CS158 《数据结构（荣誉）》 期末复习整理

Galaxies

2019.6

Shanghai, China

1 引言

1.1 算法的运行时间函数$T(n)$，若存在两个正常数c, N_0，使得$N \geq N_0$时$T(N) \leq cF(N)$，则记为$T(N) = O(F(N))$。大 O 表示法称为（渐进）时间复杂度。

- 若存在正常数c, N_0，使得$N \geq N_0$时有$T(N) \geq cF(N)$，则记为$T(N) = \Omega(F(N))$;
- 若$T(N) = O(F(N))$且$T(N) = \Omega(F(N))$，称$T(N) = \Theta(F(N))$;
- 若$T(N) = O(F(N))$但$T(N) \neq \Omega(F(N))$，称$T(N) = o(F(N))$.

1.2 求和定理和求积定理

1.2.1 $T_1(n) = O(f(n)), T_2(n) = O(g(n))$，则$T_1(n) + T_2(n) = O(\max(f(n), g(n)))$;

1.2.2 $T_1(n) = O(f(n)), T_2(n) = O(g(n))$，则$T_1(n)T_2(n) = O(f(n)g(n))$

2 线性表

2.1 线性表的顺序实现（void doubleSpace(); 空间扩大一倍）。

2.2 线性表的链接实现（单链表，双链表，循环链表（单循环，双循环））

操作	顺序实现复杂度	链接实现复杂度
insert()	O(n)	O(1) *
remove()	O(n)	O(1) *
visit()	O(1)	O(n)
search()	O(n)	O(n)
traverse()	O(n)	O(n)
doubleSpace()	均摊 O(1)	\
注：“*”表示需要先定位到需要进行操作的位置		

表 2.1 线性表的两种实现方式复杂度分析

- doubleSpace() 复杂度的均摊分析：自上一次扩容后，设目前数组大小为 M，则再插入 $M/2$ 个元素才会触发 doubleSpace() 进行扩充，单次 doubleSpace() 的复杂度是 O(M) 的，于是将此复杂度均摊到 $M/2$ 个 insert 操作，即可知道均摊每次 O(1)。

3 栈

3.1 基本定义（栈，栈顶，栈底，栈顶元素，进栈，入栈，出栈，空栈）与性质（**先进先出**）。

3.2 栈的顺序实现（顺序栈），栈的链接实现（链接栈，使用单链表即可）。

·各类操作的复杂度均为 O(1)，其中顺序栈的 doubleSpace() 与前相同，均摊 O(1)。

3.3 栈的应用

3.3.1 系统内部的递归过程是栈堆叠的过程（调用进栈，结束出栈）。

3.3.2 括号匹配 (p82)（构造括号栈，检查是否匹配）

3.3.3 表达式计算 (p92)（中缀表达式转换为**后缀表达式**(逆波兰式)，计算后缀表达式）

- 中缀转后缀：从左至右读入表达式的各个运算数和运算符，读入运算数直接输出，读入运算符先利用栈存起来，如果后一个运算符比他优先级低，则输出；遇到开括号进栈，遇到闭括号，将栈中开括号后的所有内容输出，并将开括号出栈，最后将所有操作符出栈。运算符复杂度：乘方>乘除>加减。
- 后缀求值：利用栈，读入数则进栈，读入符号则将栈中前两个数出栈并进行操作，再将

得出的新数进栈即可，直到数字栈仅剩一个数（或后缀表达式结束）。

目前表达式	栈	输出
2*(2+3*5^6+7)+1	空（起始状态）	空（起始状态）
2+3*5^6+7)+1	*(	2
3*5^6+7)+1	*(+	2 2
5^6+7)+1	*(+*	2 2 3
6+7)+1	*(+*^	2 2 3 5
7)+1	*(+	2 2 3 5 6 ^ * +
1	+	2 2 3 5 6 ^ * + 7 + *
空	空	2 2 3 5 6 ^ * + 7 + * 1 +

表 3.1 中缀转后缀举例

目前中缀表达式	（数字）栈
2 2 3 5 6 ^ * + 7 + * 1 +	空（起始状态）
^ * + 7 + * 1 +	2 2 3 5 6
* + 7 + * 1 +	2 2 3 15625
+ 7 + * 1 +	2 2 46875
+ * 1 +	2 46877 7
* 1 +	2 46884
+	93768 1
空	93769（答案）

表 3.2 后缀求值举例

3.3.4 * 双栈共享向量空间：有两个栈，一个存的东西很多，一个很少，可以将他们存在同一个空间内，一个下标从 0 开始往上存储，一个下标从 MaxSize-1 开始向下存储，这样可以节省存储空间同时防止栈溢出（上溢）

4 队列

4.1 基本定义（队头，队头元素，队尾，队尾元素，空队列，队列长度，出队）与性质（**先进先出**）。

4.2 队列的顺序实现（队头位置固定，队头位置不固定，循环队列）

4.2.1 队头位置固定：则对于队头的操作的复杂度变为 O(n)；

4.2.2 队头位置不固定：空间利用率低，有空间浪费；

4.2.3 循环队列

- front, rear 在操作时候均需要对于 MaxSize 取模，且课本利用左开右闭实现，即(front, rear];
- 循环队列满和循环队列空的判据都是 front==rear，因此需要牺牲一个单元，即 front 指向的单元不能存元素，故队列空判据：front==rear，队列满判据(rear+1)%MaxSize==front.
- 除上述特殊说明时间复杂度外，队列基本操作时间复杂度均为 O(1)；

4.3 队列的链接实现（单链表，单循环链表）

4.4 队列的应用

4.4.1 火车重排问题（每个缓冲轨道是队列，缓冲轨道内按照从小到大次序排列）

4.4.2 单服务台排队系统模拟(离散事件模拟系统)(离散的时间驱动模拟 vs **事件驱动模拟**)

5 字符串

5.1 字符串的顺序实现和链接实现、基本操作

- 基本操作：length(), equal()等比较操作，display()，字符串赋值 copy(s1,s2)，字符串连接 cat(s1, s2)，取子串 substr(s, start, len)，字符串插入 insert(s1, start, s2)，子串删除 remove(s, start, len)。

5.2 字符串的匹配（**KMP 算法**）

- 用两个指针 i,j 表示 A[i-j … i]和 B[0…j]完全匹配，当 j=m-1 时说明匹配成功。每次匹配 A[i+1]和 B[j+1]，如果相同则 i++,j++；否则减小 j 值使得 A[i-j … i]和 B[0 … j]仍然完全匹配，然后继续尝试匹配 A[i+1]和 B[j+1]。
- 关键在于：j 值减小的策略。j 值减小到的值事实上仅和 B 串相关，定义数组 P[j]表示 B 数组的前 j 个字母与 A 数组匹配而 j+1 个字母失配时新的 j 应该是多少。实际上，可以理解为**最长的前后缀相等的长度**。p 数组实际上就是一个自我匹配的过程。
- 可参见书 p151 代码，代码中 p 数组的具体实现为最长的前后缀相等的长度减 1.
- j 减小的次数不超过 j 增加的次数，而 j 增加的次数最多为 n，则 j 减小的次数最多也为 n，因此复杂度为 O(n)。

