第 4 章：数据结构设计　　作者：希赛教育软考学院　来源：希赛网　2014年05月06日

考情分析

数据结构设计在软件设计师考试中的一个重点，它主要考查考生对数据结构部分的理解程度。对数据结构部分概念及应用掌握的好坏能直接反映出考生对于软件设计的基础掌握是否充分，以及对于算法的设计是否具备扎实的功底。因此，熟练掌握数据结构中的基本概念以及应用不仅有助于提高考生在考试中的得分能力，同时也能帮助考试为算法设计及编程打下一个良好的基础。

根据考试大纲，数据结构设计是每年必考的知识点，一般以C语言的形式进行描述。本章从基本的数据结构概念入手，着重描述了有关数据结构中栈，链表，二叉树和图等考试的主要知识点。

上一节　本书简介　下一节

第 4 章：数据结构设计　　作者：希赛教育软考学院　来源：希赛网　2014年05月06日

考试大纲要求分析

根据考试大纲，数据结构设计要求考试掌握如下知识点：

（1）掌握C语言标准函数库。从以往考试来看，C语言的标准函数库是需要考生们注意掌握的，特别是一些常用的标准库函数，比如字符串中函数和一些常用的数学函数。

（2）能熟练掌握数据结构中的一些重要概念及使用C语言实现。数据结构中的算法通常是由C语言代码来实现的，而常用的数据结构有栈，树，图等概念。从考试角度出发，考生一定要理解这些概念并熟悉其性质。

（3）掌握数据结构设计和实现：线性表、查找表、树、图的顺序存储结构和链表存储结构的设计和实现。

在数据结构中，栈，队列，链表，二叉树和图的存储方式基本上有两种，一种是顺序存储，一种是链式存储。而在近5年的考试中，考试的重点放在的链式存储结构的考查上，这要求考生加深对链式存储结构的了解。

上一节　本书简介　下一节

第 4 章：数据结构设计　　作者：希赛教育软考学院　来源：希赛网　2014年05月06日

命题特点与趋势分析

近5年来考查了数据结构设计部分的知识点是栈，链表，二叉树，拓扑排序，在近5年的8次考试中，有关树的知识点占了2次，图形处理考了1次，栈考查了2次，链表，二叉树，拓扑各考查了1

次。从近几次命题的方向来看，图和栈相继被考查到了，估计在以后的考试中，会加强对二叉树和链式存储的有关操作的考查。望广大考试在备战的时候注意。

第 4 章：数据结构设计　作者：希赛教育软考学院　来源：希赛网　2014年05月06日

考点精讲

在数据结构的考查中，重点考察有关链表，栈，队列，二叉树，图等基本概念以及存储方式。本节就从以上这几个部分开始讲解。

第 4 章：数据结构设计　作者：希赛教育软考学院　来源：希赛网　2014年05月06日

链表

线性结构是一种基本的数据结构，主要用于对客观世界中具有单一的前驱和后继的数据关系进行描述。线性结构的特点是数据元素之间呈现一种线性关系，即元素“一个接一个地排列”。线性表是一种最基本，最常用的线性结构。链表是一种特殊的线性表，为了更好的了解链表，我们先来简单介绍下线性表。

1．线性表的定义

一个线性表是n（n>=0）个元素的有限序列，通常表示为（a_1,a_2,…,a_n）。非空线性表的特点如下。

（1）存在唯一的一个被称作“第一个元素”的元素。

（2）存在唯一的一个被称作“最后一个元素”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

线性表的存储结构分为顺序存储和链式存储。本节重点讲解链式存储。

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4-1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。

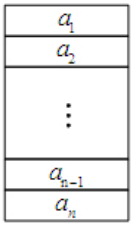
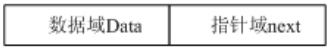


图4-1 线性表元素的顺序存储

线性表的链式存储与顺序存储不一样，它不要去用一块连续的存储单元来存放数据元素，它一般分配一个节点来存放一个元素，基本的节点结构如下所示：



其中数据域用来存放数据，而指针与用来存放下一个节点的地址，采用链式存储结构存储的线性表就是链表，它具有如下两个特点：

（1）数据元素的存储空间不一定连续。线性表的链式存储使用一组任意的存储单元来存储线性表的数据元素，不同数据元素的存储单元之间可以是连续的，也可以是不连续的。因此，线性表中的元素与其直接前驱和直接后继之间仅存在逻辑上的先后次序，在物理存储上并无前后关联。

（2）采用结点存储数据元素。在顺序表中元素的寻址可以通过数组来实现，但是链表中由于逻辑上相关联的元素的物理地址之间没有直接关联，因此，每个存储单元除了存储数据元素本身的信息外，还必须额外存储与其相关联的元素的物理地址，一般称前者为数据域，后者为指针域。指针域中存储的信息又称为指针或链，这个包含了数据域和指针域的存储单元就称为(链表的)结点，链表中每个结点都唯一对应了线性表中的一个元素。

根据链表指针域个数的不同，链表又可以分为单链表和双链表。下面我们分别来描述这两类链表。

3．单链表

节点之间通过指针域构成一个链表，若节点中只有一个指针域，则称为线性链表（或单链表），如图4-2所示。

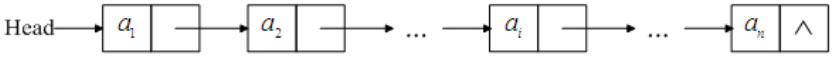


图4-2 线性表元素的单链表存储

设线性表中的元素是整型，则单链表节点类型的定义为：

```
typedef struct node{
    int data;        /*节点的数据域，此处假设为整型*/
    struct node *link;/*节点的指针域*/
}NODE，*LinkList;
```

在链式存储结构中，只需要一个指针（称为头指针，如图8-2中的head）指向第一个节点，就可以顺序访问到表中的任意一个元素。

在单链表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。

（1）插入操作

在单链表中，若在p所指节点后插入新元素节点（s所指节点，已经生成），如图4-3所示。其基本步骤如下：

```
s->link = p->link;
p->link = s;
```

即先将p所指节点的后继节点指针赋给s所指节点的指针域，然后将p所指节点的指针域修改为指

向s所指节点。

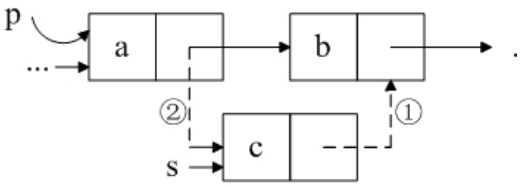


图4-3 单链表中插入节点

【函数代码】

```
int Insert_List( LinkList L,int k,int newelem)/*L为带头节点单链表的头指针*/
 /*该元素newelem插入表中的第k个元素之前，若成功则返回0，否则返回-1*/
 /*该插入操作等同于将元素newelem插入在第k-1个元素之后*/
{  LinkList p,s;              /*p、s为临时指针*/
   if( k= =1)p = L;           /*元素newelem插入在第1个元素之前时*/
   else  p = Find_List( L,k-1); /*查找表中的第k-1个元素并令p指向该元素节点*/
   if( !p||->link)return -1;/*表中不存在第k个元素*/
   s =( NODE *)malloc( sizeof( NODE));  /*创建新元素的节点空间*/
   if( !s)return -1;
   s->data = newelem;
   s->link = p->link;p->link = s;        /*元素newelem插入第k-1个元素之后*/
   return 0;
}/*Insert_List*/
```

（2）删除操作

即先将p所指节点的后继节点指针赋给s所指节点的指针域，然后将p所指节点的指针域修改为指向s所指节点。在单链表中删除p所指节点的后继节点时（过程如图4-4所示），步骤如下：

□q = p->link;

□p->link = p->link->link;

□free(q);

即先令临时指针q指向待删除的节点，然后修改p所指节点的指针域为指向p所指节点的后继的后继节点，从而将元素b所在的节点从链表中摘除，最后释放q所指节点的空间。

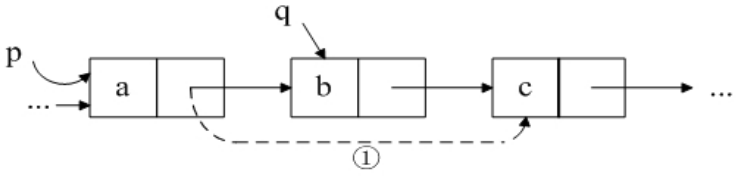


图4-4 单链表中删除节点

【函数代码】

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*/
 /*删除第k个元素，相当于令第k-1个元素节点的指针域指向第k+1个元素所在节点*/
{  LinkList p,q;              /*p、q为临时指针*/
   if(k = = 1)p = L;          /*若删除的是第一个元素节点*/
```

```
    else  p = Find_List(L，k-1);  /*查找表中的第k-1个元素并令p指向该元素节点*/
    if(!p||!p->link)return -1;     /*表中不存在第k个元素*/
    q = p->link;                          /*令q指向第k个元素节点*/
    p->link = q->link; free(q);          /*删除节点*/
    return 0;
    }/*Delete_List*/
```

4．双链表

与单链表一样，双链表也是通过指针域来连接的链表，其不同的是双链表节点中有两个指针域，其结构如图4-5所示。

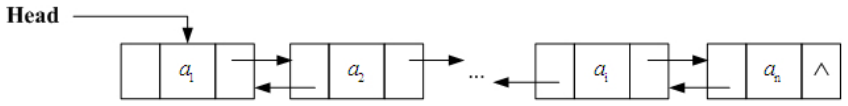


图4-5 线性表元素的双链表存储

设线性表中的元素是整型，则双链表节点类型的定义为：

```
typedef struct node{
    int data;         /*节点的数据域，此处假设为整型*/
    struct node *front;/*节点的左指针域*/
    struct node *next;/*节点的右指针域*/
}DouNODE，*DouLinkList;
```

（1）双链表的插入

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节点*s时指针的变化情况如图4-6所示，其操作过程可表示为：

```
  s->front = p;
  s->next = p->next;
  p->next ->front = s;
  p->next = s;
```

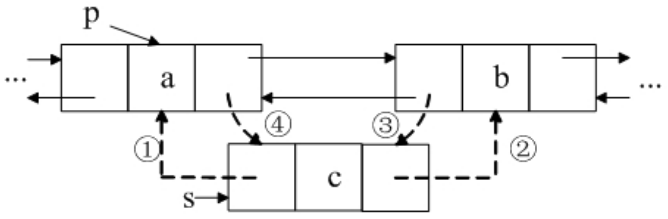


图4-6 双向链表中插入节点

【函数代码】

```
int Insert_List(DouLinkList L, int k, int newelem)
{
//L为指向双向链表头结点的指针，并且不为NULL，k为1时表示插入首元素
DouLinkList p = L, s;
int j = k;
if ( k<1) return 0;//k值不合法，线性表中不存在第k个元素
if (!p) return 0;// k值不合法，线性表中不存在第k个元素
if ( ( s = ( DouLinkList )malloc( sizeof( DouNODE ) ) ) = = NULL ) return 0;
```

```
s->data = newelem;// 赋值新结点数据域
s->front = p;// 赋值新结点前驱指针域
s->next = p->next;// 赋值新结点后继指针域
p->next->front = s;            // 赋值p结点前驱指针域
p->next  =  s;// 赋值p结点后继指针域
return 1;
}
```

（2）双链表删除

在双向链表中删除节点时指针的变化情况如图4-7所示，其操作过程可表示为：

```
 p->front->next = p->next;
 p->next->front = p->front;
```

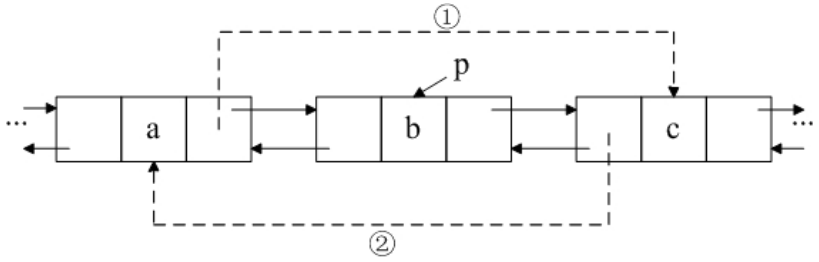


图4-7 双向链表中删除节点

【函数代码】

```
 int ListDelete( DouLinkList L, int k )
{
//L为指向双向链表头结点的指针，并且不为NULL，i为1时表示删除首元素
DouLinkList p = L,s ;
int j = k;
if ( k<1 ) return 0;
if ( !p||!( p->next ) ) return 0;        //p不为空而且p的后继不为空
s = p;          //s指向待删除的结点
p->front->next = p->next;//更新后继指针域
p->next->front = p->front;    //更新前驱指针域
free( s )//释放结点所占用的存储空间
return 1;
}
```


上一节　　本书简介　　下一节

栈

栈本质上是一种线性表，是只能在表的一端进行插入和删除操作的线性表。称表中允许插入和

删除的一端为栈顶（top），不允许插入和删除的一端称为栈底（botton）。其中栈底一直不变，而栈顶随着栈内元素的变化而变化，在顺序栈中栈顶永远指向下一个要写入的结点。

入栈操作是指在栈顶插入元素的操作，出栈操作是指在栈顶删除元素的操作。栈是一种后进先出（LIFO）的数据结构。

1．栈的存储结构

栈是一种特殊的线性表，它也和线性表一样，有顺序存储结构和链式存储结构，下面分别来讲解这两种存储结构的栈。

（1）顺序存储的栈。

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称为顺序栈。在该存储方式下，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。

（2）链式存储栈。

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4-8所示。

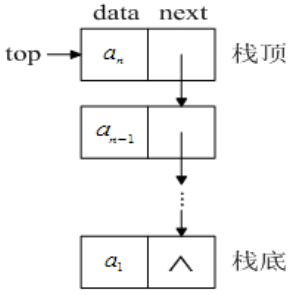


图4-8 链栈示意图

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)定义为一个返回栈顶元素值的函数。

（5）读栈顶元素Top(S)：返回栈顶元素的值，但不修改栈顶指针。

上一节　本书简介　下一节

第 4 章：数据结构设计　作者：希赛教育软考学院　来源：希赛网　2014年05月06日

队列

队列本质上也是一种线性表，标准队列只能在一端插入数据元素，在另一端删除数据元素，只能删除元素的一端称为队列头或队头（front），只能插入元素的一端称为队列尾或队尾（rear）。

向队列尾插入元素的操作称为入队列，从队列头删除元素的操作称为出队列。先入队列的数据也要先出队列，即队列是一种先进先出（FIFO）的数据结构，因此队列并不能更改数据串的次序。

1．队列的存储结构

队列也是一种特殊的线性表，它也和线性表一样，有顺序存储结构和链式存储结构，下面分别来讲解这两种存储结构的队列。

（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。由于队列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。

下面设顺序队列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4-9所示。

在顺序队列中，为了降低运算的复杂度，元素入队时只修改队尾指针，元素出队时只修改头指针。由于顺序队列的存储空间是提前设定的，所以队尾指针会有一个上限值，当队尾指针达到该上限时，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，可通过整除取余运算将顺序队列假想成一个环状结构，如图4-10所示，称之为循环队列。

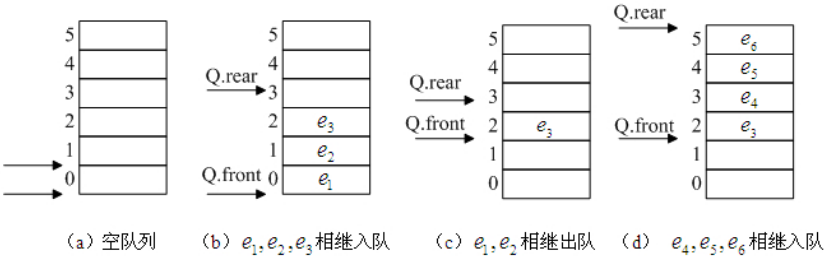


图4-9 队列的头、尾指针与队列中元素之间的关系

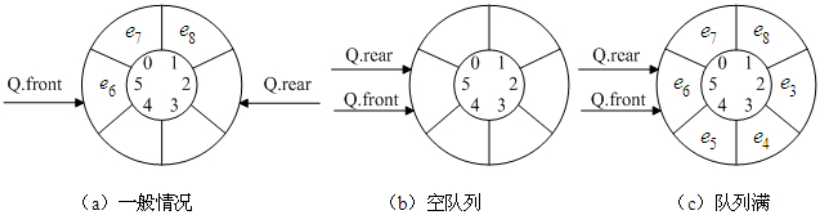


图4-10 循环队列的头、尾指针示意图

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4-11（a）所示。

元素入队时，修改队尾指针Q.rear =（Q.rear+1)% MAXSIZE，如图4-11（b）所示。

元素出队时，修改队头指针Q.front =（Q.front-1)% MAXSIZE，如图4-11（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，则有Q.rear = = Q.front，如图4-11（d）所示：若队列满，则Q.rear = = Q.front，如图4-11（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之间的关系无法判定队列的状态。为了区别队空和队满的情况，可采用以下两种处理方式：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；其二是牺牲一个存储单元，约定以“队列的尾指针所指位置的下一个位置是队头指针”表示队列满，如图4-9（f）所示，而头、尾指针的值相同时表示队列为空。

设队列中的元素类型为整型，则循环队列的类型定义为：

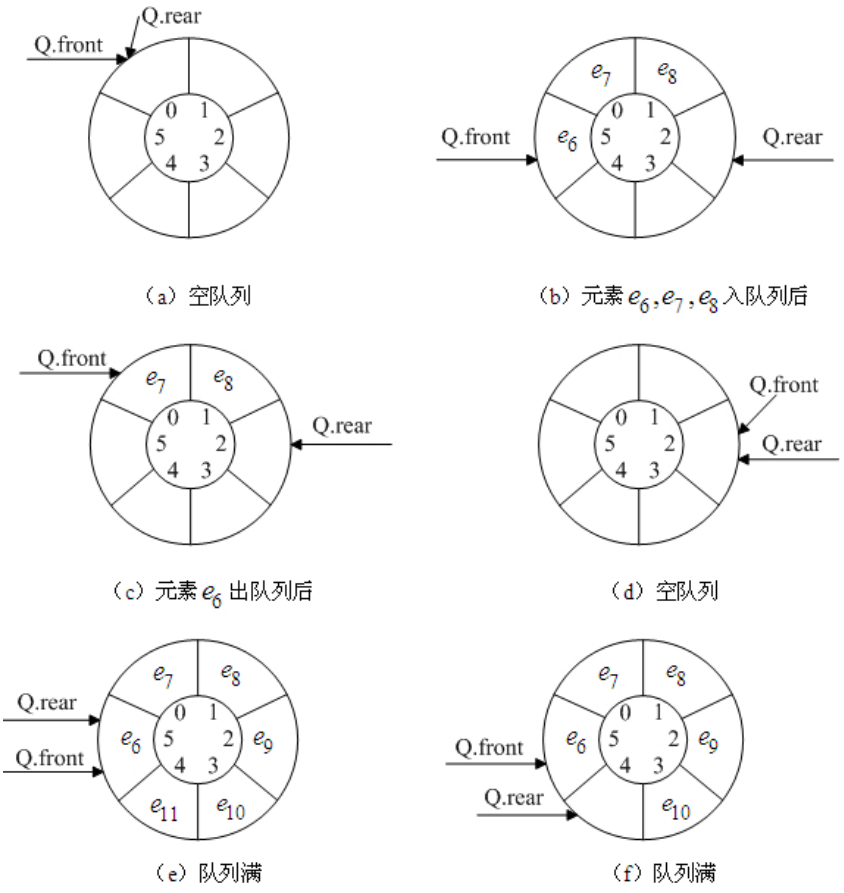


图4-11 循环队列的头、尾指针示意图

