第 3 章　数据结构与算法

数据结构是指数据元素的集合及元素间的相互关系和构造方法，结构就是元素之间的关系。在数据结构中，元素之间的相互关系是数据的逻辑结构。按照逻辑关系的不同将数据结构分为线性结构和非线性结构，其中，线性结构包括线性表、栈、队列、串，非线性结构主要包括树和图。数据元素及元素之间关系的存储形式称为存储结构，可分为顺序存储和链接存储两种基本方式。

算法与数据结构密切相关，数据结构是算法设计的基础，合理的数据结构可使算法简单而高效。

3.1　线性结构

线性结构的特点是数据集合中的元素之间是一种线性关系，数据元素“一个接一个地排列”，也就是一个序列。

3.1.1　线性表

线性表是指一个序列，常采用两种存储方法：顺序存储和链式存储，主要的操作是插入、删除和查找。

1．线性表的定义

一个线性表是 n 个元素的有限序列（$n \geq 0$），通常表示为（$a_1, a_2, \cdots, a_n$），其特点是在非空的线性表中：

（1）存在唯一的一个称作“第一个”的元素。

（2）存在唯一的一个称作“最后一个”的元素。

（3）除第一个元素外，序列中的每个元素均只有一个直接前驱。

（4）除最后一个元素外，序列中的每个元素均只有一个直接后继。

2．线性表的存储结构

1）线性表的顺序存储

线性表的顺序存储是指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，如图 3-1 所示。在这种存储方式下，元素间的

逻辑关系无须占用额外的空间来存储。

一般地，以 LOC(a_1)表示线性表中第一个元素的存储位置，L 表示每个元素所占空间的大小，则顺序存储结构中，第 i 个元素 a_i 的存储位置为

$$\mathrm{LOC}(a_i) = \mathrm{LOC}(a_1)+(i-1)\times L$$

线性表采用顺序存储结构的优点是可以随机存取表中的元素，按序号查找元素的速度很快。缺点是插入和删除操作需要移动元素，插入元素前要移动元素以挪出空的存储单元，然后再插入元素；删除元素时同样需要移动元素，以填充被删除的元素空出来的存储位置。

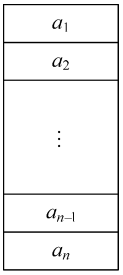


图 3-1 线性表的顺序存储

在表长为 n 的线性表中插入新元素时，共有 n+1 个可插入位置，在位置 1（元素 a_1 所在位置）插入元素时需要移动 n 个元素，在位置 n+1（元素 a_n 所在位置之后）插入元素时不需要移动元素，因此，等概率下插入一个元素时平均的移动元素次数 E_{insert} 为

$$E_{\mathrm{insert}} = \sum_{i=1}^{n+1} P_i \times (n-i+1) = \frac{1}{n+1}\sum_{i=1}^{n+1}(n-i+1) = \frac{n}{2}$$

其中，P_i 表示在表中位置 i 插入元素的概率。

在表长为 n 的线性表中删除元素时，共有 n 个可删除的元素，删除元素 a_1 时需要移动 n–1 个元素，删除元素 a_n 时不需要移动元素，因此，等概率下删除一个元素时平均的移动元素次数 E_{delete} 为

$$E_{\mathrm{delete}} = \sum_{i=1}^{n} q_i \times (n-i) = \frac{1}{n}\sum_{i=1}^{n}(n-i) = \frac{n-1}{2}$$

其中，q_i 表示删除元素 a_i 的概率。

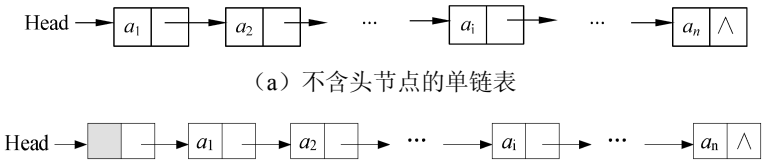
2）线性表的链式存储

线性表的链式存储是用节点来存储数据元素，元素的节点地址可以连续，也可以不连续，因此，存储数据元素的同时必须存储元素之间的逻辑关系。另外，节点空间只有在需要的时候才申请，无须事先分配。基本的节点结构如下所示：

数据域	指针域

节点中的数据域用于存储数据元素的值，指针域则存储当前元素的直接前驱或直接后继元素的位置信息，指针域中所存储的信息称为指针（或链）。

n 个节点通过指针连成一个链表，若节点中只有一个指针域，则称为线性链表（或单链表），如图 3-2 所示。



（a）不含头节点的单链表

（b）含头节点的单链表

图 3-2　线性表元素的单链表存储

在链式存储结构中，只需要一个指针（称为头指针，如图 3-2 中的 Head）指向第一个节点，就可以按照链接关系顺序地访问表中的任意一个元素。为了简化对链表状态的判定和处理，特别引入一个不存储数据元素的节点，称为头节点，将其作为链表的第一个节点并令头指针指向该节点。

在链式存储结构下进行插入和删除，其实质都是对相关指针的修改。

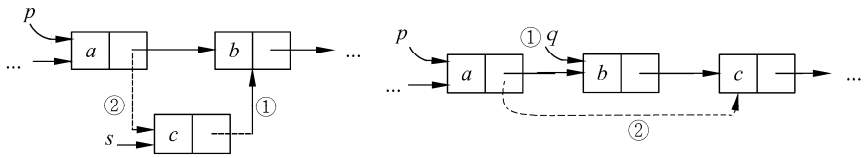
设单链表节点类型的定义为：

```
typedef struct node{
        int data;                   /*数据域*/
        struct node *next;          /*指针域*/
}NODE, *LinkList;
```

在单链表 p 所指节点（图 3-3 中元素 a 所在节点）后插入新元素节点（s 所指节点，图 3-3（a）中元素 c 所在节点）时，操作如下。

① s->next = p->next;　　/*s 所指节点的指针域改为指向 p 所指节点的后继节点*/

② p->next = s;　　/*p 所指节点的指针域改为指向 s 所指节点*/



（a）单链表中插入节点　　（b）单链表中删除节点

图 3-3　在单链表中插入和删除节点时的指针变化示意图

在单链表中删除 p 所指节点的后继节点时，操作如下。

① q = p->next;　　/*备份被删除节点的指针*/

② p->next = p->next->next;　　/*修改节点间的链接关系，从链表中摘除要删除的节点*/

③ free(q);　　/*释放被删除节点的空间*/

在图 3-3（b）中，若需删除元素 b，则令 p 节点的指针域指向其后继的后继节点（即图 3-3（b）中元素 c 所在节点），从而将元素 b 所在的节点从链表中摘除。

下面给出单链表上的插入和删除运算的实现过程。

【函数】单链表的插入运算。

```
int Insert_List (LinkList L, int k, int elem) /*L 为带头节点单链表的头指针*/
  /*将 elem 插入表 L 的第 k 个元素之前（即第 k–1 个元素之后），若成功则返回 0，否则返回–1*/
{     LinkList p,s;    /*p 的作用是指向第 k–1 个元素节点，s 则指向新申请的节点*/
      int i;

      i = 0; p = L;      /*初始时，令 p 指向头节点，i 为元素个数计数器*/
      /*顺着节点的链接关系依次向前查找，直到 p 指向第 k–1 个元素节点或到达表尾*/
      while (p && i < k-1) {
        p = p->next;   i++;
      }
      /*查找结束时 p 应指向第 k–1 个元素所在节点，若不存在第 k–1 个元素，则 p 为空指针*/
      if (!p)   return –1;                  /*表中不存在第 k–1 个元素，插入操作失败，返回*/
      /*若表中存在第 k–1 个元素，则生成新元素的节点并将其插入第 k–1 个元素之后*/
      s = (NODE *)malloc(sizeof(NODE));
      if (!s) return –1;                    /*生成新节点操作失败，无法完成插入操作，返回*/
      s->data = elem;                       /*元素存入新节点的数据域*/
      s->next = p->next;   p->next = s;   /*新节点插入第 k–1 个元素节点之后*/
      return 0;
} /* Insert_List */
```

【函数】单链表的删除运算。

```
int Delete_List (LinkList L, int k)         /*L 为带头节点单链表的头指针*/
/*删除表中的第 k 个元素节点，若成功返回 0；否则返回–1*/
{     LinkList p,q;                  /*p 的作用指向待删除节点的前驱节点，q 指向待删除的节点*/
      int i;
      /*删除第 k 个元素，需要将第 k–1 个元素节点中的指针域改为指向第 k+1 个元素的节点*/
      i = 0; p = L;                  /*初始时，令 p 指向头节点，i 为元素个数计数器*/
      /*顺着节点的链接关系依次向前查找，直到 p 指向第 k–1 个元素节点或到达表尾*/
      while (p && i < k-1) {
        p = p->next;   i++;
      }
      /*查找结束时 p 应指向第 k–1 个元素所在节点，若不存在第 k–1 个元素，则 p 为空指针*/
```

```
    if (!p)    return –1; /*表中不存在第 k–1 个元素（也不存在第 k 个元素），删除操作失败，返回*/
    if (!p->next)    return –1;              /*表中不存在第 k 个元素，删除操作失败，返回*/
    q = p->next;                             /*令 q 指向待删除的第 k 个元素节点*/
    p->next = q->next;     free(q);          /*删除节点并释放节点空间*/
    return 0;
} /* Delete_List */
```

线性表采用链表作为存储结构时，只能顺序地访问元素，而不能对元素进行随机存取。但其优点是插入和删除操作不需要移动元素。

根据节点中指针信息的实现方式，还有双向链表、循环链表和静态链表等链表结构。

- 双向链表：每个节点包含两个指针，分别指明当前元素的直接前驱和直接后继信息，可在两个方向上遍历链表中的元素。
- 循环链表：表尾节点的指针指向表中的第一个节点，可从表中任意节点开始遍历整个链表。
- 静态链表：借助数组来描述线性表的链式存储结构。

若双向链表中节点的 front 和 next 指针域分别指示当前节点的直接前驱和直接后继，则在双向链表中插入 s 所指节点时相关节点的指针域变化情况如图 3-4（a）所示，其操作过程如下。

① s -> front = p -> front;

② p -> front -> next = s;　　/*或者表示为 s -> front -> next = s;* /

③ p -> front = s;

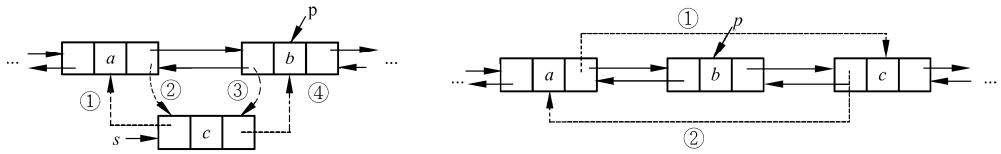
④ s -> next = p;

在双向链表中删除 p 所指节点时相关节点的指针域变化情况如图 3-4（b）所示，其操作过程如下。

① p -> front -> next = p -> next;

② p -> next -> front = p -> front;

③ free(p);



（a）双向链表中插入节点　　（b）双向链表中删除节点

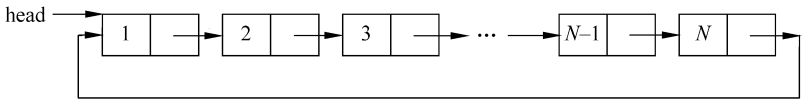
图 3-4　双向链表中插入和删除节点时的指针变化示意图

3．线性表的应用示例

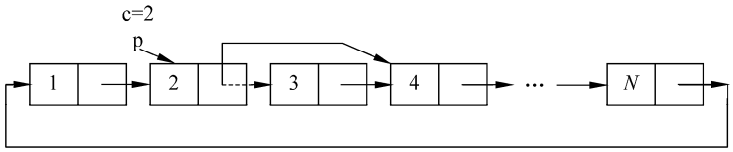
【例 3.1】 选首领。*N* 个游戏者围成一圈，从第一个人开始顺序报数 1，2，3。凡报到 3 者

退出圈子，最后留在圈中的人为首领。

N 个游戏者围成的圈可用一个包含 *N* 个节点的单循环链表来模拟，head 为头指针，如图 3-5（a）所示，其中节点的数据域存放游戏者的编号。以删除节点模拟人退出圈子的处理，整型变量 c（初值为 1）用于计数，指针变量 p 的初始值为 head。运行时，从 p 所指向的节点开始计数，p 沿链表中的指针每次向后指一个节点，c 值随 p 指针的移动相应地递增。当 c 计数到 2 时，就删除下一个节点（因其计数值将等于 3），如图 3-5（b）所示，然后将 c 置为 0，为下一次计数做准备。在表中剩下最后一个节点时应该结束游戏，因此设置一个计数器 k，其初值为参加游戏的人数。每当删除一个节点时，k 值就减 1，当 k 等于 1 时（即圈中只留下一个人），首领就选出来了。



（a）单循环链表模拟 *N* 个游戏者围成的圈



（b）c 计数到 2 时，p 所指节点的后继（下一个游戏者）出圈

图 3-5　选首领问题的存储结构示意图

【函数】选首领。

```
void play(LinkList head,int n) /*选首领*/
{   LinkList p,q;
    int c = 0, k;
    p = head; c = 1;   k = n;   /*当初始时 p 指向第 1 个游戏者，圈中有 n 个游戏者*/
    while (k > 1){
        if (c == 2) {   /*当 c 等于 2 时，p 指向的节点的后继即为将被删除的节点*/
            q = p->next; p->next = q->next;
            printf("%4d",q->data);   free(q);
            c = 0;   k--;
        }/*if*/
        else {c++; p = p->next;}
    }/*while*/
    printf("\n%4d was the winner.", p->data);                /*输出最后留在圈子内的游戏者编号*/
    free(p);
}/*play*/
```

3.1.2　栈和队列

栈和队列是常用的两种数据结构，它们的逻辑结构与线性表相同。其特点在于运算受到了限制：栈按“后进先出”的规则进行操作，队列按“先进先出”的规则进行操作，故称运算受限的线性表。

1．栈

1）栈的定义及基本运算

（1）栈的定义。

栈是只能通过访问它的一端来实现数据存储和检索的一种线性数据结构。换句话说，栈的修改是按先进后出的原则进行的。因此，栈又称为先进后出（FILO）或后进先出（LIFO）的线性表。在栈中进行插入和删除操作的一端称为栈顶（Top），相应地，另一端称为栈底（Bottom）。不含数据元素的栈称为空栈。

（2）栈的基本运算。

① 初始化栈 InitStack(S)：创建一个空栈 S。

② 判栈空 StackEmpty(S)：当栈 S 为空栈时返回“真”值，否则返回“假”值。

③ 入栈 Push(S,x)：将元素 x 加入栈顶，并更新栈顶指针。

④ 出栈 Pop(S)：将栈顶元素从栈中删除，并更新栈顶指针。若需要得到栈顶元素的值，可将 Pop(S)定义为一个函数，它返回栈顶元素的值。

⑤ 读栈顶元素 Top(S)：返回栈顶元素的值，但不修改栈顶指针。

2）栈的存储结构

（1）栈的顺序存储。栈的顺序存储是指用一组地址连续的存储单元依次存储自栈顶到栈底的数据元素，同时附设指针 top 指示栈顶元素的位置。采用顺序存储结构的栈也称为顺序栈。在顺序存储方式下，需要预先定义或申请栈的存储空间，也就是说栈空间的容量是有限的。因此在顺序栈中，当一个元素入栈时，需要判断是否栈满（栈空间中没有空闲单元），若栈满，则元素入栈会发生上溢现象。

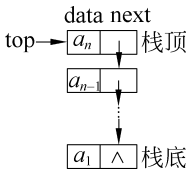


图 3-6　链栈示意图

（2）栈的链式存储。为了克服顺序存储的栈可能存在上溢的不足，可以用链表存储栈中的元素。用链表作为存储结构的栈也称为链栈。由于栈中元素的插入和删除仅在栈顶一端进行，因此不必设置头节点，链表的头指针就是栈顶指针。链栈的表示如图 3-6 所示。

3）栈的应用

栈的典型应用包括表达式求值、括号匹配等，在计算机语言的实现以及将递归过程转变为非递归过程的处理中，栈有重要的作用。

【例 3.2】 表达式求值。

计算机在处理算术表达式时，可将表达式先转换为后缀形式，然后利用栈进行计算。例如，表达式“46+5*(120–37)”的后缀表达式形式为“46 5 120 37 – * +”。

计算后缀表达式时，从左至右扫描后缀表达式：若遇到运算对象，则压入栈中；遇到运算符，则从栈中弹出相应运算对象进行计算，并将运算结果压入栈中。重复以上过程，直到后缀表达式结束。例如，后缀表达式“46 5 120 37 – * +”的计算过程为：

（1）依次将 46、5、120、37 压入栈中。

（2）遇到“–”，弹出 37、120，计算 120–37，得 83，将其压入栈中。

（3）遇到“*”，弹出 83、5，计算 5*83，得 415，将其压入栈中。

（4）遇到“+”，弹出 415、46，计算 46+415，得 461，将其压入栈中。

（5）表达式结束，则计算过程完成。

下面的函数 computing(char expr[],int *result)功能是基于栈计算后缀形式的表达式（以串形式存入字符数组 expr）的值，并通过参数 result 带回该值。函数的返回值为–1/0，分别表示表达式有/无错误。假设表达式中仅包含数字、空格和算术运算符号，其中所有项均以空格分隔，且运算符仅包含加（“+”）、减（“–”）、乘（“*”）、除（“\”）。

栈的基本操作的函数原型说明如下。

void InitStack(STACK *s)：初始化栈。
void Push(STACK *s, int e)：将一个整数压栈，栈中元素数目增 1。
void Pop(STACK *s)：栈顶元素出栈，栈中元素数目减 1。
int Top(STACK s)：返回非空栈的栈顶元素值，栈中元素数目不变。
int IsEmpty(STACK s)：若 s 是空栈，则返回 1；否则返回 0。

