五邑大学计算机学院

数据结构实验指导手册

2017 年 3 月

目　　录

说　　明

1．每人至少完成 4 个实验，实验内容可任选指导手册上的提供实验题目。

2．实验完成形式：现场检查+提交实验报告（电子版）。

3．实验报告格式要求：内容一律用宋体小四号字，程序代码用 Times New Roman 五号字。

4．实验报告 word 文档命名规范：学号_姓名_实验 x.doc，x 为实验序号。

实验一　线性表

1．实验目的

（1）了解线性表的逻辑结构特性是数据元素之间存在着线性关系，在计算机中表示这种关系有顺序存储结构和链式存储结构；

（2）掌握这两种存储结构的描述方法；

（3）掌握线性表的基本操作（查找、插入、删除）；

（4）考虑时间和空间复杂度来设计算法。

2．实验内容

（1）创建一个顺序表，存放在数组 A[N]中，元素的类型为整型，设计算法调整 A，使其左边的所有元素小于 0，右边的所有元素大于 0（要求算法的时间复杂度和空间复杂度均为 O(n)）。

（2）建立一个循环单链表，其节点有 prior，data 和 next 三个域，其中 data 为数据域，存放元素的有效信息，next 域为指针域，指向后继节点，prior 为指针域，它的值为 NULL。编写一个算法将此表改为循环双链表。

3．解题思路

（1）如图 1-1 所示，设立两个工作指针 i 和 j，i 由数组的左端向右移动，查找大于等于 0 的数，j 由数组的右端向左端移动，查找小于 0 的数，然后交换，如图 1-2 所示，直到 i>=j，调整结束。

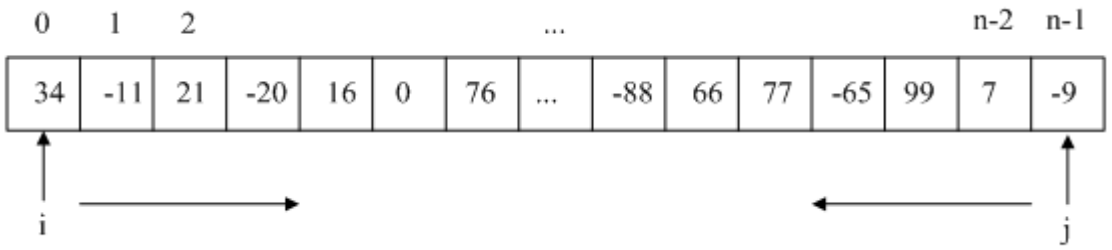

图 1-1

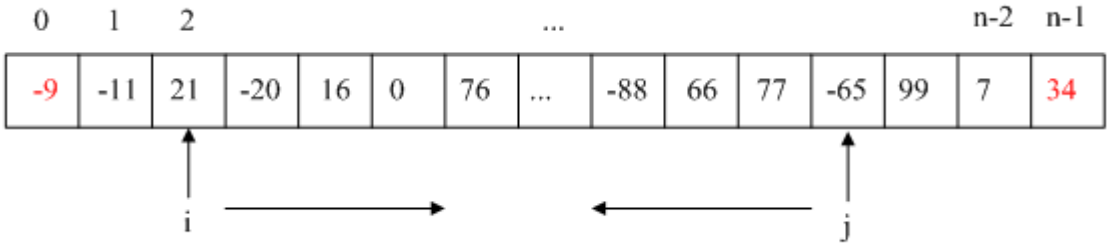

图 1-2

算法描述（C 语言）如下：

```
void quickSwapList(int a[],int n)
{
    int i = 0, j = n - 1 ;   // n 为顺序表的长度，即数的个数
    while( i < j )
    {
        while( a[i] < 0 && i < j   ) i ++ ; //  找到左边大于等于  0 的数
        while( a[j] >= 0 && i < j   ) j -- ; //  找到右边小于  0  的数
        if(i < j )                          //  交换
        {
            int t = a[i] ; a[i] = a[j] ; a[j] = t ;
        }
        i++;
        j--;
    }
}
```

（2）如图 1-3 所示，已建有一个单循环链表（带头结点），first 指向头结点。设立两个工作指针 p 和 q，分别指向头结点和第 1 个结点；执行 q->prior=p;，建立第 1 个结点的前驱指针，如图 1-4 所示；同步移动工作指针 p 和 q 分别指向下一个结点，如图 1-5 所示，建立 q 指向结点的前驱，直到 q==first 为止，再将头结点的前驱设为最后一个结点，如图 1-6 所示。