```
#define  MAXQSIZE  100
typedef  struct
{
int  *base;        /*循环队列的存储空间*/
int  front;        /*指示队头元素*/
int  rear;         /*指示队尾元素*/
}SqQueue;
```

创建一个空的循环队列

【函数代码】

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

元素入循环队列

【函数代码】

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

队列

元素出循环队列

【函数代码】

```
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0；否则返回-1*/
{  if( Q->rear = = Q->front )return -1;
   *e=Q->base[Q->front];
   Q->front = ( Q->front+1 )% MAXQSIZE;
   return 0;
}/*DeQueue*/
```

（2）队列的链式存储。

队列的链式存储也称为链队列。这里为了便于操作，给链队列添加一个头节点，并令头指针指向头节点。因此，队列为空的判定条件是：头指针和尾指针的值相同，且指向头节点。队列的链式存储结构如图4-12所示。

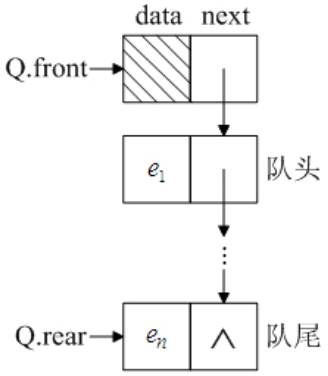


图4-12 链队列示意图

循环队列的头、尾指针示意图

（1）初始化队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FrontQue(Q)：返回队头元素的值，但不更新队头指针。

上一节 本书简介 下一节

二叉树

树型结构是一种非常重要的非线性结构。其中以二叉树最为常用。下面我们就简单介绍下二叉树的定义、性质、存储及应用。

1．二叉树的定义

二叉树是n（n> = 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且分别称为左、右子树的二叉树所组成。

2．满二叉树

在一棵二叉树中，如果所有分支结点都有左孩子和右孩子，并且叶子结点都集中在二叉树的最下一层，这样的二叉树称为满二叉树。

可以对二叉树结点进行连续编号，令树根的编号为1，并按照层数从小到大、同一层从左到右的次序进行，如图4-13(a)所示为编号后的满二叉树。

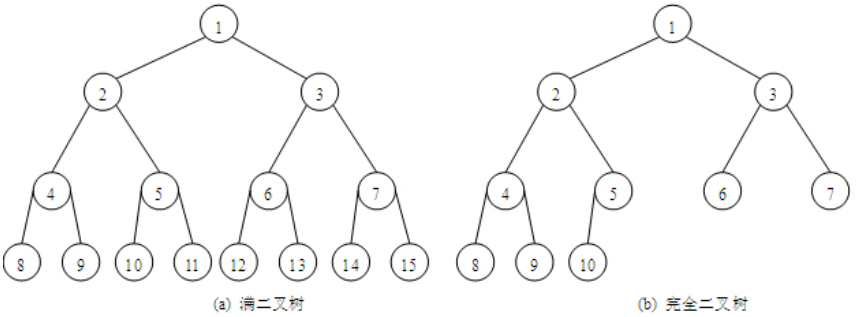


图4-13 满二叉树和完全二叉树

3．完全二叉树

若二叉树中最多只有最下面两层的结点度数可以小于2，并且最下面一层的叶子结点都依次排列在该层最左边的位置上，则这样的二叉树称为完全二叉树。

完全二叉树还可以这样形象地定义：一棵有n个结点的二叉树进行编号，如果编号后该二叉树的结点与满二叉树中编号为1～n的结点一一对应，那么称该二叉树为完全二叉树。图4-13(b)中的完全二叉树的各个结点恰好和图4-13(a)中满二叉树编号为1～10的结点一一对应。

满二叉树与完全二叉树的区别体现在以下2个方面：

（1）在一棵深度为h的完全二叉树中，前h-1层中的结点都是满的，且第 h 层的结点都集中在左边。

（2）满二叉树本身是完全二叉树的一种特例，高度相同的完全二叉树与满二叉树对应位置结点的编号也相同。

4．二叉树的性质

二叉树有很多非常重要的性质，下面我们将给出二叉树的一些常用性质。

性质1：在非空二叉树的第i层上至多有2^{i-1}个结点（i≥1）。

性质2：深度为k的二叉树中至多含有2^k-1个结点（k≥1）。

性质3：满二叉树第k层的结点个数比其1～k-1层所有的结点个数多一个（k≥1，设第0层有0个结点）。

性质4：对任何一棵二叉树而言，若叶子结点（度为0）个数为n0，度为2的结点数为n_2，则有：n_0–n_2 = 1。

性质5：具有n个结点的完全二叉树的深度为 $\lfloor \log_2 n \rfloor + 1$。（$\lfloor x \rfloor$表示不大于x的最大整数）。

性质6：如果对一棵有n个结点的完全二叉树（其深度为$\lfloor \log_2 n \rfloor + 1$）的结点按层序（从根结点开始，按照层数从小到大、同一层从左到右的次序）从1起开始编号，则对任一编号为i的结点（1≤i≤n），有：

（1）若i = 1，则结点i是二叉树的根，无双亲。

（2）若i＞1，则其双亲编号为$\lfloor i/2 \rfloor$。

（3）若2*i＞n，则编号为i的结点没有左孩子，否则其左孩子结点的编号是2i。

（4）若2*i+1＞n，则编号为i的结点没有右孩子，否则其右孩子结点的编号是2i+1。

5．二叉树的存储

二叉树的存储也可以采用顺序存储和链式存储两种结构。其顺序存储结构就是用一组地址连续的存储单元来存放二叉树的数据元素。其存储规则如下：

（1）将二叉树补齐到完全二叉树，并将原二叉树中不存在的结点（即补齐的结点）的数据置为某个特殊值。

（2）对此完全二叉树进行从上到下、从左到右的层次编号，其中根结点编号为1，其余结点依此类推。

（3）将此完全二叉树的结点存储到一维数组中，为了便于计算，从数组下标为1的元素开始存储，则编号为i的结点信息存储在下标为i的数组元素处。

如图4-14所示为二叉树的顺序存储示意图，其中使用“NO”表示结点不存在。

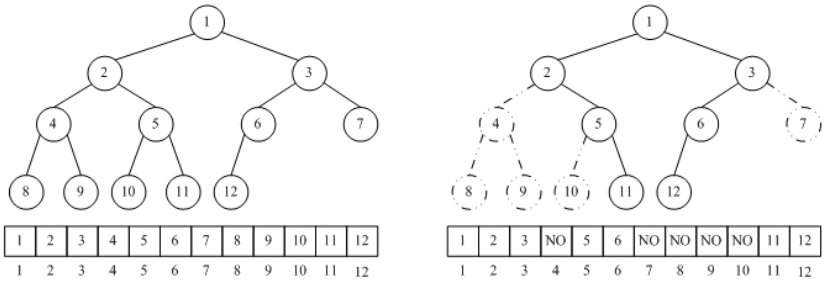


图4-14 二叉树顺序存储示意图

在二叉树的顺序存储中，两个物理地址相连的结点只表明其层次遍历顺序相连，并不能表明其血缘关系。事实上，如果结点data[i]存在左、右孩子，则其左、右孩子分别存储在data[2*i]和data[2*i+1]处。

顺序结构存储适合于完全二叉树，但为了避免存储一般二叉树时造成大量的空间浪费，一般采用链式存储二叉树。如图4-15所示为二叉链表结构示意图。

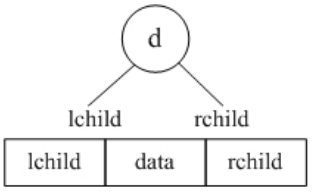


图4-15 二叉树的链式存储结构图

图4-16（b）和为图4-16（a）中二叉树的二叉链表描述树。

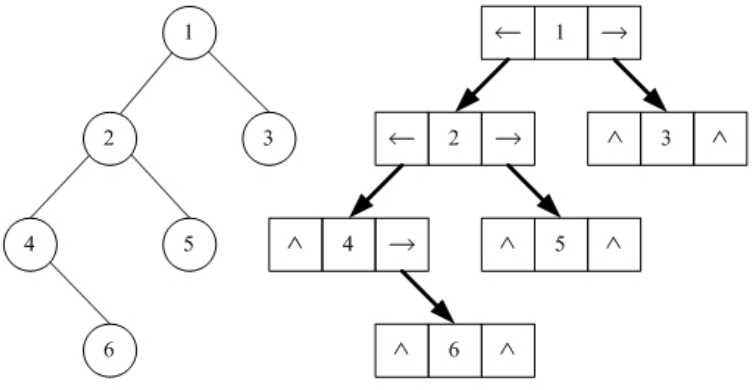


图4-16 二叉链表描述树

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;
struct BiTnode *lchild, *rchild;
}BiTnode, *BiTree;
```

在不同的存储结构中，实现二叉树的运算方法也不同，具体应采用什么存储结构，除考虑二叉树的形态外还应考虑需要进行的运算特点。

6．二叉树的遍历

二叉树的遍历是指按照某种顺序访问二叉树上所有结点，并且每个结点恰好只被访问一次。二叉树的变量存在先序遍历、中序遍历和后序遍历三种方式。

（1）先序遍历

如果二叉树为空，则直接返回。否则首先访问根结点，然后按先序遍历根结点的左子树，最后再按先序遍历根结点的右子树。

【函数代码】

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

（2）中序遍历

如果二叉树为空，则直接返回。否则首先按中序遍历根结点的左子树，然后访问根结点，最后再按中序遍历根结点的右子树。

【函数代码】

```
void InOrder( BiTree root )
{
    if( root != NULL ){
            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*/
```

（3）后序遍历

如果二叉树为空，则直接返回。否则首先按后序遍历根结点的左子树，然后按后序遍历根结点的右子树，最后再访问根结点。

【函数代码】

```
void PostOrder( BiTree root )
{
    if( root ！=NULL ){
        PostOrder( root->lchild );    /*后序遍历根节点的左子树*/
        PostOrder( root->rchild );    /*后序遍历根节点的右子树*/
        printf( “%d” ，root->data );  /*访问根节点*/
}/*if*/
}/*PostOrder*/
```

7．线索二叉树

在n个结点的二叉树链式存储中存在n+1个空指针，造成了巨大的空间浪费，为了充分利用存储资源，可以将这些空链域存放指向结点的直接前驱或直接后继的指针，这种空链域就称为“线索”，含有线索的二叉树就是“线索二叉树”。

二叉树的遍历实质上是对一个非线性结构进行线性化的过程，它使得每个节点（除第一个和最后一个外）在这些线性序列中有且仅有一个直接前驱和直接后继。但在二叉链表存储结构中，只能找到一个节点的左、右孩子，而不能直接得到节点在任一遍历序列中的前驱和后继，这些信息只有在遍历的动态过程中才能得到，因此，引入线索二叉树来保存这些动态过程得到的信息。

若n个节点的二叉树采用二叉链表作存储结构，则链表中必然有n+1个空指针域，可以利用这些空指针域来存放节点的前驱和后继信息。线索链表的节点结构如下所示。

ltag	lchild	data	rchild	rtag

其中：

$$ltag=\begin{cases}0 & lchild\ 域指示节点的左孩子\\ 1 & lchild\ 域指示节点的直接前驱\end{cases}$$

$$rtag=\begin{cases}0 & rchild\ 域指示节点的右孩子\\ 1 & rchild\ 域指示节点的直接后继\end{cases}$$

若二叉树的二叉链表采用以上所示的节点结构，则相应的链表称为线索链表，其中指向节点前驱、后继的指针称为线索。加上线索的二叉树称为线索二叉树。

二叉树常见的遍历有前序、中序和后序三种方法，不同的遍历方法将产生不同的序列，因此同一结点在不同的序列中有不同的前驱和后继，线索的取值随着遍历方式的不同而不同，故线索树可分为前序线索二叉树、中序线索二叉树和后序线索二叉树三种。把二叉树以某种方式遍历使其变成线索二叉树的过程就叫做“线索化”。如图4-14（b）~（d）分别为图4-14（a）中的树的前序、中序和后序线索二叉树。

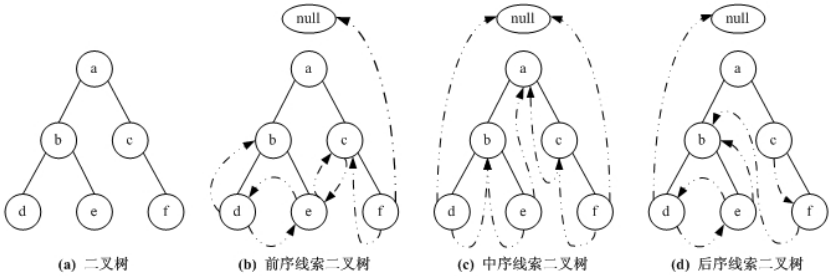


图4-17 线索二叉树

从图中发现线索化的过程，其实就是将指针域赋值为结点的直接前驱或后继地址的过程。

8．哈夫曼树

从树中一个结点到另一个结点之间的分支构成这两个结点之间的“路径”，路径上的分支数目就称“路径长度”。一般情况下，“树的路径长度” 是指从树的根结点到树中其余每个结点的路径长度之和。在不考虑带权路径的情况下，在相同结点数构成的二叉树中，完全二叉树的路径长度最短。

在带权树中，“结点的带权路径长度”定义为从树根到该结点之间的路径长度与该结点上权的乘积。假设树上有n个叶子结点，且每个叶子结点的权值为W_k（$1 \le k \le n$），到根结点的路径长度为l_k（$1 \le k \le n$），则定义“树的带权路径长度（WPL）”为树中所有叶子结点的带权路径长度之和，即：

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

假设三个叶子结点a、b和c，其权值分别为3、4、5，如图4-18是其三种二叉树组织形式，那么这三棵带权树的带权路径长度分别为：

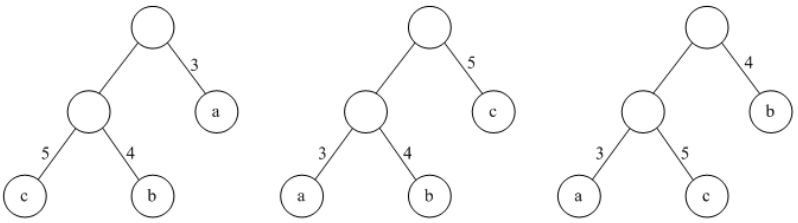


图4-18 带权二叉树

图4-18（a）：WPL = 3×1 + 4×2 + 5×2 = 22。

图4-18（b）：WPL = 3×2 + 4×2 + 5×1 = 19。

图4-18（c）：WPL = 3×2 + 5×2 + 4×1 = 20。

很明显，图4-18（b）的带权路径长度最小，事实上，这也是a、b、c三个叶子结点构成的带权路径长度最小的树。

在由n个权值{w_1,w_2，…，w_n}构造的一棵具有n个叶子结点的二叉树中，每个叶子结点的权值也为w_i，那么带权路径长度WPL最小的二叉树称为最优二叉树或哈夫曼树。

