第1章 绪论

**1.1 数据结构的基本概念**

**数据元**是数据的基本单位，一个**数据元素**可由若干个**数据项**完成，**数据项**是构成**数据元素**的不可分割的最小单位。例如，学生记录就是一个数据元素，它由学号、姓名、性别等数据项组成。

**数据对象**是具有**相同性质的数据元素**的集合，是数据的一个子集。

**数据类型**是一个值的集合和定义在此集合上一组操作的总称。

* 原子类型：其值不可再分的数据类型
* 结构类型：其值可以再分解为若干成分（分量）的数据类型
* **抽象数据类型**：抽象数据组织和与之相关的操作

**抽象数据类型（ADT）是指一个数学模型以及定义在该模型上的一组操作。抽象数据类型的定义仅取决于它的一组逻辑特性**，而与其在计算机内部如何表示和实现无关。通常用（**数据对象、数据关系、基本操作集**）这样的三元组来表示。

#关键词：数据，数据元素，数据对象，数据类型，数据结构

**数据结构的三要素：**

1. **逻辑结构**是指数据元素之间的逻辑关系，即从逻辑关系上描述数据，独立于计算机。分为线性结构和非线性结构，线性表、栈、队列属于线性结构，树、图、集合属于非线性结构。
2. **存储结构**是指数据结构在计算机中的表示（又称映像），也称物理结构，包括数据元素的表示和关系的表示，依赖于计算机语言，分为顺序存储（随机存取）、链式存储（无碎片）、索引存储（检索速度快）、散列存储（检索、增加、删除快）。
3. **数据的运算**：包括运算的定义和实现。运算的定义是针对逻辑结构的，指出运算的功能；运算的实现是针对存储结构的，指出运算的具体操作步骤。

**1.2 算法和算法评价**

算法是对特定问题求解步骤的一种描述，有五个特性：**有穷性、确定性、可行性、输入、输出**。一个算法有**零个或多个的输入**，有**一个或多个的输出**。

**时间复杂度**是指该语句在算法中被重复执行的次数，不仅依赖于问题的规模n，也取决于待输入数据的性质。一般指最坏情况下的时间复杂度。

**空间复杂度**定义为该算法所耗费的存储空间。**算法原地工作**是指算法所需辅助空间是常量，即O(1)。

第2章 线性表

**2.1 线性表的定义和基本操作**

**线性表**是具有**相同数据类型**的n个**数据元素**的**有限**序列。除第一个元素外，每个元素有且仅有一个直接前驱；除最后一个元素外，每个元素有且仅有一个直接后继。

**注意：线性表是一种逻辑结构，表示元素之间一对一的相邻关系。顺序表和链表是指存储结构**。两者属于不同层面的概念，因此不要混淆。

**2.2 线性表的顺序表示**

**线性表的顺序存储又称为顺序表**。它是用一组地址连续的存储单元，依次存储线性表中的数据元素，从而使得逻辑上相邻的两个元素在物理位置上也相邻。

顺序表的特点是表中元素的逻辑顺序与其物理顺序相同。  
顺序表最主要的特点是**随机访问（随机存取），即通过首地址和元素序号可以在O(1)的时间内找到指定的元素。但插入和删除操作需要移动大量元素**。  
顺序表的存储密度高，每个结点只存储数据元素。

**顺序表插入操作：  
最好情况：在表尾插入，元素后移语句不执行，时间复杂度为O(1)  
最坏情况：在表头插入，元素后移语句将执行n次，时间复杂度为O(n)  
平均情况：平均时间复杂度为O(n)**

**顺序表删除操作：  
最好情况：删除表尾元素，无需移动元素，时间复杂度为O(1)  
最坏情况：删除表头元素，需要移动除第一个元素外的所有元素，时间复杂度为O(n)  
平均情况：平均时间复杂度为O(n)**

**顺序表按值查找（顺序查找）：  
最好情况：查找的元素就在表头，仅需比较一次，时间复杂度为O(1)  
最坏情况：查找的元素在表尾或不存在时，需要比较n次，时间复杂度为O(n)  
平均情况：平均时间复杂度为O(n)**

**2.3 线性表的链式表示**

链式存储线性表时，不需要使用地址连续的存储单元，即它不要求逻辑上相邻的两个元素在物理位置上也相邻，对线性表的插入和删除不需要移动元素，只需要修改指针。

**单链表是非随机存储的存储结构**，即不能直接找到表中某个特定的结点。

**头结点和头指针的区分**：不管带不带头结点，头指针始终指向链表的第一个结点，而头结点是带头结点链表中的第一个结点，结点内通常不存储信息。

**采用头插法建立单链表**：从一个空表开始，生成新结点，并将读取到的数据存放在新结点的数据域中，然后将新结点插入到当前链表的表头，即头结点之后。  
读入数据的顺序与生成的链表中元素的顺序是相反的。  
每个结点插入的时间为O(1)，设单链表长为n，则总的时间复杂度为O(n)。

**采用尾插法建立单链表**：将新结点插入到当前链表的表尾上，为此必须增加一个尾指针r，使其始终指向当前链表的尾结点。  
时间复杂度和头插法相同。

**按序号查找结点值**：在单链表中从第一个结点出发，顺指针next域逐个往下搜索，直到找到第i个结点为止，否则返回最后一个结点指针域NULL。  
时间复杂度为O(n)。

**按值查找表结点**：从单链表第一个结点出发，由前往后依次比较表中各结点数据域的值，若某结点数据域的值等于给定值e，则返回该结点的指针，否则返回NULL。  
时间复杂度为O(n)。

**插入结点操作**：先检查插入位置的合法性，然后找到待插入位置的前驱结点，即第i-1个结点，再在其后插入新结点。  
主要的时间开销在于查找第i-1个元素，时间复杂度为O(n)。若是在给定的结点后面插入新结点，则时间复杂度仅为O(1)。

**删除结点操作**：先检查删除位置的合法性，然后找到待删除位置的前驱结点，即第i-1个结点，再将其删除。  
时间复杂度为O(n)。

**双链表**结点中有两个指针prior和next，分别指向其前驱结点和后继结点。双链表中执行按值查找和按位查找的操作和单链表相同，但双链表插入、删除操作的时间复杂度仅为O(1)。

**循环单链表**和单链表的区别在于，表中最后一个结点的指针不是NULL，而改为指向头结点，从而整个链表形成一个环。故表中没有指针域为NULL的结点。  
**循环双链表** L中，某结点\*p为尾结点时，p->next=L；当循环双链表为空表时，其头结点的prior域和next域都等于L。