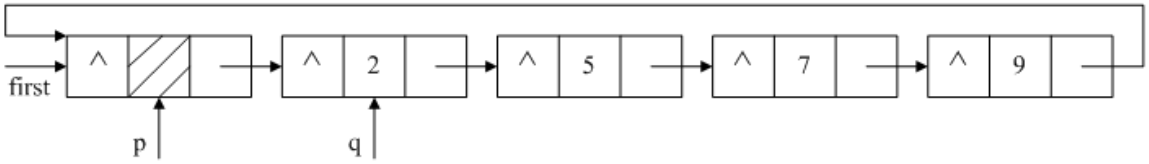

图 1-3

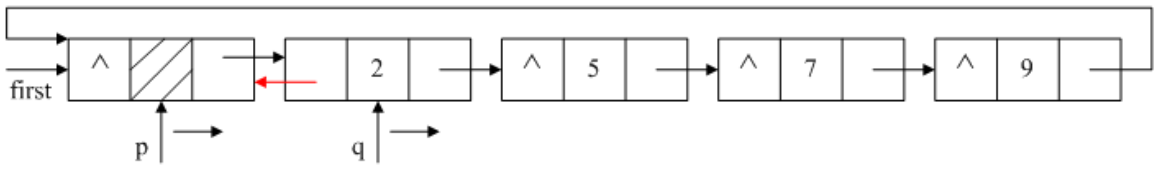

图 1-4

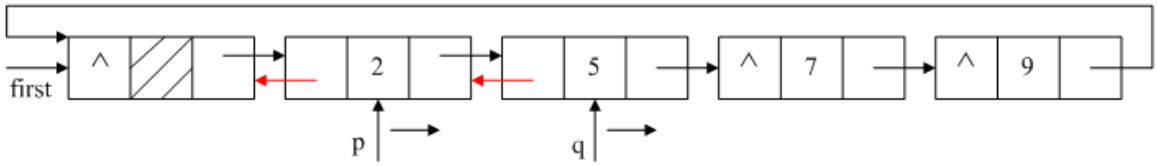

图 1-5

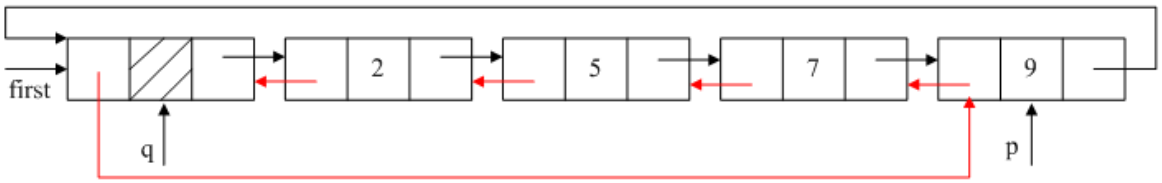

图 1-6

算法描述（C 语言）如下：

```
void singleCircleToDoubleCircleLinkList(struct Node *first)
{
    struct Node *p,*q;
    p=first;            //工作指针 p 指向头结点
    q=first->next;      //工作指针 q 指向第 1 个结点
    while(q!=first)
    {
        q->prior=p;  //置结点 q 的前驱为指针 p 指向的结点
        p=p->next;   //移动工作指针 p
        q=q->next;   //移动工作指针 q
    }
    q->prior=p;         //置头结点的前驱为最后一个结点
}
```

实验二　栈

1．实验目的

（1）理解栈的定义、特点及与线性表的异同；

（2）熟悉顺序栈的组织方法，栈满、栈空的判断条件及其描述；

（3）掌握栈的基本操作（进栈、退栈等）。

2．实验内容

（1）设计一个算法，将一般算术表达式转化为逆波兰表达式，并求逆波兰表达式的值。

（2）设计两个栈 S1、S2 都采用顺序栈方式，并且共享一个存储区[0，MaxLen-1]，为了尽量利用空间，减少溢出的可能，可采用栈顶相向、迎面增长的存储方式，如图 2-1 所示。设计一个有关栈的入栈和出栈算法。