【函数】

```
int computing(char expr[], int *result)
{
    STACK s;     int tnum, a,b;     char *ptr;
    InitStack(&s);
    ptr = expr;                              /*字符指针指向后缀表达式串的第一个字符*/
    while (*ptr!='\0') {
        if (*ptr==' ') {                     /*当前字符是空格，则取下一字符*/
            ptr++;   continue;
        }
        else   if (isdigit(*ptr)) {          /*当前字符是数字，则将数字串转换为数值*/
            tnum = 0;
            while (*ptr>='0' && *ptr <=’9’) {
```

```
                tnum = tnum * 10 + *ptr –'0';
                ptr++;
            }
            Push(&s,tnum);
        }
        else                                        /*当前字符是运算符或其他符号*/
            if ( *ptr=='+' || *ptr=='-' || *ptr =='*' || *ptr =='/' ){ /*是运算符号，取运算数进行相应运算*/
              if (!IsEmpty(s)) {
                  a = Top(s); Pop(&s);              /*取运算符的第二个运算数*/
                  if (!IsEmpty(s)) {
                      b = Top(s); Pop(&s);          /*取运算符的第一个运算数*/
                  }
                  else     return –1;               /*缺第一运算数*/
              }
              else   return –1;                     /*缺第二运算数*/
              switch (*ptr) {
                  case '+': Push(&s,b+a);   break;
                  case '-': Push(&s,b–a);    break;
                  case '*': Push(&s,b*a);   break;
                  case '/': Push(&s,b/a);    break;
              }
          }
          else       return –1;                     /*是其他符号，无法运算*/
      ptr++;                                        /*取下一字符*/
} /* while */
if (IsEmpty(s))     return –1;
else {
          *result = Top(s);   Pop(&s);              /*取运算结果*/
           if (!IsEmpty(s))   return -1;
           return 0;
        }
}
```

2．队列

1）队列的定义及基本运算

（1）队列的定义。

队列是一种先进先出（FIFO）的线性表，它只允许在表的一端插入元素，而在表的另一端删除元素。在队列中，允许插入元素的一端称为队尾（Rear），允许删除元素的一端称为队

头（Front）。

（2）队列的基本运算。

① 初始化队列 InitQueue(Q)：创建一个空的队列 Q。

② 判队空 Empty(Q)：当队列为空时返回“真”值，否则返回“假”值。

③ 入队 EnQueue(Q,x)：将元素 x 加入到队列 Q 的队尾，并更新队尾指针。

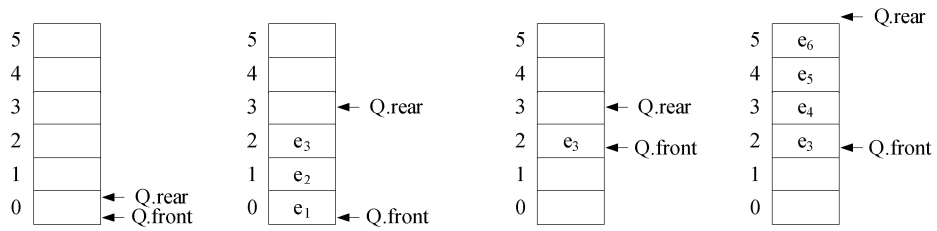
④ 出队 DeQueue(Q)：将队头元素从队列 Q 中删除，并更新队头指针。

⑤ 读队头元素 FrontQueue(Q)：返回队头元素的值，但不更新队头指针。

2）队列的存储结构

（1）队列的顺序存储。队列的顺序存储结构又称为顺序队列，它也是利用一组地址连续的存储单元存放队列中的元素。由于队中元素的插入和删除限定在表的两端进行，因此设置队头指针和队尾指针，分别指示出当前的队首元素和队尾元素。

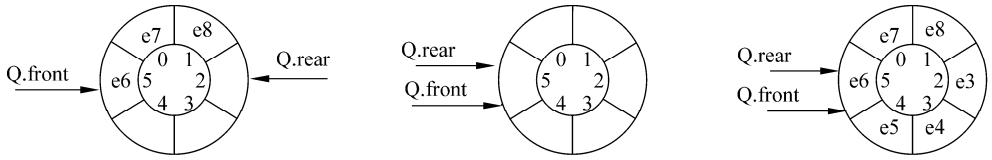
设顺序队列 Q 的容量为 6，其队头指针为 front，队尾指针为 rear，头、尾指针和队列中元素之间的关系如图 3-7 所示。



（a）空队列　（b）e_1、e_2、e_3相继入队　（c）e_1、e_2相继出队　（d）e_4、e_5、e_6相继入队

图 3-7　队列的头、尾指针与队列中元素之间的关系

在顺序队列中，为了简化运算，元素入队时，只修改队尾指针；元素出队时，只修改队头指针。由于顺序队列的存储空间是提前设定的，因此队尾指针会有一个上限值，当队尾指针达到其上限时，就不能只通过修改队尾指针来实现新元素的入队操作了。此时，可将顺序队列假想成一个环状结构，如图 3-8 所示，称之为循环队列。

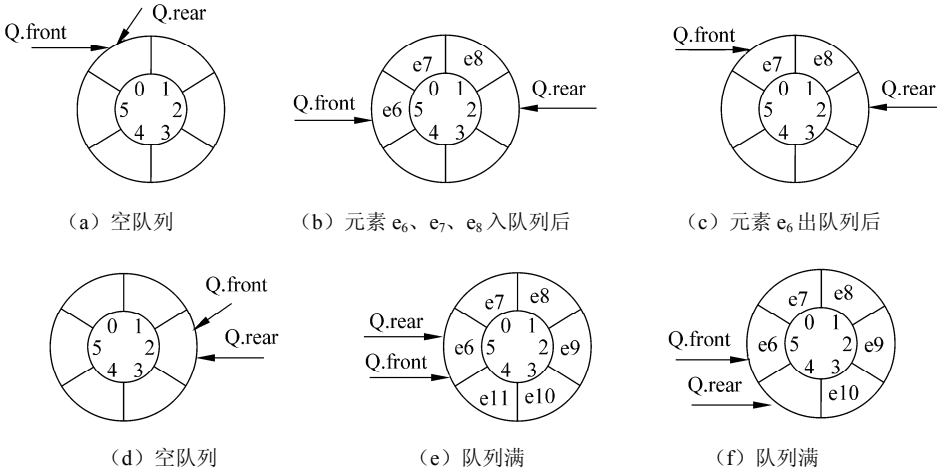


（a）一般情况　（b）空队列　（c）队列满

图 3-8　循环队列的头、尾指针示意图

设循环队列 Q 的容量为 MAXSIZE，初始时队列为空，且 Q.rear 和 Q.front 都等于 0，如图 3-9（a）所示。元素入队时修改队尾指针，即令 Q.rear = (Q.rear+1)% MAXSIZE，如图 3-9（b）所示。元素出队时修改队头指针，即令 Q.front = (Q.front+1)% MAXSIZE，如图 3-9（c）所示。

根据出队列操作的定义，当出队操作导致队列变为空时，有 Q.rear==Q.front，如图 3-9（d）所示；若队列满，则 Q.rear==Q.front，如图 3-9（e）所示。在队列空和队列满的情况下，循环队列的队头、队尾指针指向的位置是相同的，此时仅仅根据 Q.rear 和 Q.front 之间的关系无法断定队列的状态。为了区分队空和队满的情况，可采用两种处理方式：其一是设置一个标志位，以区别头、尾指针的值相同时队列是空还是满；其二是牺牲一个元素空间，约定以“队列的尾指针所指位置的下一个位置是头指针”表示队列满，如图 3-9（f）所示，而头、尾指针的值相同时表示队列为空。



（a）空队列　（b）元素 e_6、e_7、e_8 入队列后　（c）元素 e_6 出队列后

（d）空队列　（e）队列满　（f）队列满

图 3-9　循环队列的头、尾指针示意图

设队列中的元素类型为整型，则循环队列的类型定义为：

```
#define   MAXQSIZE   100
typedef   struct   {
     int   *base;          /*循环队列的存储空间，假设队列元素类型为整型*/
     int   front, rear;     /*队头、队尾指针*/
}SqQueue;
```

【函数】创建一个空的循环队列。

```
int InitQueue(SqQueue *Q)
```

```
/*创建容量为 MAXQSIZE 的空队列，若成功返回 1；否则返回 0*/
{    Q->base = (int *)malloc(MAXQSIZE*sizeof(int));
     if (!Q->base) return 0;
     Q->front = 0; Q->rear = 0; return 1;
}/*InitQueue*/
```

【函数】元素入循环队列。

```
int EnQueue(SqQueue *Q, int e) /*元素 e 入队，若成功返回 1；否则返回 0*/
{    if ( (Q->rear+1)% MAXQSIZE == Q->front) return 0;
     Q->base[Q->rear] = e;
     Q->rear = (Q->rear + 1)% MAXQSIZE;
     return 1;
}/*EnQueue*/
```

【函数】元素出循环队列。

```
int DeQueue(SqQueue *Q,int *e)
/*若队列不空，则删除队头元素，由参数 e 带回其值并返回 1；否则返回 0*/
{    if (Q->rear == Q->front) return 0;
     *e = Q->base[Q->front];
     Q->front = (Q->front + 1) % MAXQSIZE ;
     return 1;
}/*DeQueue*/
```

（2）队列的链式存储。队列的链式存储也称为链队列。为了便于操作，可给链队列添加一个头节点，并令头指针指向头节点，如图 3-10 所示。因此，队列为空的判定条件是头指针和尾指针的值相同，且均指向头节点。

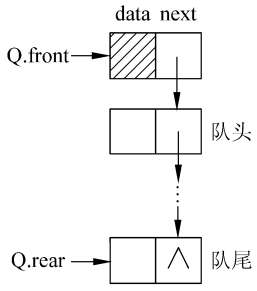


图 3-10　链队列示意图

3）队列的应用

队列常用于处理需要排队的场合，如操作系统中处理打印任务的打印队列、离散事件的计算机模拟等。

3.1.3　串

字符串是一串文字及符号的简称，是一种特殊的线性表。字符串的基本数据元素是字符，计算机中非数值问题处理的对象经常是字符串数据，如在汇编和高级语言的编译程序中，源程序和目标程序都是字符串数据；在事务处理程序中，姓名、地址等一般也是作为字符串处理的。另外，串还具有自身的特性，常常把一个串作为一个整体来处理。这里简单介绍串的定义、基

本运算和存储结构。

1．串的定义及基本运算

1）串的定义

串是仅由字符构成的有限序列，是取值范围受限的线性表。一般记为 S='$a_1a_2\cdots a_n$'，其中 S 是串名，单引号括起来的字符序列是串值。

- 串长：即串的长度，指字符串中的字符个数。
- 空串：长度为 0 的串，空串不包含任何字符。
- 空格串：由一个或多个空格组成的串。虽然空格是一个空白符，但它也是一个字符，计算串长度时要将其计算在内。
- 子串：由串中任意长度的连续字符构成的序列称为子串。含有子串的串称为主串。子串在主串中的位置指子串首次出现时，该子串的第一个字符在主串的位置。空串是任意串的子串。
- 串相等：指两个串长度相等且对应位置上的字符也相同。
- 串比较：两个串比较大小时以字符的 ASCII 码值作为依据。比较操作从两个串的第一个字符开始进行，字符的 ASCII 码值大者所在的串为大；若其中一个串先结束，则以串长较大者为大。

2）串的基本操作

- 赋值操作 StrAssign(s,t)：将串 t 的值赋给串 s。
- 连接操作 Concat(s,t)：将串 t 接续在串 s 的尾部，形成一个新串。
- 求串长 StrLength(s)：返回串 s 的长度。
- 串比较 StrCompare(s,t)：比较两个串的大小。返回值–1、0 和 1 分别表示 s<t、 s=t 和 s>t 三种情况。
- 求子串 SubString(s,start,len)：返回串 s 中从 start 开始的、长度为 len 的字符序列。

2．串的存储结构

字符串可以采用顺序存储和链式存储方式。

（1）顺序存储。该方式是用一组地址连续的存储单元来存储串值的字符序列。由于串中的元素为字符，所以可通过程序语言提供的字符数组定义串的存储空间（即存储空间的容量固定），也可以根据串长的需要动态申请字符串的空间（即存储空间的容量可扩充或缩减）。

（2）链式存储。字符串也可以采用链表作为存储结构，当用链表存储串中的字符时，每个节点中可以存储一个字符，也可以存储多个字符，需要考虑存储密度问题。节点大小为 4 的块链如图 3-11 所示。

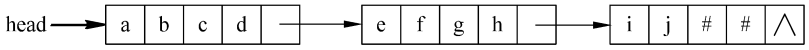


图 3-11　串值的链表存储方式

在链式存储结构中，节点大小的选择与顺序存储方法中数组空间大小的选择一样重要，它直接影响对串处理的效率。

通常情况下，字符串存储在一维字符数组中，每个字符串的末尾都有一个串结束符，在 C 语言中以特殊字符“\0”作为结束标记。

3．字符串运算

大多数的程序语言在其开发资源包中都提供了字符串的赋值（拷贝）、连接、比较、求串长、求子串等基本运算，利用它们就可以实现关于串的其他运算。下面简要介绍求串长和串比较运算的实现。

【函数】求串长，即计算给定串中除结束标志字符'\0'之外的字符数目。

```
int strlen(char *s)
{
   int n = 0;
   while (s[n]!='\0')
         n++;
   return n;
}/*strlen*/
```

【函数】串比较。

对于串 s1 和 s2，比较过程为：从两个串的第一个字符开始，若串 s1 和 s2 的对应字符相同，则继续比较下一对字符；若串 s1 的对应字符大于 s2 的相同位置字符，则串 s1 大于 s2，否则 s1 小于 s2。返回值 0 表示 s1 和 s2 的长度及对应字符完全相同，其他返回值则表示两个串中第一个不同字符的编码差值。

```
int strcmp(char *s1,char *s2)
{   int i = 0;
    while (s1[i]!='\0'|| s2[i]!='\0') {
          if (s1[i] == s2[i])     i++;
          else    return s1[i] - s2[i];
    }
    return 0;
}/*strcmp*/
```

4．串的模式匹配

子串（也称为模式串）在主串中的定位操作通常称为串的模式匹配，它是各种串处理系统中最重要的运算之一。

1）基本的模式匹配算法

该算法也称为布鲁特-福斯算法，其基本思想是从主串的第一个字符起与模式串的第一个字符比较，若相等，则继续逐个字符的后续比较，否则从主串的第二个字符起与模式串的第一个字符重新开始比较，直至模式串中每个字符依次和主串中的一个连续的字符序列相等时为止，此时称为匹配成功，否则称为匹配失败。

【函数】以字符数组存储字符串，实现朴素的模式匹配算法。

```
int Index(char S[], char T[], int pos)
/*查找并返回模式串 T 在主串 S 中从 pos 开始的位置(下标)，若 T 不是 S 的子串，则返回–1*/
{    i = pos; j = 0;   /*i，j 分别用于指示出主串字符和模式串字符的位置(下标)*/
     slen = strlen(S); tlen = strlen(T); /*计算主串和模式串的长度*/
     while (i < slen && j < tlen)
     {   if (S[i] == T[j]) {i++;j++;}
         else
            { i = i-j+1;  /*主串字符的位置指针回退*/
              j = 0;      /*模式串重新从起始字符开始*/
            }
      } /*while*/
      if (j >= tlen)
         return i – tlen;
      return –1;
} /*Index*/
```

假设主串的长度为 n，模式串的长度为 m，下面分析朴素模式匹配算法的时间复杂度，位置序号从 1 开始计算。设从主串的第 i 个位置开始与模式串匹配成功，而在前 i–1 趟匹配中，每趟不成功的匹配都是模式串的第一个字符与主串中相应的字符不相同，则在前 i–1 趟匹配中，字符的比较共进行了 i–1 次，因第 i 趟成功匹配的字符比较次数为 m，所以总的字符比较次数为$(i–1+m)$且 $1\leqslant i\leqslant n–m+1$。若在这 $n–m+1$ 个起始位置上匹配成功的概率相同，则在最好情况下，匹配成功时字符间的平均比较次数为

$$\sum_{i=1}^{n-m+1} p_i(i-1+m)=\frac{1}{n-m+1}\sum_{i=1}^{n-m+1}(i-1+m)=\frac{1}{2}(n+m)$$

因此，在最好情况下匹配算法的时间复杂度为 $O(n+m)$。而在最坏的情况下，每一趟不成功

的匹配都是模式串的最后一个字符与主串中相应的字符不相等，则主串中新一趟的起始位置为 $i-m+2$。若设第 i 趟匹配时成功，则前 $i-1$ 趟不成功的匹配中，每趟都比较了 m 次，总共比较了 $i\times m$ 次。因此，最坏情况下的平均比较次数为

$$\sum_{i=1}^{n-m+1} p_i(i\times m)=\frac{m}{n-m+1}\sum_{i=1}^{n-m+1} i=\frac{1}{2}m(n+m)$$

由于 $n>>m$，所以该算法在最坏情况下的时间复杂度为 $O(n\times m)$。

2）改进的模式匹配算法

改进的模式匹配算法又称为 KMP 算法（由 D.E.Knuth、V.R.Pratt 和 J.H.Morris 提出），其改进之处在于：每当匹配过程中出现相比较的字符不相等时，不需要回溯主串字符的位置指针，而是利用已经得到的“部分匹配”的结果，将模式串向后“滑动”尽可能远的距离，再继续进行比较。此算法可在 $O(n+m)$的时间内完成。

3.2 数组和矩阵

数组可看作是线性表的推广，其特点是多维数组的数据元素仍然是一个表。这里主要介绍多维数组的逻辑结构和存储结构、特殊矩阵和矩阵的压缩存储。

1．数组

1）数组的定义及基本运算

一维数组是长度固定的线性表，数组中的每个数据元素类型相同。n 维数组是定长线性表在维数上的扩张，即线性表中的元素又是一个线性表。

设有 n 维数组 $A[b_1,b_2,\cdots,b_n]$，其每一维的下界都为 1，b_i 是第 i 维的上界。从数据结构的逻辑关系角度来看，A 中的每个元素 $A[j_1,j_2,\cdots,j_n](1\leqslant j_i\leqslant b_i)$都被 n 个关系所约束。在每个关系中，除第一个和最后一个元素外，其余元素都只有一个直接后继和一个直接前驱。因此就单个关系而言，这 n 个关系仍是线性的。