**静态链表是借助数组来描述线性表的链式存储结构**，结点也有数据域data和指针域next，这里的指针是结点的相对地址（数组下标），又称为游标。

**顺序表和链表的比较：**

1. 存取方式：顺序表可以顺序存取，也可以随机存取，链表只能从表头顺序存取元素。
2. 逻辑结构与物理结构：采用顺序存储时，逻辑上相邻的元素，其对应的物理存储位置也相邻。而采用链式存储时，逻辑上相邻的元素，其物理存储位置则不一定相邻。
3. 查找、插入和删除操作：对于按值查找，当顺序表在无序的情况下，两者的时间复杂度均为O(n)；而当顺序表有序时，可采用折半查找，时间复杂度为O(logn)。对于按序号查找，顺序表支持随机访问，时间复杂度仅为O(1)，而链表的平均时间复杂度为O(n)。链表插入、删除较优，只需修改相关结点的指针域即可。
4. 空间分配：链式存储的结点空间只在需要的时候申请分配。
5. 通常较稳定的线性表选择顺序存储，而频繁做插入、删除操作的线性表（即动态性较强）宜选择链式存储。

第3章 栈和队列

**3.1 栈**

栈：后进先出（LIFO），又称为后进先出的线性表

**顺序栈**的实现：  
**栈顶指针**：S.top，初始时设置**S.top=-1**；**栈顶元素**：S.data[S.top]。  
**进栈操作**：栈不满时，**栈顶指针先加1，再送值到栈顶元素。S.data[++S.top]=x**。  
**出栈操作**：栈非空时，**先取栈顶元素值，再将栈顶指针减1。x=S.data[S.top–]**。  
**栈空条件：S.top==-1；栈满条件：S.top==Maxsize-1；栈长：S.top+1**

**注意**：如果栈顶指针初始化为S.top=0，即栈顶指针指向栈顶元素的下一个位置，则入栈操作变为S.data[S.top++]=x；出栈操作变为x=S.data[–S.top]。相应的栈空、栈满条件也会发生变化。

**共享栈**：利用栈底位置相对不变的特性，可以让两个顺序栈共享一个一维数据空间，将两个栈的栈底分别设置在共享空间的两端，两个栈顶向共享空间的中间延伸。  
两个栈的栈顶指针都指向栈顶元素，**top0=-1时0号栈为空，top1=Maxsize时1号栈为空；仅当两个栈顶指针相邻（top1-top0=1）时，判断为栈满**。**当0号栈进栈时top0先加1再赋值，1号栈进栈时top1先减1再赋值；出栈时刚好相反，0号栈先出栈再减1，1号栈先出栈再加1.**  
共享栈是为了更有效地利用存储空间，其存取数据的时间复杂度均为O(1)。

**链栈**的优点是便于多个栈共享存储空间和提高其效率，且不存在栈满上溢的情况。通常**采用单链表实现，并规定所有操作都是在单链表的表头进行的**。这里规定链栈没有头结点，Lhead指向栈顶元素。  
采用链式存储，便于结点的插入和删除。对于带头结点和不带头结点的链栈，在具体的实现方面有所不同。

**3.2 队列**

**队列**也是一种操作受限的线性表，只允许在表的一端进行插入，而在表的另一端进行删除。（先进先出的线性表）

**队列的顺序实现**是指分配一块连续的存储单元存放队列中的元素，并附设两个指针front和rear分别指示队头元素和队尾元素的位置。设队头指针指向队头元素，队尾指针指向队尾元素的下一个位置（也可以让rear指向队尾元素，front指向队头元素的前一个位置）。  
**初始状态（队空条件）：Q.front==Q.rear == 0.  
进队操作：队不满时，先送值到队尾元素，再将队尾指针加1。  
出队操作：队不空时，先取队头元素值，再将队头指针加1.**

**循环队列**，即把存储队列元素的表从逻辑上看成一个环。  
**初始时：Q.front==Q.rear == 0.  
出队：队首指针进1，Q.front=(Q.front+1)%MaxSize  
入队：队尾指针进1，Q.rear=(Q.rear+1)%MaxSize  
队列长度：(Q.rear+MaxSize-Q.front)%MaxSize  
出队入队时：指针都按顺时针方向进1.**

**为了区分队空还是队满的情况，有三种处理方式：**  
1）牺牲一个单元来区分队空和队满，队头指针在队尾指针的下一位置作为队满的标志：  
**队满条件为：(Q.rear+1)%MaxSize == Q.front  
队空条件仍为：Q.front==Q.rear  
队列中元素的个数：(Q.rear-Q.front+MaxSize)%MaxSize**  
2）类型中增设表示元素个数的数据成员：  
队空条件为：Q.size== 0  
队满条件为：Q.size == Maxsize  
3）类型中增设tag数据成员，以区分是队满还是队空。

**队列的链式表示**称为链队列，是一个同时带有队头指针和队尾指针的单链表。头指针指向队头结点，尾指针指向队尾结点，即单链表的最后一个结点。  
当Q.front==NULL && Q.rear == NULL时，链式队列为空。

**双端队列**是指允许两端都可以进行入队和出队操作的队列，其元素的逻辑结构仍是线性结构。  
**输出受限的双端队列**：允许在一端进行插入和删除，但在另一端只允许插入。  
**输入受限的双端队列**：允许在一端进行插入和删除，但在另一端只允许删除。