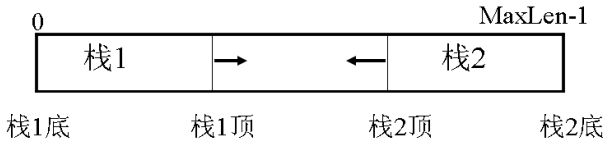

图 2-1

3．解题思路

（1）一般算术表达（中缀表达），如#3*(4+2)/2-5#，#为表达式界定符，逆波兰表达式（后缀表达式），如前述表达的后缀表达式为：3 4 2 + * 2 / 5 -。设中缀表达式的运算符有+、-、*、/、#五种，运算符的优先级别从高到低为()、*、/、+、-、#；有括号先算括号内，再算括号外的，多层括号由内向外进行。中缀表达式转换为后缀表达式需要用到一个临时栈 optr 暂存运算符。

转换算法描述（伪代码）如下：

1．将栈 optr 初始化为#；

2．从左至右依次扫描表达式的每个字符，执行下述操作

2.1 若当前字符是运算对象，则输出该字符，处理下一个字符；

2.2 若当前字符是运算符且优先级别比栈 optr 的栈顶运算符的优先级高，则将该字符入栈 optr，处理下一个字符；

2.3 若当前字符是运算符且优先级别比栈 optr 的栈顶运算符优先级别低，则将栈 optr 的栈顶元素弹出并输出；

2.4 若当前字符是运算符且优先级别与栈 optr 的栈顶运算符的优先级相等，则将栈 optr 的栈顶元素弹出，处理下一个字符。

后缀表达式求值，由于后缀表达中所有的计算按运算符出现的顺序从左向右进行，不用再考虑运算符的优先级。设定一个临时堆栈 opnd 暂存计算过程的中间结果。

后缀表达式计算算法描述（伪代码）如下：

1．初始化栈 opnd 为空；

2．从左到右依次扫描表达式的每一个字符，执行下述操作；

2.1 若当前是运算对象，则入栈 opnd，处理下一个字符；

2.2 若当前字符是运算符，则从栈 opnd 出栈两个运算对象，执行运算并将结果入栈 opnd，处理下一个字符；

3．输出栈 opnd 的栈顶元素，即表达式的运算结果。

（2）两栈共享存储空间，需要解决的关键问题在于如何判断栈满和栈空，由于两栈共享存储空间，入栈和出栈的时候还需要区分是哪个栈。设 lsTop 是左栈（数组下标为 0 的一端为栈底）栈顶指针，rsTop 为右栈（数组下标为 MaxLen-1 的一端为栈底）栈顶指针，显然，当两个栈都为空时，lsTop==-1，rsTop==MaxLen，如图 2-2 所示；当栈满时，rsTop 和 lsTop 存在以下关系：rsTop==lsTop+1，如图 2-3 所示。

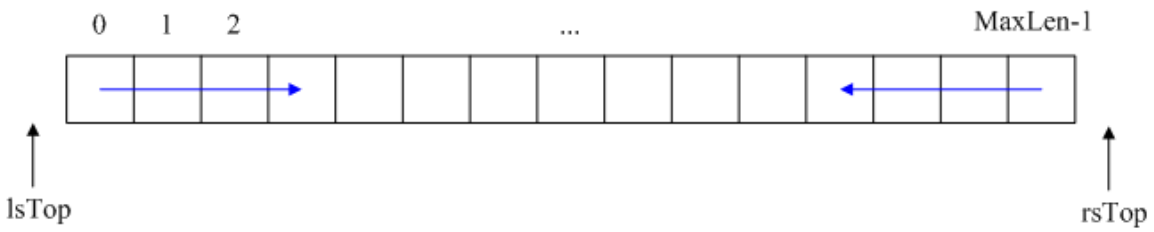

图 2-2

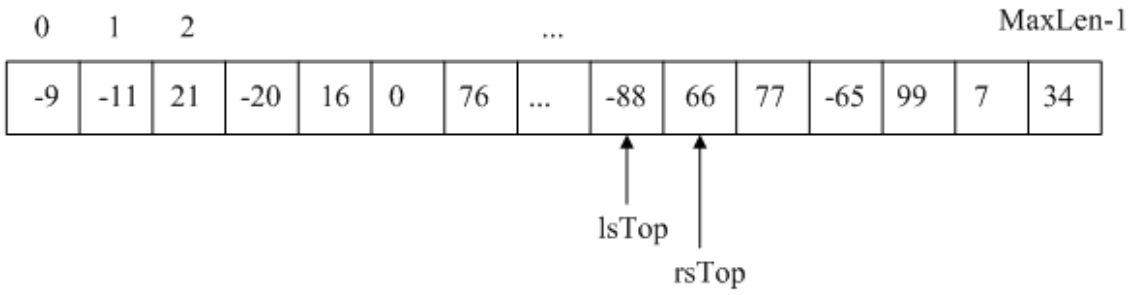

图 2-3

入栈算法描述（C 语言）如下：

```
//入栈，s=1，入左栈，s=2，入右栈，x 为入栈元素
void push(int s,int x)
{
    if(rsTop==lsTop+1)//栈满
    {
        printf("栈已满！\n");
        return;
    }
    if(s==1)//入左栈
    {
        lsTop++;
        dsStack[lsTop]=x;
    }
    else if(s==2)//入右栈
    {
        rsTop--;
        dsStack[rsTop]=x;
    }
    else
        printf("该栈不存在！\n");
    return;
}
```

