数据结构与算法学习

1.数据结构绪论

1.1基本概念和术语

（1）数据：描述客观事物的符号，是计算机中可以操作的对象，是能被计算机识别，并输入给计算机处理的符号集合。不仅包括整型、实型等数值类型，还包括字符及声音、图像、视频等非数值类型。

（2）数据元素：是组成数据的、有一定意义的基本单位，在计算机中通常作为整体处理。也被称为记录。

（3）数据项：一个数据元素可以由若干数据项组成。数据项是数据不可分割的最小单位。

（4）数据对象：是性质相同的数据元素的集合，是数据的子集。

（5）数据结构：是相互之间存在一种h或多种特定关系的数据元素的集合。

1.2逻辑结构与物理结构

（1）逻辑结构：是指数据对象中数据元素之间的相互关系，分为四种：集合结构、线性结构、树形结构、图形结构。

（2）物理结构：是指数据的逻辑结构在计算机中的存储形式。两种：顺序存储（地址连续的空间）和链式存储（地址不一定连续，通过指针访问）。

1.3抽象数据类型

（1）数据类型：是指一组性质相同的值的集合及定义在此集合上的一些操作的总称。包括两类：原子类型（是不可以再分解的基本类型，包括整型、实型、字符型等）和结构类型（有若干个类型组合而成，是可以再分解的。例如整型数组是由若干整型数据组成）。

（2）抽象数据类型（Abstract Data Type，ADT）：是指一个数学模型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一组逻辑特性，而与其在计算机内部如何表示和实现无关。

2.算法

2.1数据结构和算法

程序=数据结构+算法

2.2两种算法的比较

2.3算法定义

算法是解决特定问题求解步骤的描述，在计算机中表现为指令的有限序列，并且每条指令表示一个或多个操作。

2.4算法的特性

（1）输入输出：算法具有零个或多个输入，至少有一个或多个输出。

（2）有穷性：指算法在执行有限的步骤之后，自动结束而不会出现无限循环，并且每个步骤在可接受的时间内完成。

（3）确定性：算法的每一步骤都具有确定的含义，不会出现二义性。

（4）可行性：算法的每一步都必须是可行的，也就是说，每一步都能够通过执行有限次数完成。

2.5算法的设计要求

（1）正确性：指算法至少应该具有输入、输出和加工处理无歧义性、能正确反映问题的需求、能够得到问题的正确答案。

（2）可读性：便于阅读、理解和交流。

（3）健壮性：当输入数据不合法时，算法也能够做出相关处理，而不是产生异常或莫名其妙的结果。

（4）时间效率高和存储量低。

2.6算法效率的度量方法

（1）事后统计方法：这种方法主要是通过设计好的测试程序和数据，利用计算机计时器对不同算法编制的程序的运行时间进行比较，从而确定算法效率的高低。

（2）事前分析估计方法：在计算机程序编制前，依据统计方法对算法进行评估。

2.7函数的渐近增长

判断一个算法的效率时，函数的常数和其他次要项常常可以忽略，而更应该关注（最高级项）的阶数。

2.8算法的时间复杂度

（1）定义：在进行算法分析时，语句总的执行次数T(n)是关于问题规模n的函数，进而分析T(n)随n的变化情况并确定T(n)的数量级。算法

的时间复杂度，也就是算法的时间度量，记作：T(n)=O(f(n))。它表示随着问题规模n的增大，算法执行时间的增长率和f(n)的增长率相同，称作算法的渐近时间复杂度，简称为时间复杂度。其中f(n)是问题规模n的某个函数。

（2）大O阶方法：

- 用常数1取代运行时间中的所有加法常数；
- 在修改后的运行次数函数中，只保留最高级项；
- 如果最高阶项存在且不是1，则去除与这个项相乘的常数。

2.9常见的时间复杂度

执行次数函数	阶	非正式术语
12	O(1)	常数阶
2n+3	O(n)	线性阶
3n^2+2n+1	O(n^2)	平方阶
5log2n+20	O(logn)	对数阶
2n+3nlog2n+19	O(nlogn)	nlogn阶
6n^3+2n^2+3n+4	O(n^3)	立方阶
2^n	O(2^n)	指数阶

O(1) < O(logn) < O(n) < O(nlogn) < O(n^2) < O(n^3) < O(2^n) < O(n!) < O(n^n)

2.10最坏情况和平均情况

（1）最坏情况：最坏情况运行时间是一种保证，那就是运行时间将不会再坏了。在应用中，这是一种重要的需求，通常，除非特别指定，提到的运行时间都是最坏情况的运行时间。

（2）最好情况。

（3）平均情况：平均运行时间是所有情况中最有意义的，因为它是期望的运行时间。

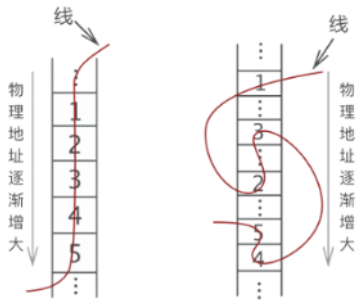
2.11算法空间复杂度

算法的空间复杂度通过计算算法所需的存储空间实现，计算公式记作：S(n)=O(f(n))，其中，n是问题规模，f(n)为语句关于n所占存储空间的函数。

3.线性表

3.1线性表的定义

线性表，全名为线性存储结构。使用线性表存储数据的方式可以这样理解，即“把所有数据用一根线儿串起来，再存储到物理空间中”。



使用线性表存储的数据，如同向数组中存储数据那样，要求数据类型必须一致，也就是说，线性表存储的数据，要么全部都是整形，要么全部都是字符串。一半是整形，另一半是字符串的一组数据无法使用线性表存储。

线性表存储数据可细分为以下 2 种：

（1）将数据依次存储在连续的整块物理空间中，这种存储结构称为顺序存储结构（简称顺序表）；

（2）数据分散的存储在物理空间中，通过一根线保存着它们之间的逻辑关系，这种存储结构称为链式存储结构（简称链表）；

3.2线性表的顺序存储结构（顺序表）及实现

3.2.1顺序表（顺序存储结构）

（1）定义

顺序表，全名顺序存储结构，是线性表的一种。线性表用于存储逻辑关系为“一对一”的数据，顺序表自然也不例外。不仅如此，顺序表对数据的物理存储结构也有要求。顺序表存储数据时，会提前申请一整块足够大小的物理空间，然后将数据依次存储起来，存储时做到数

据元素之间不留一丝缝隙。

（2）地址计算方法

LOC(ai)=LOC(a1)+(i-1)*c，顺序表序号从1开始（a1，a2，...），数组从0开始。

（3）数据长度和线性表长度

数组长度是存放线性表的存储空间的长度。线性表的长度是线性表中数据元素的个数。

3.2.2顺序表的实现

顺序存储结构定义

（2）构造函数（无参、有参）

（3）求线性表长度

（4）查找操作（按位查找、按值查找）

（5）插入操作

（6）删除操作

（7）遍历操作

（8）反转操作

3.2.3顺序表的优缺点

（1）优点

- 逻辑与物理顺序一致，顺序表能够按照下标直接快速的存取元素；
- 无须为了表示表中元素之间的逻辑关系而增加额外的存储空间；

（2）缺点

- 线性表长度需要初始定义，常常难以确定存储空间的容量，所以只能以降低效率的代价使用扩容机制；
- 插入和删除操作需要移动大量的元素，效率较低；
- 造成存储空间的“碎片”；

3.2.4总结

读取数据的时候，它的时间复杂度为 O(1)，插入和删除数据的时候，它的时间复杂度为 O(n)，所以线性表中的顺序表更加适合处理一些元素个数比较稳定，查询读取多的问题。

3.3线性表的链式存储结构及实现

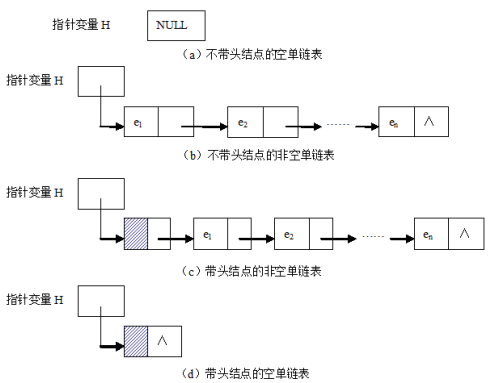
3.3.1单链表

3.3.1.1单链表的存储方法

单链表是用一组任意的存储单元存放线性表的元素，这组存储单元可以连续也可以不连续，甚至可以零散分布在内存中的任意位置。为了能正确表示元素之间的逻辑关系，每个存储单元在存储数据元素的同时，还必须存储其后继元素所在的地址信息，这个地址信息称为指针，这两部分组成了数据元素的存储映像，称为结点（node）。

```
template<class DataType>
struct Node
{
    DataType data;
    Node<DataType> * next;
};
```

单链表中每个结点的存储地址存放在其前驱结点的next域中，而第一个元素无前驱，所以设头指针指向第一个元素所在结点，整个单链表的存取必须从头指针开始进行，因而头指针具有标识一个单链表的作用。同时，最后一个元素无后继，故最后一个元素所在结点的指针域为空，即NULL，也称尾标志。



通常使用带头结点的单链表。

3.3.1.2单链表的实现

（1）遍历操作

所谓遍单链表是指按序号依次访问单链表中的所有结点且仅访问一次。可以设置一个工作指针p依次指向各结点，当指针p指向某结点时输出该结点的数据域，直到p为NULL为止。

```
template<class DataType>
void LinkList<DataType>::PrintList()
{
    p = first->next;
    while(p != NULL)
    {
        cout << p->data;
        p = p->next;
    }
}
```

（2）求线性表的长度

（3）查找操作（按位查找、按值查找）

（4）插入操作

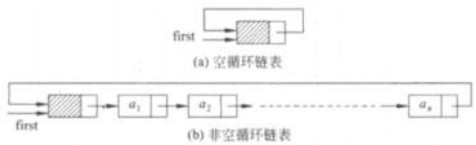
（5）构造函数（无参、有参（头插法、尾插法））

（6）删除操作

（7）析构函数

3.3.2循环链表

在单链表中，如果将终端结点的指针域由空指针改为指向头结点，就使整个单链表形成一个环，这种头尾相接的单链表称为循环单链表，简称循环链表。为了使空表和非空表的处理一致，通常也附设一个头结点。



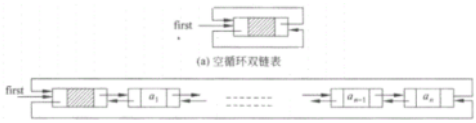
在用头指针指示的循环链表中，找到开始结点的时间是O(1)，然而要找到终端结点，则需从头指针开始遍历整个循环链表，其时间是O(n)。在很多实际问题中，操作是在表的首或尾位置上进行，此时头指针指示的循环链表就显得不够方便。如果改用指向终端结点的尾指针来指示循环链表，则查找开始结点和终端结点都很方便，它们的存储地址分别是（rear->next）->next和rear，显然，时间都是O(1)。因此实际中多采用尾指针指示的循环链表。

循环链表没有增加任何存储量，仅对单链表的链接方式稍作改变，因而其抽象数据类型相同。循环链表的基本操作的实现与单链表类似，不同之处仅在于循环条件不同。从循环链表中任一结点出发，可扫描到其他结点，从而增加了链表操作的灵活性。但这种方法的危险在于循环链表中没有明显的尾端，可能会使循环链表的处理操作进入死循环，所以，需要格外注意循环条件。通常判断用作循环变量的工作指针是否等于某一指定指针（如头指针或尾指针），以判定工作指针是否扫描了整个循环链表。

3.3.3双链表

在循环链表中，虽然从任一结点出发可以扫描到其他结点，但要找到其前驱结点，则需要遍历整个循环链表。如何希望快速确定表中任一结点的前驱结点，可以在单链表的每个结点中再设置一个指向其前驱结点的指针域，这样就形成了双链表。

和单链表类似，双链表一般也是由头指针唯一确定，增加头结点也能使双链表的某些操作变得方便，将头结点和尾结点链接起来也能构成循环双链表，这样，无论是插入还是删除操作，对链表中开始结点、终端结点和中间任意结点的操作过程相同。实际应用中通常采用带头结点的循环双链表。



循环双链表具有对称性，即某一结点的存储地址既存放在其前驱结点的后继指针中，也存放在它的后继结点的前驱指针中。

在循环双链表中求表长、按位查找、按值查找、遍历等操作的实现与单链表基本相同，不同的只是插入和删除操作。由于循环双链表是一种对称结构，使得在某一结点的之前或之后执行插入和删除操作都很容易。

（1）插入

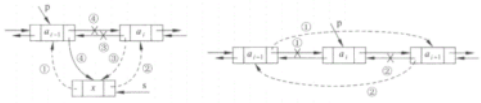
在结点p的后面插入一个新结点s，需要修改4个指针：

- s->prior=p
- s->next=p->next
- p->next->prior=s
- p->next=s

（2）删除

设指针p指向待删除结点，删除操作可以通过下述两条语句完成：

- (p->prior)->next=p->next
- (p->next)->prior=p->prior

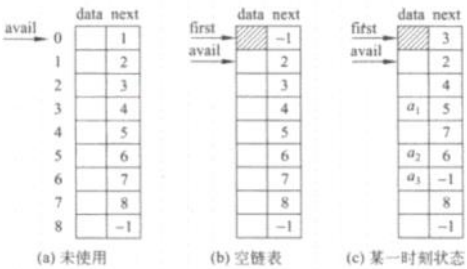


3.4线性表的其他存储方法

3.4.1静态链表

静态链表是用数组来表示单链表，用数组元素的下标来模拟单链表的指针。静态链表的每个数组元素由两个域构成：data域存放数据元素，next域存放该元素的后继元素所在的数组下标。由于它是利用数组定义的，属于静态存储分配，因此叫做静态链表。

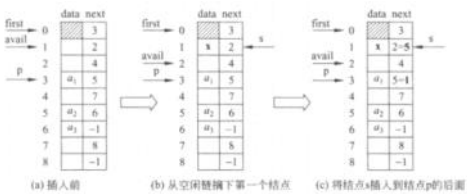
```
const int MaxSize = 100;
template<class DataType>
struct SNode
{
    DataType data;
    int next;
}SList[MaxSize];
```



上图是静态链表的存储示意图，其中，avail是空闲链（所有空闲数组单元组成的单链表）头指针，first是静态链表头指针，为了运算方便，通常静态链表也带头结点。

（1）在静态链表中进行插入操作时，首先从空闲链的最前端摘下一个结点，将该结点插入静态链表中，如下图所示。假设新结点插在结点p的后面，则修改指针的操作为：

- s=avail
- avail=SList[avail].next
- SList[s].data=x;
- SList[s].next=SList[p].next
- SList[p].next=s



3.4.2间接寻址

线性表的顺序存储利用了数组单元在物理位置上的邻接关系来表示线性表中数据元素的逻辑关系，这一特点使得顺序表可以实现随机存取；线性表的链接存储利用指针将表中元素依次串联在一起，这一特点使得在修改线性表中元素之间的逻辑关系时，只需修改相应指针而不需要移动元素。间接寻址是将数组和指针结合起来的一种方法，它将数组中存储元素的单元改为存储指向该元素的指针。

在间接寻址存储的线性表中，在位置i处执行插入操作，也需要将i，i+1，…，n处的元素指针移到位置i+1，i+2，…，n+1处，然后将指向新元素的指针填入位置i处。与顺序表不同的是，这里移动的不是元素而是指向元素的指针。虽然该算法的时间复杂度仍为O(n)，但当每个元素占用的空间比较大时，比顺序表的插入操作快得多。在间接寻址的线性表中执行删除操作同插入操作类似。

所以，线性表的间接寻址保持了顺序表随机存取的优点，同时改进了插入和删除操作的时间性能，但是它也没有解决连续存储分配带来的表长难以确定的问题。

4.栈和队列

4.1栈

4.1.1栈的逻辑结构

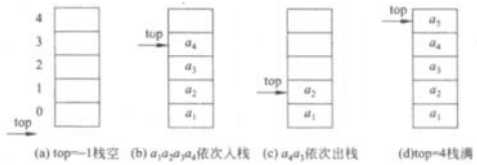
4.1.1.1定义

4.1.1.2抽象数据类型定义

4.1.2栈的顺序存储结构

4.1.2.1顺序栈

栈的顺序存储结构称为顺序栈。顺序栈被指上是顺序表的简化，唯一需要确定的是用数组的哪一端表示栈底。通常把数组中下标为0的一端作为栈底，同时附设指针top指示栈顶元素在数组中的位置。设存储栈元素的数组长度为StackSize，则栈空时栈顶指针top=-1；栈满时栈顶指针top=StackSize-1。入栈时，栈顶指针top加1；出栈时，栈顶指针top减1。