以下面的二维数组 $A[m][n]$为例，可以把它看成是一个定长的线性表，它的每个元素也是一个定长线性表。

$$\boldsymbol{A}_{m*n}=\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n-1} & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n-1} & a_{2n} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn-1} & a_{mn} \end{bmatrix}$$

可将 $\boldsymbol{A}$ 看作一个行向量形式的线性表：

$$\boldsymbol{A}_{m*n}=\left[\left[a_{11}a_{12}\cdots a_{1n}\right]\left[a_{21}a_{22}\cdots a_{2n}\right]\cdots\left[a_{m1}a_{m2}\cdots a_{mn}\right]\right]$$

也可将 $\boldsymbol{A}$ 看作列向量形式的线性表：

$$\boldsymbol{A}_{m*n}=\left[\left[a_{11}a_{21}\cdots a_{m1}\right]\left[a_{12}a_{22}\cdots a_{m2}\right]\cdots\left[a_{1n}a_{2n}\cdots a_{mn}\right]\right]$$

数组结构的特点如下。

（1）数据元素数目固定。一旦定义了一个数组结构，就不再有元素的增减变化。

（2）数据元素具有相同的类型。

（3）数据元素的下标关系具有上下界的约束且下标有序。

在数组中通常做下面两种操作。

（1）取值操作。给定一组下标，读取对应的数据元素。

（2）赋值操作。给定一组下标，存储或修改与其相对应的数据元素。

几乎所有的程序设计语言都提供了数组类型。实际上，在语言中把数组看成是具有共同名字的同一类型多个变量的集合。需要注意的是，不能对数组进行整体的运算，只能对单个数组元素进行运算。

2）数组的顺序存储

由于数组一般不作插入和删除运算，也就是说，一旦定义了数组，则结构中的数据元素个数和元素之间的关系就不再发生变动，因此数组适合于采用顺序存储结构。

对于数组，一旦确定了它的维数和各维的长度，便可为它分配存储空间。反之，只要给出一组下标便可求得相应数组元素的存储位置，也就是说，在数据的顺序存储结构中，数据元素的位置是其下标的线性函数。

二维数组的存储结构可分为以行为主序（按行存储）和以列为主序（按列存储）两种方法，如图 3-12 所示。

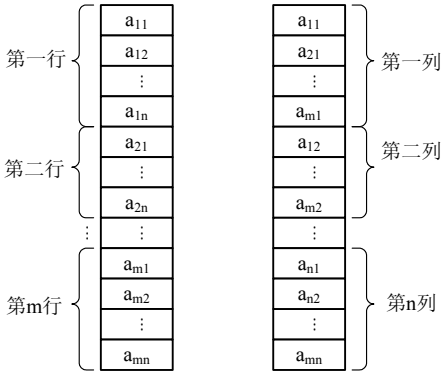


图 3-12　二维数组的两种存储方式

设每个数据元素占用 L 个单元，m、n 为数组的行数和列数，那么以行为主序优先存储的地址计算公式为

$$\mathrm{Loc}(a_{ij}) = \mathrm{Loc}(a_{11}) + ((i-1)\times n + (j-1))\times L$$

同理，以列为主序优先存储的地址计算公式为

$$\mathrm{Loc}(a_{ij}) = \mathrm{Loc}(a_{11}) + ((j-1)\times m + (i-1))\times L$$

2．矩阵

矩阵是很多科学与工程计算问题中研究的数学对象。在数据结构中主要讨论如何在尽可能节省存储空间的情况下，使矩阵的各种运算能高效地进行。

在一些矩阵中，存在很多值相同的元素或者是零元素。为了节省存储空间，可以对这类矩阵进行压缩存储。压缩存储的含义是为多个值相同的元素只分配一个存储单元，对零元素不分配存储单元。

1）特殊矩阵

常见的特殊矩阵有对称矩阵、三角矩阵和对角矩阵等。对于特殊矩阵，由于其非零元素的分布都有一定的规律，所以可将其压缩存储在一维数组中，并建立起每个非零元素在矩阵中的位置与其在一维数组中的位置之间的对应关系。

若矩阵 $\boldsymbol{A}_{n\times n}$ 中的元素有 $a_{ij} = a_{ji}$（$1 \leqslant i, j \leqslant n$）的特点，则称之为对称矩阵。

若为对称矩阵中的每一对元素分配一个存储单元，那么就可将 n^2 个元素压缩存储到能存放 $n(n+1)/2$ 个元素的存储空间中。不失一般性，以行为主序存储下三角（包括对角线）中的元素。假设以一维数组 $B[n(n+1)/2]$作为 n 阶对称矩阵 $\boldsymbol{A}$ 中元素的存储空间，则 $B[k]$（$0 \leqslant k < n(n+1)/2$）与矩阵元素 a_{ij} (a_{ji})之间存在着一一对应的关系。

$$k = \begin{cases} \dfrac{i(i-1)}{2} + j - 1 & \text{当 } i \geqslant j \\ \dfrac{j(j-1)}{2} + i - 1 & i < j \end{cases}$$

对角矩阵是指矩阵中的非零元素都集中在以主对角线为中心的带状区域中，即除了主对角线上和直接在对角线上、下方若干条对角线上的元素外，其余的矩阵元素都为零。一个 n 阶的三对角矩阵如图 3-13 所示。

若以行为主序将 n 阶三对角矩阵 $\boldsymbol{A}_{n\times n}$ 的非零元素存储在一维数组 $B[k](0 \leqslant k < 3\times n-2)$中，则元素位置之间的对应关系为

$$k = 3\times(i-1) - 1 + j - i + 1 = 2i + j - 3 \qquad (1 \leqslant i, j \leqslant n)$$

其他特殊矩阵可作类似的推导和计算，这里不再一一说明。

2）稀疏矩阵

在一个矩阵中，若非零元素的个数远远少于零元素的个数，且非零元素的分布没有规律，则称之为稀疏矩阵。

对于稀疏矩阵，存储非零元素时必须同时存储其位置（即行号和列号），所以三元组(i,j,a_{ij})可唯一确定矩阵中的一个元素。由此，一个稀疏矩阵可由表示非零元素的三元组及其行、列数唯一确定。

一个 6 行 7 列的稀疏矩阵如图 3-14 所示，其三元组表为((1,2,12),(1,3,9),(3,1,–3),(3,6,14),(4,3,24), (5,2,18),(6,1,15),(6,4,–7))。

$$\boldsymbol{A}_{n\times n}=\begin{bmatrix} a_{1,1} & a_{1,2} & & & & & \\ a_{2,1} & a_{2,2} & a_{2,3} & & & & 0 \\ & a_{3,2} & a_{3,3} & a_{3,4} & & & \\ & & \cdots & \cdots & \cdots & & \\ & & & a_{i,i-1} & a_{i,i} & a_{i,i+1} & \\ & 0 & & & \cdots & \cdots & \cdots \\ & & & & & a_{n,n-1} & a_{n,n} \end{bmatrix}$$

图 3-13　三对角矩阵示意图

$$\boldsymbol{M}_{6\times 7}=\begin{bmatrix} 0 & 12 & 9 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -3 & 0 & 0 & 0 & 0 & 14 & 0 \\ 0 & 0 & 24 & 0 & 0 & 0 & 0 \\ 0 & 18 & 0 & 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & -7 & 0 & 0 & 0 \end{bmatrix}$$

图 3-14　稀疏矩阵示意图

稀疏矩阵的三元组表构成一个线性表，其顺序存储结构称为三元组顺序表，其链式存储结构称为十字链表。

3.3　树和图

3.3.1　树

树结构是一种非常重要的非线性结构，该结构中一个数据元素可以有两个或两个以上的直接后继元素，可以用来描述客观世界中广泛存在的层次关系。

1．树的定义

树是 $n(n\geqslant 0)$个节点的有限集合。当 $n=0$ 时称为空树。在任一非空树（$n>0$）中，有且仅有一个称为根的节点；其余节点可分为 $m(m\geqslant 0)$ 个互不相交的有限集 $T_1, T_2, \cdots, T_m$，其中每个集合又都是一棵树，并且称为根节点的子树。

树的定义是递归的，它表明了树本身的固有特性，也就是一棵树由若干棵子树构成，而子树又由更小的子树构成。该定义只给出了树的组成特点，若从数据结构的逻辑关系角度来看，树中元素之间有明显的层次关系。对树中的某个节点，它最多只和上一层的一个节点（即其双亲节点）有直接关系，而与其下一层的多个节点（即其子树节点）有直接关系，如图 3-15 所示。通常，凡是分等级的分类方案都可以用具有严格层次关系的树结构来描述。

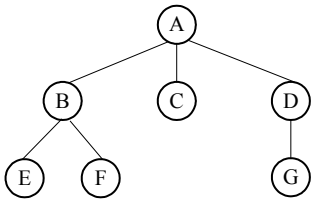


图 3-15　树结构示意图

- 双亲、孩子和兄弟：节点的子树的根称为该节点的孩子，相应地，该节点称为其子节点的双亲。具有相同双亲的节点互为兄弟。例如，图 3-15 中，节点 A 是树根，B、C、D 是 A 的孩子节点，B、C、D 互为兄弟；B 是 E、F 的双亲，E、F 互为兄弟。
- 节点的度：一个节点的子树的个数记为该节点的度。例如，图 3-15 中，A 的度为 3，B 的度为 2，C 的度为 0，D 的度为 1。
- 叶子节点：也称为终端节点，指度为零的节点。例如，图 3-15 中，E、F、C、G 都是叶子节点。
- 内部节点：度不为零的节点称为分支节点或非终端节点。除根节点之外，分支节点也称为内部节点。例如，图 3-15 中，B、D 都是内部节点。
- 节点的层次：根为第一层，根的孩子为第二层。以此类推，若某节点在第 i 层，则其孩子节点就在第 i+1 层。例如，图 3-15 中，A 在第 1 层，B、C、D 在第 2 层，E、F 和 G 在第 3 层。
- 树的高度：一棵树的最大层次数记为树的高度（或深度）。例如，图 3-15 所示树的高度为 3。
- 有序（无序）树：若将树中节点的各子树看成是从左到右具有次序的，即不能交换，则称该树为有序树，否则称为无序树。
- 森林：$m(m\geq 0)$棵互不相交的树的集合。

2．二叉树的定义

二叉树是 $n(n\geq 0)$个节点的有限集合，它或者是空树（n=0），或者是由一个根节点及两棵不相交的、分别称为左子树和右子树的二叉树所组成。

尽管树和二叉树的概念之间有许多联系，但它们有区别。树和二叉树之间最主要的区别是：二叉树中节点的子树要区分左子树和右子树，即使在节点只有一棵子树的情况下也要明确指出该子树是左子树还是右子树，树中则不区分，如图 3-16 所示。另外，二叉树中节点的最大度为 2，而树中不限制节点的度数。

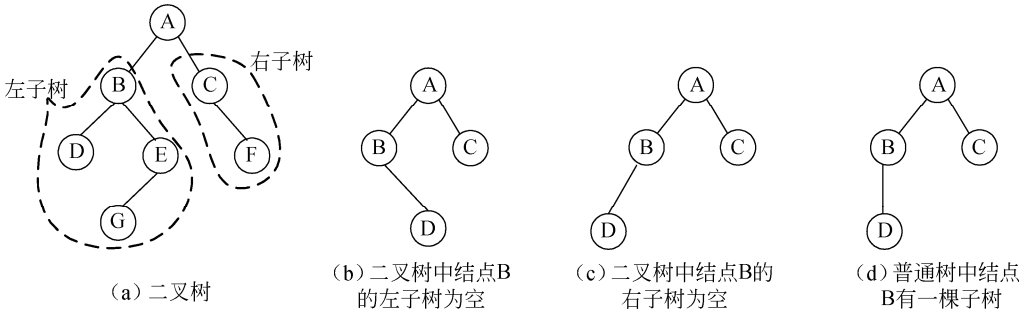


图 3-16　二叉树与普通树

3．二叉树的性质

性质 1：二叉树第 i 层（$i \geqslant 1$）上至多有 2^{i-1} 个节点。

可用归纳法证明性质 1。

性质 2：深度为 k 的二叉树至多有 2^k-1 个节点（$k \geqslant 1$）。

由性质 1，每一层的节点数都取最大值即得 $\sum_{i=1}^{k} 2^{i-1} = 2^k - 1$，因此性质 2 得证。

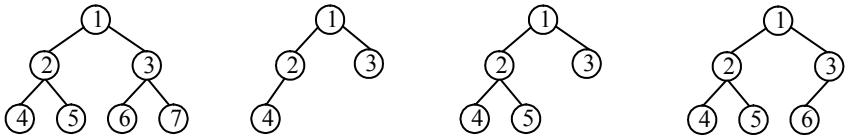
性质 3：对任何一棵二叉树，若其终端节点数为 n_0，度为 2 的节点数为 n_2，则 $n_0=n_2+1$。

对二叉树中节点的度求总和也就是分支的数目，而二叉树中节点总数恰好比分支数目多 1，因此性质 3 得证。

性质 4：具有 n 个节点的完全二叉树的深度为 $\lfloor \log_2 n \rfloor + 1$。

若深度为 k 的二叉树有 2^k-1 个节点，则称其为满二叉树。可以对满二叉树中的节点进行连续编号，约定编号从根节点起，自上而下、自左至右依次进行。即根节点的编号为 1，其左孩子节点编号为 2，右孩子节点编号为 3，以此类推，编号为 i 的节点的左孩子编号为 $2i$、右孩子编号为 $2i+1$。

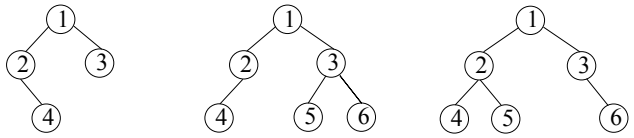
当深度为 k、有 n 个节点的二叉树，当且仅当其每一个节点都与深度为 k 的满二叉树中编号为 1～n 的节点一一对应时，称之为完全二叉树。高度为 3 的满二叉树和完全二叉树如图 3-17（a）～（d）所示，显然，满二叉树也是完全二叉树。



（a）满二叉树　（b）完全二叉树　（c）完全二叉树　（d）完全二叉树

图 3-17　满二叉树和完全二叉树示意图

在一个高度为 h 的完全二叉树中，除了第 h 层（即最后一层），其余各层都是满的。在第 h 层上的节点必须从左到右依次放置，不能留空。图 3-18 所示的高度为 3 的二叉树都不是完全二叉树，其中，（a）中 4 号节点、（b）中 5 号节点、（c）中 6 号节点的左边有空节点。



（a）　（b）　（c）

图 3-18　非完全二叉树

显然，具有 n 个节点的完全二叉树的高度为 $\lfloor \log_2 n \rfloor + 1$。

4．二叉树的存储结构

1）二叉树的顺序存储结构

用一组地址连续的存储单元存储二叉树中的节点，必须把节点排成一个适当的线性序列，并且节点在这个序列中的相互位置能反映出节点之间的逻辑关系。对于深度为 k 的完全二叉树，除第 k 层外，其余各层中含有最大的节点数，即每一层的节点数恰为其上一层节点数的两倍，由此从一个节点的编号可推知其双亲、左孩子和右孩子的编号。

假设有编号为 i 的节点，则有：

- 若 $i=1$，则该节点为根节点，无双亲；若 $i>1$，则该节点的双亲节点为 $\lfloor i/2 \rfloor$。
- 若 $2i \leqslant n$，则该节点的左孩子编号为 $2i$，否则无左孩子。
- 若 $2i+1 \leqslant n$，则该节点的右孩子编号为 $2i+1$，否则无右孩子。

完全二叉树的顺序存储结构如图 3-19（a）所示。

显然，顺序存储结构对完全二叉树而言既简单又节省空间，而对于一般二叉树则不适用。因为在顺序存储结构中，以节点在存储单元中的位置来表示节点之间的关系，那么对于一般的二叉树来说，也必须按照完全二叉树的形式存储，也就是要添上一些实际并不存在的“虚节点”，这将造成空间的浪费，如图 3-19（b）所示。

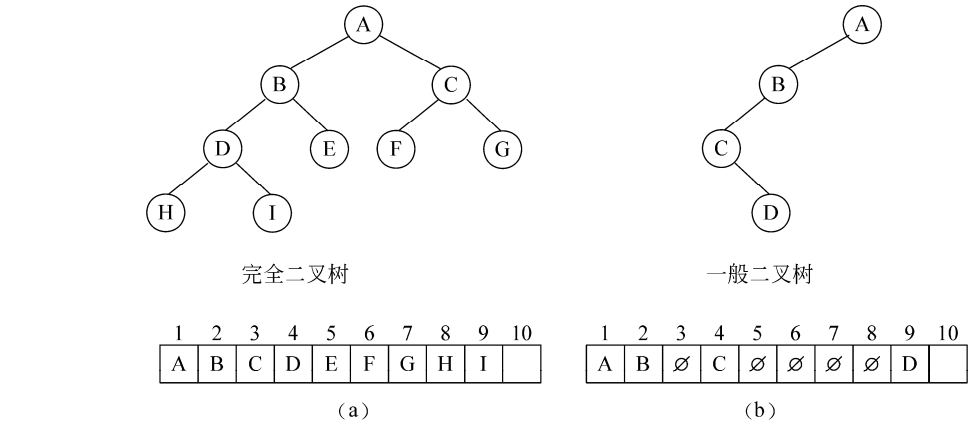


图 3-19　二叉树的顺序存储结构

在最坏的情况下，一个深度为 h 且只有 h 个节点的二叉树（单枝树）却需要 2^h–1 个存储单元。

2）二叉树的链式存储结构

由于二叉树中节点包含有数据元素、左子树根、右子树根及双亲等信息，因此可以用三叉链表或二叉链表（即一个节点含有三个指针或两个指针）来存储二叉树，链表的头指针指向二叉树的根节点，如图 3-20 所示。