出栈算法描述（C 语言）如下：

```
//出栈，s=1，出左栈，s=2，出右栈，x 返回出栈元素，成功函数返回 1，否则返回
int pop(int s,int *x)
{
    if(s==1)//出左栈
    {
        if(lsTop==-1)
        {
            printf("栈已空！\n");
```

```
            return 0;
        }
        *x=dsStack[lsTop];
        lsTop--;
    }
    else if(s==2)//出右栈
    {
        if(rsTop==MaxLen)
        {
            printf("栈已空！\n");
            return 0;
        }
        *x=dsStack[rsTop];
        rsTop++;
    }
    else
    {
        printf("该栈不存在！\n");
        return 0;
    }
    return 1;
}
```

实验三　队列

1．实验目的

（1）理解队列的定义、特点及与线性表的异同；

（2）熟悉队列的组织方法，队列满、队列空的判断条件及其描述；

（3）掌握队列的基本操作（入队、出队等）。

2．实验内容

（1）假设以数组 sequ[MaxSize]存放环形队列的元素，同时 Rear 和 Len 分别指示环形队列中队尾元素的位置和内含元素的个数。设计相应的入队和出队算法。

（2）某汽车轮渡口，过江渡船每次能载 10 辆车过江。过江车辆分别为客车类和货车类，上船有如下规定：同类车先到先上船，客车先于货车上渡船，且每上 4 辆客车，才允许上一辆货车；若等待客车不足 4 辆则以货车代替；若无货车等待则允许客车都上船。设计一个算法模拟渡口管理。

3．解题思路

（1）解决问题的关键在于以下两点：如何使用软件的方法构造环形队列和如何通过 Rear 和 Len 来确定队头元素所在的位置。对于第 1 点，通过模运算来实现，即 Rear=(Rear+1)%MaxSize；第 2 个问题，当队尾元素的下标大于队头元素的下标时，如图 3-1 所示，显然，Rear-Len+1 即为队头元素的下标；当队尾元素的下标小于队头元素的下标时，如图 3-2 所示，此时队头元素的下标为：(Rear+MaxSize)-Len+1；为使这两种情况统一处理，采用以下表达式来计算队头元素的下标：((Rear+MaxSize)-Len+1)%MaxSize。队空和队满的问题通过 Len 的值即可判断出，Len=0 显然是队空，Len=MaxSize 则为队满。