4.1.2.2顺序栈的实现

（1）栈的初始化

初始化一个空栈只需将栈顶指针top设置为-1。

（2）入栈操作

在栈中插入一个元素x只需将栈顶指针top加1，然后在top指向的位置填入元素x。

（3）出栈操作

删除栈顶元素只需取出栈顶元素，然后将栈顶指针top减1。

（4）取栈顶元素

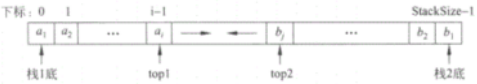
取栈顶元素只需将top指向的栈顶元素取出，并不修改栈顶指针。

（5）判空操作

顺序栈的判空操作只需判断top==-1是否成立，如果成立，则栈为空，返回1；如果不成立，则栈非空，返回0。

4.1.2.3两栈共享空间

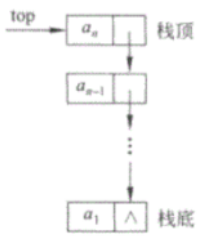
在一个程序中如果需要同时使用具有相同数据类型的两个栈时，最直接的方法是为每个栈开辟一个数组空间，不过这样做的结果可能出现一个栈的空间已被占满而无法再进行插入操作，同时另一个栈的空间仍有大量剩余而没有利用的情况，从而造成存储空间的浪费。一种可取的方法是充分利用顺序栈单项延伸的特性，使用一个数组来存储两个栈，让一个栈的栈底为该数组的始端，另一个栈的栈底为该数组的末端，每个栈从各自的端点向中间延申。



4.1.3栈的链接存储结构

4.1.3.1链栈

栈的链接存储结构称为链栈。通常链栈用单链表表示，因此其结点结构与单链表的结点结构相同。因为只能在栈顶执行插入和删除操作，显然以单链表的头部做栈顶是最方便的，而且没有必要像单链表那样为了运算方便附加一个头结点。



4.1.3.2链栈的实现

（1）构造函数

构造函数的作用是初始化一个空链栈，由于链栈不带头结点，因此只需将栈顶指针top置空。

（2）入栈操作

（3）出栈操作

（4）取栈顶元素

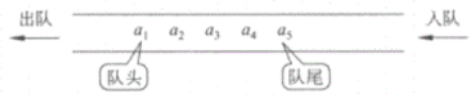
（5）判空操作

（6）析构函数

4.2队列

4.2.1队列的逻辑结构

4.2.1.1队列的定义

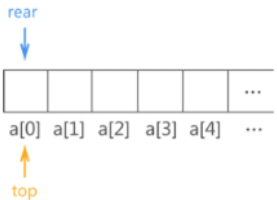


4.2.1.2队列的抽象数据类型定义

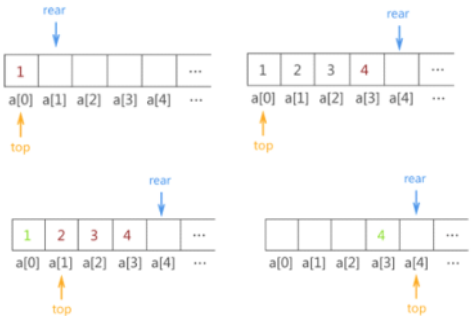
4.2.2队列的顺序存储结构

4.2.2.1顺序队列

由于顺序队列的底层使用的是数组，因此需预先申请一块足够大的内存空间初始化顺序队列。除此之外，为了满足顺序队列中数据从队尾进，队头出且先进先出的要求，我们还需要定义两个指针（top 和 rear）分别用于指向顺序队列中的队头元素和队尾元素。



由于顺序队列初始状态没有存储任何元素，因此 top 指针和 rear 指针重合，且由于顺序队列底层实现靠的是数组，因此 top 和 rear 实际上是两个变量，它的值分别是队头元素和队尾元素所在数组位置的下标。当有数据元素进队列时，对应的实现操作是将其存储在指针 rear 指向的数组位置，然后 rear+1；当需要队头元素出队时，仅需做 top+1 操作。



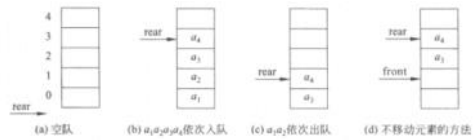
存在问题：通过对比两张图，你会发现，指针 top 和 rear 重合位置指向了 a[4] 而不再是 a[0]。也就是说，整个顺序队列在数据不断地进队出队过程中，在顺序表中的位置不断后移。

顺序队列整体后移造成的影响是：

- 顺序队列之前的数组存储空间将无法再被使用，造成了空间浪费；
- 如果顺序表申请的空间不足够大，则直接造成程序中数组 a 溢出，产生溢出错误；

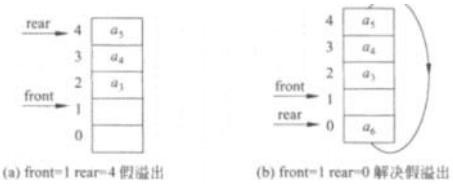
4.2.2.2循环队列

假设线性表有n个数据元素，顺序表要求把表中的所有元素都存储在数组的前n个单元。结社队列有n个元素，顺序存储的队列也应该把队列的所有元素都存储在数组的前n个单元。如果把队头元素放在数组下标为0的一端，则入队操作的时间开销仅为O(1)，此时的入队操作相当于追加，不需要移动元素；但是出队操作的时间开销为O(n)，因为要保证剩下的n-1个元素仍然存储在数组的前n-1个单元，所有元素都要向前移动一个位置。



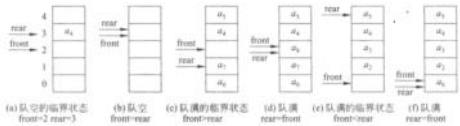
如果放宽队列的所有元素必须存储在数组的前n个单元这一条件，只要求队列的元素存储在数组中连续的位置，就可以得到一种更为有效的方法，如上图，此时入队和出队操作的时间开销都是O(1)，因为没有移动任何元素，但是队列的队头和队尾都是活动的，因此，需要设置队头、队尾两个指针，并且约定：队头指针front指向队头元素的前一个位置，队尾指针rear指向队尾元素。

但是这种方法有一个新问题。随着队列的插入和删除操作，整个队列向数组下标较大的位置移动，从而产生了队列的“单向移动性”。当元素被插入到数组中下标最大的位置上之后，队列的空间就用尽了，尽管此时数组的低端还有空闲空间，这种现象叫做“假溢出”，如下图。



解决假溢出的方法是将存储队列的数组看成是头尾相接的循环结构，即允许队列直接从数组中下标最大的位置延续到下标最小的位置，如下图。者通过取模操作很容易实现。队列的这种头尾相接的顺序存储结构称为循环队列。

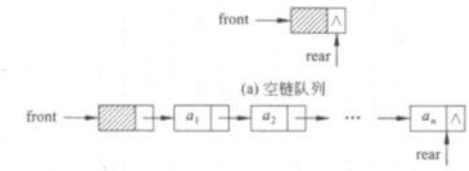
在循环队列中有一个很重要的问题：队空和队满的判定问题。如下图(a)所示，队列中只有一个元素，执行出队操作，则队头指针加1后与队尾指针相等，即队空的条件是front=rear；(c)和(e)所示数组中只有一个空闲单元，执行入队操作，则队尾指针加1后与队头指针相等，即队满的条件也是front=rear。如何将队空和队满的判定条件区分？可以浪费一个数组的元素空间，把(c)和(e)所示的情况视为队满，此时队尾指针和队头指针正好差1，即队满的条件是：(rear+1)%QueueSize=front。



4.2.3队列的链接存储结构

4.2.3.1链队列

链队列是在单链表的基础上做了简单的修改，为了使空队列和非空队列的操作一致，链队列也加上了头结点。根据队列的先进先出特性，为了操作上的方便，设置队头指针指向链队列的头结点，队尾指针指向终端结点。



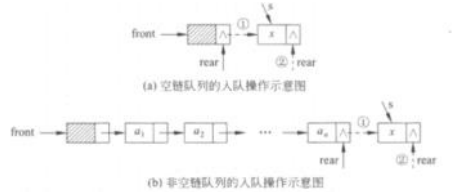
4.2.3.2链队列的实现

（1）构造函数

初始化一个空的链队列，只需申请一个头结点，然后让队头指针和队尾指针均指向头结点。

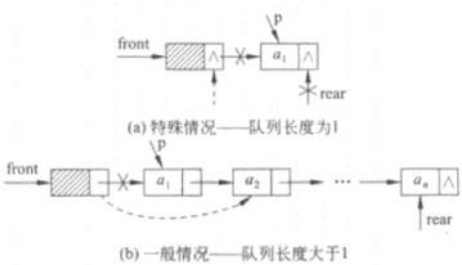
（2）入队操作

链队列的插入操作只考虑在链表的尾部进行，由于链队列带头结点，空链队列和非空链队列的入队操作一致。



（3）出队操作

链队列的删除只考虑在链队列的头部进行，需要注意队列长度等于1的特殊情况。



（4）取队头元素

（5）判空操作

（6）析构函数

5.字符串和多维数组

5.1字符串

5.1.1字符串的定义

（1）定义

字符串是零个或多个字符组成的有限序列，只包含空格的串称为空格串。串中所包含的字符个数称为串的长度，长度为0的串称为空串。

（2）字符串的比较

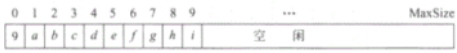
5.1.2字符串的存储结构

字符串是数据元素为单个字符的线性表，一般采用顺序存储。类似于顺序表，字符串的顺序存储结构是用数组来存储串中的字符序列。在Pascal、C/C++、Java等语言中，字符串都是采用顺序存储。在字符串的顺序存储中，一般有三种方法表示串的长度：

（1）用一个变量来表示串的长度。

（2）在串尾存储一个不会在串中出现的特殊字符作为字符串的终结符，例如，在C/C++中用'\0'来表示串的结束。这种存储方法不能直接得到串的长度，而是通过判断当前字符是否为'\0'来确定串是否结束，从而求得串的长度。

（3）用数组的0号单元存放串的长度，串值从1号单元开始存放。

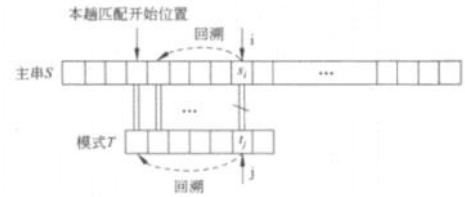


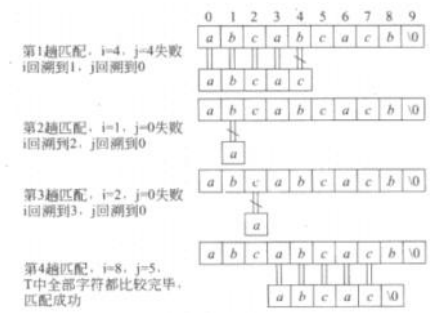
5.1.3模式匹配

给定两个字符串S="s1s2……sn"和T="t1t2……tm"，在主串S中寻找子串T的过程称为模式匹配，T称为模式。如果匹配成功，返回T在S中的位置；如果匹配失败，返回0。

5.1.3.1朴素的模式匹配算法

这是一种带回溯的匹配算法，简称BF算法，其基本思想是：从主串S的第一个字符开始和模式T的第一个字符进行比较，若相等，则继续比较二者的后续字符；否则，从主串S的第二个字符开始和模式T的第一个字符进行比较，重复上述过程，直至S或T中的所有字符比较完毕。若T中的所有字符比较完毕，则匹配成功，返回本趟匹配的开始位置；否则匹配失败，返回0。

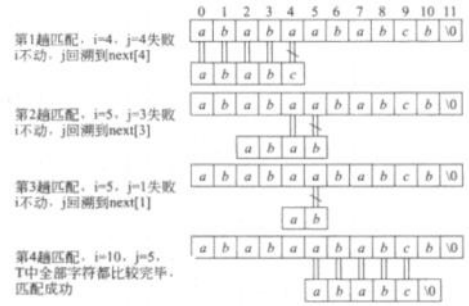





```
int BF(char S[], char T[])
{
    int i = 0, j = 0;
    while((S[i] != '\0') && (T[i] != '\0'))
    {
        if(S[i] == T[j]){i++;j++}
        else{i=i-j+1;j=0;}
    }
        if(T[j] == '\0') return (i-j+1);
        else return 0;
}
```

5.1.3.1改进的模式匹配算法

BF算法简单但效率低，一种对BF算法做了很大改进的模式匹配算法是KMP算法，其基本思想是主串不进行回溯。



5.2多维数组

5.2.1数组的定义

5.2.2数组的存储结构与寻址

5.3矩阵的压缩存储

矩阵是很多科学与工程计算问题中的处理对象。在实际应用中，经常出现一些阶数很高的矩阵，同时在矩阵中有很多值相同的元素并且它们的分布有一定的规律——称为特殊矩阵，或者矩阵中有很多零元素——稀疏矩阵，可以对这类矩阵进行压缩存储，从而节省存储空间，并使矩阵的各种运算能有效地进行。压缩存储的基本思想是：为多个值相同的元素只分配一个存储空间；对零元素不分配存储空间。

5.3.1对称矩阵的压缩存储

5.3.2三角矩阵的压缩存储

5.3.3对角矩阵的压缩存储

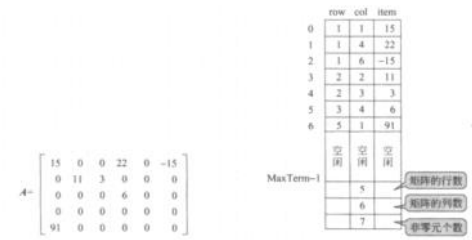
5.3.4稀疏矩阵的压缩存储

稀疏矩阵是零元素居多的矩阵。在工程应用中，通常会遇到阶数很高的大型稀疏矩阵，如果按常规方法存储，势必会存储大量的零元素，造成存储浪费。一个显然的存储方法是仅存非零元素。但对于这类矩阵，通常非零元素的分布是没有规律的，为了能找到相应的元素，仅存储非零元素是不够的，还要存储该元素所在的行号和列号，即每个非零元素表示为三元组（行号，列好，非零元素值）。将稀疏矩阵的非零元素对应的三元组所构成的集合，按行优先的顺序排列成一个线性表，称为三元组表，则稀疏矩阵的压缩存储转化为三元组表的存储。

5.3.4.1三元组顺序表

采用顺序存储结构存储的三元组表称为三元组顺序表，显然，要唯一表示一个稀疏矩阵，还需要在存储三元组表的同时存储该矩阵的行数、列数和非零元素个数。

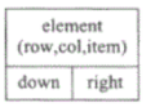
```
const int MaxTerm=100;
struct SparseMatrix
{
    element data[MaxTerm];
    int mu,nu,tu //行数、列数和非零元素个数
}
```



5.3.4.2十字链表

采用三元组顺序表存储稀疏矩阵，对于矩阵的加法、乘法等操作，非零元素的个数及位置都会发生变化，这时顺序存储方法就十分不便。稀疏矩阵的链接存储结构称为十字链表，它具备链接存储的特点，因此，在非零元素的个数及位置都发生变化的情况下，通常采用十字链表存储稀疏矩阵。

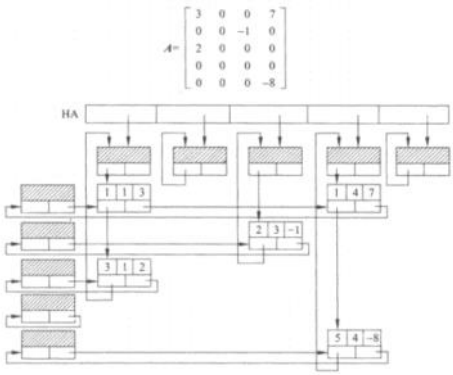
十字链表存储稀疏矩阵的基本思想是：将每个非零元素对应的三元组存储为一个链表结点，结点由5个域组成。



其中，element为数据域，存储非零元素对应的三元组；right为指针域，指向同一行中的下一个三元组；down为指针域，指向同一列中的下一个三元组。

稀疏矩阵中每一行的非零元素按其列号从小到大顺序由right域链成一个带头结点的循环行链表，每一列的非零元素按其行号从小到大顺序由down域也链成一个带头结点的循环列链表，即每个非零元素aij既是第i行循环链表中的一个结点，又是第j列循环链表中的一个结点。由

于各链表头结点的row域、col域和item域均为空，行链表头结点只用right域，列链表头结点只用down域，故这两组头结点可以合用，也就是说，对于第i行的循环链表和第i列的循环链表可以公用同一个头结点。为了实现对某一行（或某一列）头指针的快速查找，将指向这些头结点的头指指针存储在一个数组HA中。