6 树

6.1 树的定义（递归定义），基本术语（根结点，叶结点，内部结点，度（**直接后继的数量**），子结点，父结点，祖先结点（到根路径），子孙结点（子树），兄弟结点，层次（**根结点为第 1 层**），高度（最大层次），有序树（儿子有顺序），无序树，森林（树的集合））

6.2 二叉树的定义（递归定义）（**有序树**），基本术语（满二叉树（即丰满树），完全二叉树），顺序实现（利用 6.3 性质 5 的完全二叉树标号性质），链接实现（标准存储结构（二叉链表存左右儿子），广义标准存储结构（三叉链表存左右儿子、父亲））。

6.3 二叉树的常用性质及简单证明

【性质 1】一棵非空二叉树第 i 层上最多有 2^{i-1} 个结点。（归纳）

【性质 2】一棵高度为 k 的二叉树，最多具有 2^k-1 个结点。（利用性质 1）

【性质 3】对于非空二叉树，如果叶子结点度数为 n_0，度为 1 的结点数为 n_1，度为 2 的结点数为 n_2，则 n_0=n_2+1。（利用结点数 n=n_0+n_1+n_2 与树枝数 b=n-1=n_1+2n_2 联立解得）

【性质 4】具有 n 个结点的完全二叉树高度$k = [\log_2 n] + 1$。（利用性质 2）

【性质 5】如果对一棵有 n 个结点的完全二叉树的结点按层自上而下，每一层从左到右编号，若根结点编号为 1，则对任一编号为 i 的结点$(1 \le i \le n)$，有：（归纳）

(1) 如果 i=1，则该结点为二叉树的根；否则其父亲编号为$[i/2]$；

(2) 如果 2i>n，则编号为 i 的结点为叶子结点，没有儿子，否则其左儿子编号为 2i；

(3) 如果 2i+1>n，则编号为 i 的结点无右儿子，否则其右儿子编号为 2i+1。

6.4 二叉树的遍历

6.4.1 前序遍历（根左右），中序遍历（左根右），后序遍历（左右根）的递归实现，层次遍历的队列实现。

6.4.2 已知前序、中序确定二叉树及后序；已知中序、后序确定二叉树及前序（若仅前序后序无法确定）

6.4.3 二叉树遍历非递归的实现

6.4.3.1 前序遍历：开始仅加入树根，每次从栈中取出结点并输出值，按次序将该结点的右

子树、左子树加入栈中，重复该过程。

6.4.3.2 中序遍历：由于根在访问完左子树才访问，因此根结点第一次出栈后，依次将根结点、左子树加入栈中，第二次出栈时候输出根结点并加入右子树。因此需要有 TimesPop 记录已有的出栈次数。

6.4.3.3 后序遍历：同理，根结点第一次出栈后将根结点、左子树加入栈中，第二次出栈时将根结点、右子树加入栈中，第三次出栈输出。同样需要 TimesPop 记录。

6.5 二叉树的应用（表达式的计算）：中缀表达式建立表达式树（顺序扫描到优先级最低的（若有多个，由左结合特性选择最后一个）作为根，然后分左右子树递归分解），后序遍历得到结果。

6.6 **哈夫曼树**

6.6.1 等长编码和不等长编码，编码的代价$cost = \sum_i L_i w_i$。前缀编码：不等长编码，每个字符的编码和其他字符编码的前缀不同。

6.6.2 哈夫曼算法：维护一个森林，每棵树的权值由所有叶结点出现频率之和组成。每次选出权值最小的两棵树 $T1$,$T2$，将 $T1$ 和 $T2$ 作为两棵子树构建新树 T，新的树的权值为 $T1$ 和 $T2$ 权值和。直到只剩下一棵树，则每个字符的编码为从根结点走到该字符所在结点的路径，定义左支为 0，右支为 1。（事实上，如果编码集合不仅限于{0,1}，而是{0,1,2,…,k-1}可以将两树合并变为 k 树合并，最终形成一棵 k 叉树）。哈夫曼算法可利用**优先队列优化**。

6.7 树和森林

6.7.1 树的存储实现：孩子链表示法（链表存储每个结点的所有孩子），孩子兄弟链表示法（左指针指第一个儿子，右指针指向兄弟，根没有右指针，**可将任何树转化为二叉树**），双亲表示法（仅存储父亲，应用如不相交集）。

6.7.2 树的遍历：仅定义前序遍历（先根然后按顺序对每棵子树前序遍历），后序遍历（先按顺序对每棵子树后序遍历然后根），层次遍历。（注：两个按顺序是相同的顺序）

6.7.3 森林转二叉树：先将每棵树转二叉树，然后由于孩子兄弟链表示法根没有右指针，因此利用根的右指针与另外一棵树的根相连，即合并完成两棵树，此时另一棵树的根仍然没有右指针，因此可以继续合并。依次合并完所有树即可得到二叉树。

7 优先级队列

7.1 基于线性表的优先队列（无意义，要么进队 O(n)，要么出队 O(n)）

7.2 **二叉堆**和 D 堆

7.2.1 特性：结构性(完全二叉树)、有序性(结点大于/小于子孙)；

7.2.2 入堆操作：先放在下一个可用位置，然后向上过滤（向上检查是否符合二叉堆的要求，不符合则交换），找到应该插入的位置，O(log$N$)；

7.2.3 出堆操作：将第一个元素和最后一个元素交换位置，然后将第一个元素（指交换过后的）向下过滤（找儿子中较小的交换），O(log$N$)；

7.2.4 建堆操作：直接对于除叶子外的每个点进行向下过滤（从下往上进行），时间复杂度 O(n)（注：代码中 percolateDown(x)即为向下过滤）；证明即对树上的每个点深度求和即得，如下述【定理】所述。

【定理】对于一棵高度为 h，包含了 n 个结点的满二叉树，向下过滤中交换次数的和的最大

值为 n-h。（证明为级数求和，参考 p220）

7.2.5 D 堆，即每个结点最多有 D 个儿子的堆。入堆操作复杂度为$O(\log_D N)$，但是向下过滤中找最小儿子需要 $O(D)$，因此向下过滤总复杂度$O(D\log_D N)$；选取 D 为 2 的次幂可以使用位运算加速；主要应用于插入比删除多很多的应用，或是需要存储在外存的应用场景中。

7.3 归并优先级队列

7.3.1 左堆（**左偏树**）

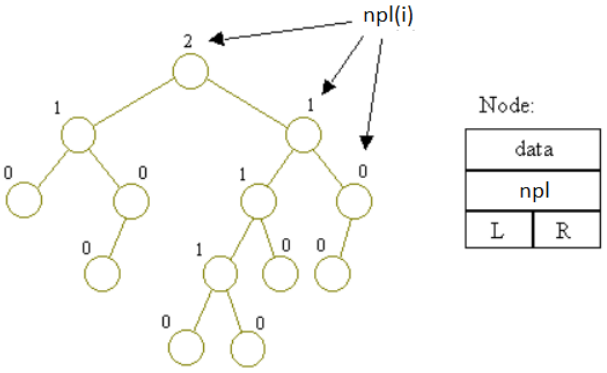

图 7.1 左堆（左偏树）的实例以及存储方式

7.3.1.1 空路径长度 *npl*(*x*)：结点 *x* 到不满两个孩子的结点的最短路径，具有 0 或 1 个孩子的 *npl* 为 0，空结点 *npl* 为-1.