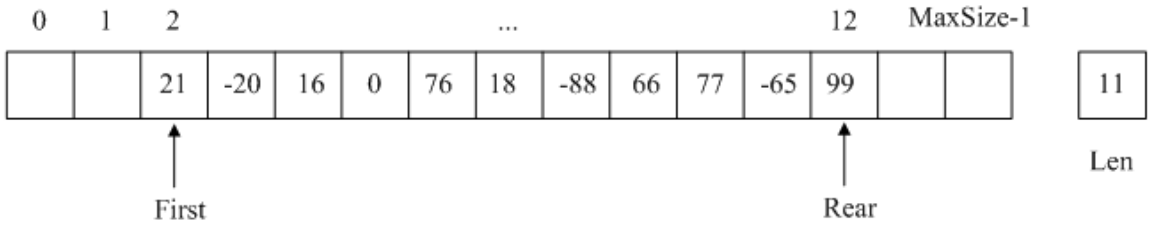

图 3-1

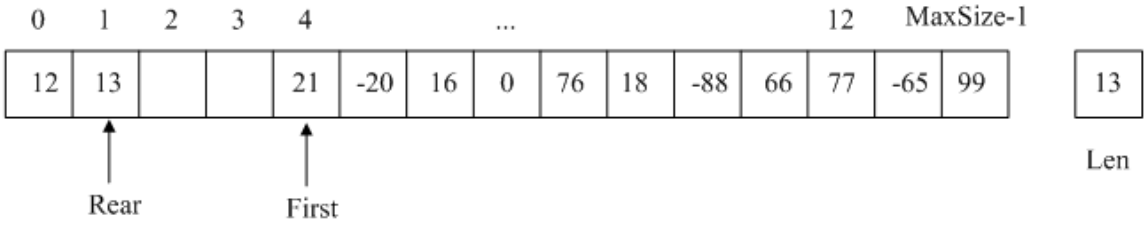

图 3-2

环形队列入队算法描述（C 语言）如下：

```
//入队
void entryQueue(int x)
{
    if(Len==MaxSize)//队满
    {
        printf("队列已满，不能入队\n");
        return;
    }
    Rear=(Rear+1)%MaxSize;//移动队尾指针
    sequ[Rear]=x;//入队
    Len++;//长度加 1
    return;
}
```

环形队列出队算法描述（C 语言）如下：

```
//出队，返回出队元素
int exitQueue(void)
{
    int x,First;//队头元素的下标
    if(Len==0)//队空
    {
        printf("队列已空，不能出队\n");
        return -1;
    }
    First=((Rear+MaxSize)-Len+1)%MaxSize;//计算队头元素的下标
    x=sequ[First];//获得队头元素
    Len--;//长度减 1
    return x;//返回队头元素
}
```

（2）建立 3 个队列，分别为客车队列 bus，货车队列 truck 和渡船队列 ferry，设置 3 个变量标识已上渡船的客车数量 busNum、货车数量 truckNum 和总数量 totalNum。