6.数和二叉树

6.1树的逻辑结构

6.1.1树的定义和基本术语

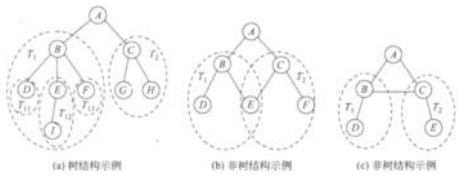
6.1.1.1树的定义

在树中常常将数据元素称为结点。树是n（n≥0）个结点的有限集合。当n=0时，称为空树；任意一棵非空树满足以下条件：

（1）有且仅有一个特定的称为根的结点；

（2）当n＞1时，除根结点之外的其余结点被分成m（m＞0）个互不相交的有限集合T1，T2，…，Tm，其中每个集合又是一棵树，并成为这个根结点的子树。

显然，树的定义是递归的。



6.1.1.2树的基本术语

（1）结点的度、树的度

某结点所拥有的子树的个数称为该结点的度；树中各结点度的最大值称为该树的度。上图(a)所示的树中，结点A的度为2，结点B的度为3，该树的度为3。

（2）叶子结点、分支结点

度为0的结点称为叶子结点，也成为终端结点；度不为0的结点称为分支结点，也成为非终端结点。上图(a)中，结点D、I、F、G和H是叶子结点，其余为分支结点。

（3）孩子结点、双亲结点、兄弟结点

某结点的子树的根结点称为该结点的孩子结点；反之，该结点称为其孩子结点的双亲结点；具有同一双亲的孩子结点互称为兄弟结点。上图(a)中，结点B是结点A的孩子结点，结点A是结点B的双亲结点，结点B和C互为兄弟，结点I没有兄弟。

（4）路径、路径长度

如果树的结点序列n1，n2，…，nk满足如下关系：结点ni是结点ni+1的双亲（1≤i＜k），则把n1，n2，…，nk称为一条由n1至nk的路径；路径上经过的边数称为路径长度。显然，在树中路径是唯一的。上图(a)中，从结点A到结点I的路径是A、B、E和I，路径长度为3。

（5）祖先、子孙

如果从结点x到结点y有一条路径，那么结点x就称为y的祖先，y称为x的子孙。显然，以某结点为根的子树中的任一结点都是该结点的子孙。如上图(a)中，结点A、B、E称为结点I的祖先，结点B的子孙有D、E、F、I。

（6）结点的层数、树的深度（高度）

规定根结点的层数为1，对其余任何结点，若某结点在第k层，则其孩子结点在第k+1层；树中所有结点的最大层数称为树的深度，也成为树的高度。如上图(a)中，结点D的层数为3，树的深度为4。

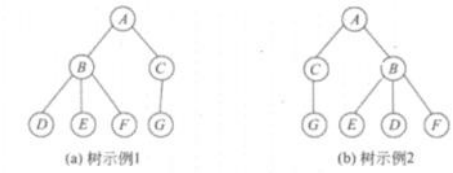
（7）层序编号

将树中结点按照从上层到下层、同层从左到右的次序依次给它们编以从1开始的连续自然数，树的这种编号方式称为层序编号。显然，通过

层序编号可以将一棵树变成线性序列。上图(a)中，结点A的编号为1，结点F的编号为6。

（8）有序树、无序树

如果一棵树中结点的各子树从左到右是有次序的，即若交换了结点各子树的相对位置，则构成不同的树，称这棵树为有序树；反之，称为无序树。下图中，若为有序树，则(a)和(b)为两棵不同的树；若为无序树，则(a)和(b)为同一棵树。除特殊说明，在数据结构中讨论的树一般都是有序树。



（9）森林

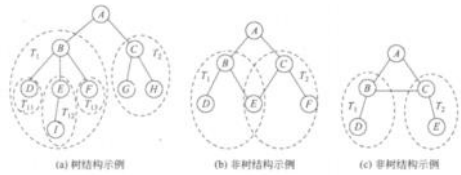
m（m≥0）棵互不相交的树的集合构成森林。任何一棵树，删去根结点就变成了森林。如上图(a)中，删去根结点A就变成了由两棵树构成的森林。

6.1.2树的抽象数据类型定义

6.1.3树的遍历操作

树中最基本的操作是遍历。树的遍历是指从根结点出发，按照某种次序访问树中所有结点，使得每个结点被访问一次且仅被访问一次。“访问”的含义很广，是一种抽象操作，在实际应用中，可以是对结点进行的各种处理，比如输出结点信息、修改结点的某些数据等。

由树的定义可知，一棵树由根结点和m棵子树构成，因此，只要依次遍历根结点和m棵子树，就可以遍历整棵树。通常有前序遍历、后序遍历和层序遍历三种方式。



（1）前序遍历

若树为空，则空操作返回；否则：

- 访问根结点；
- 按照从左到右的顺序前序遍历根结点的每一棵子树。

上图(a)中前序遍历结果为：ABDEIFCGH。

（2）后序遍历

若树为空，则空操作返回；否则：

- 按照从左到右的顺序后序遍历根结点的每一棵子树；
- 访问根结点。

上图(a)中后序遍历结果为：DIEFBGHCA。

（3）层序遍历

树的层序遍历也称为树的广度遍历，其操作定义为从树的第一层（即根结点）开始，自上而下逐层遍历，在同一层中，从左到右的顺序对结点逐个访问。

上图(a)中层序遍历结果为：ABCDEFGHI。

6.2树的存储结构

6.2.1双亲表示法

基于数组或称静态链表。

下标	data	parent
0	A	-1
1	B	0
2	C	0
3	D	1
4	E	1
5	F	1
6	G	2
7	H	2
8	I	4

(a) 树的双亲表示法

下标	data	parent	rightsib
0	A	-1	-1
1	B	0	2
2	C	0	-1
3	D	1	4
4	E	1	5
5	F	1	-1
6	G	2	7
7	H	2	-1
8	I	4	-1

(b) 带右兄弟的双亲表示法

6.2.2孩子表示法

树的孩子表示法是一种基于链表的存储方法，主要有两种形式。

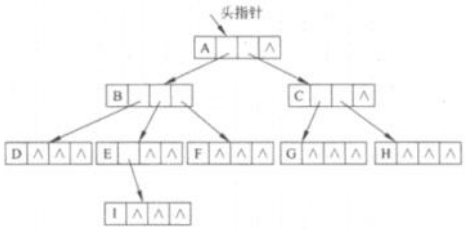
6.2.2.1多重链表表示法

（1）指针域的个数等于该结点的度

节省空间，实现困难。

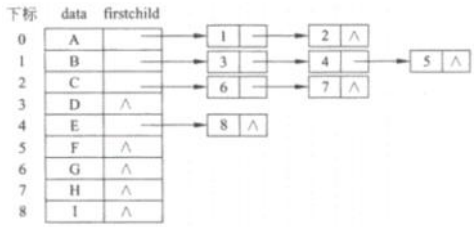
（2）指针域的个数等于树的度

浪费存储空间，链表中各结点同构，实现容易，适合于各结点度相差不大的情况。



6.2.2.2孩子链表表示法

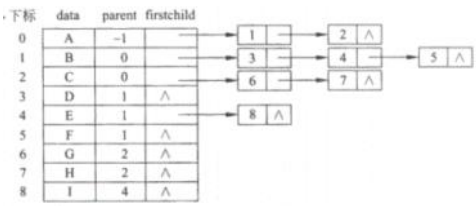
孩子链表表示法是一种用多个单链表表示树的方法，即把每个结点的孩子排列起来，看成是一个线性表，且以单链表存储，称为该结点的孩子链表。



孩子链表表示法不仅表示了孩子结点的信息，而且链在同一个单链表中的结点具有兄弟关系。与双亲表示法相反，在汉字链表表示法中查找双亲比较困难。把双亲表示法和孩子链表表示法结合起来，就形成了双亲孩子表示法。

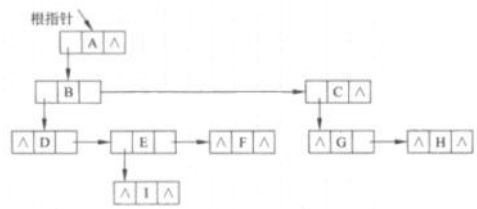
6.2.3双亲孩子表示法

双亲孩子表示法是将双亲表示法和孩子链表表示法相结合的存储方法。仍将各结点的孩子结点分别组成单链表，同时用一维数组顺序存储树中的各结点，数据元素除了包括结点的数据信息和该结点的孩子链表的头指针外，还增设一个域存储该结点的双亲结点在数组中的下标。



6.2.4孩子兄弟表示法

树的孩子兄弟表示法又称为二叉链表表示法，其方法是链表中的每个结点除数据域外，还设置了两个指针分别指向该结点的第一个孩子和右兄弟。



这种存储方法便于实现树的各种操作。例如，要访问某结点x的第i个孩子，只需从该结点的第一个孩子指针找到第1个孩子后，沿着孩子结点的右兄弟域连续走i-1步，便可找到结点x的第i个孩子。

6.3二叉树的逻辑结构

6.3.1二叉树的定义

二叉树是n（n≥0）个结点的有限集合，该集合或者为空集（称为空二叉树），或者由一个根结点和两棵互不相交的、分别称为根结点的左子树和右子树的二叉树组成。

二叉树特点：每个结点最多有两棵子树，所以二叉树中不存在度大于2的结点；二叉树是有序的，其次序不能任意颠倒，即使树中的某个结点只有一棵子树，也要区分它是左子树还是右子树。所以，二叉树和树是两种树结构。

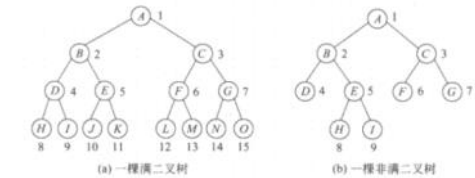
五种基本形态：

（1）斜树

所有结点只有左子树的二叉树称为左斜树；所有结点都只有右子树的二叉树称为右斜树。

（2）满二叉树

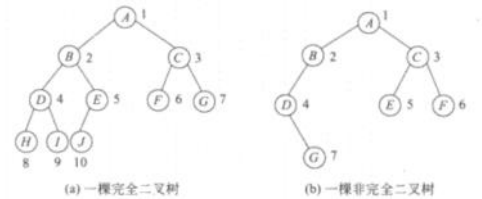
在一棵二叉树中，如果所有分支结点都存在左子树和右子树，并且所有叶子都在同一层上，这样的二叉树称为满二叉树。



满二叉树的特点：叶子只能出现在最下一层；只有度为0和度为2的结点。

（3）完全二叉树

对一棵具有n个结点的二叉树按层序编号，如果编号为i（1≤i≤n）的结点与同样深度的满二叉树中编号为i的结点在二叉树中的位置完全相同，则这棵二叉树称为完全二叉树。显然，一棵满二叉树必定是一棵完全二叉树。完全二叉树的特点是：叶子结点只能出现在最下两层，且最下层的叶子结点都集中在二叉树左侧连续的位置；如果有度为1的结点，只可能有一个，且该结点只有左孩子。



6.3.2二叉树的基本性质

（1）性质一：二叉树的第i层上最多有2^(i-1)个结点（i≥1）。

（2）性质二：在一棵深度为k的二叉树中，最多有2^k-1个结点，最少有k个结点。

（3）性质三：在一棵二叉树中，如果叶子结点的个数为n0，度为2的结点个数为n2，则n0=n2+1。

（4）具有n个结点的完全二叉树的深度为floor($\log_2 n$)+1。

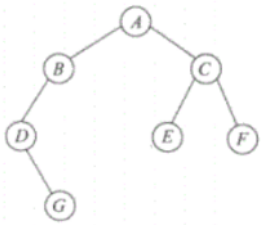
（5）对一棵具有n个结点的完全二叉树中的结点从1开始按层序编号，则对于任意的编号i（1≤i≤n）的结点，有：

- 如果i＞1，则结点i的双亲的编号为floor(i/2)；否则结点i是根结点，无双亲；
- 如果2i≤n，则结点i的左孩子的编号为2i；否则结点i无左孩子；
- 如果2i+1≤n，则结点i的右孩子编号为2i+1；否则结点i无右孩子。

6.3.3二叉树的抽象数据类型定义

6.3.4二叉树的遍历操作

遍历是二叉树中最基本的操作。二叉树的遍历是指从根结点出发，按照某种次序访问二叉树中的所有结点，使得每个结点被访问一次且仅被访问一次。由于二叉树中每个结点都可能有两棵子树，因而需要寻找一条合适的搜索路径。



（1）前序遍历

若二叉树为空，则空操作返回；否则：

- 访问根结点；
- 前序遍历根结点的左子树；
- 前序遍历根结点的右子树。

上图中，前序遍历的结果为：ABDGCEF。

（2）中序遍历

若二叉树为空，则空操作返回；否则：

- 中序遍历根结点的左子树；
- 访问根结点；
- 中序遍历根结点的右子树。

上图中，中序遍历的结果为：DGBAECF。

（3）后序遍历

若二叉树为空，则空操作返回；否则：

- 后序遍历根结点的左子树；
- 后序遍历根结点的右子树；
- 访问根结点。

上图中，后序遍历的结果为：GDBEFCA。

（4）层序遍历

ABCDEFG。

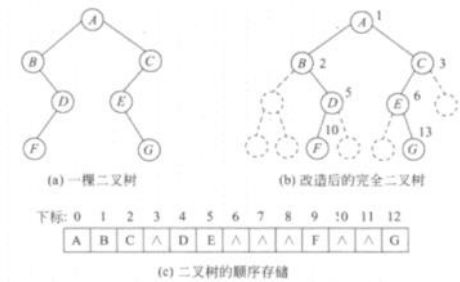
6.4二叉树的存储结构及实现

6.4.1顺序存储结构

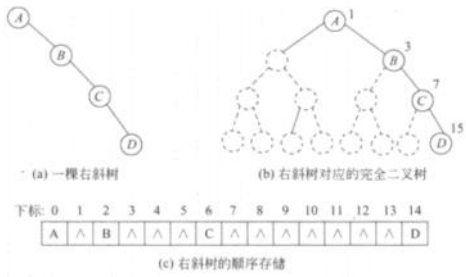
二叉树的顺序存储结构就是用一维数组存储二叉树中的结点，并且用结点的存储位置（下标）表示结点之间的逻辑关系——父子关系。由于二叉树本身不具有顺序关系，所以二叉树的顺序存储结构要解决的关键问题是如何利用数组下标来反映结点之间的父子关系。由二叉树的性质5可知，完全二叉树中结点的层序编号可以唯一地反映结点之间的逻辑关系，对于一般的二叉树，可以增添一些并不存在的空结点，使之成为一棵完全二叉树的形式，然后再用一维数组顺序存储。具体步骤如下：

（1）将二叉树按完全二叉树编号。根结点的编号为1，若某结点i有左孩子，则其左孩子的编号为2i；若某结点i有右孩子，则其右孩子的编号为2i+1；

（2）将二叉树中的结点以编号顺序存储到一维数组中。注意，C++的数组下标从0开始，因此，编号为i的结点存储到下标为i-1的位置。



显然，这种存储方法会造成存储空间的浪费，最坏的情况是右斜树，如下图所示，一棵深度为k的右斜树，只有k个结点，却需分配2^k-1个存储单元。事实上，二叉树的顺序存储结构一般仅适合于存储完全二叉树。

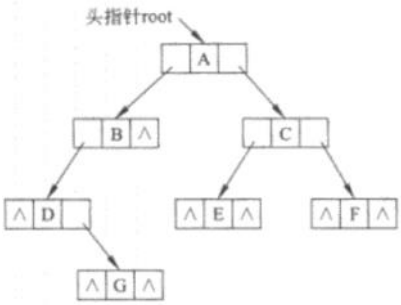


(c) 右斜树的顺序存储

6.4.2二叉链表

二叉树一般多采用二叉链表存储，其基本思想是：令二叉树的每个结点对应一个链表结点，链表结点除了存放与二叉树结点有关的数据信息外，还要设置指示左右孩子的指针，如下图所示。

其中，data为数据域，存放该结点的数据信息；lchild为左指针域，存放指向左孩子的指针，当左孩子不存在时为空指针；rchild为右指针域，存放指向右孩子的指针，当右孩子不存在时为空指针。二叉链表存储如下图所示。




```
template <class DataType>
struct BiNode
{
    DataType data;
    BiNode <DataType> * lchild, * rchild;
};
```

将二叉树的抽象数据类型定义在二叉链表存储结构下用C++语言的类实现。为了避免类的调用者访问BiTree类的私有变量root，在构造函数，析构函数以及遍历函数中调用了相应的私有函数。

```
template<class DataType>
class BiTree
{
public:
    BiTree(){root=Creat(root);} //构造函数，建立一棵二叉树
    ~BiTree(){Release(root);} //析构函数，释放各结点的存储空间
    void PreOrder(){PreOrder(root);} //前序遍历二叉树
    void InOrder(){InOrder(root);} //中序遍历二叉树
    void PostOrder(){PostOrder(root);} //后序遍历二叉树
    void LeverOrder(); //层序遍历二叉树
private:
    BiNode<DataType> * root; //指向根结点的头指针
    BiNode<DataType> * Creat(BiNode<DataType> *bt); //构造函数调用
    void Release(BiNode<DataType> * bt); //析构函数调用
    void PreOrder(BiNode<DataType> * bt); //前序遍历函数调用
    void InOrder(BiNode<DataType> * bt); //中序遍历函数调用
    void PostOrder(BiNode<DataType> * bt); //后续遍历函数调
};
```