由n个权值{w_1,w_2，…，w_n}构造一棵具有n个叶子结点的哈夫曼树的算法如下：

（1）构造n个只有根结点的二叉树集合F={T_1,T_2，…，T_n}，其中每棵二叉树 的根结点带权为w_i（$1 \le k \le n$）；

（2）在集合F中选取两棵根结点的权值最小的树作为左右子树构造一棵新的二叉树，令新二叉树根结点的权值为其左、右子树上根结点的权值之和；

（3）在F中删除这两棵树，同时将新得到的二叉树加入到F中；

（4）重复第（2）步和第（3）步，直到F只含有一棵树为止，这棵树便是哈夫曼树。

例如权值{2，3，4，8}的最优二叉树构造过程如图4-19所示。

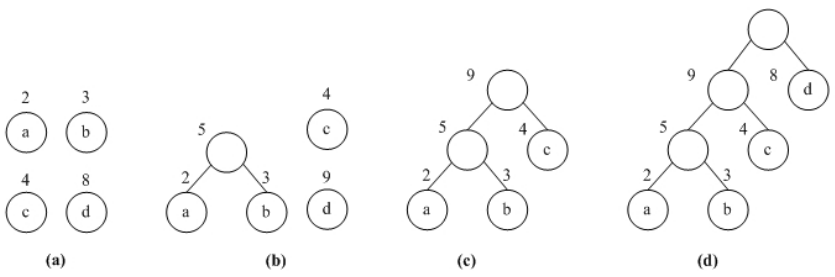


图4-19 最优二叉树构建过程

其中图4-19（d）的带权路径长度为：WPL = 2×3 + 3×3 + 4×2 + 8×1 = 31。

9．树的存储

树的存储结构多种多样，但最常见的存储结构有三种，分别是双亲存储结构、孩子链存储结构和孩子兄弟链存储结构等。

（1）双亲存储结构

其实这是一种静态链表的存储方法，它使用一组连续的地址空间存储树的结点，同时在每个结点中增加一个指针域存储双亲结点的位置（即数组下标）。图4-20（a）中的树采用双亲存储结构如图4-20（b）所示，其中定义根结点的指针域取值为“-1”，代表根结点没有双亲结点。

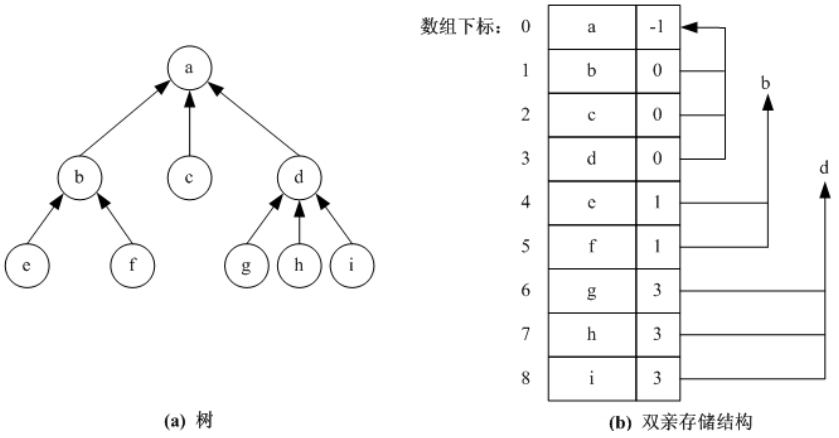


图4-20 树的双亲存储结构示例图

（2）孩子的存储结构

为了既避免浪费空间，也使编程相对容易些，可以采用链表结构存储结点及其孩子信息，即采用线性链表描述线性表（结点，孩子结点1，孩子结点2，…，孩子结点n），那么一个n个结点的树就具有n个线性链表，再把这n个链表的头结点串成一个顺序链表即可。

图4-21（a）中树的链式孩子链结构存储实例如图4-21（b）所示，其结点的指针域中增加了双亲结点的地址，如此就拥有了双亲存储和孩子链存储等两种存储结构的优势，极大方便了根结点和叶子结点之间的双向搜索。

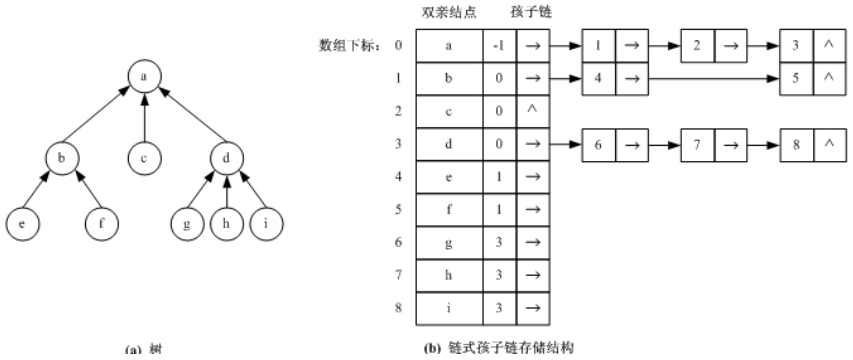


图4-21 树的孩子链存储结构示例图

（3）孩子兄弟链存储结构

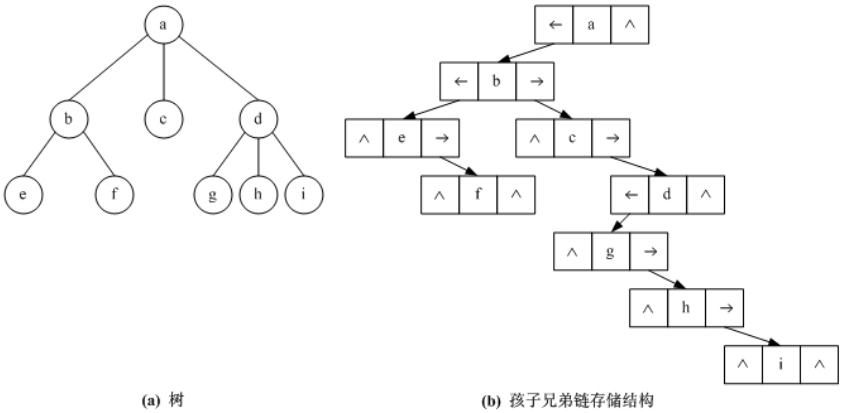
孩子兄弟链表示法又称为二叉链表表示法，就是使用二叉链表来存储树，存储方式为：

① 结点数据域代表树的结点数据信息；

② 左指针域指向结点的第一个孩子结点；

③ 右指针域指向结点的下一个兄弟结点。

如图4-22（b）所示为图4-22（a）中树的孩子兄弟链存储示意图：



(a) 树　　(b) 孩子兄弟链存储结构

图4-22 树的孩子兄弟链存储结构示例图

10．树，森林和二叉树之间的相互转换

树、森林和二叉树之间可以互相进行转换，即任何一个森林或一棵树可以对应一棵二叉树，而任一棵二叉树也能对应到一个森林或一棵树上。

（1）树与二叉树的转换

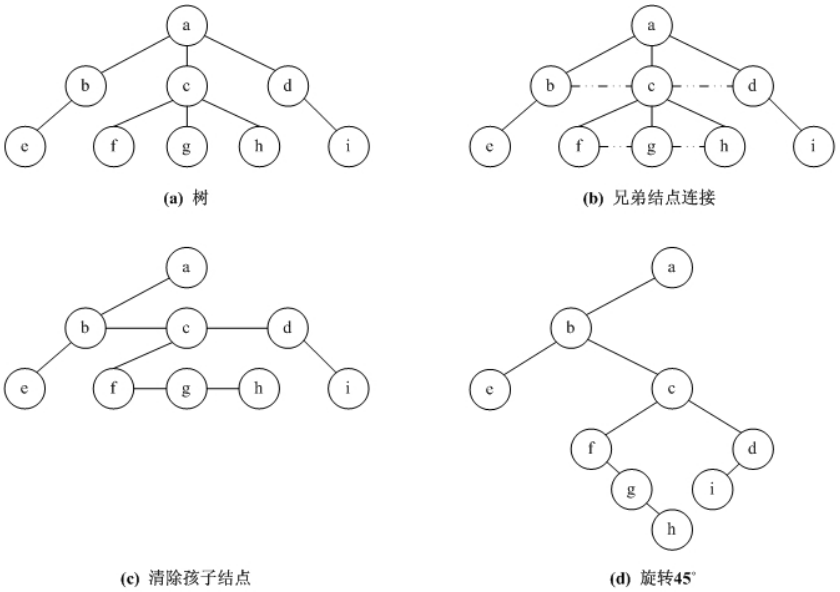
将一棵树转换为二叉树的方法是以二叉链表结构存储树，例如，将如图4-23(a)中树转换为二叉树的实例步骤如下：

① 在所有相邻兄弟结点之间加一水平连线，如图4-23(b)中虚线部分所示；

② 对每个非叶子结点而言，除了它最左边的孩子结点外，删除它与其他孩子结点之间的连线，如图3-23(c)所示。

③ 所有水平连线都以左边结点为轴心顺时针旋转45°即可。图3-23(d)就是最终转换后的二叉树。

将二叉树还原为树的过程恰恰相反，



(a) 树　　(b) 兄弟结点连接

(c) 清除孩子结点　　(d) 旋转45°

图4-23 树与二叉树的转换示例图

（2）二叉树与森林的转换

下面以图4-24(a)中的森林为例，介绍将森林转换为二叉树的方法。其步骤为：

① 新增一个结点，将森林中所有树的根结点设置为该新增结点的孩子结点，于是新产生了一棵树，如图4-24(b)所示；

② 将新产生的树转换为二叉树，如图4-24(c)所示；

③ 在新二叉树中删除根结点，则原根结点的第一个孩子成为新的根结点，森林转换二叉树完

毕，如图4-24(d)所示。

从图4-24中可以看出，在由森林转换的二叉树中，根结点一般都具有右子树，这是与树的最大区别。

将以上过程逆行就可以将二叉树还原为森林。

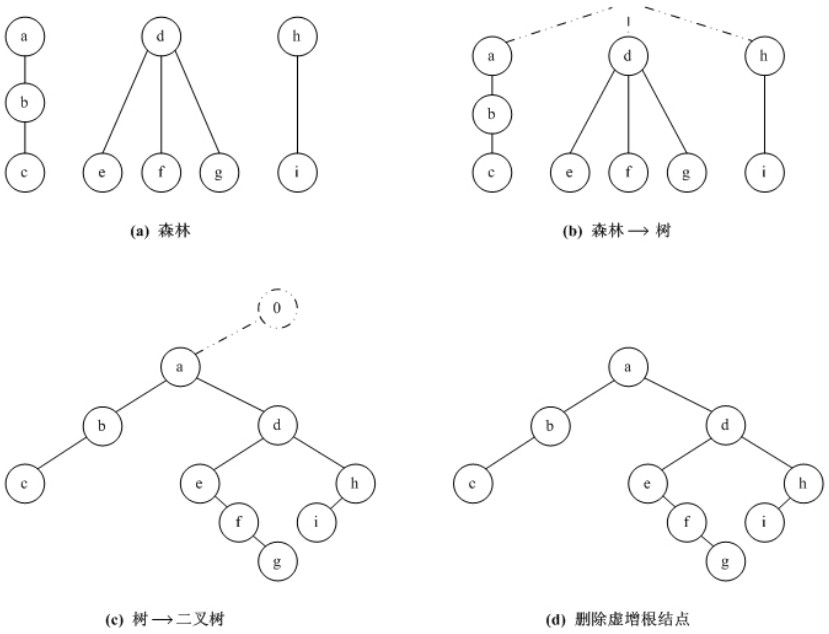


图4-24 森林转换为二叉树

上一节 本书简介 下一节

 作者：希赛教育软考学院 来源：希赛网 2014年05月06日

图

图是一种较线性表和树更为复杂的数据结构，它在人工智能、工程、数学、物理、化学、生物和计算机科学等领域中，有着广泛的应用。

1．图的定义

图G是由集合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，数据元素用顶点表示，数据元素之间的关系用边表示。下面我们将分别介绍下图中一些基本概念。

（1）有向图。图中每条边都是有方向的图称为有向图。有向图顶点之间的关系用$<v_i,v_j>$ 表示，它说明从v_i到v_j有一条有向边（也称为弧）。v_i是有向边的起点，称为弧尾；v_j是有向边的终点，称为弧头。

（2）无向图。图中的每条边都是无方向的图称为无向图，无向图中顶点v_i和v_j之间的边用(v_i,v_j)表示。因此，在有向图中$<v_i,v_j>$与$<v_j,v_i>$分别表示两条边，而在无向图中(v_i,v_j)与(v_j,v_i)表示的是同一条边。

（3）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$\frac{n(n-1)}{2}$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有方向相反的两条弧存在。

（4）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D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）。无论是有向图还是无向图，顶点数n、边数e与各顶点的度之间有以下关系：

$$e=\frac{1}{2}\sum_{i=1}^{n}D(v_i)$$

（5）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$(v_p,v_{i1}),(v_{i1},v_{i2}),\cdots,(v_{in},v_q)$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，则称其为简单路径。

（6）网。边（或弧）带权值的图称为网。

（7）有向树。如果一个有向图恰有一个顶点的入度为0，其余顶点的入度均为1，则是一棵有向树。

2．图的存储结构

图有两种基本的存储方式，它们分别是邻接矩阵法和邻接表法。

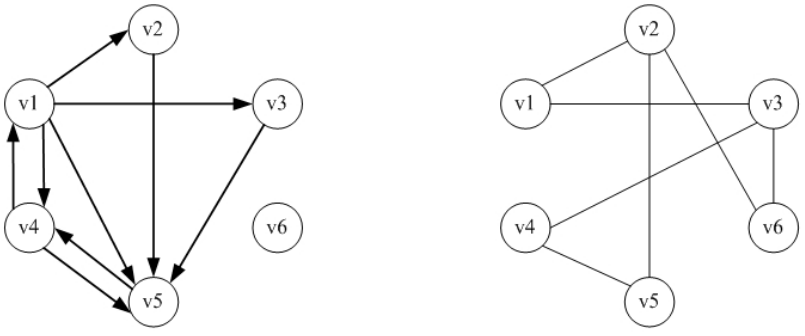
（1）邻接矩阵法

邻接矩阵法又称“数组表示法”，是存储图的最简单方法，它的基本思想是用一个n×n的邻接矩阵表示图中的点或边的信息，用一维向量来存储n个顶点的信息。

设G=（V，E）是具有n个顶点的图，其中V是顶点的集合，E是边的集合，那么邻接矩阵（AdjMatrix arcs）中的每一个元素的含义定义如下：

$$A[i,j]=\begin{cases}1 & (v_i,v_j)\in E \quad 或 \quad <v_i,v_j>\in E\\ 0 & (v_i,v_j)\notin E \quad 或 \quad <v_i,v_j>\notin E\end{cases}$$

如果顶点v_i和v_j之间存在一条边或边，定义A[i,j]为1，否则定义为0。



（a）**有向图**　　（b）**无向图**

图4-25 图的示意图

比如有向图4-25（a）和无向图4-25（b）中的邻接矩阵表分别如图4-25中矩阵A1和矩阵A2所示：

$$A1=\begin{bmatrix}0&1&1&1&1&0\\0&0&0&0&1&0\\0&0&0&0&1&0\\1&0&0&0&1&0\\0&0&0&1&0&0\\0&0&0&0&0&0\end{bmatrix}\qquad A2=\begin{bmatrix}0&1&1&0&0&0\\1&0&0&0&1&1\\1&0&0&1&0&1\\0&0&1&0&1&0\\0&1&0&1&0&0\\0&1&1&0&0&0\end{bmatrix}$$

矩阵 A1　　**矩阵** A2

图4-26 矩阵图

（2）邻接表法

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，其结构分别如图4-27所示。

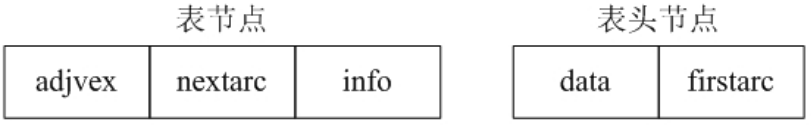


图4-27 邻接表的另种类型

其含义如下：

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：存储与边或弧有关的信息，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（邻接顶点）。

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

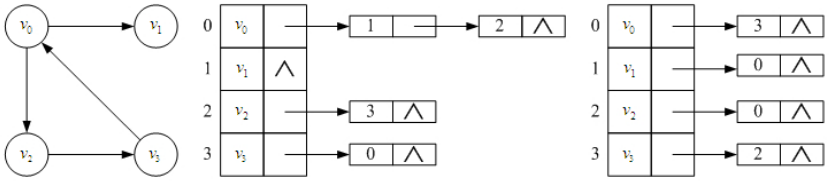
```
#define MaxN  30              /*图中顶点数目的最大值*/
typedef struct AreNode
{      /*邻接链表的表节点*/
int adjvex;               /*邻接顶点的顶点序号*/
double weight;            /*边（弧）上的权值*/
struct AreNode *nextarc;  /*下一个邻接顶点*/
}EdgeNode;
typedef struct Vnode
{        /*邻接链表的头节点*/
char  data;               /*顶点表示的数据，以一个字符表示*/
struct AreNode *firstarc; /*指向第一条依附于该顶点的弧的指针*/
}AdjList[MaxN];
typedef struct
{
int Vnum;                 /*图中顶点的数目*/
```

AdjList Vertices;

}Graph;

