网络。
- 对于二个顶点 u,v 间的剩余容量定义为：
- $c_f(u,v)=c(u,v)-f(u,v)$
- 则剩余网络可定义为：
- 对于给定的流网络 G=(V,E)以及流 f，则 $G_f(V,E_f)$是相对与流 f 的剩余网络。其中 $E_f=\{(u,v)\in V\times V：c_f(u,v)>0\}$。
- E_f的构成：
- ① if f(u,v)<c(u,v)
- ② if f(u,v)>0
- 由流网络到剩余网络的构造见 P401，图 26-3
- 原流网络中边数与剩余网络的边数满足关系：$|E_f|\leq 2|E|$
- 引理 26.2
- 令 G=(V,E)是一个源为 s，汇为 t 的流网络，f 是 G 中的一个流。令 G_f是相关于 f 的剩余网络，f’ 是 G_f的一个流，则 f+f’ 也是 G 中的一个流且流值|f+f’ |=|f|+|f’ |。

增广路径：

- 所谓增广路径是指在 G_f中一条从 s 到 t 的简单路径 。根据剩余网络的定义，在增广路径中的每条边在不违背容量约束的条件下可以增加一定量的正流。如图 26.3(b), P401

- 在一条增广路径 P 中可以增加流的最大量称为增广路径 P 的剩余容量，定义为：
- $c_f(P)=\min\{c_f(u,v):(u,v)\in P\}$
- 引理 26.3
- 令 f 是流网络 G=(V,E)的一个流，令 P 是剩余网络 G_f 的一条增广路径，定义函数 f_p：$V\times V\rightarrow R$，并有：
- $$f_p(u,v)=\begin{cases} c_f(P) & \text{if } (u,v)\in P \\ -c_f(P) & \text{if } (v,u)\in P \\ 0 & \text{otherwise} \end{cases}$$
- 那么 f_p 是 G_f 的一个流且流值为$|f_p|=c_f(P)>0$
- 推论 26.4
- 令 f 是流网络 G=(V,E)的一个流，P 是 G_f 中的一条增广路径，f_p 如上定义。定义函数 $f':V\times V\rightarrow R$ 且 $f'=f+f_p$，那么 f' 是 G 中的一个流，其流值$|f'|=|f|+|f_p|>|f|$。

截：

- 流网络的截(S,T)是顶点 V 的一种划分，其中 T=V-S，$s\in S$，$t\in T$。
- 相关于截(S,T)的表述：
- ①流函数 f(S,T)：穿越截(S,T)的净流，即流出总正流－流入总正流。
- ②C(S,T)：截(S,T)的容量，即从 S 到 T 的容量之和。
- ③最小截：所有截中容量最小的截
- 如 P403，图 26.4
- 例：截 $S=\{s,v_1,v_2\}$，$T=\{v_3,v_4,t\}$
- f(S,T)=19，c(S,T)=26
- 例：截 $S=\{s,v_1,v_2,v_4\}$，$T=\{v_3,t\}$
- f(S,T)=19，c(S,T)=23
- 引理 26.5
- 令 f 是源为 s 汇为 t 的流网络 G 中的一个流，(S,T)是 G 的一个截，那么穿越截(S,T)的净流 f(S,T)=|f|。
- 推论 26.6
- 流网络 G 中的任何流值均受限于 G 中截的容量限制。
- 定理 26.7
- 如果 f 是源为 s 汇为 t 的流网络 G 中的一个流，那么以下条件相等：
- ①f 是 G 中的最大流
- ②剩余网络没有增广路径
- ③G 中存在截(S,T)，使|f|=C(S,T)
- 由最大流最小截定理可对 Ford-Fulkerson 方法细化，算法见书 P404 及图例 26.5

相关算法及时间：

Edmonds-Karp 算法是 Ford-Fulkerson 方法的一种具体实现。它采用 BFS 搜索方法在剩余网络中寻找一条增广路径。具体做法是先对每条边的权值统一赋值为 1，然后找一条从 s 到 t 的最短路径。基于此方法，Edmonds-Karp 算法求最大流的时间为 $O(VE^2)$。

上课老师说的话：
排除：
循环不变式不作要求
分析算法的平均时间不作具体要求（只考最好时间或最坏时间）
递归算法中　替换法和递归树方法不作要求 主要掌握 master 方法
红黑树 基本性质、操作时间为重点 插入删除结点不作要求
贪心方法中 理解基本概念（包括胚、最优子集、最大独立子节）
平摊分析方法中 只掌握势函数方法（定义、分析、验证）验证很重要

红黑树删一个节点，时间复杂度多大
二项堆完成一次抽取操作后是怎样的
用 Ford-Falkerson 方法得到的最大流或最小截
选择填空判断（20-30 分）
快排最坏时间
哈夫曼是一种贪心算法
综合题（简答、算法分析）（50-60 分）
动态规划的基本步骤
平摊分析
求最大流
红黑树的概念
算法设计（20-30 分）
用贪心法求解、证明等