（1）前序遍历

```
template<class DataType>
void BiTree<DataType>::PreOrder(BiNode<DataType> * bt)
{
    if(bt == NULL) return; //递归调用的结束条件
    else{
        cout << bt -> data; //访问根结点bt的数据域
        PreOrder(bt->lchild); //前序递归遍历bt的左子树
        PreOrder(bt->rchild); //前序递归遍历bt的右子树
    }
}
```

（2）中序遍历

```
template<class DataType>
void BiTree<DataType>::InOrder(BiNode<DataType> * bt)
{
    if(bt == NULL) return; //递归调用的结束条件
    else{
        InOrder(bt->lchild); //中序递归遍历bt的左子树
        cout << bt->data; //访问根结点bt的数据域
        InOrder(bt->rchild); //中序递归遍历bt的右子树
    }
}
```

（3）后序遍历

```
template<class DataType>
void BiTree<DataType>::InOrder(BiNode<DataType> * bt)
{
    if(bt == NULL) return; //递归调用的结束条件
    else{
        PostOrder(bt->lchild); //后序递归遍历bt的左子树
        PostOrder(bt->rchild); //后序递归遍历bt的右子树
        cout << bt->data;
    }
}
```

（4）层序遍历

算法伪代码：

```
1.队列Q初始化；
2.如果二叉树非空，将根指针入队；
```

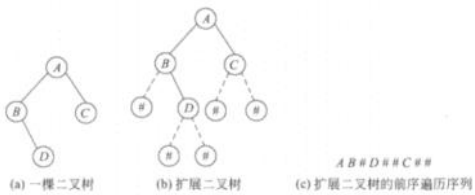
```
3.循环直到队列Q为空：
    3.1 q=队列Q的队头元素出队；
    3.2 访问结点q的数据域；
    3.3 若结点q存在左孩子，则将左孩子指针入队；
    3.4 若结点q存在右孩子，则将右孩子指针入队；
```

C++描述：

```
template<class DataType>
void BiTree<DataType>::LeverOrder()
{
    front = rear = -1 //采用顺序队列，并假定不会发生上溢
    if(root == NULL) return; //二叉树为空，算法结束
    Q[++rear] = root; //根指针入队
    while(front != rear) //当队列非空时
    {
        q = Q[++front]; //出队
        cout << q->data;
        if(q->lchild != NULL) Q[++rear] = q->lchild;
        if(q->rchild != NULL) Q[++rear] = q->rchild;
    }
}
```

（5）构造函数

构造函数的功能是建立一个二叉树。建立二叉树可以有多种方法，一种较为简单的方法是根据一个结点序列来建立二叉树。由于前序、中序和后序序列中任何一个都不能唯一确定一棵二叉树，因此不能直接使用。三种序列之一不能唯一确定二叉树的原因是：不能确定其左右子树的情况。针对以上问题，可以对二叉树做如下处理：将二叉树中每个结点的空指针引出一个虚结点，其值为一特定值，如“#”，以标识其为空。把这样处理后的二叉树称为原二叉树的扩展二叉树。如下图所示给出一棵二叉树的扩展二叉树，以及该扩展二叉树的前序遍历序列。



扩展二叉树的一个遍历学列就能唯一确定一棵二叉树。为简化问题，设二叉树中的结点均为一个字符。假设扩展二叉树的前序遍历序列由键盘输入，bt为指向根结点的指针，二叉链表的建立过程是：首先输入根结点，若输入的是“#”字符，则表明该二叉树为空树，即bt=NULL；否则输入的字符应该赋给bt->data，之后依次递归建立它的左子树和右子树。算法如下：

```
template<class DataType>
BiNode<DataType> *BiTree<DataType>::Creat(BiNode<DataType> * bt)
{
    cin >> ch; //输入结点的数据信息，假设为字符
    if(ch == '#') bt = NULL; //建立一棵空树
    else{
        bt = new BiNode;bt->data = ch; //生成一个结点，数据域为ch
        bt->lchild = Creat(bt->lchild); //递归建立左子树
        bt->rchild = Creat(bt->rchild); //递归建立右子树
    }
        return bt;
}
```

（6）析构函数

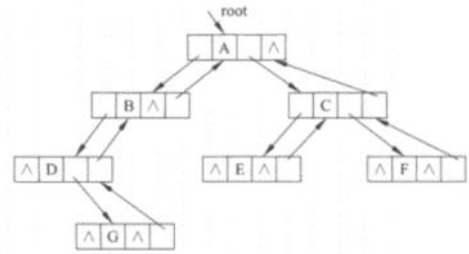
二叉链表属于动态存储分配，需要在析构函数中释放二叉链表中的所有结点。在释放某结点时，该结点的左右子树都已经释放，所以，应该采用后序遍历，当访问某结点时将该结点的存储空间释放掉。

```
template<class DataType>
void BiTree<DataType>::Release(BiNode<DataType> * bt)
{
    if(bt != NULL){
        Release(bt->lchild);
        Release(bt->rchild);
        delete bt;
    }
}
```

6.4.3三叉链表

在二叉链表中，从某结点出发可以直接访问它的孩子结点，但要找到它的双亲结点，则需要从根结点开始搜索，最坏情况下，需要遍历整个二叉链表。此时，应该采用三叉链表存储二叉树。

其中，data、lchild、rchild三个域的含义同二叉链表的结点结构；parent域为指向该结点的双亲结点的指针。



这种存储结构既便于查找孩子结点，又便于查找双亲结点。但是，相对于二叉链表而言，它增加了空间开销。

6.4.4线索链表

按照某种遍历次序对二叉树进行遍历，可以把二叉树中的所有结点排成一个线性序列。在具体应用中，有时需要访问二叉树中中的结点在某种遍历序列中的前驱和后继，此时，在存储结构中应保存结点在某种遍历序列的前驱和后继信息。考虑到一个具有n个结点的二叉链表，

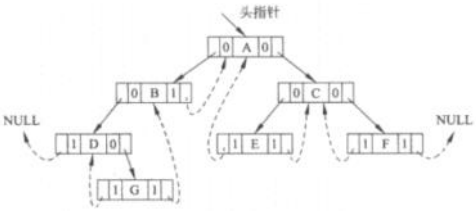
在2n个指针域中只有n-1个指针域用来存储孩子结点的地址，存在n+1个空指针域，可以利用这些空指针域存放指向该结点在某种遍历顺序中的前驱和后继结点的指针。这些指向前驱和后继结点的指针称为线索，加上线索的二叉树称为线索二叉树，相应地，加上线索的二叉链表称为线索链表。

在线索链表中，对任意结点，若左指针域为空，则用左指针域存放该结点的前驱线索；若右指针域为空，则用右指针域存放该结点的后继线索。为了区分某结点的指针域存放的是指向孩子的指针还是指向前驱或后继的线索，每个结点都增设两个标志位ltag和rtag。

ltag	lchild	data	rchild	rtag

$$ltag=\begin{cases}0 & \text{lchild 指向该结点的左孩子}\\ 1 & \text{lchild 指向该结点的前驱}\end{cases}\qquad rtag=\begin{cases}0 & \text{rchild 指向该结点的右孩子}\\ 1 & \text{rchild 指向该结点的后继}\end{cases}$$

下图为一棵普通二叉树以及该二叉树的中序线索链表的存储示意图。图中实线表示指向孩子结点的指针，虚线表示指向前驱和后继的线索。




```
template<class DataType>
class InThrBiTree
{
public:
    InThrBiTree(); //构造函数，建立中序线索链表
    ~InThrBiTree(); //析构函数，释放各结点的存储空间
    ThrNode * Next(ThrNode<DataType> * p); //查找结点p的后继
    void InOrder(); //中序遍历线索链表
private:
    ThrNode<DataType> * root; //指向线索链表的头指针
    ThrNode<DataType> * Creat(ThrNode<DataType> * bt); //构造函数调用
    void ThrBiTree(ThrNode<DataType> * bt, THrNode<DataType> * pre) //构造函数调用
}
```

（1）构造函数

构造函数的功能是建立一个中序线索链表，实质上就是将二叉链表中的空指针改为指向前驱或后继的线索，而前驱或后继的信息只有在遍历该二叉树时才能得到。因此，建立线索链表首先要建立二叉链表，然后在遍历的过程中修改空指针。建立二叉链表（带线索标志）的算法如下：

```
ThrNode<DataType> * InThrBiTree<DataType>::Creat(ThrNode<DataType> * bt)
{
    cin >> ch;
    if(ch == '#') bt = NULL; //建立一棵空树
    else{
        bt = new ThrNode;bt->data = ch; //生成一个结点
        bt->ltag = 0;bt->rtag = 0; //左右标志均置0
        bt->lchild = Creat(bt->lchild); //递归建立左子树
        bt->rchild = Creat(bt->rchild); //递归建立右子树
    }
    return bt;
}
```

在遍历过程中，访问当前结点bt的操作为：

- 检查结点bt的左右指针域，如果为空，则将相应标志置1；
- 由于结点bt的前驱结点刚刚被访问过，所以若左指针域为空，则可令其指向它的前驱；但由于bt的后继尚未访问到，所以它的右指针域不能建立线索，而要等到下次访问才能进行。为实现这一过程，设指针pre始终指向刚刚访问过的结点，即若指针bt指向当前结点，则pre指向它的前驱，显然pre的初值为NULL；
- 令pre=bt，即令pre指向刚刚访问过的结点bt。

中序线索化链表的算法伪代码为：

```
1.建立二叉链表，将每个结点的左右标志置0；
2.遍历二叉链表，建立线索；
    2.1 如果二叉链表bt为空，则空操作返回；
    2.2 对bt的左子树建立线索；
    2.3 对根结点bt建立线索；
        2.3.1 如果bt没有左孩子，则为bt加上前驱线索；
        2.3.2 如果bt没有右孩子，则为bt的右标志位置1；
        2.3.3 如果结点pre的右标志位为1，则为其加上h后继线索；
        2.3.4 令pre指向刚刚访问的结点；
    2.4 对bt的右子树建立线索；
```

C++描述：

```
template<class DataType>
void InThrBiTree<DataType>::ThrBiTree(ThrNode<DataType> *bt, ThrNode<DataType> *pre)
{
    if(bt == NULL) return;
    ThrBiTree(bt->lchild, pre);
    if(bt->lchild == NULL){ //对bt的左指针进行处理
        bt->ltag = 1;
        bt->lchild = pre; //设置pre的前驱线索
    }
    if(bt->rchild == NULL) bt->rtag = 1; //对bt的右指针进行处理
```

```
    if(pre->rtag == 1) pre->rchild = bt //设置pre的后继线索
    pre = bt;
    ThrBiTree(bt->rchild, pre);
}
```

下面给出中序线索链表的构造函数算法：

```
template<class DataType>
InThrBiTree<DataType>::InThrBiTree()
{
    root = Creat(root); //建立带线索标志的二叉链表
    pre = NULL; //当前访问结点的前驱结点pre初始化为NULL
    ThrBiTree(root, pre); //遍历二叉链表，建立线索
}
```

（2）查找后继结点

对于中序线索链表上的任一结点，其后继结点有以下两种情况：

- 如果该结点的右标志为1，表明该结点的右指针是线索，则其右指针所指向的结点便是它的后继结点；
- 如果该结点的右标志为0，表明该结点有右孩子，无法直接找到其后继结点。然而，根据中序遍历的操作定义，它的后继结点应该是遍历其右子树时第一个访问的结点，即右子树中的最左下结点。这只需沿着其右孩子的左指针向下查找，当某结点的左标志为1时，它就是所要找的后继结点。

```
template<class DataType>
ThrNode<DataType> * InThrBiTree<DataType>::Next(ThrNode<DataType> * p)
{
    if(p->rtag == 1) q = p->rchild;
    else{
        q = p->rchild;
        while(q-ltag ==0)
            q = q->lchild;
    }
    return q;
}
```

（3）遍历操作

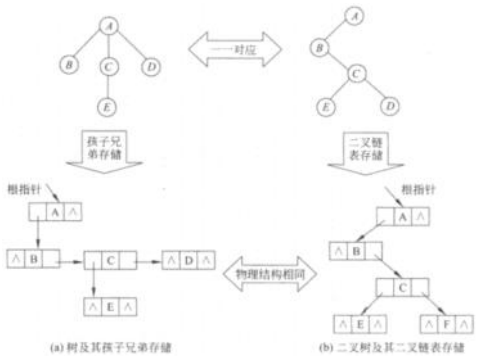
在中序线索链表上进行遍历，只需要找到中序遍历序列中的第一个结点，然后依次找每个结点的后继结点，直至某结点无后继为止。

```
template<class DataType>
void InThrBiTree<DataType>::InOrder()
{
    if(root == NULL) return;
    p = root;
    while(p->tag == 0)
        p = p->lchild;
    cout << p->data;
    while(p->rchild != NULL)
    {
        p = Next(p);
        cout << p->data;
    }
}
```

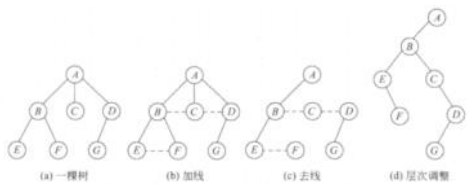
在中序线索链表上进行遍历，虽然时间复杂度亦为O(n)，但常数因子要比在二叉链表上进行的递归算法小，且不需要设工作栈。因此，若在某问题中所用的二叉树需要常遍历或查找结点在某种遍历序列中的前驱和后继，则应采用线索链表作为存储结构。

6.5二叉树遍历的非递归算法

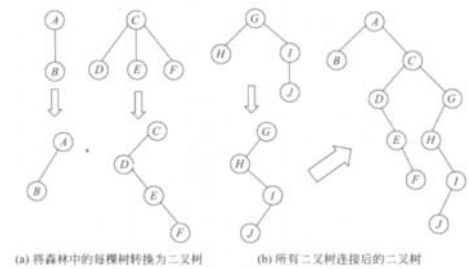
6.6树、森林与二叉树的转换



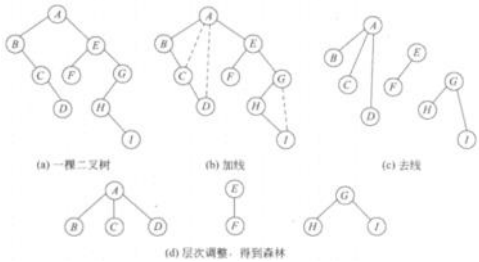
6.6.1树转换为二叉树



6.6.2森林转换为二叉树



6.6.3二叉树转换为树或森林



6.6.4森林的遍历

7.图

7.1图的逻辑结构

7.1.1图的定义和基本术语

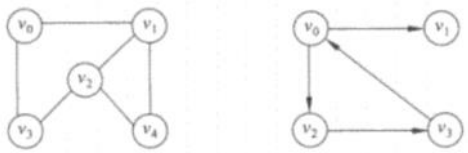
在图中常常将数据元素称为顶点。

7.1.1.1图的定义

图(graph)是由顶点的有穷非空集合和顶点之间边的集合组成，通常表示为：

$$G=(V,E)$$

其中，G 表示一个图，V 是图 G 中顶点的集合，E 是图 G 中顶点之间边的集合。若顶点 v_i 和 v_j 之间的边没有方向，则称这条边为无向边，用无序偶对(v_i,v_j)来表示；若从顶点 v_i 到 v_j 的边有方向，则称这条边为有向边(也称为弧)，用有序偶对$<v_i,v_j>$来表示，v_i 称为弧尾，v_j 称为弧头。如果图的任意两个顶点之间的边都是无向边，则称该图为**无向图**(undirected graph)，否则称该图为**有向图**(directed graph)。



7.1.1.2图的基本术语

（1）简单图

在图中，若不存在顶点到其自身的边，且同一条边不重复出现，则称这样的图为简单图。

（2）邻接、依附

在无向图中，对于任意两个顶点vi和vj，若存在边<vi，vj>，则称顶点vi和vj互为邻接点，同时称边(vi，vj)依附于顶点vi和vj；在有向图中，对于任意两个顶点vi和vj，若存在弧<vi，vj>，则称顶点vi邻接到vj，顶点vj邻接自vi，同时称弧<vi，vj>依附于顶点vi和vj。在不致混淆的情况下，通常称vj是vi的邻接点。

（3）无向完全图、有向完全图

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；在有向图中，如果任意两个顶点之间都存在方向互为相反的两条弧，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)条边。显然，在完全图中，边或弧的数目达到最多。