显然，对于有n个顶点、e条边的无向图来说，其邻接链表需用n个头节点和2e个表节点。

对于无向图的邻接链表，顶点 的度恰为第i个邻接链表中表节点的数目，而在有向图中，为求顶点的入度，必须扫描整个邻接表，这是因为第i个邻接链表中表节点的数目只是顶点v_i的出度。为此，可以建立一个有向图的逆邻接链表。有向图的邻接表和逆邻接表如图4-28所示。



（a）有向图　　（b）邻接表　　（c）逆邻接表

图4-28 一个有向图及其邻接表和逆邻接表表示

3．图的遍历

从图的某一个顶点出发，依照某种规则访问图中的其他顶点，要求所有的顶点被访问而且只被访问一次，这个过程就叫做“图的遍历”。目前遍历图的两种最常见规则是深度优先遍历和广度优先遍历，它们在无向图和有向图中均适用。

（1）图的深度优先搜索

图的“深度优先搜索”（Depth First Search，DFS）类似于树的前根遍历，是前根遍历的扩展，其基本搜索原则是尽可能先在“纵深”方向进行搜索。用深度优先搜索方法遍历到的顶点序列称为该图的深度优先遍历序列，或简称为“DFS”序列。

算法的基本思想为：

第一步，初始时定义图G的所有顶点均未被访问，任选一个顶点V为搜索的出发点，置顶点V为“已访问”；

第二步，然后依次对所有与V相连且未访问的顶点进行深度优先搜索，并置所有被搜索到的顶点为“已访问”。此后图中所有与顶点V有路径相通的顶点均将被访问；

第三步，如果图G中仍有未访问的顶点，则另选一个尚未访问的顶点作为新的搜索出发点，重复上述过程，直至图中所有顶点均已被访问。

当图G为连通图，算法在对V进行一次深度优先搜索后结束，只有在非连通图中，才需要执行第三步。

【函数代码】以邻接链表表示图的深度优先搜索算法。

```
int visited[MaxN] = {0};          /*调用遍历算法前所有的顶点都没有被访问过*/
void Dfs( Graph G, int i )
{
EdgeNode *t; int j;
printf( "%d", i );          /*访问序号为i的顶点*/
visited[i] = 1;             /*序号为i的顶点已访问过*/
   t = G.Vertices[i].firstarc;   /*取顶点i的第一个邻接顶点*/
   while( t! = NULL ){          /*检查所有与顶点i相邻接的顶点*/
      j = t->adjvex;           /*顶点j为顶点i的一个邻接顶点*/
```

```
        if( visited[j] = =0 )     /*若顶点j未被访问过*/
            Dfs( g, j );          /*从顶点j出发进行深度优先搜索*/
        t = t->nextarc;           /*取顶点i的下一个邻接顶点*/
      }/*while*/
    }/*Dfs*/
```

从函数Dfs()之外调用Dfs可以访问到所有与一个指定顶点有路径相通的其他顶点。若图是不连通的。则下一次应从另一个未被访问过的顶点出发，再次调用Dfs进行遍历，直到将图中所有的顶点都访问到为止。

（2）图的广度优先搜索

图的“广度优先搜索”（Breadth First Search，BFS）类似于树的层次遍历，是层次遍历的扩展，其基本搜索原则是尽可能先对“横向”进行搜索，即“先访问的邻接点”先于“后访问的邻接点”被访问。用广度优先搜索方法遍历到的顶点序列称为该图的广度优先遍历序列，或简称为“BFS”序列。

图的广度优先搜索方法的基本思想是：从图中某个顶点v出发，在访问了v之后依次访问v的各个未被访问过的邻接点，然后分别从这些邻接点出发依次访问它们的邻接点，并使“先被访问的顶点的邻接点”先于“后被访问的顶点的邻接点”被访问，直至图中所有已被访问的顶点的邻接点都被访问到。若此时还有未被访问的顶点，则另选图中的一个未被访问的顶点作为起点，重复上述过程，直至图中所有的顶点都被访问到为止。

【函数代码】以邻接链表表示图的广度优先搜索算法。

```
void Bfs( Graph G )
{   /*广度优先遍历图G*/
EdgeNode *t; int i, j, k;
int visited[G.Vnum] = {0};      /*调用遍历算法前所有的顶点都没有被访问过*/
InitQueue( Q );                 /*创建一个空队列*/
for( i = 0; i<G.Vnum; i++ ){
  if( !visited[i] ){          /*顶点i未被访问过*/
    EnQueue( Q, i ); printf( “%d”, i ); visited[i] = 1;   /*访问顶点i*/
    while( !Empty( Q ) ){    /*若队列不空*/
      DeQueue( Q, k );
      t = G.Vertices[k].firstarc;
      for( ; t; t = t->nextarc ){     /*检查所有与顶点k相邻接的顶点*/
        j = t->adjvex;                /*顶点j是顶点k的一个邻接顶点*/
        if( visited[j] = =0 ){        /*若顶点j未被访问过，将j加入队列*/
          EnQueue( Q , j );
          printf( “%d”, j );       /*访问序号为j的顶点*/
          visited[j] = 1;           /*置顶点j已被访问过标志*/
        }/*if*/
      }/*for*/
}/*while*/
```

}/*if*/

}/*for i*/

}/*Bfs*/

遍历图的过程实质上是通过边（弧）找邻接点的过程，因此广度优先搜索遍历图和深度优先搜索遍历图的时间复杂度相同，其不同之处仅仅在于对顶点访问的次序。

4．最小生成树

如果连通图G的一个子图是一棵包含图G的所有顶点的树，则该子图称为G的“生成树”，生成树是连通图中包含图中的所有顶点的极小连通子图，并且图的生成树并不惟一，从不同的顶点出发以不同的规则进行遍历，可以得到不同的生成树。比如按照深度优先搜索可得深度优先生成树，按照广度优先搜索可得广度优先生成树。在连通图中，构造一个生成树，如果树中所有边代表的权值之和最小，那么就称这个生成树是“最小生成树”或“最小代价生成树”，生成树的“代价”是指树上所有边的权值之和。最小生成树具有一个特别重要的性质：设N=（V，E）是一个连通网，U是顶点集V的一个非空子集。若（u，v）是一条具有最小权值的边，其中u∈U，∈V-U，则一定存在一棵包含此边（u，v）的最小生成树。

普里姆（Prim）算法和克鲁斯卡尔（Kruskal）算法正是利用这个性质来生成最小生成树的。

（1）普利姆算法

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 的边 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，T=（V，{TE}）为N的最小生成树。

由此可知，普利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。

用普利姆算法构造最小生成树的过程如图4-29所示。

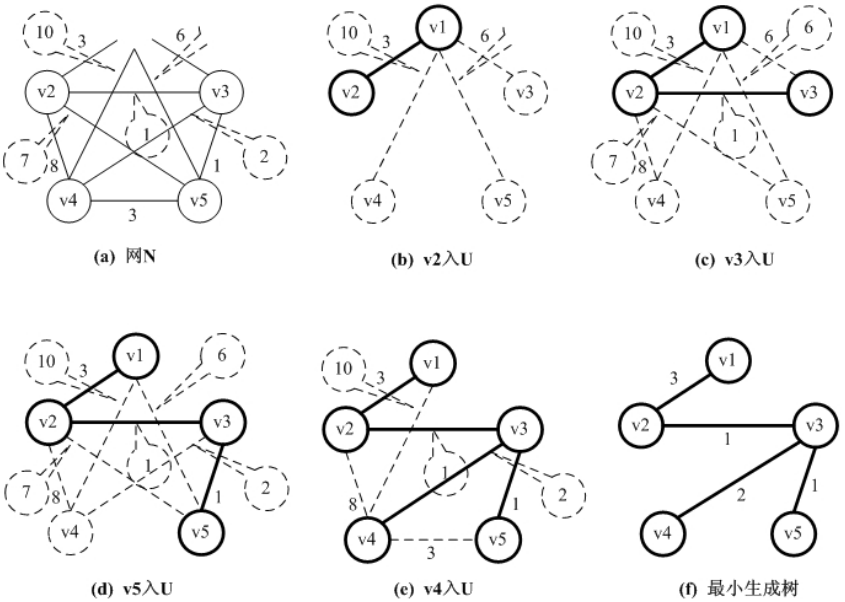


图4-29 普里姆算法求最小生成树步骤

采用普里姆算法求得代价为：权（v1,v2）+权（v2,v3）+权（v3,v4）+权（v3,v5）=3+1+2+1=7。

（2）克鲁斯卡尔算法

克鲁斯卡尔算法的基本思想是，先将n个顶点纳入子图T中，再逐渐按照某种规则向T加入n-1条边即可形成最小生成树。

设连通网N=（V，E）中，T=（V，TE）是存放最小生成树的集合，其中TE是N的最小生成树的边的集合，则克鲁斯卡尔算法创建最小生成树的详细步骤如下：

步骤1，初始状态，TE为空，此时T是一个只有n个顶点没有边的森林。

步骤2，考察图中满足以下条件的边：边一端在森林TE的一棵树中，另一端在森林TE的另一棵树中。在这些边中寻找权值最小的一条，假设为（v_i，v_j），那么将边（v_i，v_j）加入集合TE中，这样T中两棵相对较小的树就合成一棵相对较大的树，T中树的总数减小1个。

步骤3，重复执行n-1次步骤2后得到的T即为所求。

用克鲁斯卡尔算法构造最小生成树的过程如图4-30所示。

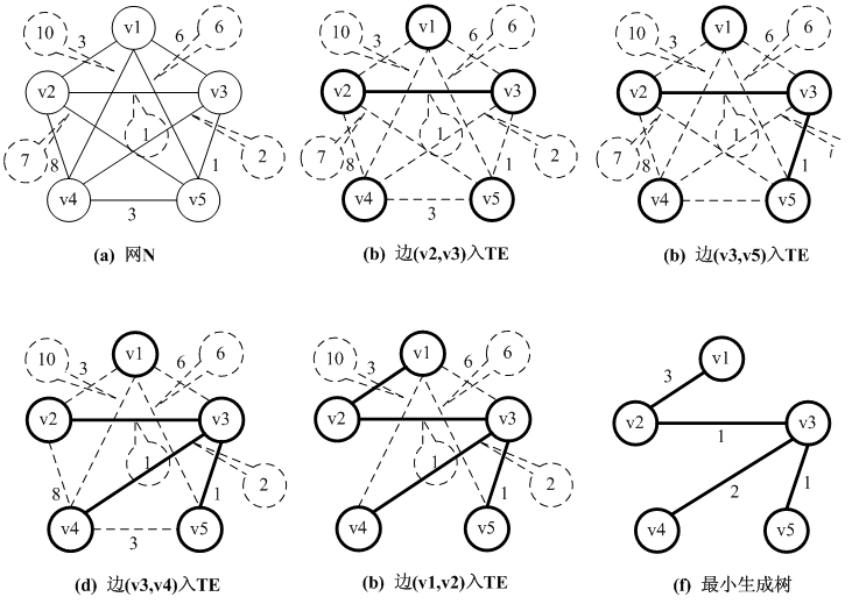


图4-30 克鲁斯卡尔算法求最小生成树步骤

5．拓扑排序、关键路径和最短路径

拓扑排序、关键路径和最短路径是考试中常见的一些内容，下面我们分别来讲述这些知识点。

（1）拓扑排序

一个无环的有向图称为有向无环图（Directed Acyclic Graph，简称DAG）。有向无环图常常用来描述一些前后相关的事件，把顶点当成活动，用边表示不同活动之间的相关性，边<i，j>表示必须先完成事件i，才可以进行事件j，这样的有向图称为“顶点表示活动的网”（Activity On Vertex Network，简称AOV-网）。在网中，存在一条从顶点v_i到顶点v_j的有向路径，则称v_i是v_j的“前驱”，v_j是v_i的“后继”，如果图中存在边<i，j>，则称v_i是v_j的“直接前驱”，v_j是v_i的“直接后继”。

在AOV图中进行拓扑排序的算法思想如下：

步骤1，在图中任意选择一个入度为0的顶点，如果没有就说明拓扑排序完成。

步骤2，将该顶点输出到拓扑排序序列。

步骤3，在图中删除此顶点及以该顶点为起点的边。

步骤4，回到第1步继续执行。

图4-31给出了一个DAG图的拓扑有序序列的产生过程。

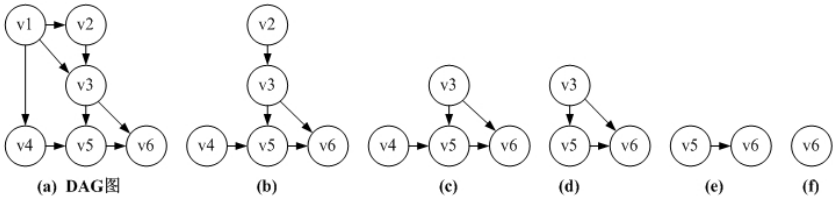


图4-31 AOV图拓扑排序示意图

（2）关键路径

AOV-网是顶点表示活动的网，而AOE-网（Activity On Edge）则是边表示活动的网，它是一个带权的有向无环图，其中顶点表示事件，边表示活动，权表示活动持续的时间，一般情况下，AOE-网用来估计工程的完成时间，图4-32定义了7个事件、10个活动的AOE-网。

在正确情况下，AOE-网表示的工程只有一个开始点和一个完成点，即只有一个入度为0的顶点（源点）和一个出度为0的顶点（汇点）。

在AOE-网表示的工程中，部分活动可以并行进行，那么完成工程的“最短时间”是从开始点到完成点的“最长路径”的长度，即该条路径上所有边的权值之和，路径长度最长的路径称为“关键路径”。

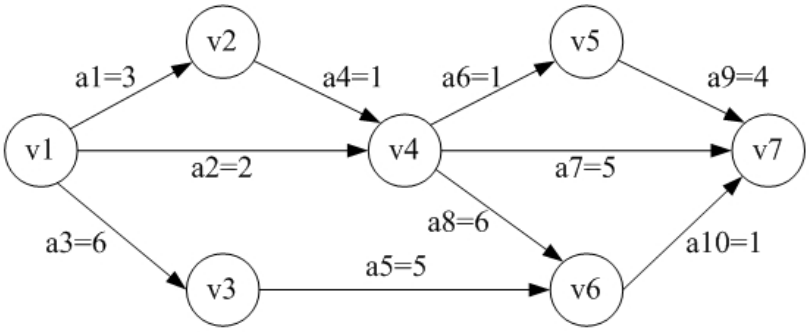


图4-32 AOE-网示意图

从源点到汇点的路径中，长度最长的路径称为关键路径。关键路径上的所有活动均是关键活动。如果任何一项关键活动没有按期完成，则会影响整个工程的进度，而提高关键活动的速度通常可以缩短整个工程的工期。假设在n个顶点的AOE网中，顶点v_0表示源点、顶点v_{n-1}表示汇点，则引入顶点事件的最早、最晚发生时间，活动的最早、最晚开始时间等术语。

顶点事件的最早发生时间ve（j）是指从源点v_0到V_j的最长路径长度（时间）。这个时间决定了所有从V_j发出的弧所表示的活动能够开工的最早时间。

ve（j）计算方法为

$$\begin{cases} ve(0)=0 \\ ve(j)=\max\{ve(i)+dut(<i,j>)\} & <i,j>\in T, 1\le j\le n-1 \end{cases}$$

其中，T是所有到达顶点j的弧的集合，dut（<i,j>）是弧<i,j>上的权值，n是网中的顶点数。

显然，上式是一个从源点开始的递推公式。必须在 的所有前驱顶点事件的最早发生时间全部得出后才能计算ve（j）。这样必须对AOE网进行拓扑排序，然后按拓扑有序序列逐个求出各顶点事件的最早发生时间。

顶点事件的最晚发生时间vl（i）。vl（i）是指在不推迟整个工期的前提下，事件 的最晚发生时间。对一个工程来说，计划用几天时间完成是可以从AOE网求得的。其数值就是汇点 的最早发生时间ve（n-1），而这个时间也就是vl（n-1）。其他顶点事件的vl应从汇点开始，逐步向源点方向递推才能求得，所以vl（i）的计算公式为

$$\begin{cases} vl(n-1) = ve(n-1) \\ vl(i) = \min\{vl(j) - dut(< i, j >)\} \quad < i, j > \in S, 1 \le i \le n-2 \end{cases}$$

其中，S是所有从顶点i发出的弧的集合。

显然，必须在顶点v_i的所有后继顶点事件的最晚发生时间全部得出后才能计算vl（i）。这样必须对AOE网逆拓扑排序，由逆拓扑序列递推计算各顶点的vl值。

活动a_k的最早开始时间e（k）是指弧<i,j>所表示的活动a_k最早可开工时间。

e（k）=ve（i）

这说明活动a_k的最早开始时间等于事件V_i的最早发生时间。

活动a_k的最晚开始时间l（k）是指在不推迟整个工期的前提下，该活动的最晚开始时间。若活动a_k由弧<i,j>表示，则

l（k）=vl（j）-dut（<i,j>）

对于活动a_k来说，若e（k）=l（k），则表示活动a_k是关键活动，它说明该活动最早可开工时间与整个工程计划允许该活动最晚的开工时间一致，施工期一点也不能拖延。若活动a_k不能按期完成，则工程将延期；若活动a_k提前完成，则可能使整个工程提前完工。

由关键活动组成的路径是关键路径。依照上述计算关键活动的方法，就可形成求AOE网的关键路径。

（3）最短路径

我们首先来看单源点最短路径，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）算法提出了一个按路径长度递增的次序产生最短路径的方法。