渡口管理示意图（某个状态）如图 3-3 所示，图中以*代表客车，#代表货车。

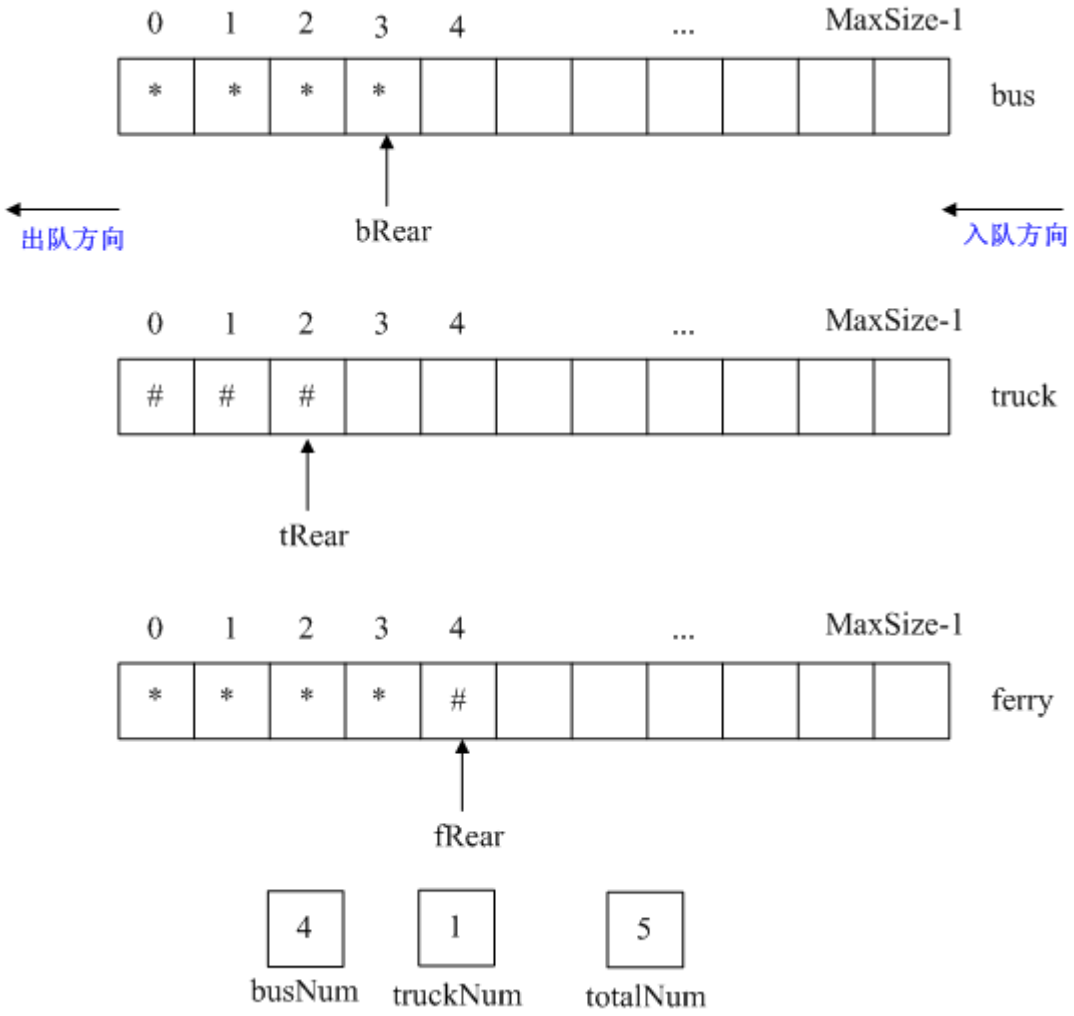

图 3-3

当车辆到达渡口时，客车入 bus 队列，货车入 truck 队列。当上渡船的汽车数量 totalNum 小于 10 时，循环扫描客车队列和货车队列，若 busNum<4 且客车队列不为空，将客车队列的队头汽车出队上船，busNum++，totalNum++；若 busNum<4 且客车队列为空但货车队列不为空，将货车队列的队头汽车出队上船，置 busNum=0，truckNum++，totalNum++；若 busNum>=4，且货车队列不为空，将货车队列的队头汽车出队上渡船，置 busNum＝0，totalNum++，truckNum++；若客车队列不空但货车队列为空，则将客车队列的队头汽车出队上渡船，置 totalNum++，busNum++，truckNum＝0；若客车队列和货车队列都为空时等待车辆的到达。

渡口管理算法描述（伪代码）如下：

1．初始化各队列为空，置 totalNum=0，busNum=0，truckNum=0；

2．输入到达渡口的汽车序列，将客车和货车分别入队；

3．当 totalNum<MaxSize 时，重复 3.1-3.5；

3.1 如果 busNum<4 且客车队列不为空，将客车队列的队头汽车出队上渡船，置 totalNum++，busNum++；

3.2 如果 busNum<4 且客车队列为空，但是货车队列不为空，将货车队列的队头汽车出队上渡船，置 totalNum++，truckNum++，busNum=0；

3.3 如果 busNum>=4 且货车队列不为空，将货车队列的队头汽车出队上渡船，置 totalNum++，truckNum++，busNum=0；

3.4 如果 busNum>=4 且货车队列为空，但客车队列不为空，将各车队列的队头汽车出队上渡船，置 totalNum++，truckNum=0，busNum++；