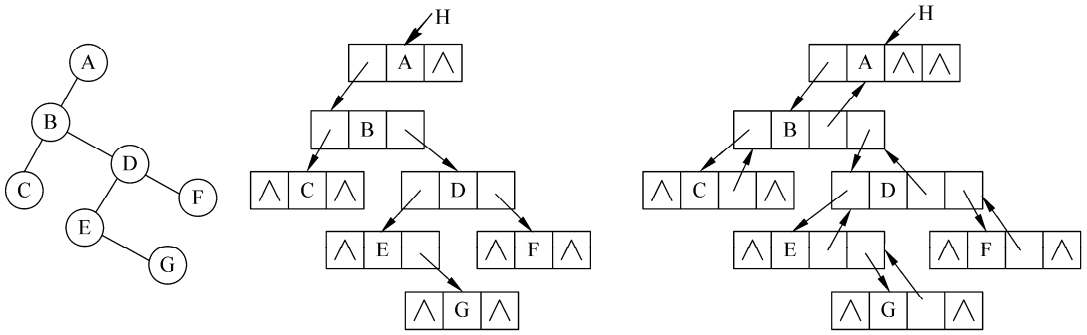


图 3-20　二叉树的链表存储结构

设节点中的数据元素为整型，则二叉链表的节点类型定义如下：

```
typedef struct BiTnode{
    int     data;                       /*节点的数据域*/
    struct BiTnode *lchild,*rchild;     /*左孩子、右孩子指针域*/
}BiTnode,*BiTree;
```

5．二叉树的遍历

遍历是按某种策略访问树中的每个节点，且仅访问一次。

由于二叉树所具有的递归性质，一棵非空的二叉树可以看作是由根节点、左子树和右子树三部分构成，因此若能依次遍历这三个部分的信息，也就遍历了整棵二叉树。按照遍历左子树要在遍历右子树之前进行的约定，依据访问根节点位置的不同，可得到二叉树的前序、中序和后序三种遍历方法。

中序遍历二叉树的操作定义如下。若二叉树为空，则进行空操作。否则：

（1）中序遍历根的左子树。

（2）访问根节点。

（3）中序遍历根的右子树。

【函数】二叉树的中序遍历算法。

```
void InOrder(BiTree root)
{   if (root == NULL) return;              /*空树*/
    else {   InOrder(root->lchild);        /*中序遍历根节点的左子树*/
             printf("%d",root->data);      /*访问根节点*/
             InOrder(root->rchild);        /*中序遍历根节点的右子树*/
    }/*if*/
}/*InOrder*/
```

实际上，将中序遍历算法中对根节点的访问操作放在左子树的遍历之前或右子树的遍历之后，就分别得到先序遍历和后序遍历算法。遍历二叉树的过程实质上是按一定规则，将树中的节点排成一个线性序列的过程。

遍历二叉树的基本操作就是访问节点，不论按照哪种次序遍历，对含有 n 个节点的二叉树，遍历算法的时间复杂度都为 $O(n)$。因为在遍历的过程中，每进行一次递归调用，都是将函数的“活动记录”压入栈中，因此，栈的容量恰为树的高度。在最坏情况下，二叉树是有 n 个节点且深度为 n 的单枝树，遍历算法的空间复杂度也为 $O(n)$。

设二叉树的根节点所在层数为 1，二叉树的层序遍历就是从树的根节点出发，首先访问第 1 层的树根节点，然后从左到右依次访问第二层上的节点，其次是第三层上的节点，以此类推，自上而下、自左至右逐层访问树中各层节点的过程就是层序遍历。

6．最优二叉树

最优二叉树又称为哈夫曼树，是一类带权路径长度最短的树。

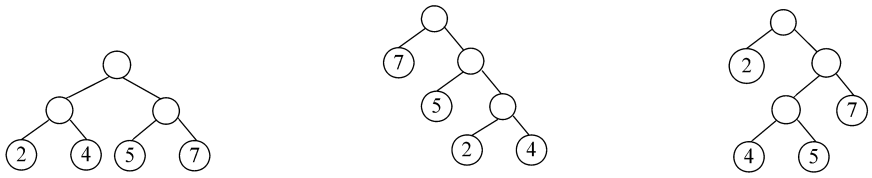
从树中一个节点到另一个节点之间的通路称为节点间的路径，该通路上分支数目称为路径长度。树的路径长度是从树根到每一个叶子之间的路径长度之和。节点的带权路径长度为从该节点到树根之间的路径长度与该节点权的乘积。

树的带权路径长度为树中所有叶子节点的带权路径长度之和，记为

$$\mathrm{WPL}=\sum_{k=1}^{n} w_k l_k$$

其中 n 为带权叶子节点数目，w_k 为叶子节点的权值，l_k 为叶子节点到根节点的路径长度。

最优二叉树是指权值为 $w_1, w_2, \cdots, w_n$ 的 n 个叶子节点的二叉树中带权路径长度最小的二叉树。例如，在图 3-21 中所示的具有 4 个叶子节点的二叉树中，以图 3-21（b）所示二叉树的带权路径长度最小。



（a）WPL= (2+4+5+7)×2 = 36　　（b）WPL= (2+4)×3+5×2+7×1 =35　　（c）WPL= (4+5)×3+7×2+2 = 43

图 3-21　不同带权路径长度的二叉树

构造最优二叉树的哈夫曼方法如下。

（1）根据给定的 n 个权值 $\{w_1, w_2, \cdots, w_n\}$ 构成 n 棵二叉树的集合 $F=\{T_1, T_2, \cdots, T_n\}$，其中每棵树 T_i 中只有一个带权为 w_i 的根节点，其左右子树均空。

（2）在 F 中选取两棵根节点的权值最小的树作为左右子树，构造一棵新的二叉树，置新构造二叉树的根节点的权值为其左、右子树根节点的权值之和。

（3）从 F 中删除这两棵树，同时将新得到的二叉树加入到 F 中。

重复（2）、（3）步，直到 F 中只含一棵树时为止，这棵树便是最优二叉树（哈夫曼树）。

最优二叉树的一个应用是对字符集中的字符进行编码和译码。

对给定的字符集 $D=\{d_1, d_2, \cdots, d_n\}$ 及权值集合 $W=\{w_1, w_2, \cdots, w_n\}$，构造其哈夫曼编码的方法为：以 $d_1, d_2, \cdots, d_n$ 作为叶子节点，$w_1, w_2, \cdots, w_n$ 作为对应叶子节点的权值，构造出一棵最优二叉树，然后将树中每个节点的左分支标上 0，右分支标上 1，则每个叶子节点代表的字符的编码就是从根到叶子的路径上的 0、1 字符组成的串。

例如，设有字符集{*a*,*b*,*c*,*d*,*e*}及对应的权值集合{0.30,0.25,0.15,0.22,0.08}，按照构造最优二叉树的哈夫曼方法：先取字符 *c* 和 *e* 所对应的节点构造一棵二叉树（根节点的权值为 *c* 和 *e* 的权值之和），然后与 *d* 对应的节点分别作为左、右子树构造二叉树，之后选 a 和 b 所对应的节点作为左、右子树构造二叉树，最后得到的最优二叉树（哈夫曼树）如图 3-22 所示。其中，字符 *a* 的编码为 00，字符 *b*、*c*、*d*、*e* 的编码分别为 01、100、11、101。译码时就从树根开始，若编码序列中当前编码为 0，则进入当前节点的左子树；为 1 则进入右子树，到达叶子时一个字符就翻译出来了，然后再从树根开始重复上述过程，直到编码序列结束。例如，若编码序列 101110000100 对应的字符编码采用图 3-22 所示的树进行构造，则可翻译出字符序列“edaac”。

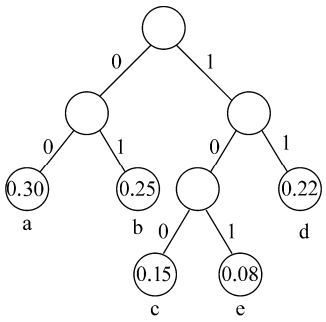


图 3-22　哈夫曼树及编码示例

7．二叉查找树

二叉查找树又称为二叉排序树，它或者是一棵空树，或者是具有如下性质的二叉树。

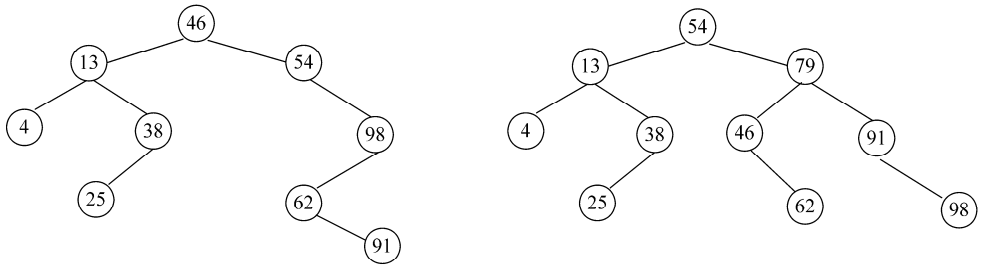
（1）若它的左子树非空，则左子树上所有节点的关键码值均小于根节点的关键码值；

（2）若它的右子树非空，则右子树上所有节点的关键码值均大于根节点的关键码值；

（3）左、右子树本身就是两棵二叉查找树。

一棵二叉查找树如图 3-23（a）所示。图 3-23（b）所示的二叉树不是二叉查找树，因为 46 比 54 小，它应该在根节点 54 的左子树上。

从二叉查找树的定义可知，对二叉查找树进行中序遍历，可得到一个关键码递增有序的节点序列。



（a） 二叉查找树　　（b） 非二叉查找树

图 3-23　二叉查找树与非二叉查找树

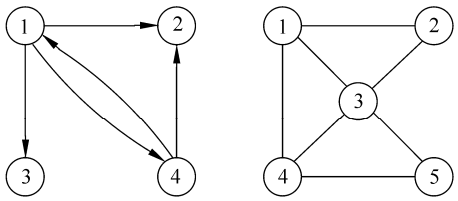
3.3.2　图

图是比树结构更复杂的一种数据结构。在树结构中，可认为除根节点没有前驱节点外，其余的每个节点只有唯一的一个前驱（双亲）节点和多个后继（子树）节点。而在图结构中，任意两个节点之间都可能有直接的关系，所以图中一个节点的前驱和后继的数目是没有限制的。图结构被用于描述各种复杂的数据对象，在自然科学、社会科学和人文科学等许多领域有非常广泛的应用。

1．图的定义及术语

图 G 是由两个集合 V 和 E 构成的二元组，记作 $G=(V, E)$，其中 V 是图中顶点的非空有限集合，E 是图中边的有限集合。从数据结构的逻辑关系角度来看，图中任一顶点都有可能与图中其他顶点有关系，而图中所有顶点都有可能与某一顶点有关系。在图中，数据结构中的数据元素用顶点表示，数据元素之间的关系用边表示。

- 有向图：若图中每条边都是有方向的，则称为有向图。从顶点 v_i 到 v_j 的有向边$<v_i, v_j>$也称为弧，起点 v_i 称为弧尾，终点 v_j 称为弧头。在有向图中，$<v_i, v_j>$与$<v_j, v_i>$分别表示两条弧，如图 3-24（a）所示。
- 无向图：若图中的每条边都是无方向的，顶点 v_i 和 v_j 之间的边用（v_i, v_j）表示。在无向图中，（v_i, v_j）与（v_j, v_i）表示的是同一条边。5 个顶点的一个无向图如图 3-24（b）所示。
- 完全图：若一个无向图具有 n 个顶点，而每一个顶点与其他 n–1 个顶点之间都有边，则称之为无向完全图。显然，含有 n 个顶点的无向完全图共有 $n(n-1)/2$ 条边。类似地，有 n 个顶点的有向完全图中弧的数目为 $n(n-1)$，即任意两个不同顶点之间都存在方向相反的两条弧。



（a）有向图　　（b）无向图

图 3-24　有向图和无向图示意图

- 度、出度和入度：顶点 v 的度是指关联于该顶点的边的数目，记作 $D(v)$。若 G 为有向图，顶点的度表示该顶点的入度和出度之和。顶点的入度是以该顶点为终点的有向边

的数目，而顶点的出度指以该顶点为起点的有向边的数目，分别记为 ID(v)和 OD(v)。例如，图 3-24（a）中，顶点 1, 2, 3, 4 的入度分别为 1, 2, 1, 1，出度分别为 3, 0, 0, 2。图 3-24（b）中，顶点 1, 2, 3, 4, 5 的度分别为 3, 2, 4, 3, 2。

- 路径：在无向图 G 中，从顶点 v_p 到顶点 v_q 的路径是指存在一个顶点序列 $v_p, v_{i1}, v_{i2}, \cdots, v_{in}, v_q$，使得 (v_p, v_{i1})，(v_{i1}, v_{i2})，…，(v_{in}, v_q)均属于 $E(G)$。若 G 是有向图，其路径也是有方向的，它由 $E(G)$中的有向边$<v_p, v_{i1}>, <v_{i1}, v_{i2}>, \cdots, <v_{in}, v_q>$组成。路径长度是路径上边或弧的数目。第一个顶点和最后一个顶点相同的路径称为回路或环。若一条路径上除了 v_p 和 v_q 可以相同外，其余顶点均不相同，这种路径称为一条简单路径。
- 子图：若有两个图 $G=(V,E)$ 和 $G'=(V',E')$，如果 $V'\subseteq V$ 且 $E'\subseteq E$，则称 G' 为 G 的子图。
- 连通图：在无向图 G 中，若从顶点 v_i 到顶点 v_j 有路径，则称顶点 v_i 和顶点 v_j 是连通的。如果无向图 G 中任意两个顶点都是连通的，则称其为连通图。图 3-24（b）所示的无向图是连通图。
- 强连通图：在有向图 G 中，如果对于每一对顶点 v_i，$v_j\in V$ 且 $v_i\neq v_j$，从顶点 v_i 到顶点 v_j 和从顶点 v_j 到顶点 v_i 都存在路径，则称图 G 为强连通图。图 3-24（a）所示的有向图不是强连通图。以顶点 1 和顶点 3 为例，顶点 1 至顶点 3 存在路径，而顶点 3 至顶点 1 没有路径。
- 网：边（或弧）具有权值的图称为网。

从图的逻辑结构的定义来看，图中的顶点之间不存在全序关系（即无法将图中的顶点排列成一个线性序列），任何一个顶点都可被看成第一个顶点；另一方面，任一顶点的邻接点之间也不存在次序关系。为便于运算，为图中每个顶点赋予一个序号。

2．图的存储结构

邻接矩阵和邻接表是两种常用的图的存储结构。

1）邻接矩阵表示法

邻接矩阵表示法利用一个矩阵来表示图中顶点之间的关系。对于具有 n 个顶点的图 $G=(V, E)$来说，其邻接矩阵是一个 n 阶方阵，且满足：

$$A[i][j]=\begin{cases}1 & \text{若}(v_i,v_j)\text{或}<v_i,v_j>\text{是 }E\text{ 中的边}\\ 0 & \text{若}(v_i,v_j)\text{或}<v_i,v_j>\text{不是 }E\text{ 中的边}\end{cases}$$

有向图和无向图的邻接矩阵如图 3-25 中的矩阵 ***A*** 和 ***B*** 所示。

由邻接矩阵的定义可知，无向图的邻接矩阵是对称的，而有向图的邻接矩阵则不一定具有

该性质。

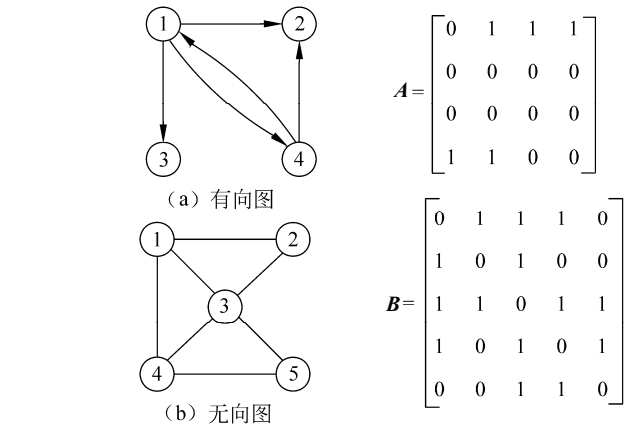


图 3-25　有向图和无向图的邻接矩阵存储示意图

借助于邻接矩阵，可判定任意两个顶点之间是否有边（或弧）相连，并且容易求得各个顶点的度。对于无向图，顶点 v_i 的度是邻接矩阵中第 i 行（或列）的值不为 0 的元素数目（或元素的和）；对于有向图，第 i 行的元素之和为顶点 v_i 的出度 $\mathrm{OD}(v_i)$，第 j 列的元素之和为顶点 v_j 的入度 $\mathrm{ID}(v_j)$。

类似地，网（赋权图）的邻接矩阵可定义为

$$\boldsymbol{A}[i][j]=\begin{cases}W_{ij} & \text{若}(v_i,v_j)\text{或}<v_i,v_j>\text{是 }E\text{ 中的边}\\ \infty & \text{若}(v_i,v_j)\text{或}<v_i,v_j>\text{不是 }E\text{ 中的边}\end{cases}$$

图 3-26 所示的是网及其邻接矩阵 $\boldsymbol{C}$。

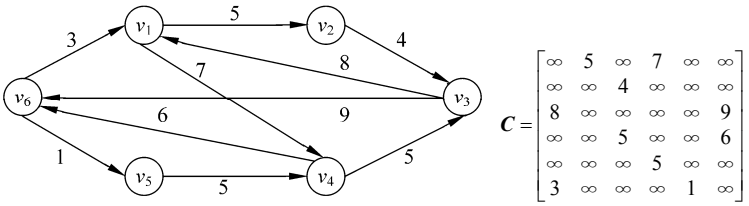


图 3-26　一个网及其邻接矩阵表示

若图用邻接矩阵表示，则图的数据类型可定义为：

```
#define MaxN     36                          /*图中顶点数目的最大值*/
typedef int AdjMatrix[MaxN][MaxN];           /*邻接矩阵*/
```