（4）稠密图、稀疏图

称边数最少的图为稀疏图，反之称为稠密图。

（5）顶点的度、入度、出度

在无向图中，顶点v的度是指依附于该顶点的边的个数，记作TD(v)。在具有n个顶点e条边的无向图中，有：

$$\sum_{i=0}^{n-1}\mathrm{TD}(v_i)=2e$$

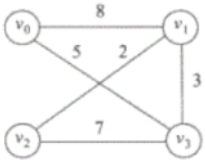
在有向图中，顶点v的入度是指以该顶点为弧头的弧的个数，称为ID(v)；顶点v的出度是指以该顶点为弧尾的弧的个数，记作OD(v)。在具有n个顶点e条边的有向图中，有：

$$\sum_{i=0}^{n-1}\mathrm{ID}(v_i)=\sum_{i=0}^{n-1}\mathrm{OD}(v_i)=e$$

（6）权、网

在图中，权通常是指对边赋予的有意义的数值量。在实际应用中，权可以有具体含义。比如，对于城市交通线路图，边上的权表示该条线路的长度或者等级；对于电子线路图，边上的权表示两个端点之间的电阻、电流或电压等。

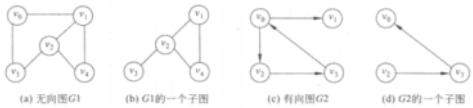
边上带权的图称为网或网图。



（7）路径、路径长度、回路

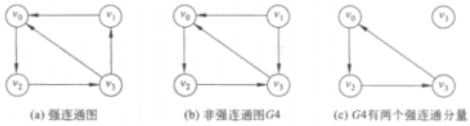
（8）简单路径、简单回路

（9）子图



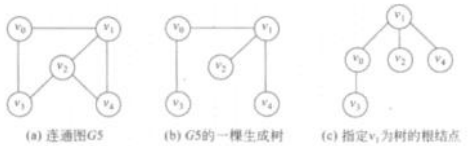
（10）连通图、连通分量

（11）强连通图、强连通分量



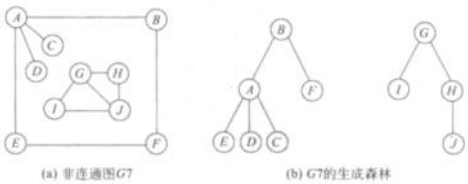
（12）生成树、生成森林

具有n个顶点的连通图G的生成树是包含G中全部顶点的一个极小连通子图。连通图的生成树是一个自由树，可以在生成树中任意指定一个顶点为树的根结点。生成树中添加任意一条属于原图中的边必定会产生回路，因为新添加的边使其所依附的两个顶点之间有了第二条路径；在生成树中减少任意一条边，则必然称为非联通。所以一棵具有n个顶点的生成树仅有n-1条边。



具有n个顶点的有向图G的生成树是包含G中全部顶点的一个子图，且子图中有一个入度为零的顶点，其他顶点的入度均为1。

在非连通图中，由每个连通分量都可以得到一棵生成树，这些联通分量的生成树构成了非联通图的生成森林。



7.1.2图的抽象数据类型定义

7.1.3图的遍历操作

7.1.3.1深度优先遍历

深度优先遍历类似于树的前序遍历。

从图中某顶点v出发进行深度优先遍历的基本思想是：

（1）访问顶点v；

（2）从v的未被访问的邻接点中选取一个顶点w，从w出发进行深度优先遍历；

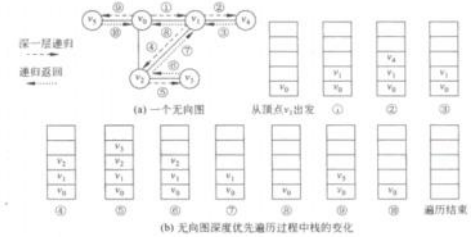
（3）重复上述两步，直至图中所有和v有路径相同的顶点都被访问到。

伪代码：

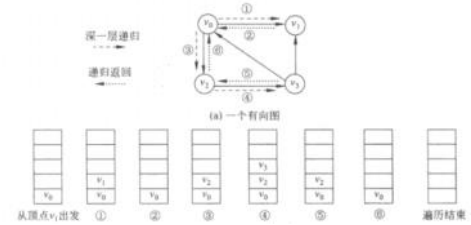
```
1.访问顶点v；visited[v] = 1;
2.w=顶点v的第一个邻接点；
3.while(w存在)
    3.1 if(w未被访问) 从顶点w出发递归执行该算法；
    3.2 w=顶点v的下一个邻接点；
```

显然，这是一个递归的过程。下图给出了(a)所示无向图的深度优先遍历路线，以及在递归执行过程中工作栈的变化情况。假设从顶点v0出发进行深度优先遍历，在访问了v0之后，将v0入栈，然后选择未被访问的邻接点v1。访问了v1之后，将v1入栈，然后选择未曾访问的邻接

点v4，访问了v4之后，将v4入栈，但是v4没有邻接点，将v4出栈；取栈顶元素v1的未曾访问的邻接点v2，则从v2出发进行遍历。以此类推，直到栈为空，得到深度优先遍历序列为v0 v1 v4 v2 v3 v5，在遍历过程中的出栈顺序为v4 v3 v2 v1 v5 v0。



有向图深度优先遍历的路线以及在递归执行过程中工作栈的变化如下图所示。



7.1.3.2广度优先遍历

广度优先遍历类似于树的层序遍历。

从图中某顶点v出发进行广度优先遍历的基本思想是：

（1）访问顶点v；

（2）依次访问v的各个未被访问的邻接点v1，v2，…，vk；

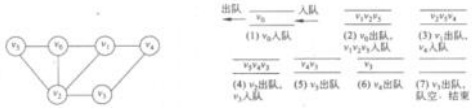
（3）分别从v1，v2，…，vk出发依次访问它们未被访问的邻接点，并使先被访问顶点的邻接点先于后被访问顶点的邻接点被访问。直至图中所有与顶点v有路径相通的顶点都被访问到。

广度优先遍历是以顶点v为起始点，由近及远，依次访问和v有路径相通且路径长度为1，2等的顶点。为了使先被访问顶点的邻接点先于后被访问顶点的邻接点被访问，需设置队列存储已被访问的顶点。

伪代码：

```
1.初始化队列Q;
2.访问顶点v;visited[v]=1;顶点v入队Q;
3.while(队列Q非空)
    3.1 v=队列Q的队头元素出队;
    3.2 w=顶点v的第一个邻接点;
    3.3 while(w存在)
        3.3.1 如果w未被访问,则访问顶点w;visited[w]=1;顶点w入队列Q;
        3.3.2 w=顶点v的下一个邻接点;
```

对下图所示无向图进行广度优先遍历，首先访问v0后将v0入队；将v0出队并依次访问v0的邻接点v1，v2和v5，并将v1，v2和v5入队；将v1出队并访问v1的未被访问的邻接点v4，并将v4入队；重复上述过程，得到顶点访问序列为v0 v1 v2 v5 v4 v3。



7.2图的存储结构及实现

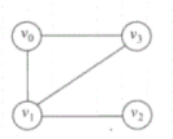
7.2.1邻接矩阵

图的邻接矩阵存储也称为数组表示法，其方法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。

无向图：

设图 $G=(V,E)$ 有 n 个顶点，则邻接矩阵是一个 $n\times n$ 的方阵，定义为：

$$\text{arc[i][j]}=\begin{cases}1 & \text{若}(v_i,v_j)\in E\text{或}<v_i,v_j>\in E\\ 0 & \text{否则}\end{cases}$$

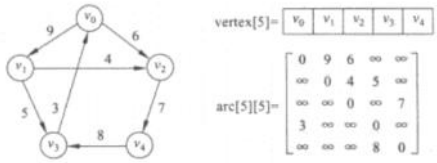


$$\text{vertex[4]}=\begin{array}{|c|c|c|c|}\hline v_0 & v_1 & v_2 & v_3\\ \hline\end{array}$$

$$\text{arc[4][4]}=\begin{bmatrix}0&1&0&1\\1&0&1&1\\0&1&0&0\\1&1&0&0\end{bmatrix}$$

有向图：

$$\text{arc[i][j]}=\begin{cases}w_{ij} & \text{若}(v_i,v_j)\in E\text{或}<v_i,v_j>\in E\\ 0 & \text{若}\ i=j\\ \infty & \text{否则}\end{cases}$$



显然，无向图的邻接矩阵一定是对称矩阵，而有向图的邻接矩阵则不一定对称。在图的邻接矩阵存储中容易解决下列问题：

- 对于无向图，顶点i的度等于邻接矩阵第i行（或第i列）非零元素的个数。对于有向图，顶点i的出度等于邻接矩阵的第i行非零元素个数；顶点i的入度等于邻接矩阵的第i列非零元素的个数。
- 要判断顶点i和j之间是否存在边，只需测试邻接矩阵中相应位置的元素arc[i][j]，若其值为1则有边；否则，顶点i和j之间不存在边。
- 找顶点i的所有邻接点，可依次判断顶点i与其他顶点之间是否有边（无向图）或顶点i到其他顶点是否有弧（有向图）。

抽象数据类型：

```
const int MaxSize = 10; //图中最多顶点个数
template<class DataType>
class MGraph
{
public:
    MGraph(DataType a[], int n, int e); //构造函数,建立具有n个顶点e条边的图
    ~MGraph(){} //析构函数为空
    void DFSTraverse(int v); //深度优先遍历图
    void BFSTraverse(int v); //广度优先遍历图
private:
    DataType vertex[MaxSize]; //存放图中顶点的数组
    int arc[MaxSize][MaxSize]; //存放图中边的数组
    int vertexNum, arcNum; //图的顶点数和边数
};
```

（1）构造函数

```
template<class DataType>
MGraph<DataType>::MGraph(DataType a[], int n, int e)
{
    vertexNum = n; arcNum = e;
    for(i = 0; i < vertexNum; i++)
        vertex[i] = a[i];
    for(i = 0; i < vertexNum; i++)
        for(j = 0; j < vertexNum; j++)
            arc[i][j] = 0;
    for(k = 0; k < arcNum; k++)
    {
        cin >> i >> j; //输入边依附的两个顶点的编号
        arc[i][j] = 1; arc[j][i] = 1;
    }
}
```

（2）深度优先遍历

```
template<class DataType>
void MGraph<DataType>::DFSTraverse(int v)
{
    cout << vertex[v]; visited[v] = 1;
    for(j = 0; j < vertexNum; j++)
        if(arc[v][j] == 1 && visited[j] == 0) DFSTraverse(j);
}
```

（3）广度优先遍历

```
template<class DataType>
void MGraph<DataType>::BFSTraverse(int v)
{
    front = rear = -1; //初始化队列，假设队列采用顺序存储并且不会发生溢出
    cout << vertex[v]; visited[v] = 1; Q[++rear] = v; //被访问顶点入队
    while(front != rear)
    {
        v = Q[++front]; //将队头元素出队并送到v中
        for(j = 0; j < vertexNum; j++)
            if(arc[v][j] == 1 && visited[j] == 0){
             cout << vertex[j]; visited[j] = 1; Q[++rear] = j;
            }
    }
}
```

在遍历时，对图中每个顶点至多调用一次遍历算法，因此一旦某个顶点被标志成已被访问，就不再从它出发进行遍历。因此，遍历图的过程实质上是对每个顶点查找其邻接点的过程。图采用邻接矩阵存储时，查找每个顶点的邻接点所需时间为O(n^2)。所以，深度优先和广度优先遍历图的时间复杂度均为O(n^2)，其中n为图中顶点个数。

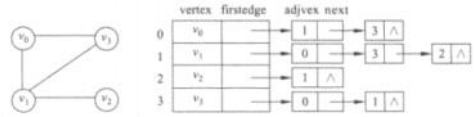
7.2.2邻接表

邻接表是一种顺序存储与链接存储相结合的存储方法，类似于树的孩子链表表示法，对于图的每个顶点vi，将所有邻接于vi的顶点链成一个单链表，称为顶点vi的边表（对于有向图则称为出边表）。为了方便对所有边表的头指针进行存取操作，可以采取顺序存储。存储边表头指针的数组和存储顶点信息的数组构成了邻接表的表头数组，称为顶点表。所以，在邻接表中存在两种结点结构：顶点表结点和边表结点，如下图所示。

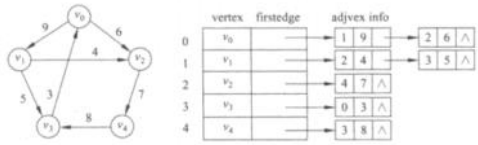
其中，vertex为数据域，存放顶点信息；firstedge为指针域，指向边表中第一个结点；adjvex为邻接点域，存放该顶点的邻接点在顶点表中的下标；next为指针域，指向边表中的下一个结点。

```
struct ArcNode //定义边表结点
{
    int adjvex; //邻接点域
    ArcNode * next;
};
template<class DataType>
struct VertexNode //定义顶点表结点
{
    DataType vertex;
    ArcNode * firstedge;
}
```

下图给出了无向图的邻接表存储示意图：



对于网图，其边表还需增设一个存储边上信息（如权值info）的域，下图给出了网图的邻接表存储示意图：



邻接表的抽象数据类型定义：

```
const int MaxSize = 10; //图的最大顶点数
template<class DataType>
class ALGraph
{
public:
    ALGraph(DataType a[], int n, int e); //构造函数，建立一个有n个顶点e条边的图
    ~ALGraph; //析构函数，释放邻接表中各边表结点的存储空间
    void DFSTraverse(int v); //深度优先遍历图
    void BFSTraverse(int v); //广度优先遍历图
private:
    VertexNode adjlist[MaxSize]; //存放顶点表的数组
    int vertexNum, arcNun; //图的定点数和边数
}
```

（1）构造函数

构造函数的功能是建立一个含有n个顶点e条边的图，假设建立有向图，算法伪代码为：

```
1.确定图的顶点个数和边的个数；
2.输入顶点信息存储在顶点表中，并初始化该顶点的边表；
3.依次输入边的信息并将边所对应的邻接点信息存储在边表中；
    3.1 输入边所依附的两个顶点的编号i和j；
    3.2 生成边表结点s，其邻接点的编号为j；
    3.3 将结点s插入到第i个边表的表头；
```

C++描述：

```
template<class DataType>
ALGraph<DataType>::ALGraph(DataType a[], int n, int e)
{
    vertexNum = n; arcNum = e;
    for(i = 0; i < vertexNum; i++) //输入顶点信息，初始化顶点表
    {
        adjlist[i].vertex = a[i];
        adjlist[i].firstedge = NULL;
    }
    for(k = 0; k < arcNum; k++) //依次输入每一条边
    {
        cin >> i >> j; //输入边所依附的两个顶点的编号
        s = new ArcNode; s->adjvex = j; //生成一个边表结点s
        s->next = adjlist[i].firstedge; //将结点s插入到第i个边表的表头
        adjlist[i].firstedge = s;
    }
}
```

（2）深度优先遍历

```
template<class DataType>
void ALGraph<DataType>::DFSTraverse(int v)
{
    cout << adjlist[v].vertex; visited[v] = 1;
    p = adjlist[v].firstedge; //工作指针p指向顶点v的边表
    while(p != NULL)
    {
        j = p->adjvex;
        if(visited[j] == 0) DFSTraverse(j);
        p = p->next;
    }
}
```

（3）广度优先遍历

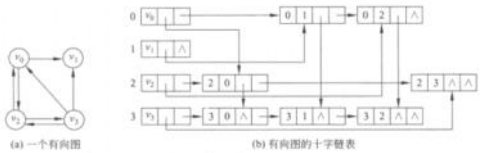
```
template<class DataType>
void ALGraph<DataType>::BFSTraverse(int v)
{
    front = rear = -1; //初始化队列，假设队列采用顺序存储且不会发生溢出
    cout << adjlist[v].vertex; visited[v] = 1; Q[++rear] = v; //被访问结点入队
    while(front != rear)
    {
        v = Q[++front];
        p = adjlist[v].firstedge' //工作指针p指向顶点v的边表
        while(p != NULL)
        {
            j = p->adjvex;
            if(visited[j] == 0){
                cout << adjlist[j].vertex; visited[j] = 1; Q[++rear] = j;
            }
            p = p->next;
        }
    }
}
```

如前所述，遍历图的过程实质上是对每个顶点查找其邻接点的过程，其耗费的时间取决于所采用的存储结构。觉以邻接表做存储结构时，算法需要访问所有n个顶点和e个边表结点，所以，深度优先和广度优先遍历图的时间复杂度均为O(n+e)。

7.2.3十字链表

十字链表是有向图的一种存储方法，它实际上是邻接表与逆邻接表的结合。在十字链表中，每条边对应的边结点分别组织到出边表和入边表中，其顶点表和边表的结点结构如下图：

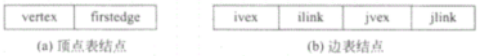
其中，vertex为数据域，存放顶点的数据信息；firstin为入边表头指针，指向以该顶点为终点的弧构成的链表中的第一个结点；firstout为出边表头指针，指向以该顶点为始点的弧构成的链表中的第一个结点；tailvex为弧的起点在顶点表中的下标；headvex为弧的终点在顶点表中的下标；headlink为入边表指针域，指向终点相同的下一条边；taillink为出边表指针域，指向起点相同的下一条边。