**设有一个双端队列，输入序列为1,2,3,4：  
（1）能由输入受限的双端队列得到，但不能由输出受限的双端队列得到的输出序列是4,1,3,2  
（2）能由输出受限的双端队列得到，但不能由输入受限的双端队列得到的输出序列是4,2,1,3  
（3）既不能由输入受限的双端队列得到，又不能由输出受限的双端队列得到的输出序列是4,2,3,1**

**3.3 栈和队列的应用**

**栈的应用**：括号匹配、表达式求值、递归  
**队列的应用**：层次遍历、计算机系统

**3.4 特殊矩阵的压缩存储**

数组是由n个相同类型的数据元素构成的有限序列。

**数组的存储结构：**  
一维数组  
多维数组：按行优先和按列优先

**矩阵的压缩存储：**  
对称矩阵、上（下）三角矩阵、对角矩阵

**对称矩阵**：将对称矩阵A[1…n][1…n]存放在一维数组B[n(n+1)/2]中，只存放主对角线和下三角区的元素。元素下标之间的对应关系如下（**数组下标从0开始**）：  
**k=i(i-1)/2+j-1（当i≥j，即下三角区和主对角线元素）  
k=j(j-1)/2+i-1（当i<j，即上三角区元素）**

**三角矩阵**：  
将**下三角矩阵**A[1…n][1…n]存放在一维数组B[n(n+1)/2+1]中，存储下三角区和主对角线上的元素，以及存储对角线上方的常量一次。元素下标之间的对应关系如下（**数组下标从0开始**）：  
**k=i(i-1)/2+j-1（当i≥j，即下三角区和主对角线元素）  
k=n(n+1)/2（当i<j，即上三角区元素）**

将**上三角矩阵**A[1…n][1…n]存放在一维数组B[n(n+1)/2+1]中，存储上三角区和主对角线上的元素，以及存储对角线下方的常量一次。元素下标之间的对应关系如下（**数组下标从0开始**）：  
**k=(i-1)(2n-i+2)/2+(j-i)（当i≤j，即上三角区和主对角线元素）  
k=n(n+1)/2（当i>j，即下三角区元素）**

**三对角矩阵**：三对角矩阵3条对角线上的元素Ai,j在一维数组B中存放的下标为**k=2i+j-3**；反之，若已知三对角矩阵中某元素Ai,j在一维数组B中存放于第k个位置，则可求得**i=(k+1)/3+1（向下取整），j=k-2i+3**.

**稀疏矩阵**：将非零元素及其相应的行和列构成一个三元组（行标，列标，值）。

第4章 串

**4.1 串的定义和实现**

**串**：由零个或多个字符组成的有限序列

**串的存储结构**：

1）定长顺序存储表示

2）堆分配存储表示

3）块链存储表示

**4.2 串的模式匹配**

**串的模式匹配**：是求第一个字符串（模式串）在第二个字符串（主串）中的位置。

**简单模式匹配算法：逐个匹配，最坏时间复杂度为O(n\*m)，n、m分别为主串和模式串的长度**。

**KMP算法**可以在O(n+m)的时间数量级上完成串的模式匹配操作，大大提高了匹配效率。  
**手工求解next数组的方法**：  
1）next[1]=0，next[2]=1（next[0]不使用）  
2）后面求解每一位的next[j]值时，根据j的前一位进行比较，令k=next[j-1]  
3）将S[j-1]与S[k]进行比较：  
a. 如果相等，则该next[j]=k+1  
b. 如果不等，令k=next[k]，若k不等于0，跳到3；若k等于0，next[j]=1

第5章 树与二叉树

**5.1 树的基本概念**

树是N个结点的有限集合，N=0时，称为空树。  
树中一个结点的子结点个数称为该**结点的度**，树中结点的最大度数称为**树的度**。  
结点的层次从树根开始定义，**根结点为第1层**。  
**结点的深度**是从根结点开始自顶向下逐层累加的；**结点的高度**时从叶结点开始自底向上逐层累加的。  
**树的高度**（又称**深度**）是树中结点的最大层数。  
树中两个结点之间的**路径**是由这两个结点之间所经过的**结点序列**构成的，而**路径长度**是路径上所经过的**边的个数**。

**树的性质：  
1）树中的结点数等于所有结点的度数加1.  
2）度为m的树中第i层上至多有个结点（i≥1）。  
3）高度为h的m叉树至多有个结点。  
4）具有n个结点的m叉树的最小高度为。**

**5.2 二叉树的概念**

**二叉树**：每个结点至多只有两棵子树（即二叉树不存在度大于2的结点），子树有左右之分，不能任意颠倒次序。

**满二叉树**：一棵高度为h，并且含有**2^h-1**个结点的二叉树称为满二叉树，即树中的每一层都含有最多的结点，除叶子结点之外的每个结点度数均为2.对于编号为i的结点，**如果有双亲，其双亲为i/2（向下取整），如果有左孩子，则左孩子为2i，如果有右孩子，则右孩子为2i+1.**

**完全二叉树**：设一个高度为h，有n个结点的二叉树，当且仅当其每一个结点都与高度为h的满二叉树中编号为1~n的结点一一对应时，称为完全二叉树。  
**1）若i≤n/2（向下取整），则结点i为分支结点，否则为叶子结点。  
2）叶子结点只可能在层次最大的两层上出现。对于最大层次中的叶子结点，都依次排列在该层最左边的位置上。  
3）如果有度为1的结点，只可能有一个，且该结点只有左孩子而无右孩子。  
4）若n为奇数，则每个分支结点都有左子女和右子女；若n为偶数，则编号最大的分支结点（编号为n/2）只有左子女，没有右子女，其余分支结点左右子女都有。**