或

```
typedef double AdjMatrix[MaxN][MaxN];        /*网（赋权图）的邻接矩阵*/
typedef struct {
    int Vnum;                                /*图中的顶点数目*/
    AdjMatrix     Arcs;
}Graph;
```

2）邻接链表表示法

邻接链表指的是为图的每个顶点建立一个单链表，第 i 个单链表中的节点表示依附于顶点 v_i 的边（对于有向图是以 v_i 为尾的弧）。邻接链表中的节点有表节点和表头节点两种类型，如下所示。

表节点

adjvex	nextarc	info

表头节点

data	firstarc

其中各参数的含义如下。

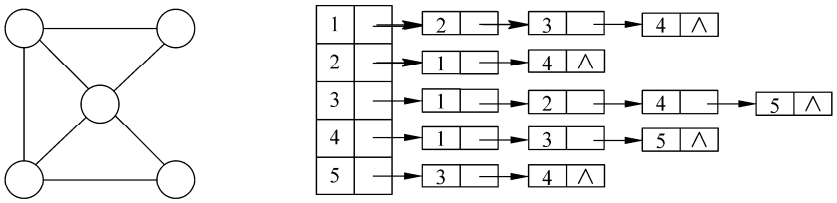
- adjvex：指示与顶点 v_i 邻接的顶点的序号。
- nextarc：指示下一条边或弧的节点。
- info：存储和边或弧有关的信息，如权值等。
- data：存储顶点 v_i 的名或其他有关信息。
- firstarc：指示链表中的第一个节点。

这些表头节点通常以顺序结构的形式存储，以便随机访问任一顶点及其邻接表。若图用邻接链表来表示，则对应的数据类型可定义如下：

```
#define MaxN    36                  /*图中顶点数目的最大值*/
typedef struct ArcNode{             /*邻接链表的表节点类型*/
    int adjvex;                     /*邻接顶点的顶点序号*/
    double weight;                  /*边(弧)上的权值*/
    struct ArcNode *nextarc;        /*下一个邻接顶点*/
}EdgeNode;
typedef struct VNode{               /*邻接链表的头节点*/
    char    data;                   /*顶点表示的数据，以一个字符表示*/
    struct ArcNode *firstarc;       /*指向第一条依附于该顶点的边（弧）的指针*/
}AdjList[MaxN];
typedef struct {
```

```
    int Vnum;                    /*图中实际的顶点数目*/
    AdjList    Vertices;
}Graph;
```

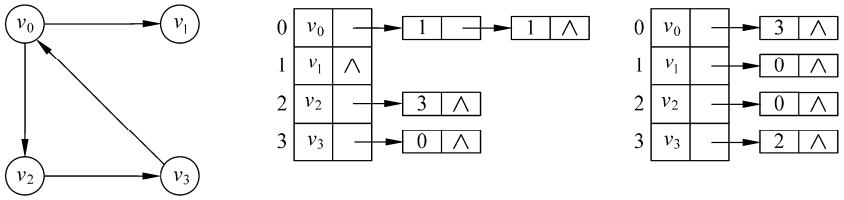
显然，对于有 n 个顶点、e 条边的无向图来说，其邻接链表需要 n 个头节点和 $2e$ 个表节点，如图 3-27 所示。对于无向图的邻接链表，顶点 v_i 的度恰为第 i 个邻接链表中表节点的数目。



（a）无向图　　（b）无向图的邻接表表示

图 3-27　无向图的邻接表表示

图 3-28（b）是图 3-28（a）所示有向图的邻接表。从中可以看出，由于第 i 个邻接链表中表节点的数目只是顶点 v_i 的出度，因此必须逐个扫描每个顶点的邻接表，才能求出一个顶点的入度。为此，可以建立一个有向图的逆邻接链表，如图 3-28（c）所示。



（a）有向图　　（b）邻接表　　（c）逆邻接表

图 3-28　有向图的邻接表及逆邻接表表示

3.4　常用算法

3.4.1　算法概述

1．算法的基本概念

算法是问题求解过程的精确描述，它为解决某一特定类型的问题规定了一个运算过程，并且具有下列特性。

（1）有穷性。一个算法必须在执行有穷步骤之后结束，且每一步都可在有穷时间内完成。

（2）确定性。算法的每一步必须是确切定义的，不能有歧义。

（3）可行性。算法应该是可行的，这意味着算法中所有要进行的运算都能够由相应的计算装置所理解和实现，并可通过有穷次运算完成。

（4）输入。一个算法有零个或多个输入，它们是算法所需的初始量或被加工的对象的表示。这些输入取自特定的对象集合。

（5）输出。一个算法有一个或多个输出，它们是与输入有特定关系的量。

因此，算法实质上是特定问题的可行的求解方法、规则和步骤。一个算法的优劣可从以下几个方面考查。

（1）正确性。正确性也称为有效性，是指算法能满足具体问题的要求。即对任何合法的输入，算法都能得到正确的结果。

（2）可读性。可读性指算法被理解的难易程度。人们常把算法的可读性放在比较重要的位置，因为晦涩难懂的算法不易交流和推广使用，也难以修改和扩展。因此，设计的算法应尽可能简单易懂。

（3）健壮性。健壮性也称为鲁棒性，即对非法输入的抵抗能力。对于非法的输入数据，算法应能加以识别和处理，而不会产生误动作或执行过程失控。

（4）效率。粗略地讲，就是算法运行时花费的时间和使用的空间。对算法的理想要求是运行时间短、占用空间小。

2．算法与数据结构

算法与数据结构密切相关，算法实现时总是建立在一定的数据结构基础之上。

计算机程序从根本上看包括两方面的内容：一是对数据的描述，二是对操作（运算）的描述。概括来讲，在程序中需要指定数据的类型和数据的组织形式就是定义数据结构，描述的操作步骤就构成了算法。因此，从某种意义上可以说“数据结构+算法=程序”。

当然，设计程序时还需选择不同的程序设计方法、程序语言及工具。但是，数据结构和算法仍然是程序中最为核心的内容。用计算机求解问题时，一般应先设计初步的数据结构，然后再考虑相关的算法及其实现。设计数据结构时应当考虑可扩展性，修改数据结构会影响算法的实现方案。

3．算法的描述

算法的描述方法有很多，若用程序语言描述，就成了计算机程序。常用的算法描述方法有流程图、NS 盒图、伪代码和决策表等。

（1）流程图。流程图（Flow Chart）即程序框图，是历史最久、流行最广的一种算法的图形表示方法。每个算法都可由若干张流程图描述。流程图给出了算法中所进行的操作以及这些操作执行的逻辑顺序。程序流程图包括三种基本成分：加工步骤，用方框表示；逻辑条件，用菱形表示；控制流，用箭头表示。流程图中常用的几种符号如图 3-29 所示。

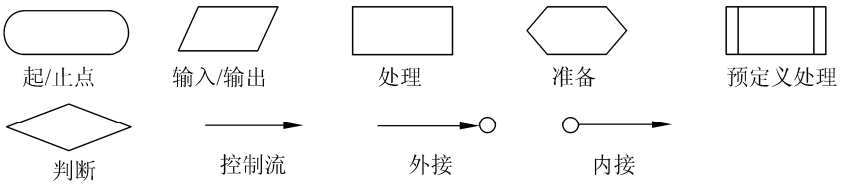
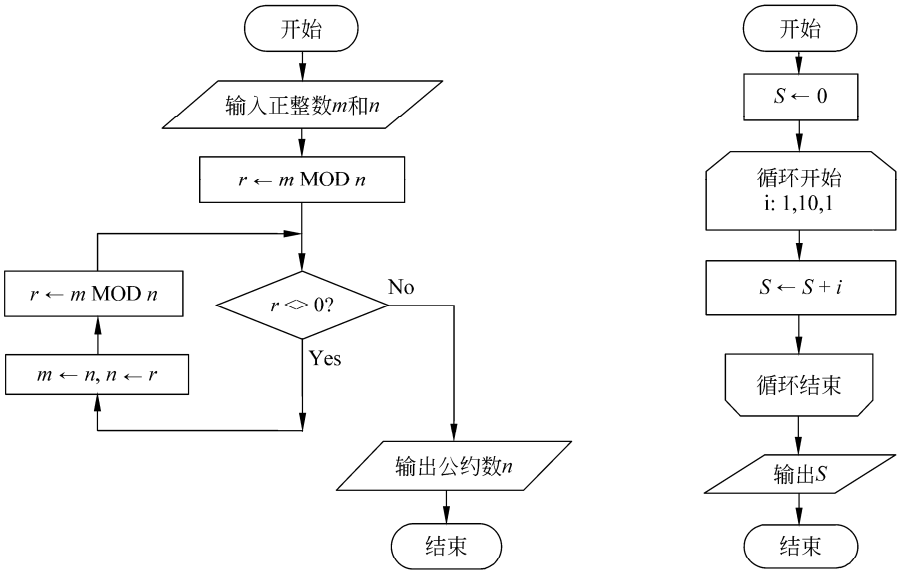


图 3-29　流程图的基本符号

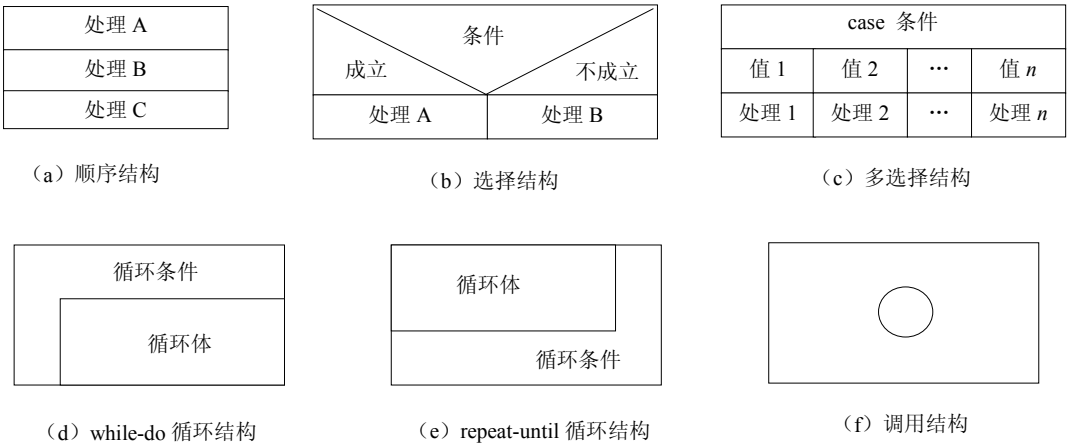
例如，求正整数 *m* 和 *n* 的最大公约数流程图如图 3-30（a）所示。若流程图中的循环结构通过控制变量以确定的步长进行计次循环，则可用⌓和⌔分别表示“循环开始”和“循环结束”，并在“循环开始”框中标注“循环控制变量：初始值，终止值，增量”，如图 3-30（b）所示。



（a）求*m*和*n*的最大公约数　　（b）求1+2+…+10

图 3-30　算法的流程图表示

（2）N/S 盒图。盒图是结构化程序设计出现之后，为支持这种设计方法而产生的一种描述工具。N/S 盒图的基本元素与控制结构如图 3-31 所示。在 N/S 图中，每个处理步骤用一个盒子表示，盒子可以嵌套。对于每个盒子，只能从上面进入，从下面走出，除此之外别无其他出入口，所以盒图限制了随意的控制转移，保证了程序的良好结构。



（a）顺序结构　（b）选择结构　（c）多选择结构

（d）while-do 循环结构　（e）repeat-until 循环结构　（f）调用结构

图 3-31　N/S 盒图的基本元素与控制结构

用 N/S 盒图描述求最大公约数的欧几里得算法，如图 3-32 所示。

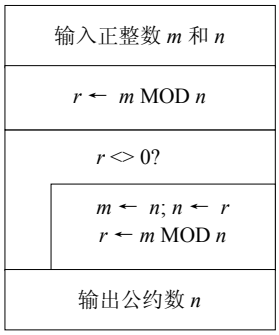


图 3-32　求 m、n 的最大公约数的 N/S 盒图

（3）伪代码。用伪代码描述算法的特点是借助于程序语言的语法结构和自然语言叙述，使算法具有良好的结构又不拘泥于程序语言的限制。这样的算法易读易写，而且容易转换成程序。

（4）决策表。决策表是一种图形工具，它将比较复杂的决策问题简洁、明确、一目了然地

描述出来。例如，如果订购金额超过 500 元，以前没有欠账，则发出批准单和提货单；如果订购金额超过 500 元，但以前的欠账尚未还清，则发不予批准的通知；如果订购金额低于 500 元，则不论以前的欠账是否还清都发批准单和提货单，在欠账未还清的情况下还要发出“催款单”。处理该问题的决策表如表 3-1 所示。

表 3-1　决策表

订 购 金 额	>500	>500	⩽500	⩽500
欠账情况	已还清	未还清	已还清	未还清
发不批准通知		√		
发出批准单	√		√	√
发出提货单	√		√	√
发出催款单				√

4．算法效率

解决同一个问题总是存在多种算法，每个算法在计算机上执行时，都要消耗时间（CPU 执行指令的时间）和使用存储空间资源。因此，设计算法时需要考虑算法运行时所花费的时间和使用的空间，以时间复杂度和空间复杂度表示。

由于算法往往和需要求解的问题规模相关，因此常将问题规模 n 作为一个参照量，求算法的时间、空间开销与 n 的关系。详细分析指令的执行时间会涉及计算机运行过程的细节，因此时间消耗情况难以精确表示，所以算法分析时常采用算法的时空开销随 n 的增长趋势来表示其时空复杂度。

对于一个算法的时间开销 $T(n)$，从数量级大小考虑，当 n 增大到一定值后，$T(n)$计算公式中影响最大的就是 n 的幂次最高的项，其他的常数项和低幂次项都可以忽略，即采用渐进分析，表示为 $T(n)=O(f(n))$。其中，n 反映问题的规模，$T(n)$是算法运行所消耗时间或存储空间的总量，O 是数学分析中常用的符号“大 O”，而 $f(n)$是自变量为 n 的某个具体的函数表达式。例如，若 $f(n)=n^2+2n+1$，则 $T(n)=O(n^2)$。

下面以语句频度为基础给出算法时间复杂度的度量。

语句频度（Frequency Count）是指语句被重复执行的次数，即对于某个基本语句，若在算法的执行过程中被执行 n 次，则其语句频度为 n。这里的“语句”是指描述算法的基本语句（基本操作），它的执行是不可分割的，因此，循环语句的整体、函数调用语句不能算作基本语句，因为它们还包括循环体或函数体。

算法中各基本语句的语句频度之和表示算法的执行时间。

例如，对于下面的三个程序段（1）、（2）、（3），其实现基本操作“x 增 1”的语句++x 的

语句频度分别为 1、n、n^2。

（1）{s=0; ++x;}

（2）for(i = 1; i <= n; i++)　{s += x; ++x;}

（3）for(k = 1; k <= n; ++k)

　　for(i = 1; i <= n; i++)

　　{s += x; ++x;}

因此，程序段（1）、（2）、（3）的时间复杂度分别为 $O(1)$、$O(n)$、$O(n^2)$，分别称为常量阶、线性阶和平方阶。若程序段（1）、（2）、（3）组成一个算法的整体，则该算法的时间复杂度为 $O(n^2)$。

3.4.2 排序

假设含 n 个记录的文件内容为$\{R_1, R_2, \cdots, R_n\}$，其相应的关键字为$\{k_1, k_2, \cdots, k_n\}$。经过排序确定一种排列$\{R_{j1}, R_{j2}, \cdots, R_{jn}\}$，使得它们的关键字满足如下递增（或递减）关系：$k_{j1} \leqslant k_{j2} \leqslant \cdots \leqslant k_{jn}$ (或 $k_{j1} \geqslant k_{j2} \geqslant \cdots \geqslant k_{jn}$)。

1）排序的稳定性

若在待排序的一组序列中，R_i 和 R_j 的关键字相同，即 $k_i = k_j$，且在排序前 R_i 领先于 R_j，那么当排序后，如果 R_i 和 R_j 的相对次序保持不变，R_i 仍领先于 R_j，则称此类排序方法为稳定的。若在排序后的序列中有可能出现 R_j 领先于 R_i 的情形，则称此类排序为不稳定的。

2）内部排序和外部排序

内部排序是指待排序记录全部存放在内存中进行排序的过程。

外部排序是指待排序记录的数量很大，以至内存不能容纳全部记录，在排序过程中尚需对外存进行访问的排序过程。

在排序过程中需进行下列两种基本操作。

（1）比较两个关键字的大小。

（2）将记录从一个位置移动到另一个位置。

1．简单排序

这里的简单排序包括直接插入排序、冒泡排序和简单选择排序。

1）直接插入排序

直接插入排序是一种简单的排序方法，具体做法是：在插入第 i 个记录时，$R_1, R_2, \cdots, R_{i-1}$ 已经排好序，这时将记录 R_i 的关键字 k_i 依次与关键字 $k_{i-1}, k_{i-2}, \cdots, k_1$ 进行比较，从而找到 R_i 应该插入的位置，插入位置及其后的记录依次向后移动。

【算法】直接插入排序算法。

```
void Insertsort(int data[], int n )
/*将数组 data[0]～data[n-1]中的 n 个整数按非递减有序的方式进行排列*/
{   int i, j;
    int tmp;
    for(i = 1; i < n; i++){
        if (data[i] < data[i-1]) {
            tmp = data[i];    data[i] = data[i-1];
            for(j = i-1; j>=0&&data[j] > tmp; j--)    data[j+1] = data[j];
            data[j+1] = tmp;
        }/*if*/
  }/*for*/
}/*Insertsort*/
```

直接插入排序法在最好情况下（待排序列已按关键码有序），每趟排序只需作 1 次比较且不需要移动元素，因此 n 个元素排序时的总比较次数为 n–1 次，总移动次数为 0 次。在最坏情况下（元素已经逆序排列），进行第 i 趟排序时，待插入的记录需要同前面的 i 个记录都进行 1 次比较，因此，总比较次数为 $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$。排序过程中，第 i 趟排序时移动记录的次数为 i+1（包括移进、移出 tmp），总移动次数为 $\sum_{i=2}^{n}(i+1) = \frac{(n+3)(n-2)}{2}$。