3.5 如果客车队列和货车队列都为空，中止循环；

4．输出当前客车、货车汽车等待情况和渡船装载汽车情况，算法结束。

实验四　二叉树

1．实验目的

（1）掌握二叉树的结构特性和二叉链表存储结构；

（2）理解二叉树、完全二叉树、满二叉树的概念和存储特点；

（3）掌握二叉树遍历的递归和非递归方法。

2．实验内容

（1）假设二叉树采用链接存储方式存储，分别编写一个二叉树先序遍历的递归算法和非递归算法。

（2）一棵完全二叉树以顺序方式存储，设计一个递归算法，对该完全二叉树进行中序遍历。

3．解题思路

（1）采用扩展二叉树的先序遍历序列来建立一棵二叉树，如图 4-1(a)所示的一棵二叉树，其扩展先序遍历序列为图 4-1(c)所示，其中#表示为空。

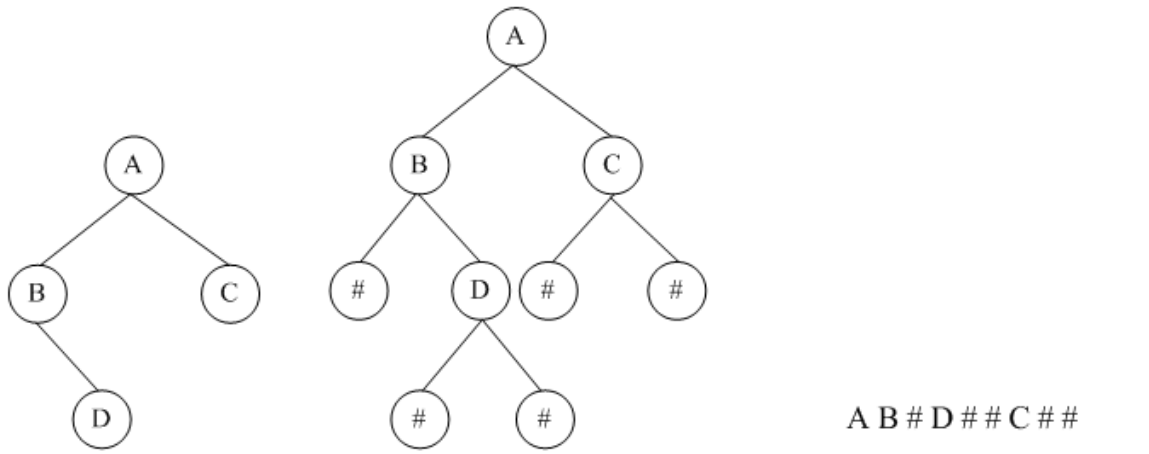

图 4-1

二叉树链表结点结构如下：

```
struct biTreeNode
{
    chardata;//二叉树元素为字符型，存放 A-Z 等
    struct biTreeNode *leftChild,*rightChild;   //左孩子、右孩子指针
```

```
};
```

创建二叉树的递归算法描述（C 语言）如下：

```
struct biTreeNode *creatBiTree(struct biTreeNode *T)
{
    char ch;
    static int i=0;
    ch=node[i];   //数组 node 存放了要创建的二叉树的扩展先序遍历序列
    i++;
    if(ch=='#')
    {
        T=NULL;     //#代表空子树；
    }
    else
    {
        T=(struct biTreeNode *)malloc(sizeof(struct biTreeNode));
        if(!T)exit(0);
        T->data = ch; //数据域为 ch
        T->leftChild=creatBiTree(T->leftChild);       //递归建立左子树
        T->rightChild=creatBiTree(T->rightChild);    //递归建立右子树
    }
    return T;
}
```

二叉树的先序遍历递归算法描述（C 语言）如下：

```
void preOrder(struct biTreeNode *T)
{
    if(T==NULL)//递归调用结束条件
        return;
    else
    {
        printf("%c ",T->data);       //访问根结点 T 的数据域
        preOrder(T->leftChild);     //前序递归遍历 T 的左子树
        preOrder(T->rightChild);   //前序递归遍历 T 的右子树
    }
}
```

二叉树的先序遍历非递归算法相对较难，其关键是：在先序遍历过某结点的整个左子树后，如何找到该结点的右子树的根指针。需要定义一个堆栈存放遍历过的结点的指针，以便能通过它找到该结点的右子树。一般情况下，设要遍历的二叉树的根指

针为 bt，可能有两种情况：

（a）若 bt!=NULL，则表明当前的二叉树不为空，此时，输出根结点 bt 的值并将 bt 保存到栈中，准备继续遍历 bt 的左子树。

（b）若 bt==NULL，则表明以 bt 为根指针的二叉树遍历完毕，并且 bt 是栈顶指针所指结点的左子树，若栈不空，则应根据栈顶指针所指结点找到待遍历右子树的根指针并赋给 bt，以继续遍历下去；若栈空，则表明整个二叉树遍历完毕，应结束。

二叉树先序遍历非递归算法描述（伪代码）如下：

1．栈 s 初始化；

2．循环直到 bt 为空且栈 s 为空；

2.1 当 bt 不空时循环

2.1.1 输出 bt->data；

2.1.2 将指针 bt 保存在栈中；

2.1.3 继续遍历 bt 的左子树；

2.2 如果栈 s 不空，则

2.2.1 将栈顶元素弹出至 bt；

2.2.2 准备遍历 bt 的右子树；

二叉树先序遍历非递归算法描述（C 语言）如下：

```
void nPreOrder(struct biTreeNode *bt)
{
    top=-1;                          //采用顺序栈，并假定不会发生上溢
    while(bt!=NULL || top!=-1)   //两个条件都不成立才退出循环
    {
        while(bt!=NULL)
        {
            printf("%c ",bt->data);
            s[++top]=bt;             //将根指针 bt 入栈
            bt=bt->leftChild;
        }
        if(top!=-1)                  //栈非空
        {
            bt=s[top--];
            bt=bt->rightChild;
        }
    }
```

```
}
```

（2）完全二叉树是指：对一棵具有 n 个结点的二叉树按层序编号，如果编号为 i(1<=i<=n)的结点与同样深度的满二叉树中编号为 i 的结点在二叉树中的位置完全相同，则这棵二叉树称为完全二叉树。如图 4-2 所示，即为一棵完全二叉树。其先序遍历为：A B D H I E J C F G，中序遍历为：H D I B J E A F C G，后序遍历为：H I D J E B F G C A。