设S为最短距离已确定的顶点集，则V-S是最短距离尚未确定的顶点集，增加一个辅助向量D和P，其中：

D的每个分量D[i]表示当前已知的从源点V_s到终点V_i的最短路径长度。

P的每个分量P[i]代表了从源点途经顶点P[i]后再到达顶点Vi的最短路径，取值为0表示已到达，取值为∞表示目前暂时不存在从原点到该顶点的路径。

迪杰斯特拉算法的基本思想如下：

步骤1，算法开始时，只有原点Vs的最短距离是已知的（D[s]=0），此时已确定顶点集S={V_s}，同时向量D的每个分量取值可以从邻接矩阵A中获得，倘若顶点V_i与V_s可达则P[i]赋值为s，否则赋值为0；

步骤2，在集合V-S中的所有顶点中，选取距离V_s路径长度最短的顶点，将其加入集合S中。

步骤3，针对每一个V_i∈V-S，重新计算源点V_s到终点V_i的最短路径长度D[i]，计算方法如下：

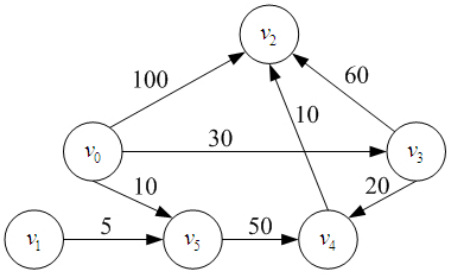
假设新加入集合S的顶点为V_j，比较原来从V_s到V_i的最短路径长度D[i]和从V_s取道V_j后到达V_i的路径长度（D[j]加上边<V_j,V_i>的权值，即D[j]+A[j,i]），小者为新的最短路径长度，并更改路径P[i]为顶点V_j：if（D[j]+A[j,i] < D[i]）{D[i] = D[j]+A[j,i]; P[i]=j;}

步骤4，重复执行步骤2和3，直到集合V-S为空或者集合V-S中所有顶点距离Vs的路径长度都为无穷大时结束。

对于图4-33所示的有向网，用迪杰斯特拉算法求解顶点 到达其余顶点的最短路径的过程如表4-

2所示。

（a）有向图G　　（b）图G的邻接矩阵



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

（a）有向图 G　　（b）图 G 的邻接矩阵

图4-33 有向图G及其邻接矩阵

表4-2 迪杰斯特拉算法求解图8-46中顶点v_0到v1、v2、v3、v4、v5最短路径的过程

终点	1	2	3	4	5
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_3v_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，则将顶点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，则将顶点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，则将顶点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，则将顶点v_2加入S集合	v_0到v_1无路径
集合S	$\{v_5\}$	$\{v_5,v_3\}$	$\{v_5,v_3,v_4\}$	$\{v_5,v_3,v_4,v_2\}$	

若每次以一个顶点为源点，重复执行迪杰斯特拉算法n次，便可求得网中每一对顶点之间的最短路径。下面介绍弗洛伊德（Floyd）提出的求最短路径的算法，该算法在形式上要简单一些。

弗洛伊德算法思想是：假设图采用邻接矩阵的方式存储，需要求从顶点v_i到v_j的最短路径。arcs[i][j]表示弧（v_i,v_j）的权值，若此弧不存在，则权值为区别于有效权值的一个数。如果存在v_i到v_j的弧，则从v_i到v_j存在一条长度为arcs[i][j]的路径，该路径不一定是最短路径，尚需进行n次试探。首先考虑路径（ v_i，v_0，v_j）是否存在（即判别路径（v_i，v_0）和（v_0,v_j）是否存在），若存在，则比较（v_i,v_j）与（ v_i，v_0，v_j）的路径长度，取较短者为从v_i到v_j的中间顶点的序号不大于0的最短路径。假如在路径上再增加一个顶点v_i，也就是说，如果（v_i,...,v_1）和（v_1,...,v_j）分别是当前找到的中间顶点的序号不大于0的 v_i到v_1 以及v_1到v_j的最短路径，那么（ v_i，v_1，v_j）就有可能是从v_i到v_j的中间顶点的序号不大于1的最短路径。将它与已经得到的从v_i到v_j的中间顶点的序号不大于0的最短路径相比较，从中选出中间顶点的序号不大于1的最短路径之后，再增加一个顶点v_2继续进行试探，依此类推。在一般情况下，若（v_i,...,v_k）和（v_k,...,v_j）分别是从v_i到v_k和v_k到v_j的中间顶点的序号不大于

k-1的最短路径，则将（ v_i，...,v_k，...,v_j）与已经得到的从v_i到v_j的中间顶点的序号不大于k-1的最短路径相比较，长度较短者便是从 到 的中间顶点的序号不大于k的最短路径。这样，在经过n次试探后，最后求得的必是从v_i到v_j的最短路径。按此方法，可以同时求得各对顶点间的最短路径。

第 4 章：数据结构设计　作者：希赛教育软考学院　来源：希赛网　2014年05月06日

典型真题解析

本节从历年考试真题中，精选出4道典型的试题进行分析，这4道试题所考查的知识点基本上覆盖了本章的所有内容，非常具有代表性。

第 4 章：数据结构设计　作者：希赛教育软考学院　来源：希赛网　2014年05月06日

例题1

B树是一种多叉平衡查找树。一棵m阶的B树，或为空树，或为满足下列特性的m叉树。

① 树中每个节点至多有m棵子树；

② 若根节点不是叶子节点，则它至少有两棵子树；

③ 除根之外的所有非叶子节点至少有$\lceil \frac{m}{2} \rceil$棵子树；

④ 所有的非叶子节点中包含下列数据信息；

$$(n, A_0, K_1, A_1, K_2, A_2, ..., K_n, A_n)$$

其中：$K_i(i=1,2,...,n)$为关键字，且$K_i < K_{i+1}(i=1,2,...,n-1)$；$A_i(i=0,1,...,n)$为指向子树根节点的指针，且指针$A_{i-1}$所指子树中所有节点的关键字均小于$K_i$，$A_n$所指子树中所有节点的关键字均大于$K_n$，n为节点中关键字的个数。

⑤所有的叶子节点都出现在同一层次上，并且不带信息（可以看作是外部节点或查找失败的节点，实际上这些节点不存在，指向这些节点的指针为空）。

例如，一棵4阶B树如图4-34所示（节点中关键字的数目省略）。

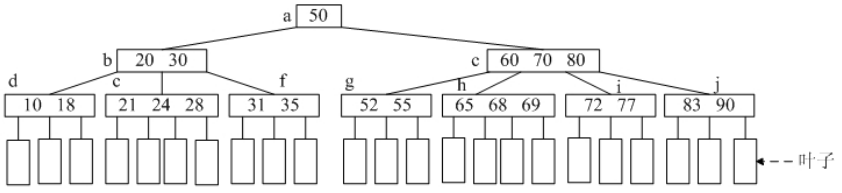


图4-34　4阶B树示例

B树的阶m、bool类型、关键字类型及B树节点的定义如下。

```
#define M 4        /*s树的阶数*/
```

```
typedef enum { FALSE = 0，TRUE = 1 } bool;
typedef int    ElemKeyType;
typedef struct BTreeNode{
int numkeys;       /*节点中关键字的数目*/
struct BTreeNode *parent;     /*指向父节点的指针，树根的父节点指针为空*/
struct BTreeNode *A[M];       /*指向子树节点的指针数组*/
ElemKeyType  K[M];            /*存储关键字的数组，K[0]闲置不用*/
}BTreeNode;
```

函数SearchBtree(BTreeNode *root, ElemKeyType akey, BTreeNode **ptr)的功能是：在给定的一棵m阶B树中查找关键字akey所在节点，若找到则返回TRUE，否则返回FALSE。其中，root是指向该m阶B树根节点的指针，参数ptr返回akey所在节点的指针，若akey不在该B树中，则ptr返回查找失败时空指针所在节点的指针。例如，在图4-34所示的4阶B树中查找关键字okey时，ptr返回指向节点e的指针。

注：在节点中查找关键字akey时采用二分法。

【函数代码1】

```
bool SearchBtree( BTreeNode *root, ElemKeyType akey, BTreeNode **ptr )
{
  int lw, hik mid;
  BTreeNode *p = root;
  *ptr = NULL;
  while( p ){
     lw = 1;  hi =__（1）__ ;
     while( lw< = hi ){
        mid = ( lw+hi )/2;
        if( p->K[mid] = =akey ){
          *ptr = p;
          return TRUE;
        }
        else
          if(__（2）__ )
            hi = mid-1;
          else
            lw = mid+1;
     }
     *ptr = p;
     p =__（3）__ ;
}
return FALSE;
}
```

【说明2】

在m阶B树中插入一个关键字时，首先在最接近外部节点的某个非叶子节点中增加一个关键字，若该节点中关键字的个数不超过m-1，则完成插入；否则，要进行节点的“分裂”处理。所谓“分裂”，就是把节点中处于中间位置上的关键字取出来并插入其父节点中，然后以该关键字为分界线，把原节点分成两个节点。“分裂”过程可能会一直持续到树根，若树根节点也需要分裂，则整棵树的高度增1。

例如，在图4-34所示的B树中插入关键字25时，需将其插入节点e中，由于e中已经有3个关键字，因此将关键字24插入节点e父节点b，并以24为分界线将节点e分裂为e1和e2两个节点，结果如图4-35所示。

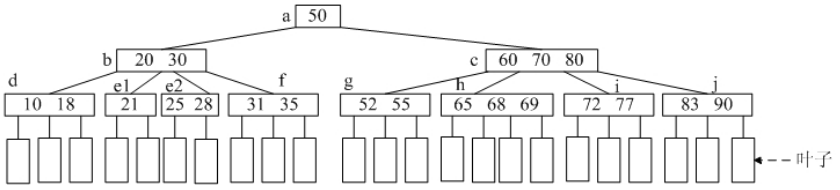


图4-35 在图4-34所示的4阶B树中插入关键字25后的B树

函数Isgrowing(BTreeNode *root，ElemKeyType akey)的功能是：判断在给定的m阶B树中插入关键字akey后，该B树的高度是否增加，若增加则返回TRUE，否则返回FALSE。其中，root是指向该m阶B树根节点的指针。

在函数Isgrowing中，首先调用函数SearchBtree(即函数代码1)，查找关键字akey是否在给定的m阶B树中，若在则返回FALSE（表明无需插入关键字akey，树的高度不会增加）；否则，通过判断节点中关键字的数目考察插入关键字akey后该B树的高度是否增加。

【函数代码2】

```
bool Isgrowing( BTreeNode *root, ElenKeyType akey )
{  BTreeNode *t, *f;
if(  ! SearchBtree(  （4）  ) ){
t = f;
while(  （5）  ){
   t = t->parent;  }
if( !t )
   return TRUE;
}
return FALSE;
}
```

例题1分析

本题考查平衡查找树相关内容。函数SearchBtree( )是采用二分法查找关键字，其实此函数中的几个空主要考的是二分法查找。二分法查找元素的过程是：首先令待查找的元素与查找表中间位置上的元素进行比较，若相等，则查找成功；否则，根据待查元素与表中间位置元素的大小关系，下一步到查找表的前半区间或后半区间进行二分查找。如果在确定的任何一个子区间都找不到指定的元素，则确定查找失败。若查找区间用一对下标lw和hi确定，则lw<=hi表示有效的查找区间，查找失败时所确定的查找区间为lw>hi。

例如，需要查找关键字21，则首先将21与根节点最中间的关键字50进行比较，比较发现

21<50，而本节点已无其他关键字，所以进行节点的左分枝；将21与节点b中的20比较发现20<21，再与30比较，发现21<30，所以进入20与30两个关键字中间的分枝e；在e中，首先与中间关键字24比较，21<24，所以与21比较，这样就查到了关键字21。

很明显，第（1）空应是填关键字的个数，而在每个节点中的关键字数目由BTreeNode中的numkeys域表示，所以函数代码1的空（1）处应填入“p->numkeys”。

若用lw和hi指示出查找区间，则由于查找表元素的递增排列特性，当待查找的元素小于表中间位置的元素时，下一步应在前半区间查找，即查找区间的一对下标为lw、mid-1，也就是说，函数代码1的空（2）处应填入“akey<p->K[mid]”。

如果在当前节点中找不到指定的关键字akey，则lw>hi，由节点中的指针对A[hi]或A[lw-1]指示出下一层的子树节点，因此函数代码1的空（3）处应填入“p->A[hi]”或“p->A[lw-1]”。

下面分析函数代码2的功能及运算过程。函数代码2用于判断在B树种插入一个关键字时，树的高度是否增加。若指定的关键字已经在B树的某节点中，就不需要插入该关键字，显然树也不会长高。

实现函数调用时实参要向形参传递信息，C语言采取传值调用的方式，根据实参向形参的值传递原则，函数代码2的第（4）空处应填入“root,akey,&f”。

根据题目中给出的描述，在m阶B树中插入一个关键字时，首先在最接近外部节点的某个非叶子节点中增加一个关键字，若该节点中关键字的个数不超过m-1，则完成插入；否则，要进行节点的“分裂”处理。所谓“分裂”，就是把节点中处于中间位置上的关键字取出来并插入其父节点中，然后以该关键字为分界线，把原节点分成两个节点。“分裂”过程可能会一直持续到树根，若树根节点也需要分裂，则整棵树的高度增加1。

显然考查插入关键字akey后树的高度是否增加，只需沿其祖先节点关系一直考查直到树根为止，判断依据就是每个待考查的节点中目前已有的关键字个数，因此函数代码2中的第（5）空处应填入“t&&t->numkeys= =m-1”。

例题1参考答案

（1）p->numkeys

（2）akey<p->K[mid]

（3）p->A[hi]

（4）root，akey，&f

（5）t&&t->numkeys = =M-1

上一节　本书简介　下一节

例题2

一般的树结构常采用孩子-兄弟表示法表示，即用二叉链表作为树的存储结构，链表中节点的两个链域分别指向该节点的第一个孩子节点和下一个兄弟节点。例如，图4-36（a）所示的树的孩子-兄弟表示如图4-36（b）所示。

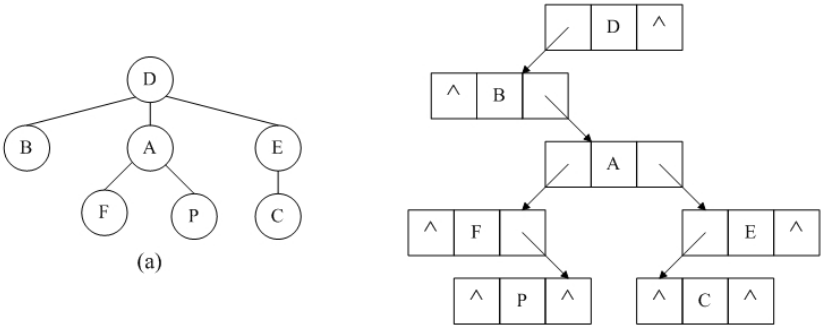


图4-36 树及其孩子-兄弟表示示意图

函数LevelTraverse( )的功能是对给定树进行层序遍历。例如，对图4-36（a）所示的树进行层序遍历时，节点的访问次序为：DBAEFPC。

对树进行层序遍历时使用了队列结构，实现队列基本操作的函数原型如表4-3所示。

表4-3 实现队列基本操作的函数原型表

函数原型	说明
Void InitQueue(Queue *Q)	初始化队列
Bool IsEmpty(Queue Q)	判断队列是否为空，若是则返回 TRUE，否则返回 FALSE
Void EnQueue(Queue *Q，TreeNode p)	元素入队列
Void DeQueue(Queue *Q，TreeNode *p)	元素出队列

Bool、Status类型定义如下：

```
typedef enum { FALSE = 0, TRUE = 1 } Bool;
typedef enum { OVERFLOW = -2, UNDERFLOW = -1, ERROR = 0, OK = 1 } Status;
```

树的二叉链表节点定义如下：

```
typedef struct Node{
    char data;
    struct Node *firstchild, *nextbrother;
}Node，*TreeNode;
```

【函数代码】

```
Status LevelTraverse( TreeNode root )
{ /*层序遍历树，树采用孩子-兄弟表示法，root是树根节点的指针*/
  Queue temQ;
  TreeNode ptr, brotherptr;
  if( !root )
return ERROR;
InitQueue( &tempQ );
 （1） ;
brotherptr = root->nextbrother;
while( brotherptr ){
    EnQueue( &tempQ, brotherptr );
     （2） ;
}/*end-while*/
while( （3） ){
   （4） ;
```

```
printf( "%c\t"，ptr->data );
if(  （5）  )continue;
   （6）;
 brotherptr = ptr->firstchild->nextbrother;
while( brotherptr ){
   EnQueue( &tempQ, brotherptr );
  （7）;
}/*end-while*/
}/*end-while*/
return OK;
}/*LevelTraverse*/
```

例题2分析

解答此题的关键在于理解用队列层序遍历树的过程。算法的流程是这样的：首先将树根节点入队，然后将其所有兄弟节点入队（当然，由于是根节点，故无兄弟节点）；完成这一操作以后，便开始出队、打印；在打印完了之后，需要进行一个判断，判断当前节点有无孩子节点，若有孩子节点，则将孩子节点入队，同时将孩子节点的所有兄弟节点入队；完了以后继续进行出队操作，出队后再次判断当前节点是否有孩子节点，并重复上述过程，直至所有节点输出。