7.2.4邻接多重表

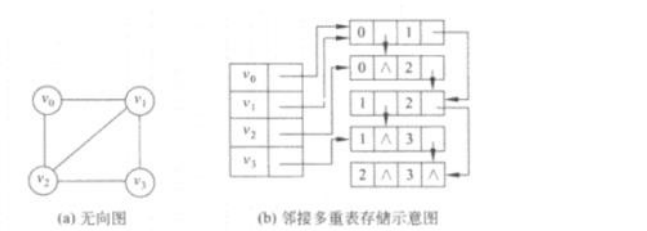
邻接多重表主要用于存储无向图。用邻接表存储无向图，每条边的两个顶点分别在以每条边的两个顶点分别在以该边所依附的两个顶点的边表中，这种重复存储给图的某些操作带来不便。例如，对已访问过的边做标记，或者要删除图中某一条边等，都需要找到表示同一条边的两个边表结点。因此，在进行这类操作的无向图中采用邻接多重表作存储结构更为适宜。

邻接多重表的存储结构和邻接表类似，也是由顶点表和边表组成，每条边用一个边表结点表示，其顶点表和边表的结点结构如下图所示。



(a) 顶点表结点　　(b) 边表结点

其中，vertex为数据域，存储有关顶点的数据信息；firstedge为边表头指针，指向依附于该顶点的第一条边的边表结点；ivex、jvex为与某条边依附的两个顶点在顶点表中的下标；ilink为指针域，指向依附于顶点ivex的下一条边；jlink为指针域，指向依附于顶点jvex的下一条边。



(a) 无向图　　(b) 邻接多重表存储示意图

7.2.5邻接矩阵和邻接表的比较

7.3最小生成树

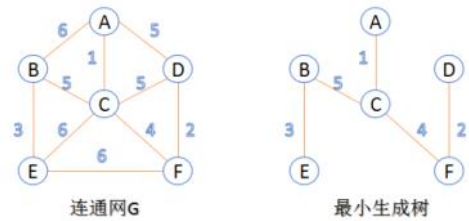
连通图：在无向图中，若任意两个顶点vi与vj都有路径相通，则称该无向图为连通图。

强连通图：在有向图中，若任意两个顶点vi和vj都有路径相通，则称该有向图为强连通图。

连通网：在连通图中，若图的边具有一定的意义，每一条边都对应着一个数，称为权；权代表着连接两个顶点的代价，则称这种连通图叫做连通网。

生成树：一个连通图的生成树是指一个连通子图，它含有图中全部n个顶点，但只有足以构成一棵树的n-1条边。一棵有n个顶点的生成树有且仅有n-1条边，如果生成树中再添加一条边，则必定成环。

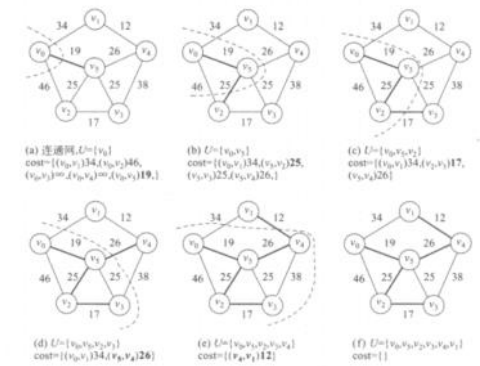
最小生成树：在连通网的所有生成树中，所有边的代价和最小的生成树，称为最小生成树。



连通网G　　最小生成树

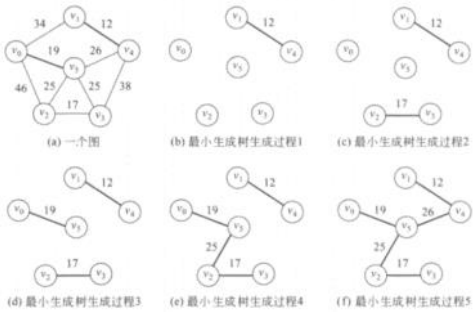
7.3.1 Prim算法

Prim算法的基本思想是：设G=(V,E)是一个无向连通网，令T=(U,TE)是G的最小生成树。T的初始状态为U={v0}(v0∈V)，TE={}，然后重复执行下述操作：在所有u∈V，v∈V-U的边中找一条代价最小的边(u,v)并入集合TE，同时将v并入U，直至U=V为止。此时TE中必有n-1条边，则T就是一棵最小生成树。



7.3.2 Kruskal算法

Kruskal算法的基本思想是：设无向连通网为G=(V,E)，令G的最小生成树为T=(U,TE)，其初始状态为U=V，TE={}，这样T中各顶点各自构成一个联通分量。然后按照边的权值由小到大的顺序，依次考察边集E中的各条边。若被考察的两个顶点属于T的两个不同的连通分量，则将此边加入到TE中，同时把两个连通分量连接为一个连通分量；若被考察边的两个顶点属于同一个连通分量，则舍去此边，以免造成回路，如此下去，当T中的连通分量个数为1时，此连通分量便是G的一棵最小生成树。



7.4最短路径

在非网图中，最短路径是指两顶点之间经历的边数最少的路径，路径上第一个顶点称为源点，最后一个顶点称为终点。在网图中，最短路径是指两顶点之间经历的边上权值之和最少的路径。

7.4.1 Dijkstra算法

Dijkstra算法用于求单源点最短路径问题，问题描述如下：给定带权有向图G=(V,E)和源点v∈V，求从v到G中其余各顶点的最短路径。

7.4.2 Floyd算法

Floyd算法用于求每一对顶点之间的最短路径问题，问题描述如下：给定带权有向图G=(V,E)，对任意顶点vi和vj(i≠j)，求顶点vi到顶点vj的最短路径。

7.5有向无环图及其应用

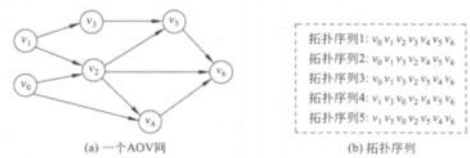
7.5.1 AOV网与拓扑排序

在一个表示工程的有向图中，用顶点表示活动，用弧表示活动之间的优先关系，称这样的有向图为顶点表示活动的网，简称AOV网。

AOV网中弧表示了活动之间存在的某种制约关系。在AOV网中不能出现回路，否则意味着某活动的开始要以自己的完成作为先决条件，显然，这是荒谬的。因此判断AOV网所代表的工程能否顺利进行，即判断它是否存在回路。而测试AOV网是否存在回路的方法，就是对AOV网进行拓扑排序。

设G=(V,E)是一个有向图，V中的顶点序列v0，v1，…，vn-1称为一个拓扑序列，当且仅当满足下列条件：若从顶点vi到vj有一条路径，则在顶点序列中顶点vi必在vj之前。对一个有向图构造拓扑序列的过程称为拓扑排序。

下图给出了一个AOV网的拓扑序列。显然，对于任何一项工程中各个活动的安排，必须按拓扑序列中的顺序进行才是可行的，并且一个AOV网的拓扑序列可能不唯一。



对AOV网进行拓扑排序的基本思想是：

（1）从AOV网中选择一个没有前驱的顶点并输出它；

（2）从AOV网中删去该顶点，并且删去所有以该顶点为尾的弧；

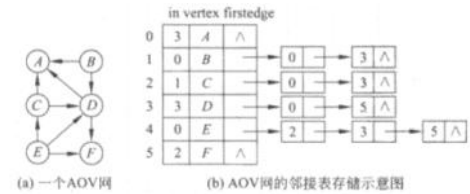
（3）重复上述两步，直到全部顶点都被输出，或AOV网中不存在没有前驱的顶点。

显然，拓扑排序的结果有两种：AOV网中全部顶点都被输出，这说明AOV网中不存在回路；AOV网中顶点未被全部输出，剩余的顶点均不存在没有前驱的顶点，这说明AOV网中存在回路。

拓扑排序算法的存储结构：

in	vertex	firstedge

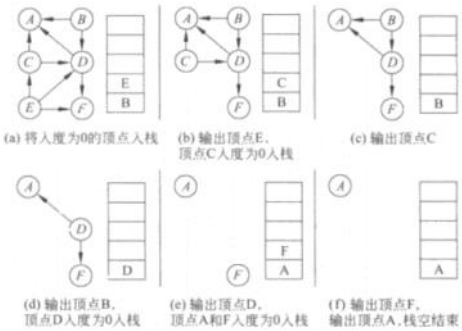
其中，vertex、firstedge的含义同邻接表；in是存储记录顶点入度的数据域。边表结点的结构同邻接表。



对没有前驱结点的查找，为了避免每次查找时都去遍历顶点表，可设置一个栈，凡是AOV网中入度为0的顶点都将其压栈，这里采用顺序栈。

伪代码如下：

```
1. 栈 S 初始化;累加器 count 初始化;
2. 扫描顶点表,将没有前驱(即入度为 0)的顶点压栈;
3. 当栈 S 非空时循环
   3.1  j=栈顶元素出栈;输出顶点 j;count++;
   3.2  对顶点 j 的每一个邻接点 k 执行下述操作:
        3.2.1  将顶点 k 的入度减 1;
        3.2.2  如果顶点 k 的入度为 0,则将顶点 k 入栈;
4. if (count<vertexNum) 输出有回路信息;
```



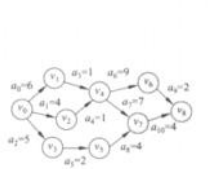
7.5.2 AOE网与关键路径

在一个表示工程的带权有向图中，用顶点表示事件，用有向边表示活动，边上的权值表示活动的持续时间，称这样的有向图为边表示活动的网，简称AOE网。AOE网中没有入边的顶点称为源点，没有出边的顶点称为终点。AOE网具有以下两个性质：

（1）只有在进入某顶点的各活动都已经结束，该顶点所代表的事件才能发生；

（2）只有在某顶点所代表的事件发生后，从该顶点出发的各活动才能开始。

下图给出了一个具有11个活动、9个事件的AOE网，顶点v0，v1，……，v8分别表示一个事件；弧<v0,v1>，<v0,v2>，…，<v7,v8>分别表示一个活动，用a0，a1，…，a10代表这些活动。其中，v0为源点，是整个工程的开始点，其入度为0；v8为终点，是整个工程的结束点，其出度为0。



事件	事件含义
v_0	源点，整个工程开始
v_1	活动a_0完成，活动a_1可以开始
v_2	活动a_1完成，活动a_4可以开始
v_3	活动a_2完成，活动a_5可以开始
v_4	活动a_3和a_4完成，活动a_6和a_7可以开始
v_5	活动a_5完成，活动a_8可以开始
v_6	活动a_6完成，活动a_9可以开始
v_7	活动a_7和a_8完成，活动a_{10}可以开始
v_8	活动a_9和a_{10}完成，整个工程结束

如果用AOE网来表示一项工程，那么，仅仅考虑各个活动之间的优先关系还不够，更多的是关心整个工程完成的最短时间是多少；哪些活动的延期将会影响整个工程的进度，而加速这些活动是否会提高整个工程的效率。

由于AOE网中的某些活动能够同时进行，故完成整个工程所必须花费的事件应改为始点到终点的最大路径长度（这里的路径长度是指该路径上的各个活动所持续的时间之和）。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。关键路径长度是整个工程所需的最短工期。也就是说，要缩短整个工期，必须加快关键活动的进度。

8.查找技术

8.1概念

8.1.1查找的基本概念

（1）关键码

（2）查找

（3）查找的结果

（4）静态查找、动态查找

不涉及插入和删除操作的查找称为静态查找，静态查找在查找不成功时，只返回一个不成功标志，查找的结果不改变查找集合；设计插入和删除操作的查找称为动态查找，动态查找在查找不成功时，需要将被查找的记录插入到查找集合中，查找的结果可能会改变查找集合。

（5）查找结构

线性表：适用于静态查找，主要采用顺序查找技术、折半查找技术。

树表：适用于动态查找，主要采用二叉排序树的查找技术。

散列表：静态查找和动态查找均适用，主要采用散列技术。

8.1.2查找算法的性能

查找算法的基本操作通常是将记录的关键码和给定的值进行比较，其运行时间主要消耗在关键码的比较上，所以，应以关键码的比较次数来度量查找算法的时间性能。比较次数又与哪些因素有关呢？显然，除了与算法本身及问题规模相关外，还与待查关键码在查找集合中的位置有关。同一查找集合，同一查找算法，待查关键码所处位置不同，比较次数往往不同。所以，查找算法的时间复杂度是问题规模n和待查关键码在查找集合中的位置k的函数，记为T(n,k)。

对于查找算法，以个别关键码的查找来衡量时间性能是不完全的。一般来讲，我们关心的是它的整体性能。将查找算法进行的关键码比较次数的数学期望值定义为平均查找长度。对于查找成功的情况，其计算公式为：

$$ASL = \sum_{i=1}^{n} p_i c_i$$

其中，n为问题规模，查找集合中的记录个数；pi为查找第i个记录的概率；ci为查找第i个记录所需的关键码的比较次数。显然，ci与算法密切相关，决定于算法；pi与算法无关，决定于具体应用。如果pi是已知的，则平均查找长度ASL只是问题规模n的函数。

对于查找不成功的情况，平均查找长度即为查找失败对应的关键码的比较次数。查找算法总的平均长度应为查找成功和查找失败两种情况下的查找长度的平均值。但在实际应用中，查找成功的可能性比查找不成功的可能性大得多，特别是在查找集合中j记录个数很多的情况下，查找不成功的概率可以忽略不计。

8.2线性表的查找技术

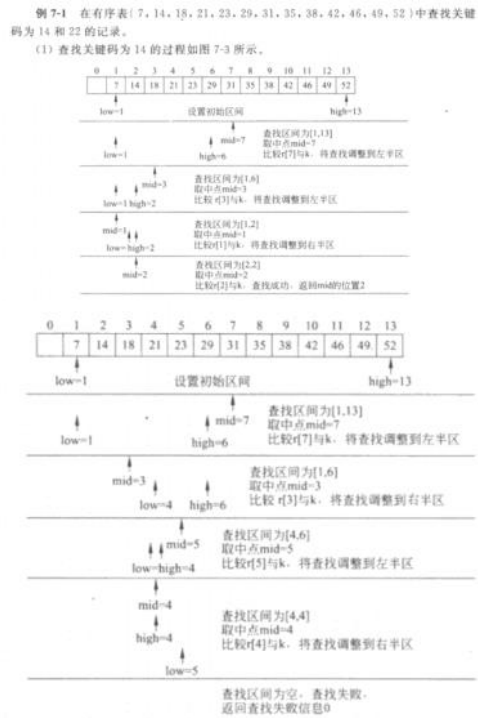
8.2.1顺序查找

8.2.1.1顺序表的顺序查找

8.2.1.2单链表的顺序查找

8.2.2折半查找

相较于顺序查找技术来说，折半查找技术的要求比较高，它要求线性表中的记录必须按关键码有序，并且必须采用顺序存储。折半查找一般只能应用于静态查找。



8.3树表的查找技术

8.3.1二叉排序树

二叉排序树又称二叉查找树，它或者是一棵空的二叉树，或者是具有下列性质的二叉树：

（1）若它的左子树不空，则左子树上所有结点的值均小于根结点的值；

（2）若它的右子树不空，则右子树上所有结点的值均大于根结点的值；

（3）它的左右子树也都是二叉排序树。

从上述定义可以看出，二叉排序树是记录之间满足一定次序关系的二叉树，中序遍历二叉排序树可以得到一个按关键码有序的序列，这也是二叉排序树的名字的由来。折半查找判定树就是一棵二叉排序树。

(a) 按63,55,90,58,70,42,10,45,83,67的顺序构造的二叉排序树　(b) 按55,42,10,70,63,58,83,67,90,45的顺序构造的二叉排序树

二叉排序树通常采用二叉链表进行存储，其结点结构为二叉链表的结点结构。二叉排序树的C++定义为：

```
class BiSortTree
{
public:
    BiSortTree(int a[], int n); //建立查找集合a[n]的二叉排序树
    ~BiSortTree(); //析构函数，同二叉链表的析构函数
    void InsertBST(BiNode<int> * root, BiNode<int> * s); //插入一个结点s
    void DeleteBST(BiNode<int> * p, BiNode<int> * f); //删除结点f的左孩子p
    BiNode<int> * SearchBST(BiNode<int> * root, int k); //查找值为k的结点
private:
    BiNode<int> * root; //二叉排序树的根指针
};
```

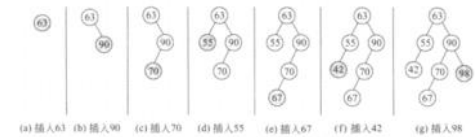
（1）二叉排序树的插入

(a) 二叉排序树　(b) 插入98后的二叉排序树

```
void BiSortTree::InsertBST(BiNode<int> * root, BiNode<int> * s)
{
    if(root == NULL) root = s;
    else if(s->data < root->data) InsertBST(root->lchild, s);
        else InsertBST(root->rchild, s);
}
```