7.3.1.2 左堆：每个结点的左孩子的空路径长度大于等于右孩子的空路径长度，即 *npl*(*leftson*) ≥*npl*(*rightson*)。

7.3.1.3 归并操作（小根堆）：将根结点大的堆与另一个堆的右子树归并，如产生的堆违反了左堆定义，则交换左右子树；若某个堆为空，另一个堆就是合并的结果，终止合并过程。合并过程需要更新 *npl* 的值。可以证明复杂度为 O(log*N*)。

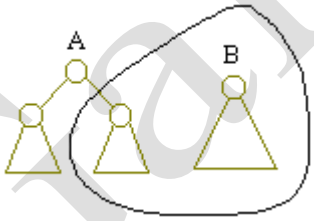

图 7.2 合并过程 1（合并 B 与 A 的右子树）

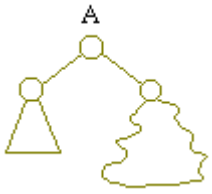

图 7.3 合并过程 2（npl 不满足则交换）

```
Merge (A,B)
     if A = NULL return B
     if B = NULL return A
     if key(B) < key(A)
          swap A, B
     right(A) ← Merge(right(A),B)
     if dist(right(A)) > dist(left(A))
          swap right(A), left(A)
     if right(A) = NULL
          dist(A) ← 0
     else
          dist(A) ← 1 + dist(right(A))
     return A
```

图 7.4 左偏树合并操作伪代码

7.3.1.4 插入操作（归并已有堆和一个仅一个结点的堆）、删除操作（删除根，将根的左右子树归并）

7.3.2 斜堆(无特别意义):每完成一次归并就交换左右孩子,防止路径过长。平均 O(logN),最坏可达到 O(n)。

7.3.3 二项堆（Bernoulli 堆）：森林表示，每棵树为二项树，满足堆的有序性。高度为 0 的二项树为一个结点，高度为 k 的二项树是将一棵高度为 k-1 的二项树加到另一棵高度为 k-1 的二项树的根上，如图 7.5。因此高度为 k 的二项树的结点个数为 2^k。一个 n 个元素的二项堆至多含有$[\log_2 n]+1$棵二项树。合并操作为两个森林的合并，两棵高度为 k 的树可以合并成一棵高度为 k+1 的树，合并过程中最多出现 3 棵高度相同的树，此时任选两棵合并即可，合并复杂度 O(log$N$)。查找最小值复杂度 O(log$N$)（即取二项树中根结点的最小值）.

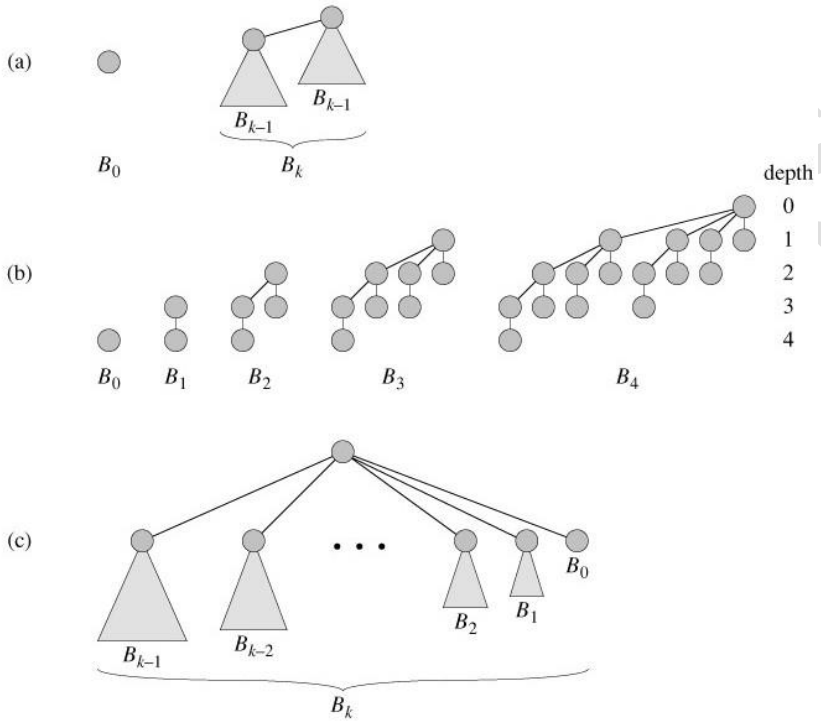

图 7.5 二项堆的实例

7.4 优先队列的应用：排队系统(p227)。

8 集合与静态查找表

8.1 一些概念（键值（关键字值），查找表，多重集（有重复关键字值），静态查找表（元素不变且个数不变），动态查找表，内部查找（元素存放在内存储器），外部查找，记录（外部查找中数据元素））

8.2 顺序查找：平均查找时间 N/2，复杂度 O(N)。

8.3 有序表的查找：

8.3.1 二分查找

8.3.2 差值查找：用下面公式来估计被查找元素的位置:（即假设平均分布来估计）

$$\text{next} = \text{low} + \left[\frac{x - a[\text{low}]}{a[\text{high}] - a[\text{low}]}(\text{high} - \text{low} - 1)\right]$$

8.3.3 分块查找：查找索引&查找块，现在查找索引中查找具体在哪一块，再在块内进行查找。（查找均可以使用顺序查找或二分查找，若均使用顺序查找则块大小为$M=\sqrt{n}$最优，单次查找复杂度为$O(\sqrt{n})$。

9 动态查找表（主要分为查找树、散列表）

9.1 二叉查找树（递归定义，左子树<结点值<右子树（有序性））

9.1.1 中序遍历二叉查找树所得到的访问序列是按照键值的递增次序排列的。

9.1.2 查找和插入时判断与根结点的大小关系进入左子树或右子树继续操作。

9.1.3 删除时分类：(1) 若无孩子则直接删除；(2) 若有一个孩子用这个孩子代替该结点并删除该结点；(3) 若有两个孩子则从其左子树最大值和右子树最小值中随便选取一个与其交换，交换后待删除结点必定只有 1 个或 0 个孩子，利用(1)、(2)删除即可。

9.1.4 平均查找时间 1.38log*N*，平均复杂度 O(log*N*)，最坏复杂度 O(*n*)。

9.2 **AVL 树**

9.2.1 要求左右子树的高度差不超过 1，利用**平衡因子**衡量结点的平衡度，**平衡因子**定义为左子树高度减去右子树高度，按照定义，AVL 树的每个结点的平衡因子只能为±1，0（存储时记录每个结点高度即可得知平衡因子）。则可以通过下述定理得知 AVL 树的高度是 O(log*N*)级别的。

【定理】一棵由 *n* 个结点组成的 AVL 树的高度小于等于 $1.44\log(N+1)-0.328$.