由此，直接插入排序是一种稳定的排序方法，其时间复杂度为 $O(n^2)$。排序过程中仅需要一个元素的辅助空间，空间复杂度为 $O(1)$。

2）冒泡排序

n 个记录进行冒泡排序的方法是：首先将第一个记录的关键字和第二个记录的关键字进行比较，若为逆序，则交换两个记录的值，然后比较第二个记录和第三个记录的关键字，以此类推，直至第 n–1 个记录和第 n 个记录的关键字比较完为止。上述过程称作一趟冒泡排序，其结果是关键字最大的记录被交换到第 n 个位置。然后进行第二趟冒泡排序，对前 n–1 个记录进行同样的操作，其结果是关键字次大的记录被交换到第 n–1 个位置。当进行完第 n–1 趟时，所有记录有序排列。

【算法】冒泡排序算法。

```
void Bubblesort(int data[],int n )
/*将数组 data[0]～data[n-1]中的 n 个整数按非递减有序的方式进行排列*/
```

```
{   int i,j,tag;   /*用 tag 表示排序过程中是否交换过元素值*/
    int tmp;
    for(i = 1,tag = 1;tag == 1&&i < n;i++){
        tag = 0;
        for(j = 0;j < n–i;j++)
            if (data[j]>data[j+1]){
                tmp = data[j]; data[j] = data[j+1]; data[j+1] = tmp;
                tag = 1;
            }/*if*/
    }/*for*/
}/*Bubblesort*/
```

冒泡排序法在最好情况下（待排序列已按关键码有序）只需作 1 趟排序，元素的比较次数为 n–1 且不需要交换元素，因此总比较次数为 n–1 次，总交换次数为 0 次。在最坏情况下（元素已经逆序排列），进行第 i 趟排序时，最大的 i–1 个元素已经排好序，其余的 n–(i–1)个元素需要进行 n–i 次比较和 n–i 次交换，因此总比较次数为 $\sum_{i=1}^{n-1}(n-i)=\frac{n(n-1)}{2}$，总交换次数为 $\sum_{i=1}^{n-1}(n-i)=\frac{n(n-1)}{2}$。

由此，冒泡排序是一种稳定的排序方法，其时间复杂度为 $O(n^2)$。排序过程中仅需要一个元素的辅助空间用于元素的交换，空间复杂度为 $O(1)$。

3）简单选择排序

n 个记录进行简单选择排序的基本方法是：通过 n–i 次关键字之间的比较，从 n–i+1 个记录中选出关键字最小的记录，并和第 i(1≤i≤n)个记录进行交换，当 i 等于 n 时所有记录有序排列。

【算法】简单选择排序算法。

```
void Selectsort(int data[],int n )
/*将数组 data[0]～data[n-1]中的 n 个整数按非递减有序的方式进行排列*/
{   int i,j,k;
    int tmp;
    for(i = 1;i < n;i++){
        k = i;
        for(j = i+1;j <= n;j++)          /*找出最小元素的下标，用 k 表示*/
            if (data[j] < data[k]) k = j;
```

```
        if (k != i) {
            tmp = data[i]; data[i] = data[k]; data[k] = tmp;
        }/*if*/
    }/*for*/
}/*Selectsort*/
```

简单选择排序法在最好情况下（待排序列已按关键码有序）不需要移动元素，因此 n 个元素排序时的总移动次数为 0 次。在最坏情况下（元素已经逆序排列），每趟排序移动记录的次数都为 3 次（两个数组元素交换值），共进行 n–1 趟排序，总移动次数为 3(n–1)。无论在哪种情况下，元素的总比较次数为 $\sum_{i=1}^{n-1}(n-i)=\frac{n(n-1)}{2}$。

由此，简单选择排序是一种不稳定的排序方法，其时间复杂度为 $O(n^2)$。排序过程中仅需要一个元素的辅助空间用于数组元素值的交换，空间复杂度为 $O(1)$。

2．希尔排序

希尔排序又称“缩小增量排序”，是对直接插入排序方法的改进。

希尔排序的基本思想是：先将整个待排记录序列分割成若干子序列，然后分别进行直接插入排序，待整个序列中的记录基本有序时，再对全体记录进行一次直接插入排序。具体做法是：先取定一个小于 n 的整数 d_1 作为第一个增量，把文件的全部记录分成 d_1 组，将所有距离为 d_1 倍数的记录放在同一个组中，在各组内进行直接插入排序；然后取第二个增量 d_2（$d_2<d_1$），重复上述的分组和直接插入排序过程，以此类推，直至所取的增量 $d_i=1(d_i<d_{i-1}<\cdots<d_2<d_1)$，即所有记录放在同一组进行直接插入排序为止。

增量序列为 5，3，1 时，希尔插入排序过程如图 3-33 所示。

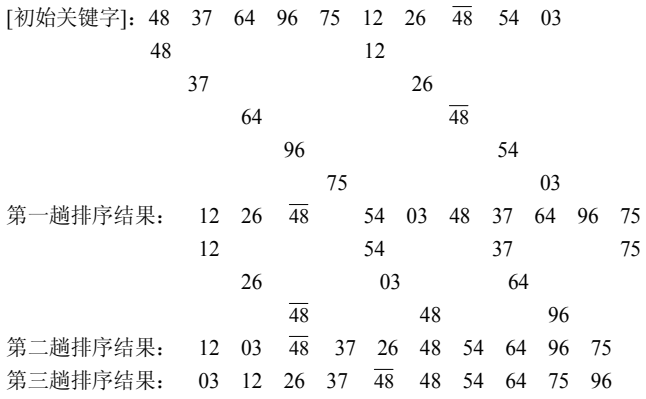


图 3-33　希尔排序示例

【函数】用希尔排序方法对整型数组进行非递减排序。

```
void shellsort(int data[],int n)
{   int *delta,k,i,t,dk,j; //构造一个增量序列存入 delta
  k = n;
  /*从 k=n 开始，重复 k=k/2 运算，直到 k=0，所得 k 值的序列作为增量序列存入 delta*/
  delta = (int *)malloc(sizeof(int)*(n/2));
  i = 0;
  do{
    k = k/2;       delta[i++] = k;
  }while(k>0);
  i = 0;
  while ((dk = delta[i]) > 0)     {
    for(k = delta[i];k < n;++k)
      if (data[k] < data[k-dk]) {   /*将元素 data[k]插入到有序增量子表中*/
        t = data[k];                /*备份待插入的元素，空出一个元素位置*/
        for(j = k-dk;j >= 0 && t < data[j];j -= dk)
          data[j+dk] = data[j];     /*寻找插入位置的同时元素后移*/
        data[j+dk] = t;             /*找到插入位置，插入元素*/
      }/*if*/
    ++i; /*取下一个增量值*/
  }/*while*/
}/* shellsort */
```

希尔排序是不稳定的排序方法。

3．快速排序

快速排序的基本思想是：通过一趟排序将待排的记录划分为独立的两部分，称为前半区和后半区，其中，前半区中记录的关键字均不大于后半区记录的关键字，然后再分别对这两部分记录继续进行快速排序，从而使整个序列有序。

一趟快速排序的过程称为一次划分，具体做法是：附设两个位置指示变量 *i* 和 *j*，它们的初值分别指向序列的第一个记录和最后一个记录。设枢轴记录（通常是第一个记录）的关键字为 pivot，则首先从 *j* 所指位置起向前搜索，找到第一个关键字小于 pivot 的记录时将该记录向前移到 *i* 指示的位置，然后从 *i* 所指位置起向后搜索，找到第一个关键字大于 pivot 的记录时将该记录向后移到 *j* 所指位置，重复该过程直至 *i* 与 *j* 相等为止。

【函数】快速排序过程中的划分。

```
int partition(int data[], int low, int high)
 /*用 data[low]作为枢轴元素 pivot 进行划分*/
 /*使得 data[low..i-1]均不大于 pivot，data[i+1..high]均不小于 pivot*/
{           int i, j;    int pivot;
            pivot = data[low];    i = low;    j = high;
            while(i < j) {                          /*从数组的两端交替地向中间扫描*/
                  while(i < j && data[j] >= pivot) j--;
                  data[i] = data[j];                /*比枢轴元素小者往前移*/
                  while (i < j && data[i] <= pivot) i++;
                  data[j] = data[i];                /*比枢轴元素大者向后移*/
             }
             data[i] = pivot;
             return i;
}
```

【函数】用快速排序方法对整型数组进行非递减排序。

```
void quickSort(int data[], int low, int high)
 /*用快速排序方法对数组元素 data[low..high]作非递减排序*/
{
     if (low < high) {
            int loc = partition(data, low, high);       /*进行划分*/
            quicksort(data,low,loc-1);                  /*对前半区进行快速排序*/
            quicksort(data,loc+1,high);                 /*对后半区进行快速排序*/
     }
}/* quickSort */
```

快速排序算法的时间复杂度为 $O(n\log_2 n)$，在所有算法复杂度为此数量级的排序方法中，快速排序被认为是平均性能最好的一种。但是，若初始记录序列按关键字有序或基本有序时，即每次划分都是将序列划分为某一半序列的长度为 0 的情况，此时快速排序的性能退化为时间复杂度是 $O(n^2)$。快速排序是不稳定的排序方法。

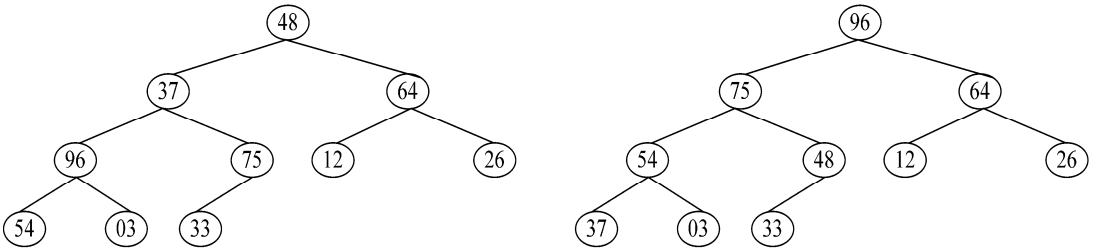
4．堆排序

对于 n 个元素的关键字序列$\{k_1, k_2, \cdots, k_n\}$，当且仅当满足下列关系时称其为堆。

$$\begin{cases} k_i \leqslant k_{2i} \\ k_i \leqslant k_{2i+1} \end{cases} \quad 或 \quad \begin{cases} k_i \geqslant k_{2i} \\ k_i \geqslant k_{2i+1} \end{cases}$$

若将此序列对应的一维数组（即以一维数组作为序列的存储结构）看成是一个完全二叉树，则堆的含义表明，完全二叉树中所有非终端节点的值均不大于（或不小于）其左、右孩子节点的值。因此，在一个堆中，堆顶元素（即完全二叉树的根节点）必为序列中的最小元素（或最大元素），并且堆中任一棵子树也都是堆。若堆顶为最小元素，则称为小顶堆；若堆顶为最大元素，则称为大顶堆。

例如，将序列（48，37，64，96，75，12，26，54，03，33）中的元素依次放入一棵完全二叉树中，如图 3-34（a）所示。显然，它既不是大顶堆（48<64），也不是小顶堆（48>37），调整为大顶堆后如图 3-34（b）所示。



（a）非堆　　（b）大顶堆

图 3-34　用完全二叉树表示堆

堆排序的基本思想是：对一组待排序记录的关键字，首先把它们按堆的定义排成一个序列（即建立初始堆），从而输出堆顶的最小关键字（对于小顶堆而言）。然后将剩余的关键字再调整成新堆，便得到次小的关键字，如此反复，直到全部关键字排成有序序列为止。

n 个元素进行堆排序时，时间复杂度为 $O(n\log_2 n)$，空间复杂度为 $O(1)$。堆排序是不稳定的排序方法。

5．归并排序

所谓“归并”，是将两个或两个以上的有序文件合并成为一个新的有序文件。从线性表的讨论可知，将两个有序表合并成为一个有序表，无论是顺序存储结构还是链式存储结构，都是容易实现的。利用归并的思想可以进行排序。归并排序是把一个有 n 个记录的无序文件看成是由 n 个长度为 1 的有序子文件组成的文件，然后进行两两归并，得到 $\left\lceil \frac{n}{2} \right\rceil$ 个长度为 2 或 1 的有序文件，再两两归并，如此重复，直至最后形成包含 n 个记录的有序文件为止。这种反复将两个有序文件归并成一个有序文件的排序方法称为两路归并排序。

n 个元素进行二路归并排序的时间复杂度为 $O(n\log_2 n)$，空间复杂度为 $O(n)$。二路归并排序是稳定的排序方法。

6．内部排序方法小结

综合比较以上所讨论的各种排序方法，大致结果如表 3-2 所示。

表 3-2　各种排序方法的性能比较

排 序 方 法	最 好 时 间	平 均 时 间	最 坏 时 间	辅 助 空 间	稳　定　性
直接插入排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
简单选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
冒泡排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
希尔排序	—	$O(n^{1.25})$	—	$O(1)$	不稳定
快速排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$	$O(\log_2 n)\sim O(n)$	不稳定
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定
归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	稳定

不同的排序方法各有优缺点，可根据需要运用到不同的场合。选取排序方法时需要考虑的主要因素有：待排序的记录个数 n；记录本身的大小；关键字的分布情况；对排序稳定性的要求；语言工具的条件，辅助空间的大小等。

依据这些因素，可以得到以下几点结论。

（1）若待排序的记录数目 n 较小时，可采用插入排序和简单选择排序。由于直接插入排序所需的记录移动操作较简单选择排序多，因此当记录本身信息量较大时，用简单选择排序方法较好。

（2）若待排序记录按关键字基本有序，则宜采用直接插入排序或冒泡排序。

（3）若 n 较大，则应采用时间复杂度为 $O(n\log_2 n)$的排序方法，例如快速排序、堆排序或归并排序。

快速排序是目前内部排序方法中被认为是最好的方法，当待排序的关键字随机分布时，快速排序的平均时间最短；堆排序只需一个辅助存储空间，并且不会出现在快速排序中可能出现的最坏情况。这两种排序方法都是不稳定的排序方法，若要求排序稳定，可选择归并排序。通常可将归并排序和直接插入排序结合起来使用。先利用直接插入排序求得较长的有序子文件，然后再两两归并。

前面讨论的内部排序算法都是在一维数组上实现的。当记录本身信息量较大时，为避免耗费大量的时间移动记录，可以采用链表作为存储结构。

7．外部排序

外部排序就是对大型文件的排序，待排序的记录存放在外存。在排序的过程中，内存只存

储文件的一部分记录，整个排序过程需要进行多次内外存间的数据交换。

常用的外部排序方法是归并排序，一般分为两个阶段：在第一阶段，把文件中的记录分段读入内存，利用某种内部排序方法对这段记录进行排序并输出到外存的另一个文件中，在新文件中形成许多有序的记录段，称为归并段；在第二阶段，对第一阶段形成的归并段用某种归并方法进行一趟趟的归并，使文件的有序段逐渐加长，直到将整个文件归并为一个有序段时为止。

3.4.3 查找

查找是非数值数据处理中一种常用的基本运算，查找运算的效率与查找表所采用的数据结构和查找方法密切相关。

1．查找表及查找效率

查找表是指由同一类型的数据元素（或记录）构成的集合。由于“集合”这种数据结构中的数据元素之间存在着完全松散的关系，因此，查找表是一种非常灵活的数据结构，分为静态查找表和动态查找表，哈希表是一种动态查找表。

（1）静态查找表。对查找表经常要进行的两种操作如下。

① 查询某个“特定”的数据元素是否在查找表中。

② 检索某个“特定”的数据元素的各种属性。

通常将只进行这两种操作的查找表称为静态查找表。

（2）动态查找表。对查找表经常要进行的另外两种操作如下。

① 在查找表中插入一个数据元素。

② 从查找表中删除一个数据元素。

若在查找过程中还可能插入查找表中不存在的数据元素，或者从查找表中删除已存在的某个数据元素，则称相应的查找表为动态查找表。

（3）关键字。它是数据元素（或记录）的某个数据项的值，用它来识别（标识）这个数据元素（或记录）。主关键字是指能唯一标识一个数据元素的关键字；次关键字是指能标识多个数据元素的关键字。

（4）查找。根据给定的某个值，在查找表中确定是否存在一个其关键字等于给定值的记录或数据元素。若表中存在这样的一个记录，则称查找成功，此时或者给出整个记录的信息，或者指出记录在查找表中的位置；若表中不存在关键字等于给定值的记录，则称查找不成功，此时的查找结果用一个“空记录”或“空指针”表示。

（5）平均查找长度。对于查找算法来说，其基本操作是“将记录的关键字与给定值进行比较”。因此，通常以“其关键字和给定值进行过比较的记录个数的平均值”作为衡量查找算法好坏的依据。

为确定记录在查找表中的位置，需和给定值进行比较的关键字个数的期望值称为查找算法在查找成功时的平均查找长度。

对于含有 n 个记录的表，查找成功时的平均查找长度定义为

$$\mathrm{ASL}=\sum_{i=1}^{n}P_iC_i$$

其中，P_i 为对表中第 i 个记录进行查找的概率，且 $\sum_{i=1}^{n}P_i=1$，一般情况下，认为查找每个记录的概率是相等的，即 $P_i=1/n$；C_i 为找到表中其关键字与给定值相等的记录时（为第 i 个记录），和给定值已进行过比较的关键字个数。显然，C_i 随查找方法的不同而不同。

2．顺序查找

从表中的一端开始，逐个进行记录的关键字和给定值的比较，若找到一个记录的关键字与给定值相等，则查找成功；若整个表中的记录均比较过，仍未找到关键字等于给定值的记录，则查找失败。

顺序查找的方法对于顺序存储和链式存储方式的查找表都适用。

从顺序查找的过程可见，C_i 取决于所查记录在表中的位置。若需查找的记录正好是表中的第一个记录时，仅需比较一次；若查找成功时找到的是表中的最后一个记录，则需比较 n 次。一般情况下，$C_i=n-i+1$，因此在等概率情况下，顺序查找成功的平均查找长度为

$$\mathrm{ASL}_{ss}=\sum_{i=1}^{n}P_iC_i=\frac{1}{n}\sum_{i=1}^{n}(n-i+1)=\frac{n+1}{2}$$