（2）二叉排序树的构造

构造二叉排序树的过程是从空的二叉排序树开始的，依次插入一个个结点。


(a) 插入63　(b) 插入90　(c) 插入70　(d) 插入55　(e) 插入67　(f) 插入42　(g) 插入98

```
BiSortTree::BiSortTree(int r[], int n)
{
    for(i = 0; i < n; i++)
    {
        s = new BiNode; s->data = r[i];
        s->lchild = s->rchild = NULL;
        InsertBST(root, s);
    }
}
```

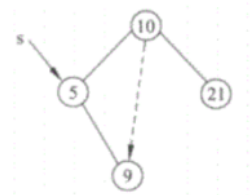
在找到插入位置后，向二叉排序树中插入结点的操作只是修改相应指针，而寻找插入位置的比较次数不超过树的深度，所以二叉排序树具有较高的插入效率。

插入结点的次序不同，所构造的二叉排序树的形状就不同。所以，对应于同一个查找集合，可以有不同的二叉排序树的形式，而不同的二叉排序树可能具有不同的深度，这直接影响着二叉排序树的插入效率。

（3）二叉排序树的删除

二叉排序树的删除操作比插入操作要复杂一些。首先，从二叉排序树中删除一个结点之后，要仍然保持二叉排序树的特性；其次，由于被插入的结点都是作为叶子结点连接到二叉排序树上，因而不会破坏结点之间的链接关系。而删除结点则不同，被删除的可能是叶子结点，也可能是分支结点，当删除分支结点时就破坏了二叉排序树中原有结点之间的链接关系，需要重新修改指针，使得删除结点后仍为一棵二叉排序树。

先讨论如何在二叉排序树中删除值最小的结点，这个方法在删除一般结点时被调用。为了删除二叉排序树中值最小的结点，首先应该找到这个结点。这只需沿左子树下移，直到最左下结点，这样就找到了值最小的结点，记作s。删除s只需要简单地把s的父结点中原来指向s的指针改为指向s的右孩子（s肯定没有左孩子，否则它就不是最小的结点）。这样修改指针，删除了s结点，且二叉排序树的特性保持不变。



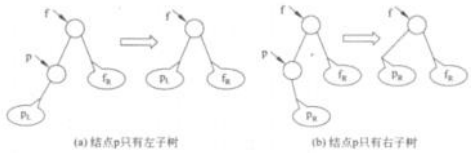
下面讨论二叉排序树的删除，不失一般性，设待删除结点为p，其双亲结点为f，且p是f的左孩子，则被删除结点有以下三种情况：

- p结点为叶子，p既没有左子树也没有右子树。

由于删除叶子结点不影响二叉排序树的特性，所以，只需将被删除结点的双亲结点的相应指针改为空指针，即f->lchild=NULL。

- p只有左子树pL或只有右子树pR。

此时，只需将pL或pR替换为f的左子树，即f->lchild=p->lchild或f->lchild=p->rchild，仍然保持二叉排序树的特性。

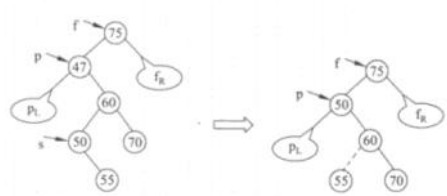


(a) 结点p只有左子树　　(b) 结点p只有右子树

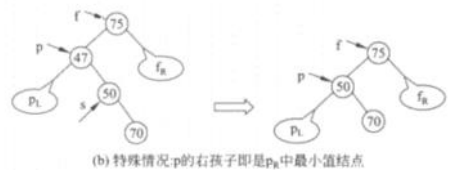
- p既有左子树pL又有右子树pR。

一种简单但代价很高的方法是让父结点的做指针指向p的任意一棵子树，然后将另一个子树中的结点重新插入，这将使得二叉排序树的结构发生变化并增加其高度。一个较好的方法是从某个子树中找出一个结点s，其值能代替p的值，这样就可以用结点s的值去替换结点p的值，再删除结点s。

由于必须再使二叉排序树的结构不发生较大变化的同时保持二叉排序树的特性，因而不是任意一个值都可以替换结点p的值。这个值应该是大于结点p的最小值（或小于结点p的最大值）。



(a) 一般情况



(b) 特殊情况:p的右孩子即是p_R中最小值结点

```
void BiSortTree::DeleteBST(BiNode<int> * p, BiNode<int> * f)
{
    if((p->lchild == NULL) && (p->rchild == NULL)){ //p为叶子
        f->lchild = NULL;
        delete p;
    }
    else if(p->rchild == NULL){ //p只有左子树
        f->lchild = p->lchild;
        delete p;
    }
    else if(p->lchild == NULL){ //p只有右子树
        f->lchild = p->rchild;
        delete p;
    }
    else{ //p的左右子树均不空
        par = p; s = p->rchild;
        while(s->lchild != NULL)
        {
            par = s;
            s = s->lchild;
        }
        p->data = s->data;
        if(par == p) par->rchild = s->rchild; //处理特殊情况
        else par->lchild = s->rchild;
        delete s;
    }
}
```

（4）二叉排序树的查找及性能分析

由于二叉排序树的定义，在二叉排序树中查找给定值k的过程是：

- 若root是空树，则查找失败；
- 若k等于root->data，则查找成功；
- 若k小于root->data，则在root的左子树上查找；
- 若k大于root->data，则在root的右子树上查找。

上述过程一直持续到k被找到或者待查找的子树为空。如果待查找的子树为空，则查找失败。当查找失败时，恰好找到了以k为键值的新结点在二叉排序树中的插入位置。二叉排序树的查找效率就在于只需要查找两个子树之一。

```
BiNode<int> * BiSortTree::SearchBST(BiNode<int> * root, int k)
{
    if(root == NULL) return NULL;
    else if (root->data == k) return root;
    else if (root->data < k) return SearchBST(root->lchild, k);
    else return SearchBST(root->rchild, k);
}
```

在二叉排序树上查找关键码等于给定值的结点的过程，恰好走了一条从根结点到该结点的路径，和给定值的比较次数等于给定值结点在二叉排序树中的层数，比较次数最少为1次（即整个二叉排序树的根结点就是待查结点），最多不超过树的深度。

考虑二叉排序树的查找性能与其他查找技术性能的比较。折半查找中关键码与给定值的比较次数也不超过折半查找判定树的深度。然而，

长度为n的判定树是唯一的，而含有n个结点的二叉排序树却不唯一，其形状取决于各个记录被插入二叉排序树的先后顺序。如果二叉排序树是平衡的（形态均匀），则有n个结点的二叉排序树的高度为floor(log2n)+1，其查找效率为O(log2n)，近似于折半查找。如果二叉排序树完全不平衡（最坏情况下为一棵斜树），则其深度可达到n，查找效率为O(n)，退化为顺序查找。一般地，二叉排序树的查找性能在O(log2n)和O(n)之间。因此，为了获得较好的查找性能，就要构造一棵平衡的二叉排序树。

8.3.2平衡二叉树

（1）平衡二叉树

平衡二叉树或者是一棵空的二叉排序树，或者是具有以下性质的二叉排序树：

- 根结点的左子树和右子树的深度最多相差1；
- 根结点的左子树和右子树也都是平衡二叉树。

（2）平衡因子

结点的平衡因子是该结点的左子树的深度与右子树的深度之差。

（3）最小不平衡子树

最小不平衡子树是指在平衡二叉树的构造过程中，以距离插入结点最近的、且平衡因子的绝对值大于1的结点为根的子树。

构造平衡二叉树的基本思想是：在构造二叉排序树的过程中，每当插入一个结点时，首先检查是否因插入而破坏了树的平衡性。若是，则找出最小不平衡子树。在保持二叉排序树特性的前提下，调整最小不平衡子树中各结点之间的链接关系，进行相应的旋转，使之成为新的平衡子树。

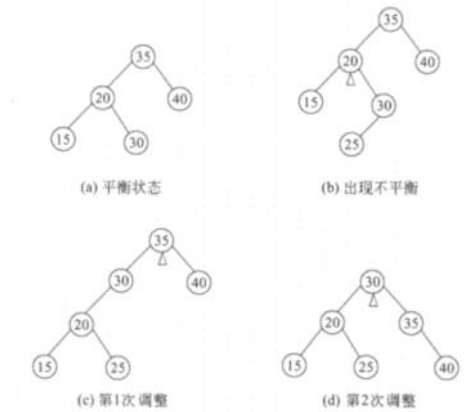
例如，假设关键码集合为{20，35，40，15，30，25}，在一棵空的二叉排序树上插入20和35，产生下图(a)所示的二叉排序树，此时显然平衡。当把40插入时，出现了不平衡现象，如下图(b)所示。这好比一条扁担出现了一头重一头轻的现象，如将支撑点（根结点）由20改为35，则扁担又恢复了平衡。仿此我们做一次逆时针旋转，旋转后称为新的平衡二叉树，如下图(c)所示。RR型调整。

再插入15和30，二叉排序树还是平衡的，如下图(a)所示。然而，当插入25时又失去了平衡，此时最小不平衡子树的根结点是35，如下图(b)所示。这时的平衡调整要复杂一些，需要支撑点调整两次，做两次旋转。

第一次旋转：最小不平衡子树的根结点不动，先调整根结点的左子树（即扁担中较重的一头）。根据扁担原理，将支撑点由20调为30，显然应进行逆时针旋转。在这次旋转中有一个冲突：25是30的左孩子，旋转后20也应成为30的左孩子，解决的办法是“旋转优先”，即30的左孩子应是旋转下来的20，而25比30小比20大，应成为20的右孩子，如下图(c)所示。

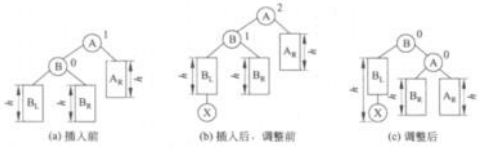
第二次旋转：调整最小不平衡子树，将支撑点由35调整为30，显然应进行顺时针旋转，如下图(d)所示。

一般情况下，设结点A为最小不平衡子树的根结点，对该子树进行平衡化调整归纳起来有以下4种情况：



（1）LL型

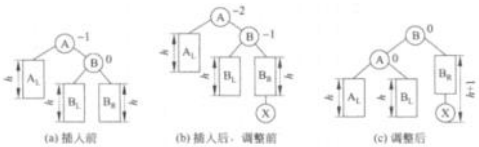
下图为插入前的平衡二叉树。其中，B为结点A的左子树的根结点，BL、BR分别为结点B的左右子树，AR为结点A的右子树，且BL、BR、AR三棵子树的深度均为h。将结点x插在结点B的左子树BL上，导致结点A的平衡因子由1变为2，以结点A为根的子树失去了平衡，如下图(b)所示。



新插入的结点是插在结点A的左孩子的左子树上，属于LL型。由上述扁担原理，将支撑点由A改为B，相应地，需要进行顺时针旋转。旋转后，结点A和BR发生冲突，按照“旋转优先原则”，将结点A成为结点B的右孩子，结点B的右子树BR成为结点A的左子树，如上图(c)所示。

（2）RR型

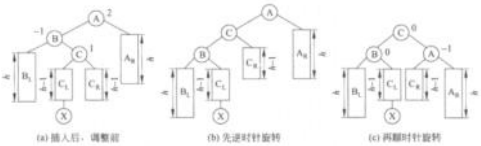
下图为插入前的平衡二叉树。其中，B为结点A的右孩子，BL、BR分别为结点B的左右子树，AL为结点A的左子树，且BL、BR、AL三棵树的深度均为h。将结点x插入在结点B的右子树BR上，导致结点A的平衡因子由-1变为-2，以结点A为根的子树失去了平衡，如下图(b)所示。



新插入的结点是插在结点A的右孩子的右子树上，属于RR型。将支撑点由A改为B，相应地，需要进行逆时针旋转。旋转后，结点A和结点B的左子树BL发生冲突，则结点A成为结点B的左孩子，结点B的左孩子BL成为结点A的右子树，如上图(c)所示。

（3）LR型

结点x插在根结点A的左孩子的右子树上，使结点A的平衡因子由1变为2，以结点A为根的子树失去了平衡，如下图(a)所示，这属于LR型，需要旋转两次。

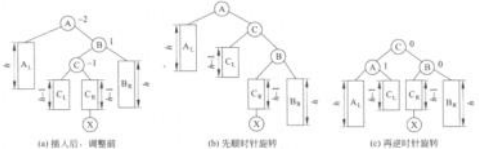


第一次旋转：根结点A不动，先调整结点A的左子树。将支撑点由结点B调整到结点C处，相应地，需要进行逆时针旋转。在旋转过程中，结点B和结点C的左子树发生冲突，按照旋转优先原则，结点B作为结点C的左孩子，结点C的左子树作为结点B的右孩子，其他结点之间的关系没有发生冲突，如上图(b)所示。

第二次旋转：调整整个最小不平衡子树。将支撑点由A调整到结点C，相应地，需要进行顺时针旋转。结点A作为结点C的右孩子，结点C的右子树作为结点A的左子树，如上图(c)所示。

（4）RL型

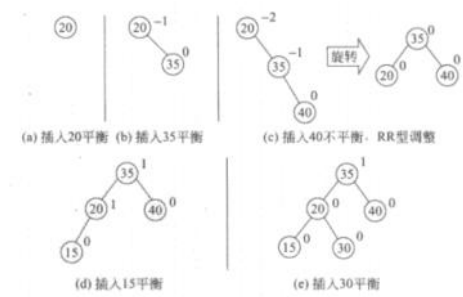
结点x插在根结点A的右孩子的左子树上，使结点A的平衡因子由-1变为2，以结点A为根的子树失去了平衡，如下图(a)所示，这属于RL型，也需要旋转两次。

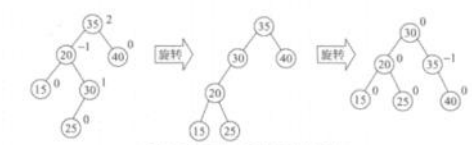


第一次旋转：根结点A不动，先调整根结点A的右子树。将支撑点由结点B调整到C处，相应地，需要进行顺时针旋转。在旋转过程中，结点B和结点C的右子树发生了冲突，按照旋转优先原则，结点B作为结点C的右孩子，结点C的右子树作为结点B的左子树，其他结点之间的关系没有发生冲突。如上图(b)所示。

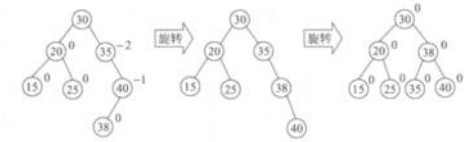
第二次旋转：调整整个最小不平衡子树。将支撑点由A调整到C，相应地，需进行逆时针旋转。结点A作为结点C的左孩子，结点C的左子树作为结点A的右子树，如上图(c)所示。

例子：





(f)插入25不平衡，LR型，需旋转两次



(g) 插入38不平衡，RL型，需旋转两次

8.4散列表的查找技术

8.4.1概述

8.4.2散列函数的设计

（1）直接定址法

直接定址法的散列函数是关键码的线性函数。

（2）保留余数法

（3）数字分析法

（4）平方取中法

（5）折叠法

8.4.3处理冲突的方法

（1）开放定址法

用开放定址法处理冲突得到的散列表叫做闭散列表。

所谓开放定址法，就是由关键码得到的散列地址一旦发生冲突，就去寻找下一个空的散列地址，只要散列表足够大，空的散列地址总能找到，并将记录存入。

找到一个空散列地址的方法很多，主要有三种：

- 线性探索法

当发生冲突时，线性探索法从冲突位置的下一个位置起，依次寻找空的散列地址，即对于键值key，设H(key)=d，闭散列表的长度为m，则发生冲突时，寻找下一个散列地址的公式为：

$$H_i=(H(\text{key})+d_i)\%m \quad (d_i=1,2,\cdots,m-1)。$$

例 7-8　关键码集合为 {47，7，29，11，16，92，22，8，3}，散列表表长为 11，散列函数为 $H(\text{key})=\text{key} \bmod 11$，用线性探测法处理冲突，得到的闭散列表如图 7-26 所示。

散列地址	0	1	2	3	4	5	6	7	8	9	10
关键码	11	22		47	92	16	3	7	29	8	
比较次数	1	2		1	1	1	4	1	2	2	

- 二次探测法

$$H_i=(H(\text{key})+d_i)\%m \quad (d_i=1^2,-1^2,2^2,-2^2,\cdots,q^2,-q^2 \text{ 且 } q\leqslant\sqrt{m})$$

散列地址	0	1	2	3	4	5	6	7	8	9	10
关键码	11	22	3	47	92	16		7	29	8	
比较次数	1	2	3	1	1	1		1	2	2	