简略证明：令 *S*(*H*)为高度为 *H* 的规模最小的 AVL 树的规模，于是 *S*(*H*)=*S*(*H*-1)+*S*(*H*-2)+1 (*H*>2)，利用 *S*(*H*)与 Fibonacci 数的性质以及 Fibonacci 数的通项公式即得。

9.2.2 AVL 树的插入

9.2.2.1 LL 情况：原来 A 左子树比右子树高，插入在 A 左儿子 B 的左子树中使得 A 左儿子 B 的高度增加。此时将 A 与 B 调换位置并进行相对应的交换即可(即进行 LL 旋转)。

9.2.2.2 RR 情况：LL 情况的镜像对称。

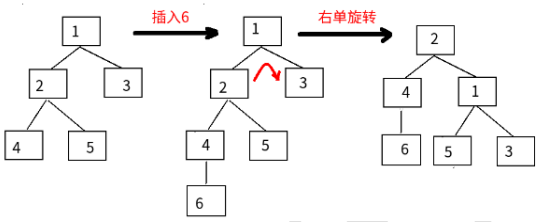

图 9.1 LL 旋转（右单旋转）

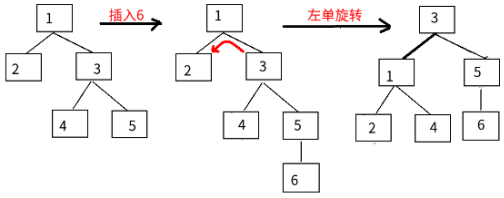

图 9.2 RR 旋转（左单旋转）

9.2.2.3 LR 情况：原来 A 左子树比右子树高，插入在 A 左儿子 B 的右子树 C 中使得 A 左儿子 B 的高度增加。此时先将 C 和 B 调换位置，再将 C 和 A 调换位置即可（旋转两次，第一次 B 进行 RR 旋转，然后 A 进行 LL 旋转）。

9.2.2.4 RL 情况：LR 情况的镜像对称。

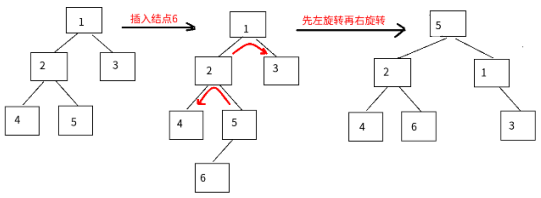

图 9.3 LR 旋转（先 RR 后 LL）

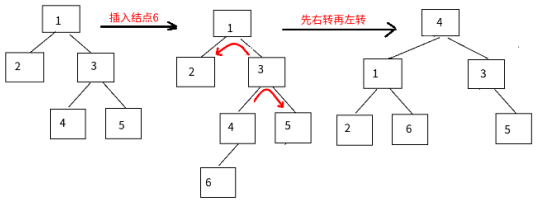

图 9.4 RL 旋转（先 LL 后 RR）

9.2.2.5 插入总结：按照二叉查找树的方式插入，然后检查插入路径上是否出现不平衡的情况，如果是进行调整（上述四种旋转）即可。

9.2.3 AVL 树的删除

9.2.3.1 与二叉查找树相同，分(1)(2)(3)三种情况讨论，但是删除结点后可能使得一些结点不平衡，于是需要旋转调整（同插入一样需要在到根的路径上依次检查）。

9.2.3.2 情况 1：P 原来平衡因子为 0，左子树变矮后导致平衡因子变为-1。无需调整，且由于 P 高度不变，不用检查 P 结点父亲平衡性。

9.2.3.3 情况 2：P 原来平衡因子为 1，左子树变矮后导致平衡因子变为 0，无需调整，但是由于 P 高度变化，需递归检查 P 结点父亲平衡性。

9.2.3.4 情况 3：P 原来平衡因子为-1，左子树变矮后导致平衡因子变为-2。

9.2.3.4.1 情况 3.1：P 右子树 Q 的平衡因子为-1，即 Q 左子树比右子树矮。对 P 进行一次 RR 旋转即可平衡，但是树高减小，需要继续检查父亲。如图 9.5 所示。

9.2.3.4.2 情况 3.2：P 右子树 Q 的平衡因子为 1，即 Q 左子树比右子树高，则对 P 进行 RL 旋转即可（先对 Q 进行 LL 旋转，再对 P 进行 RR 旋转），旋转后高度减小，需要继续检查。如图 9.6 所示。

9.2.3.4.3 情况 3.3：P 右子树 Q 的平衡因子为 0，即 Q 左右子树高度相同。对 P 进行 RR 旋转即可，旋转后高度不变，不需要继续检查。如图 9.7 所示。