也就是说，成功查找的平均比较次数约为表长的一半。若所查记录不在表中，则至少进行 n 次比较才能确定失败。

与其他查找方法相比，顺序查找方法在 n 值较大时，其平均查找长度较大，查找效率较低。但这种方法也有优点，那就是算法简单且适应面广，对查找表的结构没有要求，无论记录是否按关键字有序排列均可应用。

3．折半查找

折半查找也称为二分查找，其基本思想是：先令查找表中间位置记录的关键字和给定值比较，若相等，则查找成功；若不等，则缩小范围，直至新的查找区间中间位置记录的关键字等于给定值或者查找区间没有元素时（表明查找不成功）为止。

设查找表的元素存储在一维数组 $r[1..n]$中，那么在表中的元素已经按关键字递增（或递减）排序的情况下，进行折半查找的方法是：首先比较 key 值与表 r 中间位置（下标为 mid）的记录的关键字，若相等，则查找成功。若 key>r[mid].key，则说明待查记录只可能在后半个子

表 r[mid+1..n]中，下一步应在后半个子表中再进行折半查找；若 key<r[mid].key，说明待查记录只可能在前半个子表 r[1..mid–1]中，下一步应在 *r* 的前半个子表中进行折半查找，这样通过逐步缩小范围，直到查找成功或子表为空时失败为止。

【函数】设有一个整型数组中的元素是按非递减的方式排列的，在其中进行折半查找的算法如下：

```
int Bsearch(int r[],int low,int high,int key)
/*元素存储在数组 r[low..high]，用折半查找的方法在数组 r 中找值为 key 的元素*/
/*若找到则返回该元素的下标，否则返回–1*/
{   int mid;
    while(low <= high) {
        mid = (low+high)/2 ;
        if (key == r[mid]) return mid;
        else if (key < r[mid]) high = mid-1;
                else low = mid+1;
    }/*while*/
    return –1;
}/*Bsearch*/
```

折半查找的性能分析

折半查找的过程可以用一棵二叉树描述，方法是：以当前查找区间的中间位置上的记录作为根，左子表和右子表中的记录分别作为根的左子树和右子树上的节点，这样构造的二叉树称为折半查找判定树。例如，具有 11 个节点的折半查找判定树如图 3-35 所示，节点中的数字表示元素的序号。

从折半查找判定树可以看出，查找成功时，折半查找的过程恰好走了一条从根节点到被查节点的路径，关键字进行比较的次数即为被查找节点在树中的层数。因此，折半查找在查找成功时进行比较的关键字数最多不超过树的高度，而具有 n 个节点的判定树的高度为 $\lfloor \log_2 n \rfloor + 1$，所以折半查找在查找成功时和给定值进行比较的关键字个数至多为 $\lfloor \log_2 n \rfloor + 1$。

给判定树中所有节点的空指针域加上一个指向一个方型节点的指针，称这些方型节点为判定树的外部节点（与之相对，称那些圆形节点为内部节点），如图 3-36 所示。那么折半查找不成功的过程就是走了一条从根节点到外部节点的路径。和给定值进行比较的关键字个数等于该路径上内部节点个数。因此，折半查找在查找不成功时和给定值进行比较的关键字个数最多也不会超过 $\lfloor \log_2 n \rfloor + 1$。

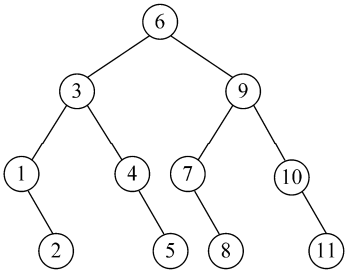


图 3-35　具有 11 个节点的折半查找判定树

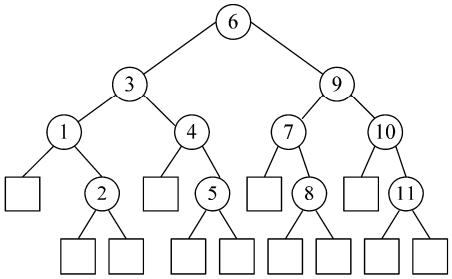


图 3-36　加上外部节点的判定树

那么折半查找的平均查找长度是多少呢？为了方便起见，不妨设节点总数为 $n=2^h-1$，则判定树是深度为 $h=\log_2(n+1)$ 的满二叉树。在等概率情况下，折半查找的平均查找长度为

$$\mathrm{ASL}_{bs}=\sum_{j=1}^{n}P_iC_i=\frac{1}{n}\sum_{j=1}^{n}j\times 2^{j-1}=\frac{n+1}{n}\log_2(n+1)-1$$

当 n 值较大时，$\mathrm{ASL}_{bs}\approx\log_2(n+1)-1$。

折半查找比顺序查找的效率要高，但它要求查找表进行顺序存储并且按关键字有序排列。因此，折半查找适用于表不易变动，且又经常进行查找的情况。

4．索引顺序查找

索引顺序查找又称分块查找，是对顺序查找方法的一种改进。

在分块查找过程中，首先将表分成若干块，每一块中关键字不一定有序，但块之间是有序的，即后一块中所有记录的关键字均大于前一个块中最大的关键字。此外，还建立了一个“索引表”，索引表按关键字有序，如图 3-37 所示。

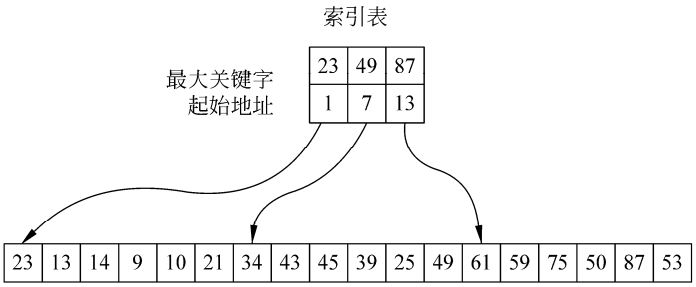


图 3-37　表及其索引表

因此，查找过程分为两步：第一步在索引表中确定待查记录所在的块；第二步在块内顺序

查找。

由于分块查找实际上是两次查找的过程，因此整个分块查找的平均查找长度应该是两次查找的平均查找长度（块内查找与索引查找）之和，所以分块查找的平均查找长度为 $ASL_{bs}=L_b+L_w$，其中 L_b 为查找索引表的平均查找长度，L_w 为块内查找时的平均查找长度。

进行分块查找时可将长度为 n 的表均匀地分成 b 块，每块含有 s 个记录，即 $b=\left\lceil\frac{n}{s}\right\rceil$。在等概率的情况下，块内查找的概率为 $\frac{1}{s}$，每块的查找概率为 $\frac{1}{b}$，若用顺序查找方法确定元素所在的块，则分块查找的平均查找长度为

$$\mathrm{ASL}_{bs}=L_b+L_w=\frac{1}{b}\sum_{j=1}^{b}j+\frac{1}{s}\sum_{i=1}^{s}i=\frac{b+1}{2}+\frac{s+1}{2}=\frac{1}{2}\left(\frac{n}{s}+s\right)+1$$

可见，其平均查找长度在这种条件下不仅与表长 n 有关，而且和每一块中的记录数 s 有关。可以证明，当 s 取 $\sqrt{n}$ 时，ASL_{bs} 取最小值 $\sqrt{n}+1$，这时的查找效率较顺序查找要好得多，但远不及折半查找。

5．树表查找

二叉查找树、B-树、红黑树等是常见的以树表方式组织的查找表，这里以二叉查找树为例简要介绍树表上的查找运算。二叉查找树是一种动态查找表，其特点是表结构本身是在查找过程中动态生成的，即对于给定值 key，若表中存在关键字等于 key 的记录，则查找成功返回，否则插入关键字等于 key 的记录。

根据定义，非空的二叉查找树中左子树上所有节点的关键字均小于根节点的关键字，右子树上所有节点的关键字均大于根节点的关键字，因此，可将二叉查找树看成是一个有序表，其查找过程与折半查找过程相似。

1）二叉查找树的查找过程

在二叉查找树上进行查找的过程：若二叉查找树非空，将给定值与根节点的关键字值相比较，若相等，则查找成功；若不等，则当根节点的关键字值大于给定值时，到根的左子树中进行查找，否则到根的右子树中进行查找。若找到，则查找过程是走了一条从树根到所找到节点的路径；否则，查找过程终止于一棵空树。

设二叉查找树以二叉链表为存储结构，节点的类型定义如下：

```
typedef struct Tnode
{
        int data;                          /*节点的键值*/
        struct Tnode *lchild,*rchild;      /*指向左、右子树的指针*/
```

```
}Tnode, *BiTree;
```

【算法】二叉查找树的查找算法。

```
Bitree SearchBST(BiTree root, int key, BiTree *father)
/*在 root 指向根的二叉查找树中查找键值为 key 的节点*/
/*若找到，则返回该节点的指针，否则返回 NULL*/
{    Bitree   p = root;     *father = NULL;
     while (p && p->data!=key)
     {
         *father = p;
         if (key < p->data) p = p->lchild;
         else p = p->rchild;
     }/*while*/
     return p;
}/*SearchBST*/
```

2）二叉查找树中插入节点的操作

二叉查找树是通过依次输入数据元素并把它们插入到二叉树的适当位置上构造起来的，即每读入一个元素，首先在二叉查找树中进行查找，若找到则不再插入，否则根据查找时得到的位置信息进行插入。其过程为：若二叉查找树为空，则为新元素创建节点并作为二叉查找树的根节点；若二叉查找树非空，则将新元素的值与根节点的值相比较，如果小于根节点的值，则继续在左子树中查找，否则在右子树中继续查找，直到某节点的值与新元素的值相等，或者到达空的子树为止，此时创建新元素的节点并替换该空的子树完成插入处理。设关键字序列为{46，25，54，13，29，91}，则对应的二叉查找树构造过程如图 3-38（a）~（g）所示。

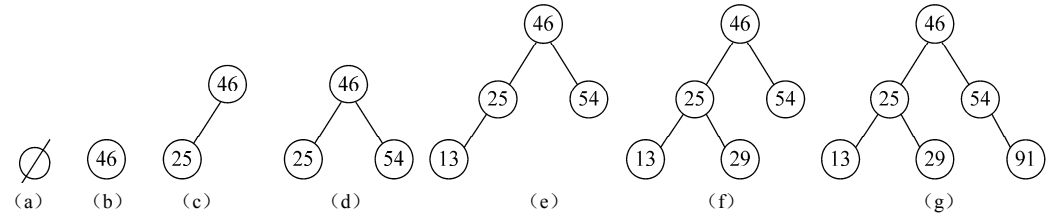


图 3-38　二叉查找树的构造过程

从上面的插入过程还可以看到，每次插入的新节点都是二叉查找树上新的叶子节点，因此在进行插入操作时，不必移动其他节点，仅需改动某个节点的指针域，由空变为非空即可。这就相当于在一个有序序列上插入一个记录而不需要移动其他记录。

另外，由于一棵二叉查找树的形态完全由输入序列决定，所以在输入序列已经有序的情况下，所构造的二叉查找树是一棵单枝树。从二叉查找树的查找过程可知，这种情况下的查找效

率与顺序查找的效率相当。

6．哈希查找

对于前面讨论的几种查找方法，由于记录的存储位置与其关键码之间不存在确定的关系，所以查找时都要通过一系列对关键字的比较，才能确定被查记录在表中的位置，即这类查找方法都建立在对关键字进行比较的基础之上。理想的情况是依据记录的关键码直接得到对应的存储位置，即要求记录的关键码与其存储位置之间存在一一对应关系，通过这个关系，能很快地由关键码找到记录。哈希表就是按这种思想组织的查找表。

1）哈希造表

根据设定的哈希函数 Hash(key)和处理冲突的方法，将一组关键字映射到一个有限的连续的地址集（区间）上，并以关键字在地址集中的“像”作为记录在表中的存储位置，这种表称为哈希表，这一映射过程称为哈希造表或散列，所得的存储位置称为哈希地址或散列地址。

在构造哈希表时，是以记录的关键字为自变量计算一个函数（称为哈希函数）来得到该记录的存储地址并存入元素，因此在哈希表中进行查找操作时，必须计算同一个哈希函数，首先得到待查记录的存储地址，然后到相应的存储单元去获得有关信息再判定查找是否成功。

对于某个哈希函数 Hash 和两个关键字 K_1 和 K_2，如果 $K_1 \neq K_2$ 而 Hash(K_1)=Hash(K_2)，则称为出现了冲突，对该哈希函数来说，K_1 和 K_2 则称为同义词。

一般情况下，只能尽可能地减少冲突而不能完全避免，所以在建造哈希表时不仅要设定一个“好”的哈希函数，而且要设定一种处理冲突的方法。

采用哈希法主要考虑的两个问题是哈希函数的构造和冲突的解决。

2）处理冲突

解决冲突就是为出现冲突的关键字找到另一个“空”的哈希地址。常见的处理冲突的方法有开放定址法、链地址法（拉链法）、再哈希法、建立公共溢出区法等，在处理冲突的过程中，可能得到一个地址序列，记为 $H_i(i=1, 2, \cdots, k)$。下面简要介绍开放定址法和链地址法。

（1）开放定址法。

$$H_i=(\text{Hash(key)}+d_i)\ \%\ m \quad i=1, 2, \cdots, k \quad (k \leqslant m-1)$$

其中，Hash(key)为哈希函数；m 为哈希表的表长；d_i 为增量序列。

常见的增量序列有如下三种。

- $d_i = 1, 2, 3, \cdots, m-1$，称为线性探测再散列。
- $d_i = 1^2, -1^2, 2^2, -2^2, 3^2, \cdots, \pm k^2$ （$k \leqslant m/2$），称为二次探测再散列。
- d_i=伪随机序列，称为随机探测再散列。

最简单的产生探测序列的方法是进行线性探测。也就是发生冲突时，顺序地到存储区的下一个单元进行探测。

例如，某记录的关键字为 key，哈希函数值 $H(key)=j$。若在哈希地址 j 发生了冲突（即此位置已存放了其他元素），则对哈希地址 $j+1$ 进行探测，若仍然有冲突，再对地址 $j+2$ 进行探测，以此类推，直到将元素存入哈希表。

【例 3.3】 设关键码序列为 47，34，19，12，52，38，33，57，63，21，哈希表表长为 13，哈希函数为 Hash(key)=key mod 11，用线性探测法解决冲突构造哈希表。

Hash(47) = 47 MOD 11 = 3，　Hash(34) = 34 MOD 11 = 1，

Hash(19) = 19 MOD 11 = 8，　Hash(12) = 12 MOD 11 = 1，

Hash(52) = 52 MOD 11 = 8，　Hash(38) = 38 MOD 11 = 5，

Hash(33) = 33 MOD 11 = 0，　Hash(57) = 57 MOD 11 = 2，

Hash(63) = 63 MOD 11 = 8，　Hash(21) = 21 MOD 11 = 10

使用线性探测法解决冲突构造哈希表的过程如下。

（a）开始时哈希表为空表。

哈希地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码													

（b）根据哈希函数，计算出关键码 47 的哈希地址为 3，在该单元处无冲突，因此插入 47。此后关键码 34 和 19 需要插入的哈希地址 1 和 8 处也没有冲突，因此在对应位置直接插入后如下所示。

哈希地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码		34		47					19				

（c）将关键码 12 存入哈希地址为 1 的单元时发生冲突，探测下一个单元（即哈希地址为 2 的单元），不再冲突，因此将 12 存入哈希地址为 2 的单元后如下所示。

哈希地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码		34	12	47					19				

（d）将关键码 52 存入哈希地址为 8 的单元时发生冲突，探测下一个单元（即哈希地址为 9 的单元），不再冲突，因此将 52 存入哈希地址为 9 的单元后如下所示。

哈希地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码		34	12	47					19	52			

（e）在哈希地址为 5 的单元存入关键码 38，没有冲突；在哈希地址为 0 的单元中存入关键码 33，没有冲突。因此将 38 和 33 先后存入哈希地址为 5 和 0 的单元后如下所示。

哈希地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码	33	34	12	47		38			19	52			

（f）在哈希地址为 2 的单元存入关键码 57 时发生冲突，探测下一个单元（即哈希地址为 3 的单元），仍然冲突，再探测哈希地址为 4 的单元，不再冲突，因此将 57 存入哈希地址为 4 的单元后如下所示。

哈希地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码	33	34	12	47	57	38			19	52			

（g）在哈希地址为 8 的单元存入关键码 63 时发生冲突，探测下一个单元（即哈希地址为 9 的单元），仍然冲突，再探测哈希地址为 10 的单元，不再冲突，因此将 63 存入哈希地址为 10 的单元后如下所示。

哈希地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码	33	34	12	47	57	38			19	52	63		

（h）在哈希地址为 10 的单元存入关键码 21 时发生冲突，用线性探测法解决冲突，算出哈希地址 11，不再冲突，因此将 21 存入哈希地址为 11 的单元后如下所示，此时得到最终的哈希表。

哈希地址	0	1	2	3	4	5	6	7	8	9	10	11	12
关键码	33	34	12	47	57	38			19	52	63	21	

线性探测法可能使第 i 个哈希地址的同义词存入第 i+1 个哈希地址，这样本应存入第 i+1 个哈希地址的元素变成了第 i+2 个哈希地址的同义词，……，因此，可能出现很多元素在相邻的哈希地址上“聚集”起来的现象，大大降低了查找效率。为此，可采用二次探测法或随机探测再散列法，以降低“聚集”现象。

（2）链地址法。

链地址法是一种经常使用且很有效的方法。它将具有相同哈希函数值的记录组织成一个链表，当链域的值为 NULL 时，表示已没有后继记录。

例如，哈希表表长为 11、哈希函数为 Hash(key)=key mod 11，对于关键码序列 47，34，13，12，52，38，33，27，3，使用链地址法构造的哈希表如图 3-39 所示。