这一描述可能过于理论，难以理解，接下来以本题为例来说明此过程。首先将树根节点D入队，并同时检查是否有兄弟节点，对于兄弟节点应一并入队。这里的D没有兄弟节点，所以队列此时应是：

D

接下来执行出队操作。D出队，出队以后检查D是否有子节点，经检查，D有子节点B，所以将B入队，同时将B的兄弟节点：A和E按顺序入队。得到队列：

B

A

E

接下来再执行出队操作。B出队，同时检查B是否有子节点，B无子节点，所以继续执行出队操作。A出队，同时检查A是否有子节点，A有子节点F，所以将F入队，同时将F的兄弟节点P入队。得到队列：

E

F

P

接下来再次执行出队操作。E出队，E有子节点C，所以C入队。得：

F

P

C

接下来再次执行出队操作。F出队，F无子节点，继续出队操作，P出队，仍无子节点，最后C出队，整个过程结束。

通过对算法的详细分析，我们现在便可轻松得到答案。（1）应是对根节点root执行入队操作，

即EnQueue(&tempQ,root)。（2）在一个循环当中，循环变量是brotherptr，此变量无语句对其进行更新，所以（2）必定是更新brotherptr。结合前面的算法分析可知（2）应填：brotherptr = brotherptr->nextbrother。（3）、（4）加上后面的语句“printf(“%c\t” ，ptr->data); ”是控制数据的输出，这些数据应是从队列中得到，所以此处必有出队操作，同时在出队之前应判断队列是否为空，所以（3）、（4）填：！IsEmpty(tempQ)和DeQueue(&tempQ,&ptr)。（5）实际上是问“在什么情况下，要持续进行出队操作？”，前面的算法分析中已指出：若出队节点无子节点，则继续进行出队操作，所以（5）填：！ptr->firstchild。（6）和（7）所在的语句段的功能是将刚出队节点的子节点及其兄弟节点入队，所以（6）填：EnQueue(&tempQ,ptr->firstchild)。（7）和（2）相同，填：brotherptr = brotherptr->nextbrother。

例题2参考答案

（1）EnQueue(&tempQ, root)

（2）brotherptr = brotherptr->nextbrother

（3）！IsEmpty(tempQ)

（4）DeQueue(&tempQ, &ptr)

（5）！ptr->firstchild

（6）EnQueue(&tempQ, ptr->firstchild)

（7）brotherptr = brotherptr->nextbrother

第 4 章：数据结构设计　作者：希赛教育软考学院　来源：希赛网　2014年05月06日

例题3

栈（Stack）结构是计算机语言实现中的一种重要数据结构。对于任意栈，进行插入和删除操作的一端称为栈顶（Stack Top），而另一端为栈底（Stack Bottom）。栈的基本操作包括：创建栈（NewStack）、判断栈是否为空（IsEmpty）、判断栈是否已满（IsFull）、获取栈顶数据（Top）、压栈/入栈（Push）、弹栈/出栈（Pop）。

当设计栈的存储结构时，可以采取多种方式。其中，采用链式存储结构实现的栈中各数据项不必连续存储（如图4-37所示）。

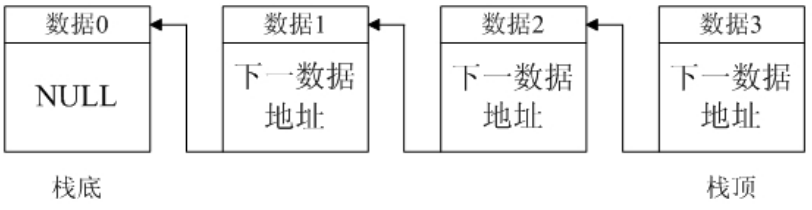


图4-37 链栈示意图

以下C代码采用链式存储结构实现一个整数栈操作。

【函数代码】

```
typedef struct List {
```

```
    int data;                    //栈数据
    struct List* next;           //上次入栈的数据地址
}List;
typedef struct Stack {
    List* pTop;                  //当前栈顶指针
}Stack;
Stack* NewStack( ) { return( Stack* )calloc( 1, sizeof( Stack ) );  }
int IsEmpty( Stack* S )
{ //判断栈S是否为空栈
if( __（1）__ )return 1;
return 0;
}
int Top( Stack* S )
{ //获取栈顶数据。若栈为空，则返回机器可表示的最小整数
if( IsEmpty( S ) )return INT_MIN;
return __（2）__;
}
void Push( Stack* S，int theData )
{ //将数据theData压栈
    List* newNode;
    newNode = ( List* )calloc( 1，sizeof( List ) );
    newNode->data = theData;
    newNode->next = S->pTop;
    S->pTop = __（3）__;
}
void Pop( Stack* S )
{ //弹栈
List* lastTop;
if( IsEmpty( S ) )return;
lastTop = S->pTop;
S->pTop = __（4）__;
free( lastTop );
}
#define MD( a ) a<<2
int main( )
{
int i;
Stack* myStack;
myStack = NewStack( );
```

```
Push( myStack, MD( 1 ) );
Push( myStack, MD( 2 ) );
Pop( myStack );
Push( myStack, MD( 3 )+1 );
while( !IsEmpty( myStack ) ){
    printf( "%d", Top( myStack ) );
    Pop( myStack );
}
return 0;
}
```

以上程序运行时的输出结果为：（5）

例题3分析

本题考查基本程序设计能力。

堆栈是软件设计中常使用的一种经典数据结构，题目给出的操作都是任何堆栈都具有的基本操作。堆栈的存储结构通常采用数组或链表形式，但无论采用哪种存储结构，整体上呈现的是后进先出的特点，即后进入堆栈的元素先出栈。题目中给出的结构体Stack仅包含一个指向栈顶元素的指针（栈顶指针），当且仅当堆栈中没有元素时，该指针应为NULL。当向堆栈中增加元素时，首先需要动态创建该元素的存储区，并且栈顶指针指向该元素。当元素出栈时，栈顶指针则指向出栈元素的紧前一个元素。结构体List表示栈中元素，包含对应的数据和指向紧上次入栈的元素指针next，对于第1个入栈的元素，指针next为NULL，而其他元素中的指针next一定不为NULL。

C语言中，如果用一个整数型表达式表示条件判定语句的话，该表达式的值为0则表示假，非0表示真。从给定程序代码可以看出，对于函数IsEmpty，若其返回值为0则表示堆栈非空，否则表示堆栈为空。因此，对于空（1），必须填写可表示堆栈为空的判定语句：S = =NULL||S->pTop = =NULL，这2个条件中只要有1个条件满足，则表明堆栈S为空。对于空（2），此时需要返回栈顶元素中的数据，而栈顶元素为S->pTop，所以对应的数据应该为S->pTop->data。

对于压栈操作Push，在为新元素获取存储空间后，必须调整堆栈的栈顶指针S->pTop指向新元素的存储区，即S->pTop=newNode。对于弹栈操作Pop，弹出栈顶元素lastTop后，需要调整栈顶指针，使其指向被弹出元素的下一个元素，即S->pTop=S->pTop->next，或S->pTop=lastTop->next。

对于main函数中宏MD(x)，在程序预编译时会按字符替换为“x<<2”。所以在main函数中，首先入栈的元素为“1<<2”，即整数4，第2个入栈的元素为“2<<2”，即整数8，其次将8弹出，然后再将“3<<2+1”入栈，C语言中“+”优先级高于“<<”，所以此时入栈者为整数24，而此时堆栈中有2个元素，其中栈顶元素为24，下一元素为4。最后，若堆栈非空，则循环完成显示栈顶元素的值、弹出栈顶元素的操作，直至堆栈为空。所以程序执行时的输出内容为“24 4”。

例题3参考答案

（1）S= = NULL||S->pTop = =NULL

（2）S->pTop->data

（3）newNode

（4）S->pTop->next，或lastTop->next

（5）24 4

例题4

对二叉树进行遍历是二叉树的一个基本运算。遍历是指按某种策略访问二叉树的每个节点，且每个节点仅访问一次的过程。函数InOrder( )借助栈实现二叉树的非递归中序遍历运算。

设二叉树采用二叉链表存储，节点类型定义如下：

```
typedef struct BtNode{
ElemType data;          /*节点的数据域，ElemType的具体定义省略*/
struct BtNode *lchild, *rchild;   /*节点的左、右孩子指针域*/
}BtNode, *Btree;
```

在函数InOrder( )中，用栈暂存二叉树中各个节点的指针，并将栈表示为不含头节点的单向链表（简称链栈），其节点类型定义如下：

```
typedef struct StNode{          /*链栈的节点类型*/
Btree elem;                 /*栈中的元素是指向二叉链表节点的指针*/
struct StNode *link;
}StNode;
```

假设从栈顶到栈底的元素为 ，则不含头节点的链栈示意图如图4-38所示。

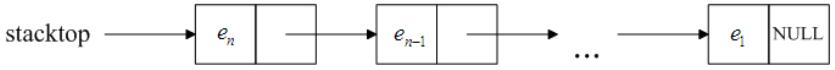


图4-38 链栈示意图

【函数代码】

```
int InOrder( Btree root )       /*实现二叉树的非递归中序遍历*/
{
Btree ptr;                  /*ptr用于指向二叉树中的节点*/
StNode *q;                  /*q暂存链栈中新创建或待删除的节点指针*/
StNode *stacktop = NULL;        /*初始化空栈的栈顶指针stacktop*/
ptr = root;                 /*ptr指向二叉树的根节点*/
while( （1） || stacktop!= NULL ){
while( ptr!= NULL ){
    q = ( StNode * )malloc( sizeof( StNode ) );
    if( q = =NULL )
return -1;
q->elem = ptr;
（2）;
```

```
    stacktop = q;          /*stacktop指向新的栈顶*/
    ptr = __（3）__;       /*进入左子树*/
        }
    q = stacktop;
     __（4）__;           /*栈顶元素出栈*/
    visit( q );            /*visit是访问节点的函数，其具体定义省略*/
    ptr = __（5）__;       /*进入右子树*/
    free( q );             /*释放原栈顶元素的节点空间*/
    }
    return 0;
}/*InOrder*/
```

例题4分析

对非空二叉树进行中序遍历的方法是：先中序遍历根节点的左子树，然后访问根节点，最后中序遍历根节点的右子树。用递归方式描述的算法如下：

```
void In_Order_Traversing( BiTree root )
{  //root是指向二叉树根节点的指针
if( root!=NULL ){
    In_Order_Traversing( root->LeftChild );
    visit( root );
    In_Order_Traversing( root->RightChild );
}
}
```

从以上算法的执行过程可知，从树根出发进行遍历时，递归调用In_Order_Traversing（root->LeftChild）使得遍历过程沿着左孩子分支一直走向下层节点，直到到达二叉树中最左下方的节点（设为f）的空左子树为止，然后返回f节点，再由递归调用In_Order_Traversing（root->RightChild）进入f的右子树，并重复以上过程。在递归算法执行过程中，辅助实现递归调用和返回处理的控制栈实际上起着保存从根节点到当前节点的路径信息。

用非递归算法实现二叉树的中序遍历时，可以由一个循环语句实现从指定的根节点出发，沿着左孩子分支一直到头（到达一个没有左子树的节点）的处理，从根节点到当前节点的路径信息（节点序列）可以明确构造一个栈来保存。

本题目的难点在于将栈的实现和使用混合在一起来处理，而且栈采用单链表存储结构。下面分析题中给出的代码。

空（1）是遍历的条件之一，由于另外一个条件stacktop！=NULL初始时是不成立的，因此空（1）所表示的条件必须满足，由于是对非空二叉树进行遍历，显然该条件代表二叉树非空。即ptr！=NULL或其等价表示形式。

临时指针ptr初始时指向整个二叉树的根节点，此后用以下代码表示一直沿左孩子指针链向下走的处理。临时指针q用于在链栈中加入新元素时使用。处理思路是：若当前节点有左子树，则将当前节点的指针存入栈中，然后进入当前节点的左子树。入栈时，先申请元素在链栈中的节点空间，然后设置节点数据域的值（即当前节点的指针），最后将新申请的节点加入链栈首部。

```
while( ptr!=NULL ){
    q = ( StNode* )malloc( sizeof( StNode ) );  /*为新入栈的元素创建节点*/
    if( q = =NULL )            /*若创建新节点失败，则退出*/
      return -1;
    q->elem = ptr;             /*在栈顶保存指向当前节点的指针*/
    q->link = stacktop;        /*新节点加入栈顶*/
stacktop = q;                  /*更新栈顶指针，即stacktop指向新的栈顶*/
ptr = ptr->lchild;             /*进入当前节点的左子树*/
}
```

当上述过程进入一棵空的子树时（ptr为空指针），循环结束。此后，应该从空的子树返回其父节点并进行访问。由于进入空的左子树前已将其父节点指针压入栈中，因此，栈顶元素即为该父节点，对应的处理就是弹栈。相应地，在链栈中要删除表头节点并释放节点空间：

```
q = stacktop;                  /*q指向链栈中需要删除的节点，即栈顶元素*/
stack = stacktop->link;    /*栈顶元素出栈*/
visit( q );                    /*访问节点*/
free( q );                     /*释放节点空间*/
```

由于还需要通过q指针进入被删除节点的右子树，因此，释放节点空间的操作free(q)操作之前，使ptr指向q所指节点的右子树指针，以得到被删除节点的数据域信息，即空（5）所在语句ptr = q->elem->rchild。

指针是C语言中灵活且非常强大的工具，是否熟练掌握C语言的判断条件之一就是对指针的理解和使用。软件设计师需要熟练掌握这些内容。

例题4参考答案

（1）ptr!=NULL，或ptr!=0，或ptr

（2）q->link = stacktop

（3）ptr->lchild

（4）stacktop = stacktop->link，或stacktop = q->link

（5）q->elem->rchild

上一节　本书简介　下一节

考前必练

根据考试大纲和近5年的试题出题的研究方向，特别精心准备了考前必做练习题，从这些必做的考前练习题中，希望帮助考生能把握住考试的考题方向。

　作者：希赛教育软考学院　来源：希赛网　2014年05月06日

试题1

考前必做的练习题

函数int Toplogical(LinkedWDigraph G)的功能是对图G中的顶点进行拓扑排序，并返回关键路径的长度。其中图G表示一个具有n个顶点的AOE网，图中顶点从1～n依次编号，图G的存储结构采用邻接表表示，其数据类型定义如下：

```
typedef struct Gnode{    /*邻接表的表节点类型*/
  int adjvex;            /*邻接顶点编号*/
  int weight;            /*弧上的权值*/
  struct Gnode *nextarc;  /*指示下一个弧的节点*/
}Gnode;
typedef struct Adjlist{  /*邻接表的头节点类型*/
  char vdata;            /*顶点的数据信息*/
  struct Gnode *Firstadj;  /*指向邻接表的第一个表节点*/
}Adjulist;
typedef struct LinkedWDigraph{  /*图的类型*/
  int n, e;              /*图中顶点个数和边数*/
  struct Adjlist *head;   /*指向图中第一个顶点的邻接表的头节点*/
}LinkedWDigraph;
```

例如，某AOE网如图4-39所示，其邻接表存储结构如图4-40所示。

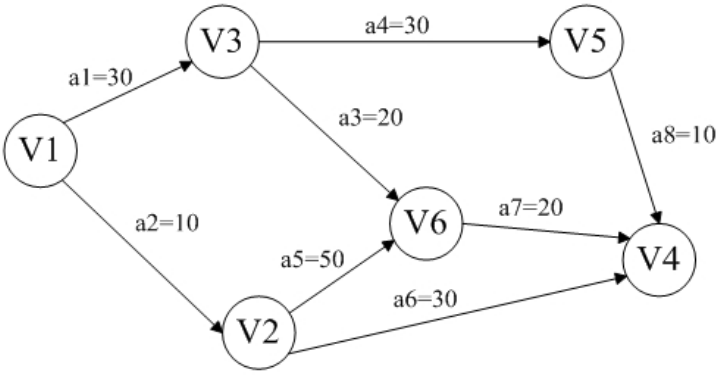


图4-39　AOE网

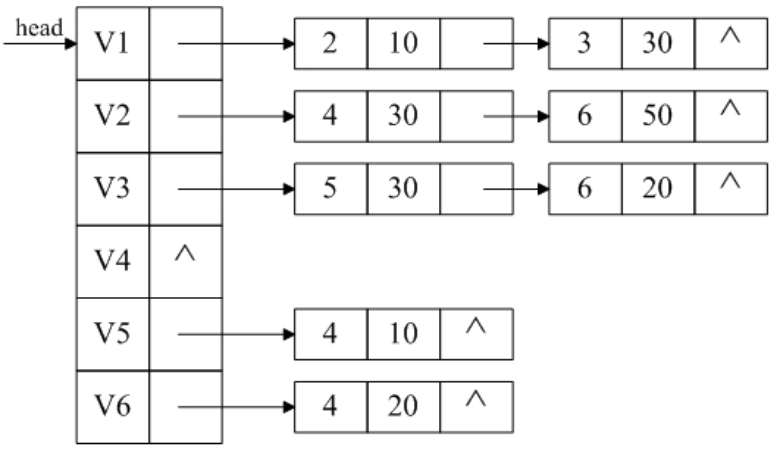


图4-40 AOE网邻接表存储结构

【函数代码】

```
int Toplogical( LinkedWDigraph G )
{
Gnode *p;
int j, w, top = 0;
int *Stack，*ve，*indegree;
ve = ( int* )malloc( ( G.n+1 )*sizeof( int ) );
indegree = ( int* )malloc( ( G.n+1 )*sizeof( int ) );  /*存储网中各项点的入度*/
Stack = ( int* )malloc( ( G.n+1 )*sizeof( int ) );    /*存储入度为0的顶点的编号*/
if( ！ve|| ! indegree|| ! Stack ） exit( 0 );
for( j = 1; j< = G.n; j++ ){
   ve[j] = 0;  indegree[j] = 0;
}/*for*/
for( j = 1; j< = G.n; j++ ){   /*求网中各项点的入度*/
   p = G.head[j].Firstadj;
   while( p ){
  （1）  ;
p = p->nextarc;
   }/*while*/
 }/*for*/
for( j = 1; j< = G.n; j++ ）/*求网中入度为0的顶点并保存其编号*/
    if( !indegree[j] ） Stack[++top] = j;
while( top>0 ){
    w =  （2）  ;
    printf(  “%c”，G.head[w].vdata );
    p = G.head[w].Firstadj;
    while( p ){
         （3）  ;
        if( !indegree[p->adjvex] )
```

```
            Stack[++top] = p->adjvex;
          if(   （4）   )
            ve[p->adjvex] = ve[w]+p->weight;
          p = p->nextarc;
        }/*while*/
      }/*while*/
      return  （5）  ;
    }/*Toplogical*/
```