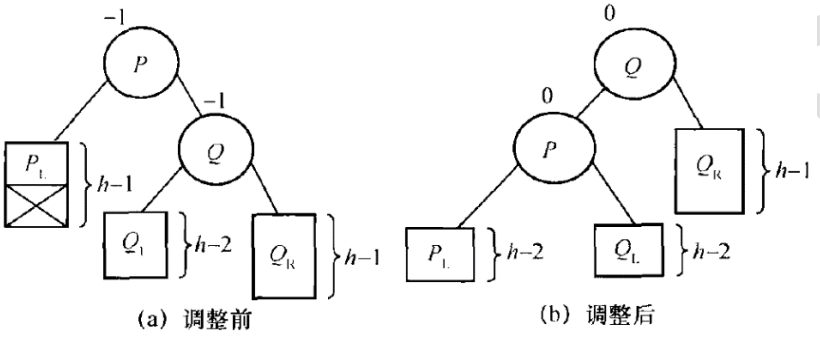

图 9.5 情况 3.1（进行 RR 旋转）

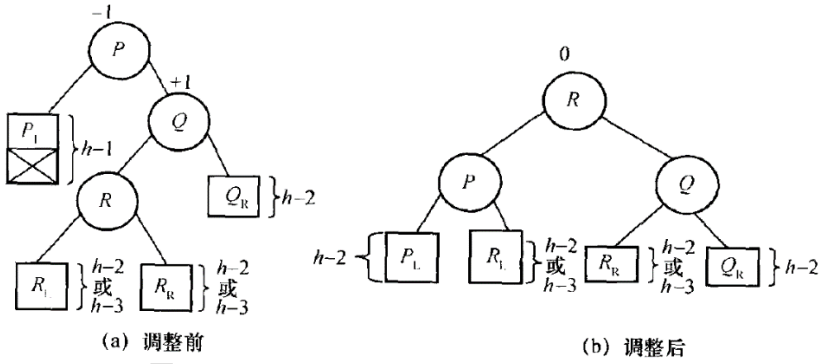

图 9.6 情况 3.2（进行 RL 旋转）

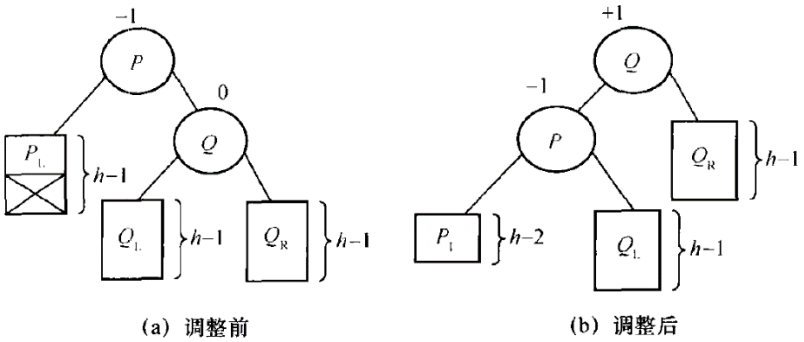

图 9.7 情况 3.3（进行 RR 旋转）

9.2.3.5 情况 4，情况 5，情况 6（情况 6.1~6.3）为情况 1，情况 2，情况 3（情况 3.1~3.3）镜像对称，类似可得。

9.2.4 【注】树上 AVL 树的 LL，LR，RL，RR 的意义是结点 P 的什么地方不平衡了（LL 表示左儿子的左儿子，以此类推），因此 LL(x)并不是表示左旋 x，相反的，表示为将 x 进行右旋（由于 x 的左子树的左儿子不平衡了，故需要右旋）。

9.3 **红黑树**

9.3.1 红黑树需要满足下列性质：(1) 每个结点被染成红色或黑色；(2) 根结点是黑色的；(3) 如果一个结点是红色的，那么他的儿子结点必须是黑色的；(4) 从任何一个结点出发到空结点（即空指针指向的结点，定义空结点为黑色的）的路径上，包含相同数目的黑色结点。

9.3.2 红黑树的树高最多为 2log(*N*+1)。（出现了红黑交替的链）

9.3.3 红黑树的插入

9.3.3.1 新结点总是作为叶子结点插入，且为不违背(4)，新结点初始着色均为红色。如果新结点 X 的父亲 P 为红色的，就需要进行旋转和颜色的调整了。

9.3.3.2 情况 1：父结点 P 的兄弟结点 S 是黑色的，设 P 的父结点为 G。

9.3.3.2.1 情况 1.1:X 是 G 的外侧结点,即 G-P-X 是左-左或右-右,分别称为 LLb 和 RRb，其中 b 表示父亲结点的兄弟是黑色的。此时对于 G 做单旋转（LL/RR），然后进行重新着色即可。如图 9.8 所示。

9.3.3.2.2 情况 1.2:X 是 G 的内侧结点,即 G-P-X 是左-右或右-左,分别称为 LRb 和 RLb。此时对于 G 做一个双旋转（LR/RL），然后进行重新着色即可。如图 9.9 所示。

9.3.3.2.3 情况 1.1 和情况 1.2 调整后根结点均为黑色的，因此不需要向上继续调整。

9.3.3.2.4 重新着色方式可统一记为从上到下三层结点为黑-红-黑；

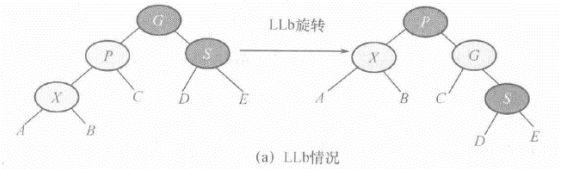

图 9.8 情况 1.1（LLb 旋转和 RRb 旋转）

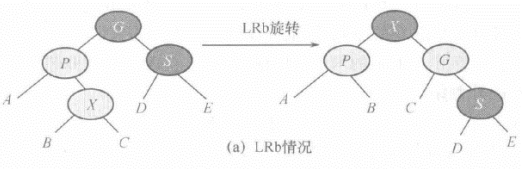

图 9.9 情况 1.2（LRb 旋转和 RLb 旋转）

9.3.3.3 情况 2：父结点 P 的兄弟结点 S 是红色的，设 P 的父结点为 G。

9.3.3.3.1 情况 2.1 与情况 2.2 和情况 1.1 与情况 1.2 类似,均为外部结点和内部结点情况，旋转方式与情况 1.1、情况 1.2 完全相同。

9.3.3.3.2 染色方式为从上到下三层结点为红-黑-红。因此根结点为红色,需向上继续调整。

9.3.4 红黑树的删除（自顶向下）

9.3.4.1 与二叉查找树相同，分(1)(2)(3)三种情况讨论，最终归结到叶子结点或只有一个孩子的结点的删除。需要注意的是书上的方法为自顶向下的删除。其主要思想就是在从根到待删除的结点的路径上遇到每个点都想办法将其变为红色，最后被删除结点为红色的叶子结点即可直接删除。

9.3.4.2 设 X 为当前遇到的结点，T 是兄弟结点，P 是他们的父结点。对于每个遇到的 X，都试图将其变为红色。可以确定的是 P 是红色的，根据 X 的颜色可以分情况讨论。

9.3.4.3 情况 1：X 有两个黑色儿子。

9.3.4.3.1 情况 1.1：T 有两个黑色儿子，则直接将 P 变为黑色，X 和 T 变为红色即可。

9.3.4.3.2 情况 1.2：T 有一个外侧的红色儿子，则对 P 进行单旋转（LL/RR），此时 T 作为

根，则将 T-P-X 三层染成红-黑-红即可。如图 9.10 所示。

9.3.4.3.3 情况 1.3：T 有一个内侧的红色儿子 R，则对 T、P 进行双旋转（LR/RL），此时 R 作为根结点。把 R-(P,T)-X 三层染成红-黑-红即可。如图 9.11 所示。