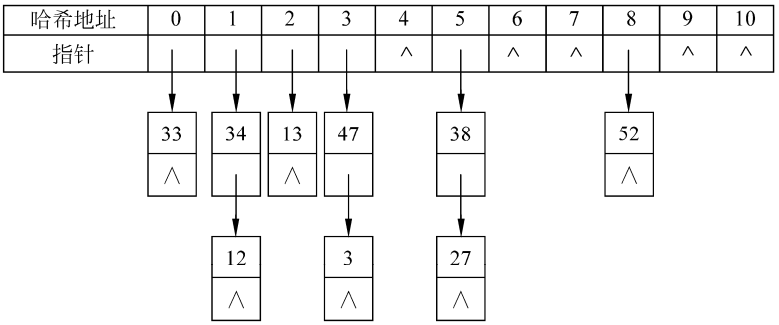


图 3-39　用链地址法解决冲突构造哈希表

3）哈希查找

在线性探测法解决冲突的方式下，进行哈希查找有两种可能：第一种情况是在某一位置上查到了关键字等于 key 的记录，查找成功；第二种情况是按探测序列查不到关键字为 key 的记录而又遇到了空单元，这时表明元素不在表中，表示查找失败。

在用链地址法解决冲突构造的哈希表中查找元素，就是根据哈希函数得到元素所在链表的头指针，然后在链表中进行顺序查找的过程。

3.4.4　递归算法

递归（Recursion）是一种描述和解决问题的基本方法，用来解决可归纳描述的问题，或者是可分解为结构自相似的问题。所谓结构自相似，是指构成问题的部分与问题本身在结构上相似。这类问题具有的特点：整个问题的解决可以分为两部分，第一部分是一些特殊或基本的情况，可直接解决；第二部分与原问题相似，可用类似的方法解决，但比原问题的规模小。

由于第二部分比整个问题的规模小，所以每次递归时第二部分的规模都在缩小，如果最终缩小为第一部分的情况则结束递归。因此，通过递归不断地分解问题，第一部分和第二部分的解密切配合，完成原问题的求解。

这类问题在数学中很常见，例如求 $n!$。

$$f(n)=n!=\begin{cases}1 & n=0\\ n\times f(n-1) & n>0\end{cases}$$

【算法】求 $n!$。

```
long fact(int n)
{    if (n == 1)   return 1;        /*第一部分，递归终止*/
     else   return n*fact(n-1);   /*第二部分，递归前进*/
}
```

在该式中，$f(n-1)$的计算与原问题 $f(n)$的计算相似，只是规模更小。

设一维数组 A 的元素 $A[k1]\sim A[k2]$中存放着整数，用递归方法求出它们中的最大者。显然，若 $k1=k2$，即数组中只有 1 个元素，则 $A[k1]$就是最大元素；若 $k1<k2$，则用类似的方法先求出 $A[k1+1]\sim A[k2]$中的最大者 m，然后令 m 与 $A[k1]$进行比较，二者中的最大者即为所求。

折半查找的过程也可用递归算法描述如下。

【算法】设有一个整型数组中的元素是按非递减的方式排列的，递归地进行折半查找。

```
int Bsearch_rec(int r[],int low,int high,int key)
/*元素存储在数组 r[low..high]，用折半查找的方法在数组 r 中找值为 key 的元素*/
/*若找到则返回该元素的下标，否则返回–1*/
```

```
{   int mid;
    if (low <= high) {
        mid = (low+high)/2 ;
        if (key == r[mid]) return mid;
        else if (key < r[mid]) return Bsearch_2(r, low, mid-1, key);
                else return Bsearch_2(r, mid+1, high, key);
                        }/*if*/
    return –1;
}/*Bsearch_rec*/
```

【算法】用递归方法求 $A[k1]$～$A[k2]$中的最大者，并作为函数值返回。

```
int maxint(int A[], int k1, int k2)     /*求 A[k1]～A[k2]中的最大者并返回*/
{    if(k1 == k2) return    A[k1];
      else{
            m = maxint(A,k1+1,k2);
            return (A[k1] > m) ? A[k1] : m;
          }
}/* maxint */
```

二叉树是用递归方式定义的，因此关于二叉树的运算可用递归算法描述。例如，二叉树的高度可定义为

$$\text{二叉树的高度} = \begin{cases} 0 & \text{空二叉树} \\ 1 + \text{MAX(左子树的高度，右子树的高度)} & \text{非空二叉树} \end{cases}$$

【算法】用递归方法求二叉树的高度。

```
int HeightBT(BiTree root)
{/*二叉树采用二叉链表存储，root 指向二叉树的根节点，求解并返回二叉树的高度*/
  if (root == NULL) return 0;                  /*空二叉树的高度为 0*/
  else
    {   LH = HeightBT(root->lchild);           /*求根节点的左子树高度*/
        RH = HeightBT(root->rchild);           /*求根节点的右子树高度*/
        if (LH > RH) return LH + 1;
        else return RH + 1;
    }
}/* HeightBT */
```

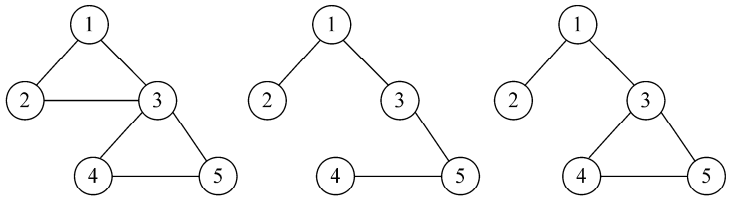
3.4.5　图的相关算法

1．求最小生成树算法

1）生成树的概念

设图 $G=(V, E)$是个连通图，如果其子图是一棵包含 G 的所有顶点的树，则该子图称为 G 的生成树(Spanning Tree)。

当从图 G 中任一顶点出发遍历该图时，会将边集 $E(G)$分为两个集合 $A(G)$和 $B(G)$。其中 $A(G)$是遍历时所经过的边的集合，$B(G)$是遍历时未经过的边的集合。显然，$G_1=(V, A)$是图 G 的子图，称子图 G_1 为连通图 G 的生成树。图 3-40 所示的是图及其生成树。



（a）图　　（b）生成树　　（c）非生成树

图 3-40　一个无向图的生成树

对于有 n 个顶点的连通图，至少有 n–1 条边，而生成树中恰好有 n–1 条边，所以连通图的生成树是该图的极小连通子图。若在图的生成树中任意加一条边，则必然形成回路。

图的生成树不是唯一的。从不同的顶点出发，选择不同的存储方式和遍历方式，可以得到不同的生成树。对于非连通图而言，每个连通分量中的顶点集和遍历时走过的边集一起构成若干棵生成树，把它们称为非连通图的生成树森林。

2）最小生成树

对于连通网来说，边是带权值的，生成树的各边也带权值，于是就把生成树各边的权值总和称为生成树的权，把权值最小的生成树称为最小生成树。求解最小生成树有许多实际的应用。普里姆（Prim）算法和克鲁斯卡尔（Kruskal）算法是两种常用的求最小生成树的算法。

（1）普里姆（Prim）算法思想。

假设 $N=(V, E)$是连通网，TE 是 N 上最小生成树中边的集合。算法从顶点集合 $U=\{u_0\}(u_0 \in V)$、边的集合 $TE=\{\}$开始，重复执行下述操作：在所有 $u \in U$，$v \in V-U$ 的边$(u, v) \in E$ 中找一条代价最小的边(u_0, v_0)，把这条边并入集合 TE，同时将 v_0 并入集合 U，直至 $U=V$ 时为止。此时 TE 中必有 n–1 条边，则 $T=(V, \{TE\})$为 N 的最小生成树。

由此可知，普里姆算法构造最小生成树的过程是以一个顶点集合 $U=\{u_0\}$作初态，不断寻找与 U 中顶点相邻且代价最小的边的另一个顶点，扩充 U 集合直至 $U=V$ 时为止。

用普里姆算法构造最小生成树的过程如图 3-41 所示。

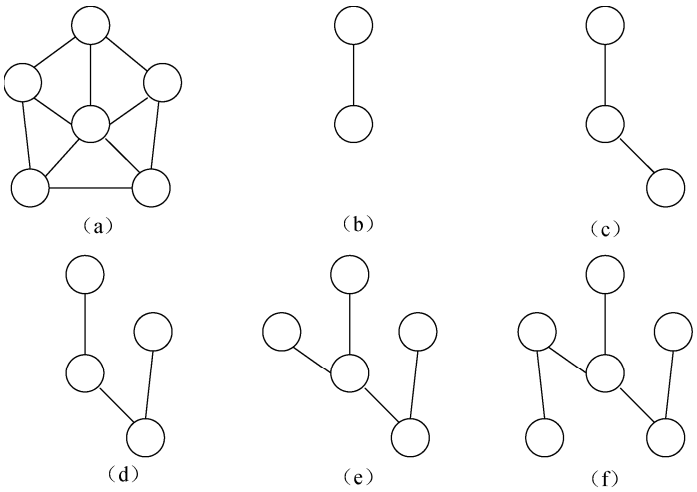


图 3-41　普里姆算法构造最小生成树的过程

（2）克鲁斯卡尔（Kruskal）算法思想。

克鲁斯卡尔求最小生成树的算法思想为：假设连通网 $N=(V, E)$，令最小生成树的初始状态为只有 n 个顶点而无边的非连通图 $T=(V, \{\})$，图中每个顶点自成一个连通分量。在 E 中选择代价最小的边，若该边依附的顶点落在 T 中不同的连通分量上，则将此边加入到 T 中，否则舍去此边而选择下一条代价最小的边。以此类推，直至 T 中所有顶点都在同一连通分量上为止。

用克鲁斯卡尔算法构造最小生成树的过程如图 3-42 所示。

克鲁斯卡尔算法的时间复杂度为 $O(e\log e)$，与图中的顶点数无关，因此该算法适合于求边稀疏的网的最小生成树。

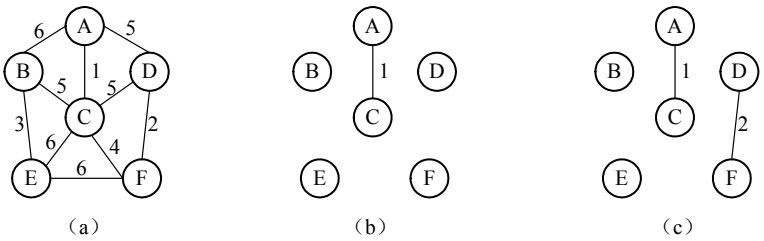
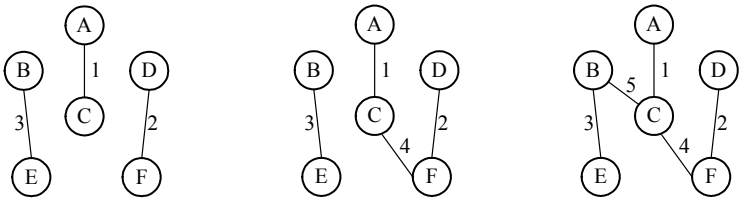


图 3-42　克鲁斯卡尔算法构造最小生成树的过程



（d）　（e）　（f）

图 3-42　克鲁斯卡尔算法构造最小生成树的过程（续）

2．拓扑排序

1）AOV 网

在工程领域，一个大的工程项目通常被划分为许多较小的子工程（称为活动），当这些子工程都完成时，整个工程也就完成了。若以顶点表示活动，用有向边表示活动之间的优先关系，则称这样的有向图为以顶点表示活动的网（Activity On Vertex network，AOV 网）。在有向网中，若从顶点 v_i 到顶点 v_j 有一条有向路径，则顶点 v_i 是 v_j 的前驱，顶点 v_j 是 v_i 的后继。若<v_i，v_j>是网中的一条弧，则顶点 v_i 是 v_j 的直接前驱，顶点 v_j 是 v_i 的直接后继。AOV 网中的弧表示了活动之间的优先关系，也可以说是一种活动进行时的制约关系。

在 AOV 网中不应出现有向环，若存在的话，则意味着某项活动必须以自身任务的完成为先决条件，显然这是荒谬的。因此，若要检测一个工程划分后是否可行，首先就应检查对应的 AOV 网是否存在回路。检测的方法是对其 AOV 网进行拓扑排序。

2）拓扑排序

拓扑排序是将 AOV 网中所有顶点排成一个线性序列的过程，并且该序列满足：若在 AOV 网中从顶点 v_i 到 v_j 有一条路径，则在该线性序列中，顶点 v_i 必然在顶点 v_j 之前。

一般情况下，假设 AOV 网代表一个工程计划，则 AOV 网的一个拓扑排序就是一个工程顺利完成的可行方案。对 AOV 网进行拓扑排序的方法如下。

（1）在 AOV 网中选择一个入度为零（没有前驱）的顶点且输出它。

（2）从网中删除该顶点以及与该顶点有关的所有边。

（3）重复上述两步，直至网中不存在入度为零的顶点为止。

按照上述步骤进行拓扑排序的过程如图 3-43 所示，得到的拓扑序列为 6,1,4,3,2,5。显然，对有向图进行拓扑排序所产生的拓扑序列有可能是多种。

执行的结果会有两种情况：一种是所有顶点已输出，此时整个拓扑排序完成，说明网中不存在回路；另一种是尚有未输出的顶点，剩余的顶点均有前驱顶点，表明网中存在回路，拓扑排序无法进行下去。

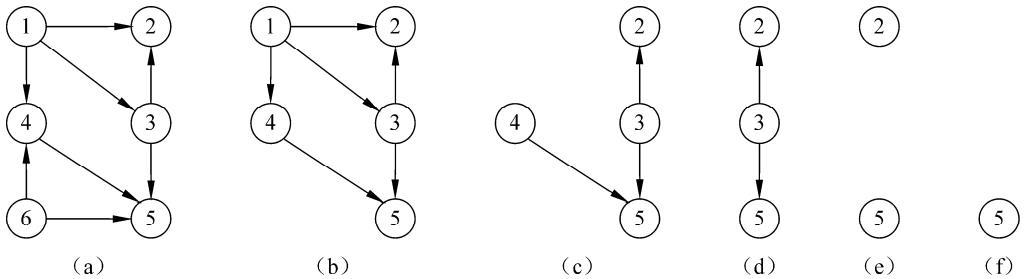


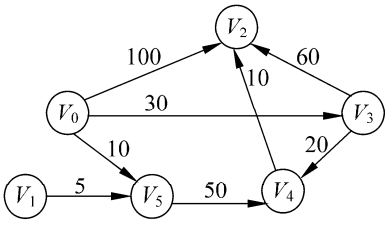
图 3-43　拓扑排序过程

3．求单源点的最短路径算法

所谓单源点最短路径，是指给定带权有向图 G 和源点 v_0，求从 v_0 到 G 中其余各顶点的最短路径。迪杰斯特拉（Dijkstra）提出了按路径长度递增的次序产生最短路径的算法，其思想是：把网中所有的顶点分成两个集合 S 和 T，S 集合的初态只包含顶点 v_0，T 集合的初态包含除 v_0 之外的所有顶点，凡以 v_0 为源点，已经确定了最短路径的终点并入 S 集合中，按各顶点与 v_0 间最短路径长度递增的次序，逐个把 T 集合中的顶点加入到 S 集合中去。

每次从 T 集合选出一个顶点 u 并使之并入集合 S 后（即 v_0 至 u 的最短路径已找出），从 v_0 到 T 集合中各顶点的路径有可能变得更短。例如，对于 T 集合中的某一个顶点 v_i 来说，其已知的最短路径可能变为（v_0, …, u, v_i），其中的…仅包含 S 中的顶点。对 T 集合中各顶点的路径进行考查并进行必要的修改后，再从中挑选出一个路径长度最小的顶点，从 T 集合中删除它，同时将其并入 S 集合。重复该过程，就能求出源点到其余各顶点的最短路径及路径长度。

在图 3-44 所示的有向网中，用迪杰斯特拉算法求解顶点 V_0 到达其余顶点的最短路径的过程如表 3-3 所示。



（a）有向网G

$$\begin{bmatrix} \infty & \infty & 100 & 30 & \infty & 10 \\ \infty & \infty & \infty & \infty & \infty & 5 \\ \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 60 & \infty & 20 & \infty \\ \infty & \infty & 10 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 50 & \infty \end{bmatrix}$$

（b）网G的邻接矩阵

图 3-44　有向网 G 及其邻接矩阵

表 3-3　迪杰斯特拉算法求解图 3-44（a）顶点 V_0 到 V_1、V_2、V_3、V_4、V_5 最短路径的过程

终　点	第 1 步	第 2 步	第 3 步	第 4 步	第 5 步
V_1	∞	∞	∞	∞	∞
V_2	100 (V_0, V_2)	100 (V_0, V_2)	90 (V_0, V_3, V_2)	60 (V_0, V_3, V_4, V_2)	
V_3	30 (V_0, V_3)	30 (V_0, V_3)			
V_4	∞	60 (V_0, V_5, V_4)	50 (V_0, V_3, V_4)		
V_5	10 (V_0, V_5)				
说明	从 V_0 到 V_1、V_2、V_3、V_4、V_5 的路径中，(V_0, V_5) 最短，将顶点 V_5 加入 S 集合，并且更新 V_0 到 V_4 的路径	从 V_0 到 V_1、V_2、V_3、V_4 的路径中，(V_0, V_3)最短，将顶点 V_3 加入 S 集合，并且更新 V_0 到 V_2、V_0 到 V_4 的路径	从 V_0 到 V_1、V_2、V_4 的路径中，(V_0, V_3, V_4)最短，将顶点 V_4 加入 S 集合，并且更新 V_0 到 V_2 的路径	从 V_0 到 V_1、V_2 的路径中，(V_0, V_3, V_4, V_2)最短，将顶点 V_2 加入 S 集合	V_0 到 V_1 无路径
集合 S	$\{V_0, V_5\}$	$\{V_0, V_5, V_3\}$	$\{V_0, V_5, V_3, V_4\}$	$\{V_0, V_5, V_3, V_4, V_2\}$	$\{V_0, V_5, V_3, V_4, V_2\}$