**二叉排序树**：左子树上所有结点的关键字均小于根结点的关键字，右子树上所有结点的关键字均大于根结点的关键字。

**平衡二叉树**：树上任一结点的左子树和右子树的深度之差不超过1。

**二叉树的性质：  
1）非空二叉树上叶子结点等于度为2的结点数加1，即N0=N2+1  
2）非空二叉树上第k层上至多有个结点  
3）高度为h的二叉树至多有****个结点。**

**4）对完全二叉树按从上到下、从左到右的顺序依次编号1~N，则有以下关系：  
a. 当i>1时，结点i的双亲结点编号为  
b. 当2i≤N时，结点i的左孩子编号为2i，否则无左孩子  
c. 当2i+1小于等于N时，结点i的右孩子编号为2i+1，否则无右孩子。  
d. 结点i所在层次（深度）为  
5）具有n个结点的完全二叉树的高度为或****。**

**二叉树的顺序存储结构**就是用一组地址连续的存储单元依次自上而下、自左至右存储完全二叉树上的结点元素。  
完全二叉树和满二叉树采用顺序存储比较合适。  
**注意**：二叉树属于树，都可以用树的存储结构来存储，但是树却不都能用二叉树的存储结构来存储。

**二叉树的链式存储结构**：链式结构是指用一个链表来存储一棵二叉树，二叉树中每一个结点用链表的一个链结点来存储。二叉链表至少包含三个域：数据域data、左指针域lchild、右指针域rchild。  
**在含有n个结点的二叉链表中含有n+1个空链域。**

**5.3 二叉树的遍历和线索二叉树**

**先序遍历PreOrder**：访问根结点，先序遍历左子树，先序遍历右子树  
**中序遍历InOrder**：中序遍历左子树，访问根结点，中序遍历右子树  
**后序遍历PostOrder**：后序遍历左子树，后序遍历右子树，访问根结点  
时间复杂度均为O(n)，在递归遍历中，递归工作栈的栈深恰好为树的高度。

**层次遍历：需要借助一个队列**。先将二叉树根结点入队，然后出队，访问该结点，如果它有左子树，则将左子树根结点入队；如果它有右子树，则将右子树根结点入队。然后出队，对出队结点访问，如此反复，直到队列为空。

由二叉树的**先序序列和中序序列**可以唯一确定一棵二叉树；  
由二叉树的**后序序列和中序序列**可以唯一确定一棵二叉树；  
由二叉树的**层序序列和中序序列**可以唯一确定一棵二叉树。  
如果只知道二叉树的先序序列和后序序列，则无法确定一棵二叉树。

**线索二叉树：为了加快查找结点前驱和后继的速度。**  
在**二叉树线索化**时，通常规定：若无左子树，令lchild指向其前驱结点；若无右子树，令rchild指向其后继结点，还需要增加那个标志域表明当前指针域所指对象时指向左（右）子结点还是直接前驱（后继）。  
ltag=0（lchild域指示结点的左孩子）  
ltag=1（lchild域指示结点的前驱）  
rtag=0（rchild域指示结点的右孩子）  
rtag=1（rchild域指示结点的后继）

**5.4 树、森林**

树的存储结构：既可以采用**顺序存储结构**，也可以采用**链式存储结构**。

**双亲表示法**：采用一组连续空间来存储每个结点，同时在每个结点中增设一个伪指针，指示其双亲结点在数组中的位置。  
**利用了每个结点（根结点除外）只有唯一双亲的性质，可以很快得到每个结点的双亲结点，但是求结点的孩子时需要遍历整个结构。**

**孩子表示法**：将每个结点的孩子结点都用单链表链接起来形成一个线性结构，则N个结点就有N个孩子链表（叶子结点的孩子链表为空表）。  
**寻找子女的操作非常直接，而寻找双亲的操作需要遍历N个结点中孩子链表指针域所指向的N个孩子链表。**

**孩子兄弟表示法**：又称为二叉树表示法，即以二叉树表作为树的存储结构，使每个结点包括三部分内容：结点值、指向结点第一个孩子结点的指针和指向结点下一个兄弟结点的指针（沿此域可以找到结点的所有兄弟结点）。  
**可以方便地实现树转换为二叉树的操作，易于查找结点的孩子等，但缺点是从当前结点查找其双亲结点比较麻烦。**

给定一棵树，可以找到唯一的一棵二叉树与之对应。

**树转换为二叉树的规则：每个结点左指针指向它的第一个孩子结点，右指针指向它在树中的相邻兄弟结点，可表示为“左孩子右兄弟”。由于根结点没有兄弟，所以由树转换而得的二叉树没有右子树。**

森林转换为二叉树的规则与树相似。

**二叉树转换为森林的规则：若二叉树非空，则二叉树根及其左子树为第一棵树的二叉树形式，二叉树根的右子树又可以看做是一个由除第一棵树外的森林转换后的二叉树，应用同样的方法，直到最后产生一棵没有右子树的二叉树为止。  
二叉树转换为树和森林是唯一的。**

**树的遍历**：  
**先根遍历**：若树非空，则先访问根结点，再按从左到右的顺序遍历根结点的每一棵子树。其访问顺序与这棵树相应二叉树的**先序遍历顺序相同**。  
**后根遍历**：若树非空，则按从左到右的顺序遍历根结点的每一棵子树，之后再访问根结点。其访问顺序与这棵树相应二叉树的**中序遍历顺序相同**。  
**层次遍历**：与二叉树的层次遍历思想基本相同，即按层序依次访问各结点。

**森林的遍历**：  
**先序遍历森林**：访问森林中第一棵树的根结点；先序遍历第一棵树中根结点的子树森林；先序遍历除去第一棵树之后剩余的树构成的森林。  
**中序遍历森林**：中序遍历森林中第一棵树的根结点的子树森林；访问第一棵树的根结点；中序遍历除去第一棵树之后剩余的树构成的森林。