- 随机探测法

当发生冲突时，随机探测法探测下一个散列地址的位移量是一个随机数列，即寻找下一个散列地址的公式为：

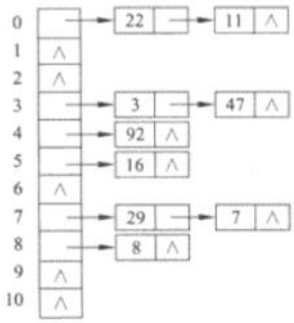
$$H_i=(H(\text{key})+d_i)\%m \quad (d_i \text{ 是一个随机数列}, i=1,2,\cdots,m-1)$$

（2）拉链法（链地址法）

用拉链法处理冲突构造的散列表叫做开散列表，如下图所示。

拉链法的基本思想是：将所有散列地址相同的记录，即所有关键码为同义词的记录存储在一个单链表中——称为同义词子表，在散列表中存储的是所有同义词子表的头指针。设n个记录存储在长度为m的开散列表中，则同义词子表的平均长度为n/m。

关键码集合{47，7，29，11，16，92，22，8，3}，散列函数为H(key)=key mod 11，则开散列表为：



开散列表查找算法C++描述如下：

```
Node<int> * HashSearch2(Node<int> * ht[], int m, int k)
{
    j = H(k); //计算散列地址
    p = ht[j]; //工作指针p初始化为指向第j个同义词子表
    while((p != NULL) && (p->data != k))
        p = p->next;
    if(p->data == k) return p; //查找成功
    else{
        q = new Node; q->data = k;
        q->next = ht[j]; ht[j] = q; //插在第j个同义词子表的表头
    }
}
```

8.4.4散列查找的性能分析

8.4.5开散列表和闭散列表的比较

9.排序技术

9.1概述

9.1.1排序的基本概念

（1）排序

（2）正序、逆序

（3）趟

（4）排序算法的稳定性

（5）排序的分类

排序的分类

根据在排序过程中待排序的所有记录是否全部被放置在内存中，可将排序方法分为内排序和外排序两大类。内排序是指在排序的整个过程中，待排序的所有记录全部被放置在内存中；外排序是指由于待排序的记录个数太多，不能同时放置在内存，而需要将一部分记录放置在内存，另一部分记录放置在外存，整个排序过程需要在内外存之间多次交换数据才能得到排序的结果。

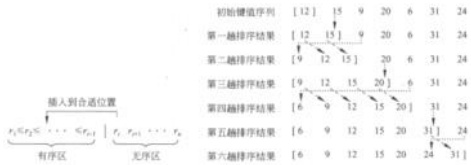
根据排序方法是否建立在关键码比较的基础，可以将排序方法分为基于比较的排序和不基于比较的排序。基于比较的排序方法主要通过关键码之间的比较和记录的移动这两种操作来实现，其时间下界为 $\Omega(n\log_2 n)$，大致可分为插入排序、交换排序、选择排序、归并排序等四类；不基于比较的排序方法是根据待排序数据的特点所采取的其他方法，通常没有大量的关键码之间的比较和记录的移动操作。

9.1.2排序算法的性能

9.2插入排序

插入排序是一类借助插入进行排序的方法，其主要思想是：每次将一个待排序的记录按其关键码的大小插入到一个已经排好序的有序序列中，直到全部记录排好序。

9.2.1直接插入排序




```
void InsertSort(int r[], int n) //0号单元用作暂存单元和监视哨
{
    for(i = 2; i <= n; i++)
    {
        r[0] = r[i]; //暂存待插关键码，设置哨兵
        for(j = i - 1; r[0] < r[j]; j--) //寻找插入位置
            r[j+1] = r[j]; //记录后移
        r[j+1] = r[0];
    }
}
```

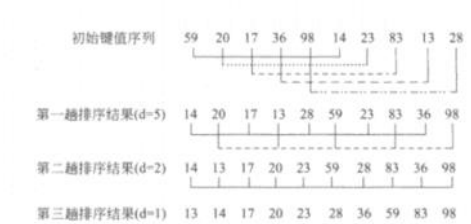
时间：最好：O(n)；最坏：O(n^2)；平均：O(n^2)。

空间：O(1)。

稳定。

9.2.2希尔排序

希尔排序的思想是：先将整个待排序记录序列分割成若干个子序列，在子序列内分别进行直接插入排序，待整个序列基本有序时，再对全体记录进行一次直接插入排序。



希尔排序算法的时间性能分析是一个复杂的问题，因为它是所取增量的函数。有人在大量实验的基础上指出，希尔排序的时间性能在$O(n^2)$和$O(n\log_2 n)$之间，当n在某个特定范围时，希尔排序的时间性能约为$O(n^{1.3})$。

希尔排序只需要一个记录的辅助空间，用于暂存当前待插入的记录。

希尔排序是一种不稳定的排序方法。

9.3交换排序

交换排序是一类借助交换进行排序的方法，其主要思想是：再待排序序列中选两个记录，将它们的关键码进行比较，如果反序则交换它们的位置。

9.3.1冒泡排序

初始键值序列	[50	13	55	97	27	38	49	65]
第一趟排序结果	[13	50	55	27	38	49	65]	**97**
第二趟排序结果	[13	50	27	38	49]	**55**	65	97
第三趟排序结果	[13	27	38	49]	**50**	55	65	97
第四趟排序结果	13	27	38	49	50	55	65	97

时间：最好：O(n)；最坏：O(n^2)；平均：O(n^2)。

空间：O(n)。

稳定。

9.3.2快速排序

快速排序又称为分区交换排序，其基本思想是：首先选一个轴值（povit，即比较的基准），将待排序记录划分成独立的两部分，左侧记录的关键码均小于或等于轴值，右侧记录的关键码均大于或等于轴值，然后分别对这两部分重复上述过程，直到整个序列有序。

时间：最好：O(nlog2n)；最坏：O(n^2)；平均：O(nlog2n)。

空间：O(nlog2n)。

不稳定。

9.4选择排序

选择排序是一类借助选择进行排序的方法，其主要思想是：每趟排序在房前待排序序列中选择关键码最小的记录，添加到有序序列中。选择排序的特点是记录移动的次数少。

9.4.1简单选择排序

时间：最好/最坏/平均：O(n^2)。

空间：O(1)。

不稳定。

9.4.2堆排序

堆排序的运行时间主要消耗在初始建堆和重建堆时进行的反复筛选上。初始建堆需要$O(n)$时间，第i次取堆顶记录重建堆需要用$O(\log_2 i)$时间，并且需要取$n-1$次堆顶记录，因此总的时间复杂度为$O(n\log_2 n)$，这是堆排序最好、最坏和平均的时间代价。堆排序对原始记录的排列状态并不敏感，相对于快速排序，这是堆排序最大的优点。

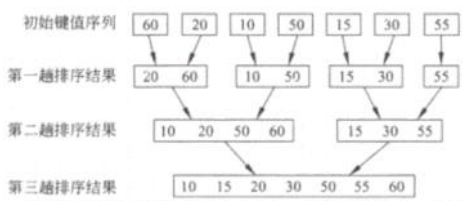
在堆排序算法中，只需要一个用来交换的暂存单元。

堆排序是一种不稳定的排序方法。

9.5归并排序

归并排序是一种借助归并进行排序的方法，归并的含义是将两个或两个以上的有序序列归并成一个有序序列的过程。归并排序的主要思想是：将若干有序序列逐步归并，最终归并为一个有序序列。

9.5.1二路归并排序的非递归实现

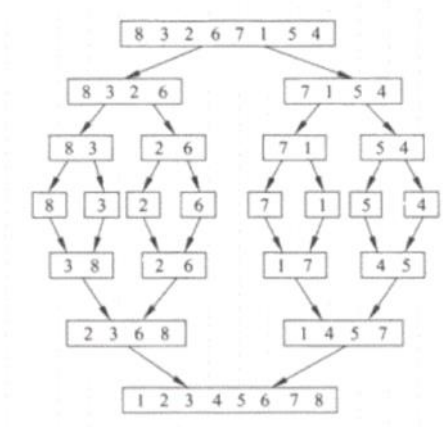


对二路归并排序算法的时间分析非常直观。一趟归并排序需要将待排序序列扫描一遍，其时间性能为$O(n)$。整个归并排序需要进行$\lceil\log_2 n\rceil$趟，因此，总的时间代价是$O(n\log_2 n)$，这是归并排序算法最好、最坏、平均的时间性能。

二路归并排序在归并过程中需要与待排序记录序列同样数量的存储空间，以便存放归并结果，因此其空间复杂度为$O(n)$。

二路归并排序是一种稳定的排序方法。

9.5.2二路归并排序的递归实现

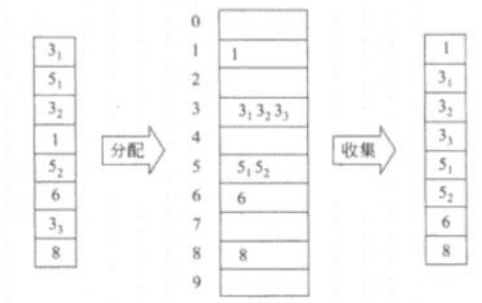


9.6分配排序

前面的排序方法都是建立在关键码比较的基础上，分配排序是基于分配和收集的排序方法。其基本思想是：先将待排序记录序列分配到不同的桶里，然后再把各桶中的记录依次收集到一起。

9.6.1桶排序

基本思想是：假设待排序记录的值都在0~m-1之间，设置m个桶，首先将值为i的记录分配到第i个桶中，然后再将各个桶中的记录依次收集起来。下图为一个桶排序的例子，下角标表示具有相同关键码的记录序号。

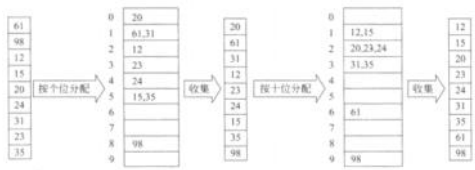


桶式排序第一个循环初始化静态链表，时间性能为 $O(n)$；第二个循环初始化静态链队列，时间性能为 $O(m)$；进行分配需要遍历静态链表，时间性能为 $O(n)$；进行收集需要遍历静态链表和静态链队列，时间性能为 $O(n+m)$。因此，桶式排序的时间复杂度为 $O(n+m)$。

桶式排序的空间复杂度是 $O(m)$，用来存储 m 个静态队列表示的桶。

由于桶采用队列作为存储结构，因此，桶式排序是稳定的。

9.6.2基数排序



假设待排序记录的关键码由 d 个子关键码复合而成，每个子关键码的取值范围为 m 个，则基数排序的时间复杂度为 $O(d(n+m))$，其中每一趟分配的时间复杂度是 $O(n)$，每一趟收集的时间复杂度为 $O(n+m)$，整个排序需要执行 d 趟。

基数排序共需要 m 个队列，因此空间复杂度为 $O(m)$。

由于桶采用队列作为存储结构，因此基数排序是稳定的。

10.索引技术

10.1索引的基本概念

（1）文件

（2）索引、索引项

（3）静态索引、动态索引

（4）线性索引、树形索引

（5）多级索引

10.2线性索引技术

10.2.1稠密索引

10.2.2分块索引

10.2.3多重表

10.2.4倒排表

10.3树形索引

树形索引将索引项组织为树结构，由于树的高度一般小于同规模的线性表的长度，所以，对树结构的查找一般也快于线性查找。
树形索引多用于动态索引，即树中结点可动态地增加或删除，因此，树形索引常采用链接存储结构实现。二叉排序树是一种最基本地树形索引，许多其他索引都是从它发展而来的。

10.3.1 2-3树

（1）定义

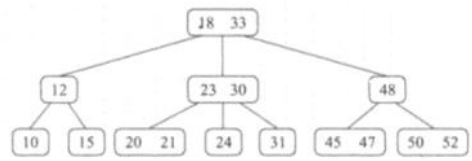
一棵2-3树是具有下列特性的树：

- 一个结点包含一个或者两个关键码；
- 每个内部结点有2个子女（包含一个关键码）或者3个子女（包含两个关键码），并因此得名2-3树；
- 所有叶子结点都在树的同一层。

当所有叶子结点在同一层时，我们就说这棵树是树高平衡的。2-3树最大的有点是，它能够以相对较低的代价保持树高的平衡。
除了形状上的特性外，2-3树还有一个类比于二叉排序树的特性：对于每一个结点，左子树中所有结点的值都小于第一个关键码的值，中间子树的值均大于第一个关键码的值。如果有右子树的话（相应地，结点存储两个关键码），那么中间子树中所有结点的值都小于第二个关键码的值，右子树的值大于第二个关键码的值。为了维护特性，当结点插入或者删除时要采取特别的处理。
从2-3树的定义可以推导出树的叶子结点的个数和树的深度之间的关系。一个高度为k的2-3树至少有2^(k-1)个叶子结点，此时每个分支结点都有2个子女，形成一棵满二叉树的形状；一个高度为k的2-3树至多有3^(k-1)个叶子结点，此时每个分支结点都有3个子女。

（2）查找

在2-3树种查找一个关键码的过程类似于在二叉排序树中的查找。查找从根结点开始，如果根结点不包含被查找的关键码k，那么查找就在可能包含关键码k的子树中继续进行。存储在根结点中的关键码确定哪一个子树是正确的子树。如下图所示的2-3中查找30，由于30在18和33之间，它只可能在中间的子树上，查找根结点中间的子树就会得到这个关键码。



（3）插入

（4）删除

10.3.2 B-树

B-树是一种平衡的多路查找树，主要面向动态查找，通常用在文件系统中。

（1）定义

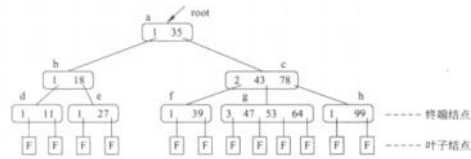
一棵m阶的B-树或者为空树，或者为满足下列特性的m叉树：

- 所有的叶子结点都出现在同一层，并且不带信息，叶子结点的双亲称为终端结点；
- 树中每个结点至多有m棵子树；
- 若根结点不是终端结点，则至少有两棵子树；
- 除根结点之外的所有非终端结点至少有cell(m/2)棵子树；
- 所有的非终端结点都包含以下数据：

$$(n, A_0, K_1, A_1, K_2, \cdots, K_n, A_n)$$

其中，$n(\lceil m/2 \rceil - 1 \leqslant n \leqslant m-1)$为关键码的个数，$K_i(1 \leqslant i \leqslant n)$为关键码，且$K_i < K_{i+1}(1 \leqslant i \leqslant n-1)$，$A_i(0 \leqslant i \leqslant n)$为指向子树根结点的指针，且指针$A_i$所指子树中所有结点的关键码均小于$K_{i+1}$，大于$K_i$。

一般情况下，B-树的叶子结点可以看做是外部结点（即查找失败）的结点，通常称为外结点。实际上这些结点不存在，指向这些结点的指针为空。所以，B-树的叶子结点可以不画出来。因为叶子都出现在同一层上，所以B-树也是树高平衡的。另外，每个结点中关键码的个数为子树的个数减1。如下图所示为一棵4阶B-树。



B-树是2-3树的推广，2-3树是一个3阶B-树。通常B-树中的一个结点的大小能够填满一个磁盘页，存储在B-树中的指针实际上是包含其孩子结点的块号，每个结点一般允许100个或者更多个孩子。

（2）查找

（3）插入

（4）删除

10.3.2 B+树

在基于磁盘的大型系统中，广泛适用的是B-树的一个变体，称为B+树。

一棵m阶B+树在结构上与m阶的B-树相同，但在关键码的内部安排上有所不同。具体如下：

- 具有m棵子树的结点含有m个关键码，即每一个关键码对应一棵子树；
- 关键码Ki是它所对应的子树的根结点中的最大（或最小）关键码；
- 所有的终端结点中包含了全部关键码信息，及指向关键码记录的指针；
- 各终端结点按关键码的大小次序链在一起，形成单链表，并设置头指针。

与B-树类似，在B+树中，结点内的关键码仍然有序排列，并且对同一结点内的 任意两个关键码Ki和Kj，若Ki<Kj，则Ki小于Kj对应的子树中的所有关键码。

与二叉排序树和2-3树最显著的区别是B+树旨在终端结点存储记录，内部结点存储关键码，但是这些关键码只是用于引导查找的。这意味着内部结点在结构上与终端结点有显著区别。内部结点存储关键码用于引导查找，把每个关键码与一个指向子女结点的指针相关联；终端结点存储实际记录，在B+树纯粹作为索引的情况下则存储关键码和指向实际记录的指针。一个B+树的终端结点一般链接起来，形成一个链表，这样，通过访问链表中的所有终端结点，就可以按照排序的顺序遍历全部记录。

下图为一棵3阶B+树，通常在B+树上有两个头指针，一个指向根结点，另一个指向关键码最小的终端结点。因此，可以对B+树进行两种查找操作：一种是从最小关键码起顺序查找，另一种是从根结点开始随机查找。