9.3.4.3.4 注：如果 T 有两个红色儿子，则选用 1.2 和 1.3 均可。

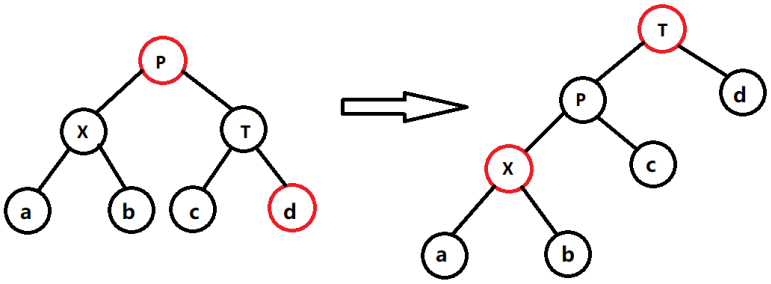

图 9.10 情况 1.2（进行 RR 旋转）

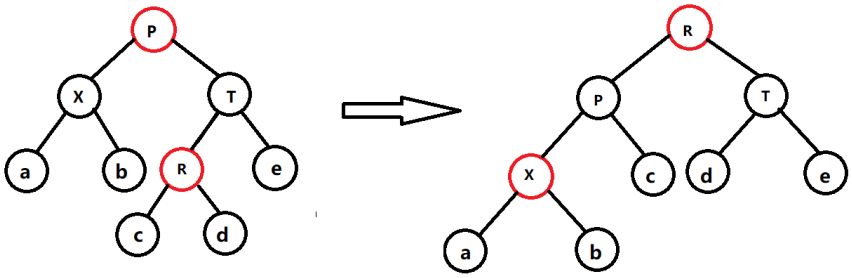

图 9.11 情况 1.3（进行 RL 旋转）

注：图 9.10 和图 9.11 中 a,b,c,d,e 染色任意（只要符合红黑树条件即可）

9.3.4.4 情况 2：X 有至少一个红儿子。

9.3.4.4.1 情况 2.1：如果 X 不是被删除结点，继续前进一次，如果前进到的 X'为红色的，那么问题解决，直接进行下一层的调整；否则，设当前前进到的 X'的父亲 P、兄弟 T，其中 T 为红色的，X'和 P 都是黑色的。此时将 P 进行一次单旋转，则 X'的父结点 P 为红色的且一定有两个黑儿子，就完成了上一层的既定任务（P 即上一层的 X 变为了红色）于是转化为情况 1 继续进行处理。如图 9.12 所示。

9.3.4.4.2 情况 2.2：如果 X 是被删结点且有两个儿子，则在右子树上寻找替身，如果 X 的右孩子为红色的，则直接往下走一层；否则左孩子一定是红色结点，将 X 进行一次右旋转（LL），则 X 变为红色，继续做即可。如果 X 是被删结点且只有一个儿子，则该儿子必为红结点，进行一次单旋转即可将 X 变为红色的叶子，直接删除即可。

9.3.4.5 注：情况 1 的时候遇到被删除结点直接寻找替身即可，因为已经将被删结点染红。

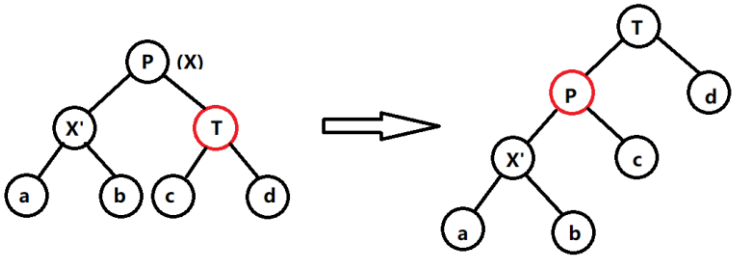

图 9.12 情况 2.1（进行 RL 旋转）

9.4 **AA 树**

9.4.1 左儿子不能为红色的红黑树，定义层次为结点到空结点的路径上左链的数量。如果结点是一个叶子，层次为 1；如果是红色结点，就是他父亲的层次（红色为水平右链）；如果是黑色的结点，就比他父亲的层次少 1；空结点的层次为 0。将层次的概念用图表示就形成了水平链表示法。

9.4.2 水平链表示法中，AA 树有如下性质：(1) 不存在水平左链（左儿子不为红色）；(2) 不存在连续右链（不会有连续的红色结点）；(3) 在层次 2 或 2 以上的结点必定有两个儿子；(4) 如果一个结点没有水平右链，它的两个儿子在同一层。

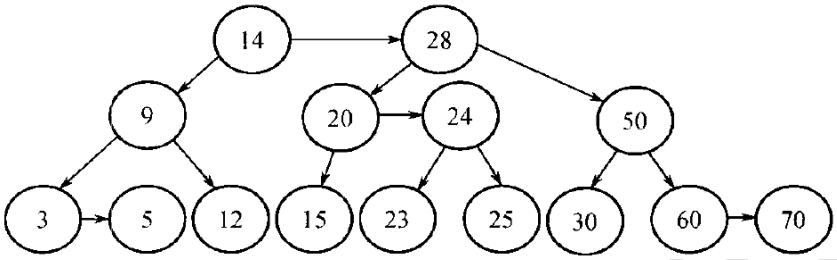

图 9.13 AA 树的水平链表示法

9.4.3 AA 树的插入

9.4.3.1 在最底层插入（按照红黑树，插入为红色结点，不会改变层次），然后进行调整。

9.4.3.2 出现水平左链，进行 LL 旋转转化为水平右链。

9.4.3.3 出现连续水平右链，则对于中间结点进行 RR 旋转即可。中间结点的层次增加 1，对于上一层递归解决即可。

9.4.4 AA 树的删除

9.4.4.1 和二叉查找树一样，分为(1)(2)(3)三种情况，归结为删除第一层的结点的问题。

9.4.4.2 情况 1：删除红结点，不影响平衡。

9.4.4.3 情况 2：删除有右红孩子的结点，用右红孩子代替该结点即可，不影响平衡。

9.4.4.4 情况 3：删除一个没有孩子的黑结点，进行调整。先降低父亲结点的层次（如果存在水平右链则一起降低），那么会出现水平左链或连续的水平右链。过程如下：如果需要，先对根结点执行 LL 旋转（水平左链）；如果需要，再对当前根的右孩子执行一次 LL 旋转（因为第一 LL 旋转可能仍然会出现水平左链，或是应该先对根结点的右孩子的水平右链进行处理）；如果需要，对当前根结点的右儿子的右儿子进行 LL 旋转（因为可能有两个水平左链需要消除）；如果需要，对当前根执行一次 RR 旋转。（消除连续右链）；如果需要，再对当前根的右孩子进行一次 RR 旋转（处理完水平左链可能有超过 3 个水平右链需要处理）。（此部分可参考 p299-p300 的图 9-39，图 9-40 和图 9-41）。

9.5 伸展树（Splay）

9.5.1 不需要维护平衡信息的二叉查找树，均摊时间复杂度 O(logN)，适用于 90-10 规则（90%的访问针对 10%的数据），将频繁访问的数据向根旋转。

9.5.2 伸展树的旋转分三类：(1) 父亲就是根的，使用单旋（LL/RR），称 zig 情况；(2) 父亲不是根，但是这个点是内部结点，使用双旋转（LR/RL），称 zig-zag 情况；(3) 父亲不是根，这个结点是外部结点，使用两次 LL/RR 旋转，称 zig-zig 情况。