树的应用——并查集

**5.5 树与二叉树的应用**

**二叉排序树（BST，又称为二叉查找树）**：左子树上所有结点关键字值均小于根结点的关键字值；右子树上所有结点关键字值均大于根结点的关键字值；左右子树本身也分别是一棵二叉排序树。  
对二叉排序树进行**中序遍历**，可以得到一个递增的有序序列。

二叉排序树的查找、插入、构造、删除  
**二叉排序树查找算法的平均查找长度**，主要取决于树的高度，即与二叉树的形态有关。  
当有序表是静态查找表时，宜用顺序表作为其存储结构，而采用二分查找实现其查找操作；若有序表是动态查找表，则应选择二叉排序树作为其逻辑结构。

**平衡二叉树（AVL）**：左子树和右子树都是平衡二叉树，且任意结点的左右子树高度差的绝对值不超过1。  
平衡因子的值只可能是-1,0或1（左子树的高度-右子树的高度）

**平衡二叉树的插入**：

**LL平衡旋转（右单旋转）：A右旋**

**RR平衡旋转（左单旋转）：A左旋**

**LR平衡旋转（先左后右双旋转）：B左旋A右旋**

**RL平衡旋转（先右后左双旋转）：B右旋A左旋**

**平衡二叉树的查找：平均查找长度为O(log2 n)**

从树根结点到任意结点的路径长度（经过的边数）与该结点上权值的乘积称为**该结点的带权路径长度**。树中所有叶结点的带权路径长度之和称为**该树的带权路径长度**。

在含有N个带权叶子结点的二叉树中，其中带权路径长度（WPL）最小的二叉树称为**哈夫曼树，也称为最优二叉树**。

**哈夫曼树的构造**：给定N个权值分别为w1~wN的结点  
1）将这N个结点分别作为N棵仅含一个结点的二叉树，构成森林F  
2）构造一个新结点，并从F中选取两棵根结点权值最小的树作为新结点的左右子树，并且将新结点的权值置为左右子树上根结点的权值之和  
3）从F中删除刚才选出的两棵树，同时将新得到的树加入F中  
4）重复2）和3），直至F中只剩下一棵树为止  
**特点**：  
1）每个初始结点最终都成为叶结点，并且**权值越小的结点到根结点的路径长度越大**  
2）构造过程中共新建了N-1个结点（双分支结点），因此哈夫曼树的**结点总数为2N-1**  
3）每次构造都选择2棵树作为新结点的孩子，因此哈夫曼树中**不存在度为1的结点**

**可变长度编码**：对频率高的字符赋以短编码，而对频率较低的字符赋以较长一些的编码，从而使字符平均编码长度减短，起到压缩数据的效果。  
如果没有一个编码是另一个编码的前缀，则称这样的编码为**前缀编码**。  
**由哈夫曼树构造哈夫曼编码**：首先将每个出现的字符当做一个独立的结点，其权值为它出现的频度，构造出相应的哈夫曼树。显然，所有字符结点都出现在叶结点中。我们可以将字符的编码解释为从根至该字符的路径上边标记的序列，其中边标记为0表示“转向左孩子”，标记为1表示“转向右孩子”。

第6章 图

**6.1 图的基本概念**

**有向图、无向图**  
**简单图**：不存在重复边，不存在顶点到自身的边（数据结构中仅讨论简单图）  
**多重图**的定义和简单图相反。

**无向完全图**：在无向图中，如果任意两个顶点之间都存在边，则称该图为无向完全图。含有n个顶点的无向完全图有**n(n-1)/2**条边。  
**有向完全图**：在有向图中，如果任意两个顶点之间都存在方向相反的两条弧，则称该图为有向完全图。含有n个顶点的有向完全图有\*\*n(n-1)\*\*条有向边。

**子图**：设有两个图G=(V,E)和G’=(V’,E’)，若V’是V的子集，且E’是E的子集，则称G’是G的**子图**。若有满足V(G’)=V(G)的子图G’，则为G的**生成子图**。

**连通、连通图和连通分量（无向图）**：在**无向图**中，若从顶点v到顶点w有路径存在，则称v和w是**连通**的。若图G中任意两个顶点都是连通的，则称图G为**连通图**，否则称为非连通图。**无向图中的极大连通子图称为连通分量**。**如果一个图有n个顶点，并且有小于n-1条边，则此图必是非连通图。**

**强连通图、强连通分量（有向图）**：在**有向图**中，若从顶点v到顶点w和从顶点w到顶点v之间都有路径，则称这两个顶点是**强连通**的。若图中任何一对顶点都是强连通的，则称该图为**强连通图**。**有向图中的极大强连通子图称为强连通分量**。

**生成树、生成森林：连通图的生成树是包含图中全部顶点的一个极小连通子图。若图中顶点数为n，则它的生成树含有n-1条边**。对于生成树而言，若砍去它的一条边，则它会变成非连通图，若加上一条边则会形成一个回路。

**对于无向图，顶点v的度是指依附于该顶点的边的条数，记为TD(v)。  
在具有n个顶点e条边的无向图中，全部顶点的度之和为2e，即无向图的全部顶点的度之和等于边数的两倍。**

**对于有向图，顶点v的度分为入度和出度，入度记为ID(v)，出度记为OD(v)。顶点v的度等于其入度和出度之和，即TD(v)=ID(v)+OD(v)。  
在具有n个顶点e条边的有向图中，有向图的全部顶点的入度之和与出度之和相等并且等于边数**。