第 4 章：数据结构设计　作者：希赛教育软考学院　来源：希赛网　2014年05月06日

试题2

在一个分布网络中，资源（石油、天然气、电力等）可从生产地送往其他地方。在传输过程中，资源会有损耗，例如，天然气的气压会减少，电压会降低。电压会降低，我们将需要输送的资源信息称为信号。在信号从信源地送往消耗地过程中，仅能容忍一定范围的信号衰减，称为容忍值。分布网络可表示为一个树型结构，如图4-41所示。信号源是树根，树中的每个结点（除了根）表示一个可以放置放大器的子结点，其中某些结点同时也是信号消耗点，信号从一个结点流向其子结点。

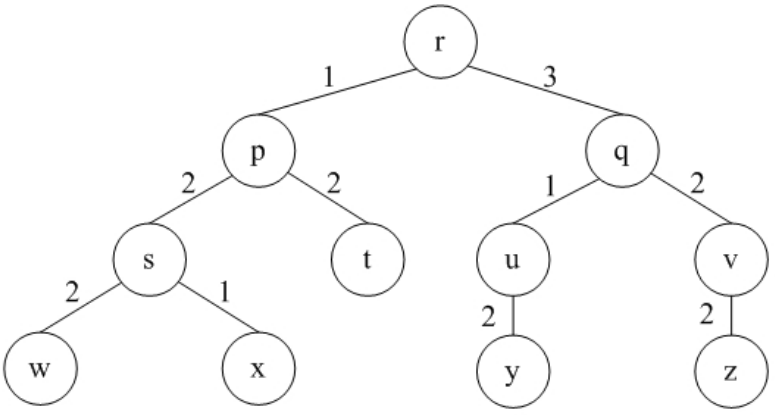


图4-41　分布网络的树型结构

每个结点有一个d值，表示从其父结点到该结点的信号衰减量。例如，在图4-39中，结点w、p、q的d值分别为2、1、3，树根结点表示信号源，其d值为0。

每个结点有一个M值，表示从该结点出发到其所有叶子的信号衰减量的最大值。显然，叶子结点的M值为0。对于非叶子结点j，M（j）=max{ M（k）+d（k）| k是j的孩子结点 }。在此公式中，要计算结点的M值，必须先算出其所有子结点的M值。

在计算M值的过程中，对于某个结点i，其有一个子结点k满足d（k）+M（k）大于容忍值，则应在k处放置放大器；否则，从结点i到某叶子结点的信号衰减量会超过容忍值，使得到达该叶子结点时信号不可用，而在结点i处放置放大器并不能解决到达叶子结点的信号衰减问题。

例如，在图4-42中，从结点p到其所有叶子结点的最大衰减值为4。若容忍值为3，则必须在s处

放置信号放大器，这样可使得结点p的M值为2。同样，需要在结点q、v处放置信号放大器，如图4-42中阴影结点所示。若在某结点放置了信号放大器，则从该结点输出的信号与信号源输出的信号等价。

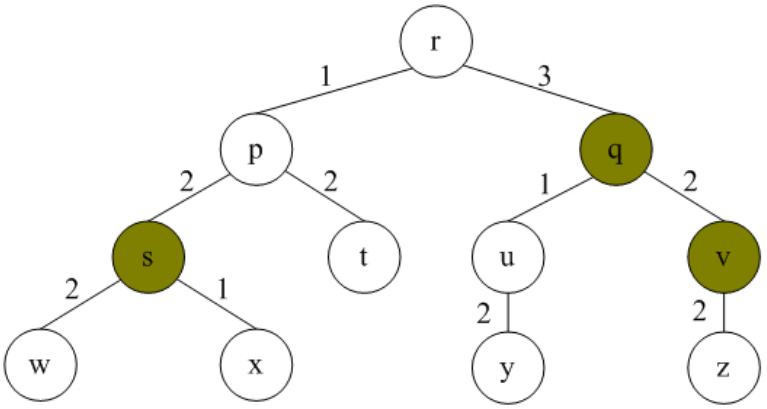


图4-42 带阴影的树型结构

函数placeBoosters(TreeNode *root)的功能是：对于给定树型分布网络中各个结点，计算其信号衰减量的最大值，并确定应在树中哪些结点放置信号放大器。

全局变量Tolerance保存信号衰减容忍值。

树的结点类型定义如下：

```
typedef struct TreeNode {
int id;                 /*当前结点的识别号*/
int ChildNum;           /*当前结点的子结点数目*/
int d;                  /*父结点到当前结点的信号衰减值*/
struct TreeNode **childptr; /*向量，存放当前结点到其所有子结点的指针*/
int M;                  /*当前结点到其所有子结点的信号衰减值中的最大值*/
bool boost;             /*是否在当前结点放置信号放大器的标志*/
}TreeNode;
```

【函数代码】

```
void placeBoosters( TreeNode *root )
{ /*计算root所指结点处的衰减值，如果衰减量超出了容忍值，则放置放大器*/
TreeNode *p;
int i, degradation;
if( __（1）__ ){
    degradation = 0; root->M = 0;
    i = 0;
    if( i> = root->ChildNum )
       return;
    p = __（2）__ ;
    for( ; i< root->ChildNum && p; i++，p = __（3）__ ){
       p->M = 0;
     __（4）__ ;
       if( p->d + p->M > Tolerance ){ /*在p所指结点中放置信号放大器*/
```

```
p->boost = true;
p->M = 0;
}
        if( p->d + p->M > degradation )
            degradation = p->d + p->M;
        }
        root->M = __( 5 )__ ;
}
}
```


第 4 章：数据结构设计　作者：希赛教育软考学院　来源：希赛网　2014年05月06日

试题3

现有n（n<1000）节火车车厢，顺序编号为1,2,3，...，n，按编号连续依次从A方向的铁轨驶入，从B方向铁轨驶出，一旦车厢进入车站（Station）就不能再回到A方向的铁轨上；一旦车厢驶入B方向铁轨就不能再回到车站，其结构如图4-43所示，其中Station为栈结构，初始为空且最多能停放1000节车厢。

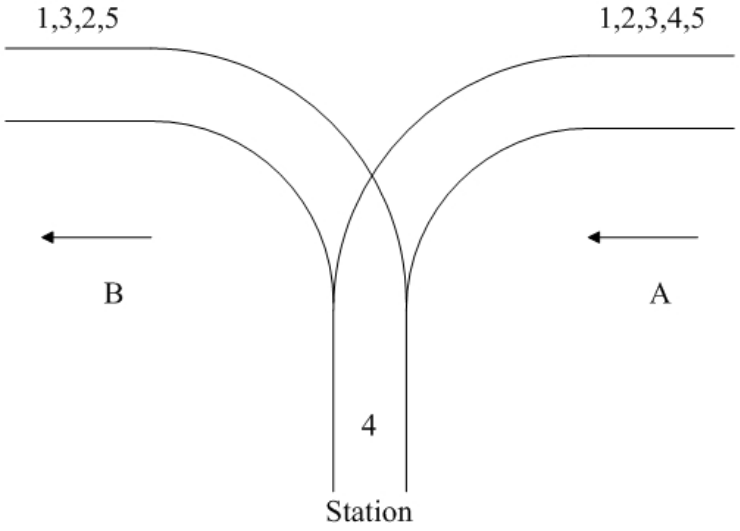


图4-43 火车进站示意图

下面的C程序判断能否从B方向驶出预先指定的车厢序列，程序中使用了栈类型STACK，关于栈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)：若是空栈则返回1，否则返回0

【程序代码】

```
#include<stdio.h>
/*此处为栈类型及其基本操作的定义，省略*/
int main( )
{
 STACK station;
 int state[1000];
 int n;                    /*车厢数*/
 int begin, i, j, maxNo;     /*maxNo为A端正待入栈的车厢编号*/
 printf( "请输入车厢数：" );
 scanf( "%d"，&n );
 printf( "请输入需要判断的车厢编号序列（以空格分隔）：" );
 if( n<1 )return -1;
 for( i = 0; i<n; i++ ) /*读入需要驶出的车厢编号序列，存入数组state[]*/
    scanf( "%d"，&state[i] );
 （1）; /*初始化栈*/
 maxNo = 1;
for( i = 0; i<n; i++ ){/*检查输出序列中的每个车厢号state[i]是否能从栈中获取*/
     if( （2） ){/*当栈不为空时*/
if( state[i] = =Top( station ) ){/*栈顶车厢号等于被检查车厢号*/
            printf( "%d"，Top( station ) );
            Pop( &station );   i++;
        }else
            if(  （3）  ){
                printf( "error\n" );
                return 1;
            }else {
                begin =（4）;
                for( j = begin+1; j< = state[i]; j++ ){
                    Push( &station, j );
                }
            }
}
else { /*当栈为空时*/
          begin = maxNo;
          for( j = begin; j< = state[i]; j++ ){
              Push( &station, j );
          }
          maxNo =（5）;
```

```
}
}
printf( “OK” );
return 0;
}
```


第 4 章：数据结构设计　作者：希赛教育软考学院　来源：希赛网　2014年05月06日

试题4

散列文件的存储单位称为桶（Bucket）。假如一个桶能存放m个记录，当桶中已有m个同义词（散列函数值相同）的记录时，存放第m+1个同义词会发生“溢出”。此时需要将第m+1个同义词存放到另一个称为“溢出桶”的桶中。相对地，称存放前m个同义词的桶为“基桶”。溢出桶和基桶大小相同，用指针链接。查找指定元素记录时，首先在基桶中查找。若找到，则成功返回，否则沿指针到溢出桶中进行查找。

例如：设散列函数为Hash（Key）=Key mod 7，记录的关键字序列为15,14,21,87,96,293,35,24,149,19,63,16,103,77,5,153,145,356,51,68,705,453，建立的散列文件内容如图4-44所示。

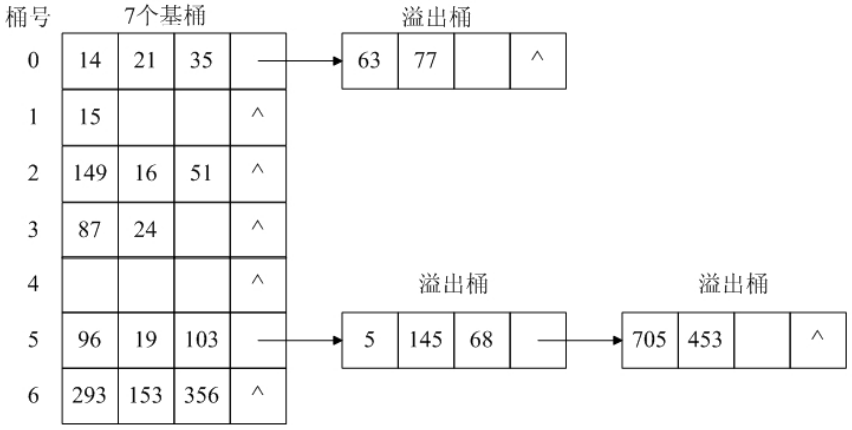


图4-44 散列文件内容示意图

为简化起见，散列文件的存储单位以内存单元表示。

函数InsertToHashTable(int NewElemKey)的功能是：若新元素NewElemKey正确插入散列文件中，则返回值1；否则返回值0。采用的散列函数为Hash(NewElemKey) = NewElemKey%P，其中，P为设定的基桶数目。函数中使用的预定义符号如下：

```
#define NULLKey -1          /*散列桶的空闲单元标识*/
#define P 7                 /*散列文件中基桶的数目*/
#define ITEMS 3             /*基桶和溢出桶的容量*/
typedef struct BucketNode {   /*基桶和溢出桶的类型定义*/
 int KeyData[ITEMS];
```

```
 struct BucketNode *Link;
}BUCKET;
BUCKET Bucket[P];              /*基桶空间定义*/
【函数】
int InsertToHashTable( int NewElemKey )
{  /*将元素NewElemKey插入散列桶中，若插入成功则返回0，否则返回-1*/
/*设插入第一个元素前基桶的所有KeyData[]、Link域已分别初始化为NULLKEY、NULL*/
int Index;          /*基桶编号*/
int i, k;
BUCKET *s, *front, *t;
  （1）;
for( i = 0; i<ITEMS; i++ )  /*在基桶查找空闲单元，若找到则将元素存入*/
       if( Bucket[Index].KeyData[i] = =NULLKEY ){
          Bucket[Index].KeyData[i] = NewElemKey; break;
        }
if(  （2）  ) return 0;
/*若基桶已满，则在溢出桶中查找空闲单元，若找不到则申请新的溢出桶*/
  （3）;
t = Bucket[Index].Link;
if( t!=NULL ){ /*有溢出桶*/
        while( t!=NULL ){
           for( k = 0; k<ITEMS; k++ )
              if( t->KeyData[k] = =NULLKEY ){ /*在溢出桶链表中找到空闲单元*/
                 t->KeyData[k] = NewElemKey; break;
               }/*if*/
           if(  （4）  ) t = t->Link;
           else break;
         }/*while*/
}/*if*/
if(  （5）  ){ /*申请新溢出桶并将元素存入*/
        s = ( BUCKET * )malloc( sizeof( BUCKET ) );
        if( !s )return -1;
        s->Link = NULL;
        for( k = 0; k<ITEMS; k++ )
           s->KeyData[k] = NULLKEY;
        s->KeyData[0] = NewElemKey;
       （6）;
}/*if*/
return 0;
```

}/*InsertToHashTable*/

　作者：希赛教育软考学院　来源：希赛网　2014年05月06日

练习题解析

试题1分析

此C语言程序题考点为拓扑排序和关键路径。在解题之前，先了解几个概念。

（1）AVO网络

一个大工程中有许多项目组，有些项目的实行存在先后关系，某些项目必须在其他一些项目完成之后才能开始实行。工程项目实行的先后关系可以用一个有向图来表示，工程的项目称为活动，有向图的顶点表示活动，有向边表示活动之间开始的先后关系。这种有向图称为用顶点表示活动网络，简称AOV网络，图4-45就是一个AOV网络。

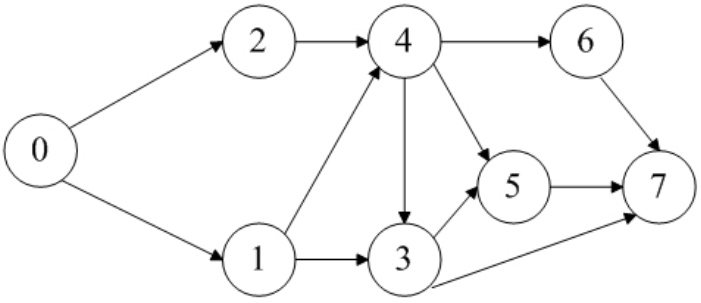


图4-45　AOV网络图

（2）拓扑排序

对AOV网络的顶点进行拓扑排序，就是对全部活动排成一个拓扑序列，使得如在AOV网络中存在一条弧（i，j），则活动i排在活动j之前。对图4-45的顶点进行拓扑排序，可以得到多个不同的拓扑序列，如02143567，01243657，02143657，01243567。

（3）AOE网络

利用AOV网络，对其进行拓扑排序能对工程中的活动的先后顺序做出安排。但一个活动的完成总需要一定的时间，为了能估算某个活动的开始时间，找出那些影响工程完成时间最大的活动，需要利用带权的有向图。图中的顶点表示一个活动结束的事件，图中的边表示活动，边上的权表示完成该活动所需的时间，这种用边表示活动的网络称为AOE网络。图4-46表示一个具有8个活动的某个工程的AOE网络。图中有6个顶点，分别表示事件V1到V6，其中V1是工程的开始状态，V4是工程的结束状态。边上的权表示完成该活动所需的时间。