9.6 **散列表**

9.6.1 散列函数导致的两个不同元素被映射到同一位置的问题称冲突或碰撞。

9.6.2 常用散列函数

9.6.2.1 直接定址法：$H(x)=x$，$H(x)=ax+b$；

9.6.2.2 除留余数法：$H(x)=x \bmod M$，一般选择 M 为素数；

9.6.2.3 数字分析法：选取分布均匀的若干位或他们的组合作为地址；

9.6.2.4 平方取中法：平方后选取中间的若干位（中间若干位涉及到了全部位置的数）；

9.6.2.5 折叠法：按照若干位折叠分段后相加后抛弃进位；

9.6.2.6 乘法散列法：$H(x) = (Ax \bmod 1) << p$ $(0<A<1)$，相当于提取小数部分。

9.6.2.7 斐波那契散列法：乘法散列法中，A=0.618……

9.6.3 碰撞解决：闭散列表（将溢出数据存到没有散列表使用的单元），开散列表(拉链法)（将映射到统一位置的数据组织成线性表来存储（利用外部的单元））

9.6.3.1 线性探测法：从冲突位置顺序向下搜索直到发现第一个空位置。

9.6.3.2 二次探测法：为了解决初始聚集问题，依次尝试 $H+1^2, H+2^2, H+3^2, \ldots$的位置。如果表大小为素数，则可以保证若表至少有一半空单元，则新元素总时能被插入，且插入过程中没有元素被探测两次（书 p313 定理 9.2）（证明利用反证法+同余性质）。利用这个定理，**只要表开到元素数量的两倍，就一定能够存下所有元素！**且增加 i^2 可以利用+1，+3，+5，……的等差数列代替优化。

9.6.3.3 再散列法：探测序列为 $H_1(x), H_1(x)+H_2(x), H_1(x)+2H_2(x), \ldots$

9.6.3.4 【注】闭散列表删除时不能直接删除元素（由于可能是占位所用，删除会导致查找失败!），一般采用迟删除（打标记表示这个单元时活动的还是被删除的，但是不删除存储内容）。

9.6.3.5 链接法：对于冲突的数据用链表存储；

9.6.3.6 公共溢出区法：对于冲突的数据存放在公共溢出区。

9.6.3.7 平均查找长度：查找散列表中包含的数据，所比较的次数的平均值。

10 排序

10.1 排序的基本分类

10.1.1 稳定：关键字值相同的元素排序后按照出现顺序排列。（交换相邻元素的排序）

10.1.2 不稳定：关键字值相同的元素排序后不一定按照出现顺序排列。

10.1.3 内排序：在内存中进行的排序。

10.1.4 外排序：在外部存储设备中进行的排序。

10.2 各种（内）排序总结

名称	主要思想（内容）	稳定	时间复杂度
（直接）插入排序	每次将一个元素插入到有序序列中	√	$O(n^2)$
（二分）插入排序	将直接插入排序寻找插入位置改为二分查找	√	$O(n^2)$
希尔排序	将相隔 h_t 的元素拿出来插入排序（增量序列）	×	(统计) $n^{1.25}$
（直接）选择排序	每次选择待排序序列的最小的数，选择 n 次	×	$O(n^2)$
堆排序	利用优先队列来选择最小的数	×	$O(n \log n)$
冒泡排序	每次冒泡可以将最大元素沉底	√	$O(n^2)$
快速排序	选择基准点划分左右区间，递归此过程。 一般选择 $a[low]$；随机选择可保证复杂度。	×	$O(n\log n)$ 最坏 $O(n^2)$
归并排序	分成两部分分别排序后，合并有序序列	√	$O(n\log n)$
基数排序	从低到高按位的大小排序	√	$O(\text{len}(n+10))$
桶排序	特殊的基数排序	√	$O(n+m)$

表 10.1 排序方法的总结整理

11 外部查找与排序

11.1 B 树

11.1.1 定义：一个 m 阶 B 树要么为空，要么满足：

(1) 根结点要么是叶子，要么至少有两个儿子，至多有 m 个儿子；

(2) 除根结点和叶子结点之外，每个结点的儿子个数 s 满足$\left\lceil \frac{m}{2} \right\rceil \le s \le m$；

(3) 有 s 个儿子的非叶结点具有 $n=s-1$ 个关键字，故 $s=n+1$，这些结点的数据信息为 $(n, A_0, (K_1,V_1), A_1, (K_2,V_2), \ldots, A_{n-1}, (K_n,V_n), A_n)$，其中 n 为关键字个数。

(4) 所有叶子结点出现在同一层上，并且不带信息（可认为是外部结点或查找失败的结点）。

11.1.2 B 树的插入：与二叉查找树类似，插入在叶子结点的父亲上，如果关键字个数达到 m 则分裂，并且往父亲插入一个关键字，如果需要继续分裂递归即可。根需要分裂的时候建立新根。

11.1.3 B 树的删除：与二叉查找树类似，需要往儿子结点寻找替身。删除后如果关键值过少需要向左右兄弟借关键字（需要注意的是这里的借关键字是指将左兄弟的最大关键值移动至父亲，再将父亲的关键值移动到该结点），或者与左右兄弟合并（需要把父亲的关键字拿下来）。

11.2 B+树

11.2.1 定义：一个 M 阶的 B+树要么为空，要么满足：

(1) 根或者是叶子，或者有 2 到 M 个孩子；

(2) 除根之外的所有结点都有不少于$\lceil M/2 \rceil$且不多于 M 个孩子；

(3) 有 k 个孩子的结点保存 k-1 个键来引导查找，键 i 代表子树 i+1 中键的最小值；

(4) 叶结点的孩子指针指向存储记录的数据块的地址。换句话说，对于索引 B+树他们是叶结点，但对于数据块来说，他们又是数据块的父结点，数据块才是真的叶结点，而在 B+树中，叶结点的孩子指针都是空指针；

(5) 每个数据块至少有$\lceil L/2 \rceil$个记录，至多有 L 个记录；

(6) 所有叶结点按照顺序组成一个单链表。

11.2.2 B+树的插入和删除和 B 树类似，只是插入和删除全部在叶子结点进行，对于内部结点只需要改变索引即可。同样插入有分裂，删除时有借用和合并的操作。

11.2.3 B+树的计算：若一个磁盘块的容量为 F，关键字占 K 字节，磁盘块号一般 4 字节，则一个 M 阶 B+树有(M-1)个键，关键字占用 K(M-1)个字节，还需要 4M 字节存储儿子，因此一个非叶结点需要 K(M-1)+4M 字节，选取最大的 M 使得 K(M-1)+4M<=F 即可。设每条完整记录(K,V)的字节为(K+V)，则选取最大的 L 使得(K+V)L<=F。

11.3 外排序

11.3.1 置换选择（预处理）：可以在容纳 p 的内存中生成平均长度为 2p 的有序序列。（例：原序列{81,94,12,11,96,35,17,99,28,58,41,75,15}，内存中最多能容纳 3 个记录）

顺序	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A[1]	12	81	94	96	11	11	17	28	35	41	58	75	99	15	15
A[2]	81	94	96	11	17	17	35	35	58	58	75	99	15		
A[3]	94	11	11	35	35	35	99	99	99	99	99	15			
输出	12	81	94	96		11	17	28	35	41	58	75	99		15
输入	11	96	35	17		99	28	58	41	75	15				