带权图也称为网。  
**稠密图**：边数很多；

**稀疏图**：边数很少。一般当图G满足|E|<|V|\*log|V|时，可以将G看成是稀疏图。

**路径**：顶点序列；

**路径长度**：路径上边的数目。第一个顶点和最后一个顶点相同的路径称为**回路或环**。**如果一个图有n个顶点，并且有大于n-1条边，则此图一定有环**。

**简单路径**：在路径序列中，顶点不重复出现的路径称为**简单路径**。

**简单回路**：除第一个顶点和最后一个顶点之外，其余顶点不重复出现的回路称为**简单回路**。

有一个顶点的入度为0，其余顶点的入度均为1的有向图称为**有向树**。

**6.2 图的存储及基本操作**

**邻接矩阵存储**，就是用一个一维数组存储图中顶点的信息，用一个二维数组存储图中边的信息（即各顶点之间的邻接关系），存储顶点之间邻接关系的二维数组称为**邻接矩阵**。  
1）**无向图的邻接矩阵是对称矩阵（并且唯一）**，在实际存储邻接矩阵时只需存储上（或下）三角矩阵的元素即可，对规模特大的邻接矩阵可采用压缩存储。  
2）邻接矩阵表示法的空间复杂度为**O()**，其中n为图的顶点数|V|  
3）**对于无向图，邻接矩阵的第i行（或第i列）非零元素（或非∞元素）的个数正好是第i个顶点的度**  
4）**对于有向图，邻接矩阵的第i行（或第i列）非零元素（或非∞元素）的个数正好是第i个顶点的出度（或入度）**  
5）**容易确定顶点之间是否有边相连，但难以确定图中有多少条边**。  
6）**稠密图适合使用邻接矩阵的存储表示**。

**邻接表**就是对图G中的每个顶点vi建立一个单链表，第i个单链表中的结点表示依附于顶点vi的边（对于有向图则是以顶点vi为尾的弧），这个单链表就称为顶点vi的边表（对于有向图则称为出边表）。边表的头指针和顶点的数据信息采用顺序存储（称为顶点表），所以在邻接表中存在两种结点：**顶点表结点和边表结点**。  
顶点表结点由顶点域（data）和指向第一条邻接边的指针（firstarc）构成，边表（邻接表）结点由邻接点域（adjvex）和指向下一条邻接边的指针域（nextarc）构成。  
1）如果G为无向图，则所需的存储空间为**O(|V|+2|E|)**；如果G为有向图，则所需的存储空间为**O(|V|+|E|)**  
2）**对于稀疏图，采用邻接表表示将极大地节省存储空间**。  
3）**容易找出顶点的所有邻边，但难以确定两个顶点之间是否存在边**  
4）**图的邻接表表示并不唯一**

**十字链表**是**有向图**的一种**链式**存储结构。  
既容易找到vi为尾的弧，也容易找到vi为头的弧，因而容易求得顶点的出度和入度。  
图的十字链表表示是不唯一的，但一个十字链表表示确定的一个图。

**邻接多重表**是**无向图**的另一种**链式**存储结构。

**6.3 图的遍历**

**广度优先搜索（BFS）**：不是一个递归的算法，必须借助一个辅助**队列**，以记忆正在访问的顶点的下一层顶点。  
类似的思想还应用于Dijkstra单源最短路径算法和Prim最小生成树算法。  
**空间复杂度为O(|V|)；采用邻接表存储方式时，时间复杂度为O(|V|+|E|)；采用邻接矩阵存储方式时，时间复杂度为O(|V|^2)**。

广度优先生成树：一给定图的邻接矩阵存储表示是唯一的，故其广度优先生成树也是唯一的，但由于邻接表存储表示不唯一，故其广度优先生成树也是不唯一的。

**深度优先搜索（DFS）**：对于同样一个图，基于邻接矩阵的遍历所得到的DFS序列和BFS序列是唯一的，基于邻接表的遍历所得到的DFS序列和BFS序列是不唯一的。  
**空间复杂度为O(|V|)；采用邻接表存储方式时，时间复杂度为O(|V|+|E|)；采用邻接矩阵存储方式时，时间复杂度为O(|V|^2)**。

**6.4 图的应用**

**最小生成树（MST）：Prim算法、Kruskal算法**  
1）最小生成树**不是唯一的**。当图G中的各边权值互不相等时，G的最小生成树是唯一的；若无向连通图G的边比顶点树少1，即G本身就是一棵树时，Ｇ的最小生成树就是它本身。  
2）最小生成树的边的权值之和总是唯一的。  
3）最小生成树的边数为顶点数减１  
**Prim算法的时间复杂度为O(|V|^2)，不依赖于|E|，因此它适用于求解边稠密的的图的最小生成树。  
Kruskal算法适用于边稀疏而顶点较多的图。**

**最短路径**分为两类：一是单源最短路径：即求图中某一顶点到其他各顶点的最短路径，可通过**Dijkstra算法**求解；二是求每一对顶点间的最短路径，可通过**Floyd-Warshall算法**来求解。  
Dijkstra算法：单源最短路径时间复杂度为O(|V|^2)。如果边上带有负权值，Dijkstra算法并不适用。  
Floyd算法的时间复杂度为O(|V|^3)。Floyd算法允许图中带负权值的边，但不允许有包含带负权值的边组成的回路。Floyd算法同样也适用于带权无向图。

**有向无环图（DAG）**  
**AOV网**：如果用DAG图表示一个工程，其顶点表示活动，用有向边<Vi,Vj>表示活动Vi必须先于活动Vj进行的这样一种关系，则将这种有向图称为顶点表示活动的网络，记为AOV网。

**拓扑排序**：由一个有向无环图的顶点组成的序列，当且仅当满足下列条件时，称为该图的一个拓扑排序：  
1）每个顶点出现且只出现一次。  
2）若顶点A在序列中排在顶点B的前面，则在图中不存在从顶点B到顶点A的路径。  
每个DAG图都有一个或多个拓扑序列。  
对一个DAG图进行拓扑排序的算法：  
1）从DAG图中选择一个没有前驱的顶点并输出  
2）从图中删除该顶点和所有以它为起点的有向边  
3）重复1和2直到当前的DAG图为空或当前图中不存在无前驱的顶点为止。而后一种情况说明有向图中必然存在环  
**时间复杂度为O(|V|+|E|)**