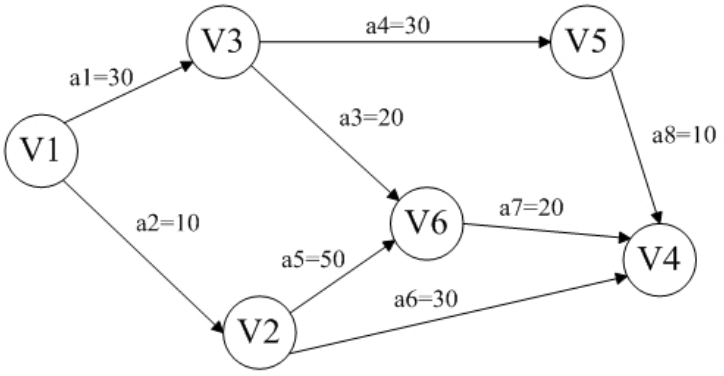


图4-46 AOE网络图

（4）关键路径

在AOE网络中某些活动可以并行地进行，所以完成工程的最少时间是从开始顶点到结束顶点的最长路径长度，我们称从开始顶点到结束顶点的最长路径为关键路径，关键路径上的活动为关键活动。如图4-46的AOE网络的关键路径为V1-V2-V6-V4，关键路径长度为80。

了解了上面的这些概念以后，解题就非常容易了。

从程序中的注释可知下段程序的作用是求网中各顶点的入度。

```
for( j = 1; j< = G.n; j++ ){
    p = G.head[j].Firstadj;
    while( p ){
      （1）
     p = p->nextarc;
}
}
```

从已知的代码结合邻接表来看，首先p指向了邻接表弟1个节点V1的Firstadj域，然后用while循环遍历了V1的Firstadj指向的链表。链表中的记录的，是当前节点可到达的节点，只要统计这些节点在邻接表中所有链表中出现的次数，就可知道其入度。又因为程序前面有：

```
indegree=( int* )malloc( ( G.n+1 )*sizeof( int ) ); /*存储网中各顶点的入度*/
```

所以第（1）空应填“indegree[p->adjvex]++”。

接下来看第（2）空，第（2）空是给w赋值，接下来是打印第w号节点的数据，这也就意味着w号节点是拓扑排序选出来的节点，所以w必是一个入度为0的节点。然而在此之前已经有程序把所有的入度为0的节点保存在Stack数组中了，而且Stack数组是模拟的一个栈，其控制指针只有top，所以我们应该从Stack中取出栈顶元素赋值给w。所以第（2）空填“Stack[top--]”。注意这里不能用“Stack[--top]”，因为前面有入栈语句“Stack[++top]=j;”。

接下来看下面的程序段。

```
while( p ){
      （3）；
    if( !indegree[p->adjvex] )
       stack[++top] = p->adjvex;
    if （4）
       ve[p->adjvex] = ve[w]+p->weight;
    p = p->nextarc;
```

}

此段程序的作用是：把选出节点所关联的边去掉，即原来V1有到V3的边a1=30和到V2的边a2=10，当V1节点选出以后，a1，a2也要随之消失。这时V3和V2的入度要更新，也就是把V3和V2的入度分别减1。所以第（3）空应填“indegree[p->adjvex]--”。第（4）空看起来比较棘手，因为前面没有说明ve是用于存放什么数据的，所以我们应该从整个程序的功能来推敲。程序有一项功能是要返回关键路径的长度，但到目前为止，都没有程序段完成此项功能。所以可以断定

if __（4）__

ve[p->adjvex]=ve[w]+p->weight;

的功能是计算关键路径长度。ve的初值最开始都是0，而且关键路径是要找从开始点到结束点的最长路径。所以我们只要保证每到一个点vx，ve[vx]中存的都是最长路径即可。也就是说，首先选出的是V1，从V1到V2只有一条路径，所以ve[v2]=a2=10，从V1到V3只有一条路径，所以ve[v3]=a1=30。然后选出V2节点，V2选出以后，因为V2到V6有a5=50，所以现在到V6的最长路径为ve[v6]=ve[v2]+a5=60。...经过若干步后，当程序选中V3节点时，会产生到V6的另外一条路径V1-V3-V6，这条路径的长度为50，这条路径比现存的路径长度ve[v6]短，所以单纯的更新语句“ve[p->adjvex]=ve[w]+p->weight”不能正确保存最长路径，为了保证ve中保存的路径最长，应该有判断(ve[w]+p->weight)>ve[p->adjvex]。所以第（4）空应该填“(ve[w]+p->weight)>ve[p->adjvex]”。

第（5）空很明显是要返回关键路径。不过具体是要返回哪个节点的最长路径长度，才是整个图的关键路径呢？这一点可以从关键路径的定义着手：“我们称从开始顶点到结束顶点的最长路径为关键路径”，所以最后一个选出节点的ve存放的便是关键路径。所以第（5）空应填“ve[w]”。

试题1参考答案

（1）indegree[p->adjvex]++

（2）Stack[top--]

（3）indegree[p->adjvex]--

（4）（ve[w]+p->weight）>ve[p->adjvex]

（5）ve[w]

试题2分析

本题考查数据结构中树的基本操作。

题目【说明】部分对树的结构，以及程序的目的有比较明确的说明。本程序的功能是在合适的位置安放信号放大器。通过对题目【说明】部分的分析可以得知放置信号放大器的原则是判断当前节点的d[i]+M[i]是否大于容忍值。若大于，则在i处设信号放大器。如：对于结点s，d[s]=2，M[s]=2，d[s]+M[s]=4，此时若容忍值为3，则d[s]+M[s]>3，需要在此放置一个信号放大器。但在题目中，结点的M值是未提供的，所以程序应完成两个操作：第一个是求出结点的M值；第二个是确定当前结点是否需要加信号放大器。

下面进行具体的代码分析。

第（1）空是一个判断条件，当条件成立时，才进入程序主体。这个空非常容易，在对树进行操作过程中，只有当前结点不为空结点时才有必要进行相应的操作，所以此处应填：root。

通过对程序主体进行分析可知，指针p用于指向子结点，其初始值应为第一个子结点“childptr[0]”的指针，因此第（2）空应填：root->childptr[0]，此后p依次指向下一个子结

点，因此第（3）空填入：root->childptr[i]。

第（4）空是关键的一步，由于“要计算结点的M值，必须先算出其所有子结点的M值”，所以需要用递归，利用递归来计算子结点的M值，故此处填：placeBoosters(p)。

第（5）空非常容易，是将已求得的M值，存入当前结点的root->M中，由于程序中计算出来的M记录在degradation中，所以此处填：degradation。

试题2参考答案

（1）root

（2）root->childptr[0]

（3）root->childptr[i]

（4）placeBoosters(p)

（5）degradation

试题3分析

本题考查栈数据结构的应用和C程序设计基本能力。

栈的运算特点是先进先出。在本题中，入栈序列为1、2、…、n-1、n，出栈序列保存在state[]数组中，state[0]记录出栈序列的第1个元素，state[1]记录出栈序列的第2个元素，依次类推。程序采用模拟元素入栈和出栈的操作过程来判断出栈序列是否恰当。需要注意的是，对于栈，应用时不一定是所有元素先入栈后，再逐个进行出栈操作，也不一定是进入一个元素接着就出来一个元素，而是栈不满且输入序列还有元素待进入就可以进栈，只要栈不空，栈顶元素就可以出栈，从而使得唯一的一个入栈序列可以得到多个出栈序列。当然，在栈中有多个元素时，只能让栈顶的元素先出栈，栈中其他的元素才能从顶到底逐个出栈。本题中入栈序列和出栈序列的元素为车厢号。

空（1）处堆栈进行初始化，根据题干中关于栈基本操作的说明，调用InitStack初始化栈，由于形参是指针参数，因此实参应为地址量，即应填入“InitStack(&station)”。

当栈不空时，就可以令栈顶车厢出栈，空（2）处应填入“！IsEmpty(station)”。

栈顶车厢号以Top(station)表示，若栈顶车厢号等于出栈序列的当前车厢号state[i]，说明截至到目前为止，出栈子序列state[0]～state[i]可经过栈运算获得。由于进栈时小编号的车厢先于大编号的车厢进入栈中，因此若栈顶车厢号大于出栈序列的当前车厢号state[i]，则对于state[i]记录的车厢，若它还在栈中，则此时无法出栈，因为它不在栈顶，所以出错；若它已先于当前的栈顶车厢出栈，则与目前的出栈序列不匹配，仍然出错，因此空（3）处应填入“state[i]<Top(station)”。

若栈顶车厢号小于出栈序列的当前车厢号state[i]，则说明state[i]记录的车厢还没有进入栈中，因此从入栈序列（A端）正待进入的车厢（即比栈顶车厢号正好大1）开始，直到state[i]记录的车厢号为止，这些车厢应依次进入栈中。程序中用以下代码实现此操作：

```
for( j = begin+1; j< = state[i]; j++ ){
    Push( &station, j );
}
```

显然，begin应获取栈顶车厢号的值，即空（4）处应填入“Top(station)”。

还有一种情况，就是待考查的出栈序列还没有结束而栈空了，则说明需要处理入栈序列，使其车厢入栈。程序中用maxNo表示A端正待入栈的车厢编号，相应的处理如下面代码所示：

```
begin = maxNo;
   for( j = begin; j< = state[i]; j++ ){
```

```
        Push( &station, j );
      }
```

接下来，A端正待入栈的车厢编号等于j或state[i]+1，即空（5）处应填入j或“state[i]+1”。

如果驶出的车厢编号序列是经由栈获得的，则程序运行时输出该序列及字符串“OK”，否则输出“error”而结束。

试题3参考答案

（1）InitStack(&station)

（2）！IsEmpty(station)

（3）state[i]<Top(station)

（4）Top(station)

（5）j或state[i]+1

试题4分析

从题目叙述和图的形象表达可以看出：题中提及的散列桶，只不过是一种加强型的采用拉链法处理冲突的散列表，普通的散列表一个空间存储一个元素，题中的散列表能存三个而已。

先来看第（1）空。该句紧挨的下面的for循环注释“在基桶查找空闲单元，若找到则将元素存入”给了我们很好的暗示，要存入元素，那么先得定好该元素要存入的地址，显然该空是要根据新元素确定散列地址，于是填Index=NewElemKey%P，或者Index=Hash(NewElemKey)，或者Index=NewElemKey%7。

题干中说函数InsertToHashTable(int NewElemKey)的功能是：若新元素NewElemKey正确插入散列文件中，则返回值1；否则返回值0。而程序中的注释又说“/*将元素NewElemKey插入散列桶中，若插入成功则返回0，否则返回-1*/”，这两处有矛盾，我们以后者为准，因为程序是跟注释一致的。

接下来看第（2）空。我们发现if语句之后，有“return 0;”语句，这表明，新元素插入成功所以才返回0，根据if语句上面紧挨的for循环可知，for循环用于在基桶中查询是否有空闲单元，如果有，则插入并用“break;”跳出整个for循环，新元素插入基桶中的第几个位置跳出之后的i就是几。例如图4-44中的14插入在0号基桶中的第1个位置（注意从0开始），可见只要插入基桶成功就有i<ITEMS。如果在基桶中没有找到空闲单元（如注释所说那样“若基桶已满，则……”），则退出for循环之后i的值就是3，不应该返回0。因此第（2）空填i<ITEMS或i<3（由对ITEMS的宏定义可知，其值是3）。

接着看第（3）空。其下数第一个if语句的功能就是“若基桶已满，并且有溢出桶则在溢出桶中查找空闲单元”，先不急着填这空，可往下看，读懂这段程序。从if语句中的while语句的for循环可以看出，for循环的功能就是在溢出桶链表中第1个有空闲单元的溢出桶中插入新元素。在该for循环之后，我们发现有语句“front=t;”，这表明front也指向t所指向的溢出桶，接下来我们发现一个if判断语句，然后是“t=t->Link;”，该句表明，如果第（4）空中的条件成立，t就后移一个位置指向下一个溢出桶，如果条件不成立就跳出整个while循环。而我们对比第（5）空后面的注释可以知道，while循环的功能就是依次在溢出桶链中依次查找到空闲单元，如今该句要跳出整个while循环，就表明在溢出桶链中已经查找到空闲单元，不需要继续依次查询到最后一个溢出桶（即直至t==NULL）。那么在什么情况下需要t指针后移一个位置呢？显然是当t当前所指的溢出桶没有空闲单元时要后移，而对while中的for循环我们可以看到，当k从1变化到ITEMS-1=2时若还没有找到空闲

单元就结束循环，结束循环之后k的值为ITEMS。这个很多人理解不了，关键是没理解for循环工作的机理，在此，我们假设进入某次循环时k=2，应该先判断条件“k<ITEMS;”是否成立，若成立，于是执行循环体，之后无条件执行“k++”，此时k的值为3，于是进入下一次循环，又先判断“k<ITEMS;”是否成立，显然不成立，于是for循环结束，不执行它的循环体。接下来应执行for循环体下面的语句，可见此时的k值是3！理清了上述思路，显然，第（4）空填k= = ITEMS或k>=ITEMS，ITEMS当然可用3代替。

从对t的初值可看出，它首先指向基桶的第一个溢出桶，根据第（5）空后面的注释可看出，第（5）空那个if语句的功能就是在基桶没有溢出桶时或者即使基桶有溢出桶但所有溢出桶都满了（即while循环查询到溢出桶链表末尾）时，申请新溢出桶并将新元素存入。于是，第（5）空填判断条件t = =NULL或者！t。新元素存在新的溢出桶s中，在基桶没有溢出桶时如何插入新的溢出桶s？显然是用指针front指向基桶，然后将基桶的指针域指向新溢出桶s，于是第（3）空必须填front = &Bucket[Index]或者front = Bucket+Index。根据上文意思可知，其中Bucket[Index]表示已满的编号为Index的基桶；第（6）空必须填front->Link = s。

至此，我们才可以看出while中的语句“front = t;”的作用：在基桶中有溢出桶的情况下，让front指向最后一个溢出桶，便于第（6）空中在溢出桶链表末尾插入新的溢出桶。

试题4参考答案

（1）Index=NewElemKey%P或者Index=Hash(NewElemKey)或者Index=NewElemKey%7

（2）i<ITEMS或者i<3

（3）front = &Bucket[Index]或者front = Bucket+Index

（4）k = =ITEMS或k>=ITEMS（ITEMS当然可用3代替）

（5）t = =NULL或！t

（6）front→Link = s

上一节　本书简介　下一节

作者：希赛教育软考学院　来源：希赛网　2014年05月06日

试题解答方法

考生在面对此类题目的解答时候，一定要主要以下几个方面，才能更好的解决问题。

（1）首先把试题通读一遍，找出知识考查点。

考生在接到题目后，首先不应该盲目的去填写有关题目的答案，而是应该把试题大概通读一遍，找出题目所要考查的内容到底是什么。用笔将考查的部分重点的圈出，以备引起注意。这部分的建议花费时间为5分钟

（2）确定考查点，回忆考查点点。

在确定了考试考查知识点后，迅速回忆一下在备战过程中的该部分知识点的一些性质，所涉及到的一些算法，比如：函数的编写，参数的类型和内容是什么，特别需要注意的地方是哪些。这部分建议花费时间为5分钟

（3）仔细阅读题目，根据试题表述的意义来进行填空。

在进行试题答题时候，结合题目所表述的意思，在填空处需要的语句一定要将前后语句放在一起去理解。有的时候遇见了某空白处不能确定填写答案的时候，可以暂时放下该处的填写，继续往下看，等填写完其他空白处的时候，再次结合题目意思把未填写的空白处补齐。这部分建议花费时间10-15分钟

（4）检查程序

在填写完所有的空白处后，再次回到题目，将题目和代码重新阅读一次。看看填写的内容是否满足题目的意思。这部分建议花费时间5-10分钟。

上一节　本书简介　下一节

第 5 章：算法设计　作者：希赛教育软考学院　来源：希赛网　2014年05月06日

考情分析

对软件设计而言，数据结构设计解决的是软件程序中的数据组织问题，如何确定数据处理的流程，即确定解决问题的步骤是软件设计的另一个重要问题。

计算机程序需要正确、详尽地描述问题的每个对象和处理逻辑，其中问题的对象由程序的数据结构、变量来描述，而用来描述问题处理逻辑的程序结构、函数和语句构成程序的算法（algorithm）。

算法是计算机程序的灵魂，算法和数据结构共同构成程序的两个重要方面。算法是一组确定的解决问题的步骤。它和机器以及语言并无关系，但最终还是需要使用特定的语言加以实现。

算法设计是计算机程序设计的核心内容之一，算法的优劣直接影响着程序的性能和运行效率。因此，算法设计能力在很大程度上决定了软件设计能力的高低，是软件设计师应具备的重要技能。

算法被公认为计算机科学的基石，算法的设计技术和分析技术是算法理论研究的主要内容。

从近几年的软件设计师考试题目来看，算法设计和数据流图、UML建模技术、数据库设计题目逐渐成为前四个必答题之一，算法设计的考查重点在于对常用算法的综合运用以及对算法复杂度等基本概念的理解。因此，在复习时应该注意加强对算法设计相关内容的理解与实践。

上一节　本书简介　下一节

第 5 章：算法设计　作者：希赛教育软考学院　来源：希赛网　2014年05月06日

考试大纲要求分析

关于算法设计，在考试大纲中对软件设计师的要求是熟练掌握常用算法。程序设计中常用的算法有迭代、穷举搜索、递推、递归、回溯、贪心、动态规划和分治等。

在考试大纲中，对软件设计师的考试要求是能够做到对常用算法的综合运用（指对所列知识要理解其确切含义及于其他知识的联系能够进行叙述和解释，并能在实际问题的分析、综合、推理和