表 11.1 置换选择举例

11.3.2 多阶段归并：如果生成的已排序片段数为 m，则 k 路归并需要归并$\log_k m$次。设待排序数据存放在磁带上，生成的已排序片段也存放在磁带上，则两路归并需要 4 根磁带，$A1$,$A2$,$B1$,$B2$，一开始将置换选择的排序片段交替存入 $A1$,$A2$；归并时归并 $A1$,$A2$ 的第一个序列、第二个序列……，得到一系列合并后的有序序列，交替存入 $B1$,$B2$；再以 $B1$,$B2$ 为输入，$A1$,$A2$ 为输出重复过程即可排序完成。如果有 $2k$ (k>2)条磁带，可以使用 k 路归并来减少归并次数。

11.3.3 多阶段归并（斐波那契归并）（以两路归并为例）

磁带	已排序片段	归并 1	归并 2	归并 3	归并 4	归并 5	归并 6	归并 7
T1	0	13	5	0	3	1	0	1
T2	21	8	0	5	2	0	1	0
T3	13	0	8	3	0	2	1	0

（交替作为输入输出端）

表 11.2 多阶段归并举例

12 不相交集

12.1 等价关系（包含自反性，对称性，传递性的关系）划分等价类的问题称为等价问题。

12.2 不相交集（并查集）基本操作 union(), find()。改进方法：按规模并、按高度并（此时可以用负数既表示高度、规模，也包含了这是个根的信息）、路径压缩

12.3 应用：迷宫问题，最近公共祖先 LCA 问题（后序遍历+并查集，查询成功的时候所在点即 LCA）。

13 图

13.1 基本概念（V 顶点集，E 边集，结点，有向图（<u,v>表示 u 到 v 的边），弧（有向图的边），无向图（双向图），边权，加权图，加权有向/无向图）

13.2 基本术语（邻接，度（入度，出度），子图，路径和路径长度（连通，非加权路径长度，加权路径长度，简单路径，回路（环），有向无环图（DAG）），连通图和连通分量（对无向图），强连通图和强连通分量（对有向图），完全图（无向完全图，有向完全图），生成树（无向连通图的极小连通子图）。

13.3 图的存储（邻接矩阵 O(V^2)，邻接表 O(V+E)）

13.4 图的遍历（深度优先搜索 DFS（深度优先生成树，深度优先生成森林，深度优先访问森林（区别于深度优先生成森林，从 1 开始依次访问，***不确定***）），广度优先搜索 BFS（广度优先生成树，广度优先生成森林，广度优先访问森林）

13.5 图的遍历的应用

13.5.1 判定无向图连通性

13.5.2 欧拉（回）路（访问每条边恰好 1 次）（判定可行，选择起点，随便搜一条起点开始的（回）路，从这条路径上的点继续搜索回路，把两个回路合并，不断重复此过程）

13.5.3 哈密尔顿回路（访问每个点恰好 1 次）

13.5.4 有向图的连通性算法（Kosaraju 算法）：从任意结点开始 DFS，如果 G 不是强连通的会得到深度优先生成森林（或树），对森林中每棵树按照生成次序进行后序遍历并给每个结点编号。然后将 G 的边逆向形成 Gr，从编号最大的结点开始 DFS 搜索 Gr，得到的深度优先遍历森林的每棵树就是 G 的强连通分量。（证明正确性需要证明如

果两个顶点 *v* 和 *w* 在 *Gr* 的同一棵深度优先生成树中，那么必然存在从 *v* 到 *w* 的路径和 *w* 到 *v* 的路径）

13.5.5 拓扑排序：将 DAG 中的结点按照边的关系进行排序，利用队列实现，时间复杂度 O(*V*+*E*)。

13.5.6 * AOE 网络：DAG，但是与 AOV 的区别是将活动定义在边上，顶点表示事件，当进入某个顶点的所有活动（边）都完成了，才可以进行该顶点引出的所有活动。AOE 网络需要解决问题：(1) 完成工程最小时间；(2) 影响工程进度的关键活动

13.5.6.1 从源点到收点的最长路径称为关键路径，关键路径上的活动称为关键活动。

13.5.6.2 从源点到某个顶点 *v* 的最长路径称为该顶点代表的事件的最早发生时间，还可以定义最晚发生时间表示不推迟整个活动的前提下，从这个顶点出发的活动的最晚开始时间。两时间之差称为该活动的时间余量，时间余量为 0 的活动即关键活动。

13.5.6.3 求出拓扑序列，则从头到尾遍历可求出最早发生时间，从尾到头遍历可以求出最迟开始时间，再从头到尾遍历即可求出关键活动和关键路径。

14 最小生成树

14.1 最小生成树的定义：边权和最小的生成树。

14.2 Kruskal 算法：优先队列（排序）+不相交集，从小到大加边，并查集判断是否连通。时间复杂度 O(ElogE)。

14.3 Prim 算法：两点集 *U*,*U'*，初始任选一点加入 *U*，其余点在 *U'*中。每次选定一条最小的两端点分别在 *U* 和 *U'*中的边加入最小生成树，并把两端点全部加入 *U* 中，直到最后 *U*=*V* 结束，时间复杂度 O(V^2)。（可利用优先队列进行优化至 O(*E*+*V*log*E*)）

14.4 * 正确性证明：事实上仅需要证明连接两不相交点集的最小边一定在整个图的最小生成树中即可。（利用反证法）假设不在，则一定存在另一条边连接两个点集而这条边的边权大于最小边，因此一定不优，假设矛盾，原定理成立。

15 最短路径

15.1 单源最短路径

15.1.1 非加权图的最短路径：BFS 实现，利用队列，时间复杂度 O(*V*)。

15.1.2 加权图的单源最短路径：Dijkstra 算法（不能有负权边）：初始点集 S 仅有起点，每次利用 *S* 中的点和 *V*-*S* 中的点的连边加上 *S* 中的点的已处理出的最短距离来更新 *V*-*S* 中的点的最短距离，每次选出更新后的 *V*-*S* 中距离最小的点加入集合 *S*，重复上述过程直到 *S*=*V* 结束，时间复杂度 O(V^2)。（可利用优先队列进行优化至 O(*E*log*E*)甚至 O(*E*log*V*)）

15.1.3 带负权边的单源最短路径：SPFA 算法：对每个节点找到更好路径后都检查所有后继节点来更新最短路径，时间复杂度 O(*EV*)。

15.1.4 有向无环图：利用拓扑排序的次序选择结点进行计算，时间复杂度 O(*V*+*E*)。

15.2 多源最短路径（所有点对间）：Floyd 算法（注意循环变量 *k*,*i*,*j*），动态规划思想。

16 算法设计基础

16.1 枚举法；

16.2 贪婪法；

16.3 分治法：大整数乘法，平面最近点对；
16.4 动态规划：斐波那契数列，硬币找零，最优二叉查找树；
16.5 回溯法：八皇后问题，分书问题；
16.6 随机算法：跳表，Millar-Rabin 素性检测算法。