**AOE网**：在带权有向图中，以顶点表示事件，有向边表示活动，边上的权值表示完成该活动的开销，则称这种有向图为用边表示活动的网络，简称为AOE网。  
1）只有在某顶点所代表的事件发生后，从该顶点出发的各有向边所代表的活动才能开始  
2）只有在进入某一顶点的各有向边所代表的活动都已经结束时，该顶点所代表的事件才能发生。  
在AOE网中，仅有一个入度为0的顶点，称为开始顶点（源点），它表示整个工程的开始，仅有一个出度为0的顶点，称为结束顶点（汇点），它表示整个工程的结束。  
从源点到汇点的所有路径中，具有最大路径长度的路径称为**关键路径**。关键路径上的活动称为关键活动。  
完成整个工程的最短时间就是关键路径的长度，也就是关键路径上各活动花费开销的总和。

**事件vk的最早发生时间ve(k)**：指从开始顶点V到Vk的最长路径长度。事件的最早发生时间决定了所有从Vk开始的活动能够开工的最早时间。（从前往后计算）  
**事件vk的最迟发生时间vl(k)**：指在不推迟整个工程完成的前提下，即保证它所指向的事件vi在ve(i)时刻能够发生时，该事件最迟必须发生的事件。（从后往前计算）  
**活动ai的最早开始时间e(i)**：指该活动的起点所表示的事件最早发生时间  
**活动ai的最迟开始时间l(i)**：指该活动的终点所表示的事件最迟发生时间与该活动所需时间之差。  
**一个活动ai的最迟开始时间l(i)和其最早开始时间e(i)的差额d(i)=l(i)-e(i)**：指该活动完成的时间余量，是在不增加完成整个工程所需的总时间的情况下，活动ai可以拖延的时间。如果为0，说明该活动必须要如期完成。

**求关键路径的算法步骤如下**：  
1）求AOE网中所有事件的最早发生时间ve()  
2）求AOE网中所有事件的最迟发生时间vl()  
3）求AOE网中所有活动的最早开始时间e()  
4）求AOE网中所有活动的最迟开始时间l()  
5）求AOE网中所有活动的差额d()，找出所有d()=0的活动构成关键路径

**注意**：  
1）关键路径上的所有活动都是关键活动，因此可通过加快关键活动来缩短整个工程的工期，但不能任意缩短关键活动，因为一旦缩短到一定程度，该关键活动可能变成非关键活动。  
2）网中的关键路径并不唯一。只提高一条关键路径上的的关键活动速度并不能缩短整个工程的工期，只有加快那些包括在所有关键路径上的关键活动才能达到缩短工期的目的。

第7章 查找

**7.1 查找的基本概念**

**查找**：  
1）查询某个特定的数据元素是否在查找表中  
2）检索满足条件的某个特定的数据元素的各种属性  
3）在查找表中插入一个数据元素  
4）从查找表中删除某个数据元素  
**静态查找表：顺序查找、折半查找、散列查找  
动态查找表：二叉排序树的查找、散列查找、二叉平衡树、B树**  
**平均查找长度**：一次查找的长度是指需要比较的关键字次数，而平均查找长度是所有查找过程中进行关键字的比较次数的平均值。**平均查找长度是衡量查找算法效率的最主要的指标**。

**7.2 顺序查找和折半查找**

**一般线性表的顺序查找：查找成功时的平均查找长度为(n+1)/2，查找不成功时的平均查找长度为n+1**；缺点是当n较大时，平均查找长度较大，效率低，优点是对数据元素的存储没有要求，顺序存储或链式存储皆可。  
**有序表的顺序查找：查找成功时的平均查找长度为(n+1)/2，查找不成功时的平均查找长度为n/2+n/(n+1)**。

**折半查找：又称为二分查找，仅适用于有序的顺序表，仅适合于线性表的顺序存储结构，不适合链式存储结构，且要求元素按关键字有序排列**。  
若有序序列有n个元素，则对应的判定树有n个圆形的非叶结点和n+1各方形的叶结点。  
**查找成功的平均查找长度**为1/n\*(1*1+2*2+…+h\*2^(h-1))。

**分块查找，又称为索引顺序查找**，吸取了顺序查找和折半查找各自的优点，既有动态结构，又适于快速查找。  
**基本思想**：将查找表分为若干个子块，块内的元素可以无序，但块之间是有序的，即第一个块中的最大关键字小于第二个块中的所有记录的关键字，以此类推。再建立一个索引表，索引表中的每个元素含有各块的最大关键字和各块总第一个元素的地址，索引表按关键字有序排列。  
**分块查找的平均查找长度为索引查找和块内查找的平均长度之和**，设将长度为n的查找表均匀分为b块，每块有s个记录，若在块内和索引表中均采用顺序查找，则**平均查找长度为(b+1)/2+(s+1)/2=(s^2+2s+n)/(2s)**，此时**若S=根号n，则平均查找长度取最小值**。若对索引表采用折半查找，则平均查找长度为**log2 (b+1)+(s+1)/2**

**7.3 B树和B+树**

B树，又称为多路平衡二叉树，B树中所有结点的孩子结点数的最大值称为B树的阶，通常用m表示。  
1）**树中每个结点至多有m棵子树（即至多含有m-1个关键字）**  
2）若根结点不是终端结点，则至少有两棵子树  
3）除根结点外的所有非叶结点至少有棵子树（即至少含有个关键字）  
4）**结点中关键字个数n满足m/2（向上取整）-1≤n≤m-1**  
B树是所有结点的平衡因子均等于0的多路查找树。  
B树中的大部分操作所需的磁盘存取次数于B树的高度成正比。应该明确B树的高度不包括最后的不带任何信息的叶结点所处的那一层。**高度logm (n+1)≤h≤logm/2 ((n+1)/2)+1其中m/2向上取整**

**B树的查找：即爱找到目标结点后，先将结点中的信息读入内存，然后再采用顺序查找法或折半查找法查找等于K的关键字  
1）在B树中找结点（在磁盘上进行）  
2）在结点内找关键字（在内存中进行）**

**B树的插入：  
1）定位  
2）插入：由于结点中的关键字个数n满足m/2（向上取整）-1≤n≤m-1，若插入后超过m-1，则必须对结点进行分裂**

**B树的删除：要使得删除后的结点中的关键字个数≥m/2（向上取整）-1，涉及到合并问题**

**m阶的B+树与m阶的B树的主要差异在于**：  
1）**在B+树中，具有n个关键字的结点只含有n棵子树，即每个关键字对应一棵子树；而在B树中，具有n个关键字的结点含有n+1棵子树**  
2）在B+树中，每个结点（非根内部结点）关键字个数n的范围是**m/2（向上取整）≤n≤m**（根结点1≤n≤m），在B树中，每个结点（非根内部结点）关键字个数n的范围是**m/2（向上取整）-1≤n≤m-1**（根结点1≤n≤m-1）  
3）在B+树中，叶结点包含信息，所有非叶结点仅起到索引作用，非叶结点中的每个索引项只含有对应子树的最大关键字和指向该子树的指针，不含有该关键字对应记录的存储地址。  
4）**在B+树中，叶结点包含了全部关键字**，即在非叶结点中出现的关键字也会出现在叶结点中；而在B树中，叶结点包含的关键字和其他结点包含的关键字是不重复的。

#B+树比B树更加适用于实际应用中的操作系统的文件索引和数据库索引，因为其磁盘读写代价更低，查询效率更加稳定。

**7.4 散列表**

**散列函数**：一个把查找表中的关键字映射称该关键字对应的地址的函数，记为Hash(key)=Addr  
**散列表**：是根据关键字而直接进行访问的数据结构，即散列表建立了关键字和存储地址之间的一种直接映射关系。  
理想情况下，对散列表进行查找的时间复杂度为O(1)，即与表中元素个数无关。

**散列函数的构造方法**：  
1.直接定址法：H(key)=a\*key+b  
2.**除留余数法**：H(key)=key%p（p是一个不大于m但最接近或等于m的质数）  
3.数字分析法  
4.平均取中法  
5.折叠法

**处理冲突的方法**：  
1.**开放定址法**：Hi=(H(key)+di)%m（m表示散列表表长，di为增量序列）  
1）线性探测法：di=0,1,2…m-1  
2）平方探测法（二次探测法）  
3）再散列法：Hi=(H(key)+i\*Hash2(key))%m  
4）伪随机序列法  
2.**拉链法**（链接法，chaining）

**散列表的查找效率取决于三个因素：散列函数、处理冲突的方法和装填因子**  
**装填因子**：一般记为a，定义为一个表的装满程度，即**a=表中记录数n/散列表长度m**  
**散列表的平均查找长度依赖于散列表的装填因子a，而不直接依赖于n或m**

第8章 排序

**8.1 排序的基本概念**

**内部排序**是指在排序期间元素全部存放在**内存**中的排序；**外部排序**是指在排序期间元素无法全部同时存放在内存中，必须在排序的过程中根据要求不断地在**内、外存**之间移动的排序。

**8.2 插入排序**

基本思想：每次将一个待排序的记录，按其关键字大小插入到前面已经排好序的子序列中，直到全部记录插入完成。

**直接插入排序：空间复杂度O(1)，时间复杂度O(n^2)，稳定，适用于顺序存储和链式存储的线性表**。

**注意：大部分排序算法都仅适用于顺序存储的线性表。**

**折半插入排序：时间复杂度O(n^2)，稳定**

**希尔排序：空间复杂度O(1)，时间复杂度O(n^1.3)，不稳定**

**8.3 交换排序**

**冒泡排序：空间复杂度O(1)，时间复杂度O(n^2)，稳定，每一趟排序都会将一个元素放置到其最终的位置**

**快速排序：空间复杂度最坏是O(n)，平均是O(log2 n)，时间复杂度O(nlog2 n)，不稳定，每一趟排序都会将一个元素（基准元素）放置到其最终的位置**，是所有内部排序算法中平均性能最优的排序算法

**8.4 选择排序**

**简单选择排序：空间复杂度O(1)，元素间比较的次数与序列的初始状态无关，始终是n(n-1)/2次，所以时间复杂度始终是O(n^2)，不稳定，每一趟排序都会将一个元素放置到其最终的位置**

**堆排序：空间复杂度O(1)，建堆时间O(n)，每次调整的时间复杂度为O(h)，时间复杂度O(nlog2 n)，不稳定**

**8.5 归并排序和基数排序**

**归并排序：空间复杂度O(n)，时间复杂度O(nlog2 n)，稳定**

**基数排序**：不是基于比较进行排序的，而是采用多关键字排序思想（即基于关键字各位的大小进行排序的），借助“分配”和“收集”两种操作对单逻辑关键字进行排序。**分为最高位优先排序和最低位优先排序**。  
*空间复杂度O®，时间复杂度O(d(n+r))，与序列的初始状态无关，稳定*\*

**8.6 各种内部排序算法的比较及应用**



**8.7 外部排序**

1）外部排序指待排序文件较大，内存一次放不下，需存放在外部介质的文件的排序  
2）为减少平衡归并中外存读写次数所采取的方法：**增大归并路数和减少归并段个数**  
3）**利用败者树增大归并路数**  
4）**利用置换-选择排序增大归并段长度来减少归并段个数**  
5）**由长度不等的归并段，进行多路平衡归并，需要构造最佳归并树**