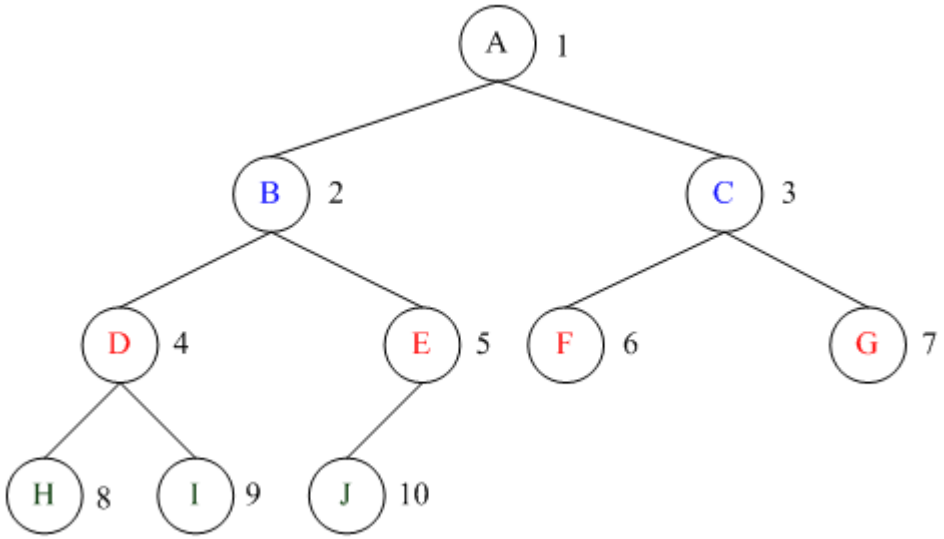

图 4-2

采用顺序存储即按层序编号的顺序存储所有结点，如图 4-3 所示，采用一维数组 node[MaxSize]存储图 4-1 所示的完全二叉树，下标为 0 的单元（即 node[0]）存放结点数，下标为 1 的单元存放根结点，以此类推。

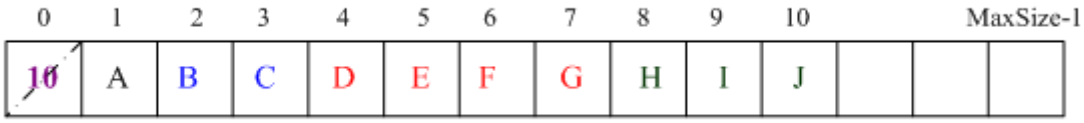

图 4-3

若要在此存储结构上实现完全二叉树的递归中序遍历，需要解决的关键问题在于如何确定一个结点（叶子结点除外）的左孩子和右孩子所在的下标，以及其双亲结点（根结点除外）的下标。

由二叉树【**性质 5-5**】可知：对一棵具有 n 个结点的完全二叉树中的结点从 1 开始按层序编号，则对任意的编号为 i(1<=i<=n)的结点（简称结点 i），有：

（a）如果 i>1，结点 i 的双亲的编号为 i/2；否则结点 i 是根结点，无双亲。

（b）如果 2i<=n，则结点 i 的左孩子的编号为 2i；否则结点 i 无左孩子。

（c）如果 2i+1<=n，则结点 i 的右孩子的编号为 2i+1；否则结点 i 无右孩子。

显然，按图 4-3 所示的存储方式，结点所在的下标与层序编号一致。在二叉链表的递归中序遍历算法的基础上，用结点下标代替结点的地址，即可实现顺序存储的递归中序遍历。

顺序存储的完全二叉树递归中序遍历算法描述（C 语言）如下：

```
void seqInOrder(int i)
{
    if(i==0)                    //递归调用的结束条件
        return;
    else
    {
        if(2*i<=node[0])
            seqInOrder(2*i);//中序遍历 i 的左子树
        else
            seqInOrder(0);
        printf("%2c",node[i]);//输出根结点
        if(2*i+1<=node[0])
            seqInOrder(2*i+1);//中序遍历 i 的右子树
        else
            seqInOrder(0);
    }
}
```

与此类似的，其递归先序和后序遍历算法分别如下：

顺序存储的完全二叉树递归先序遍历算法描述（C 语言）如下：

```
void seqPreOrder(int i)
{
    if(i==0)                    //递归调用的结束条件
        return;
    else
    {
        printf("%2c",node[i]);//输出根结点
        if(2*i<=node[0])
            seqPreOrder(2*i);//先序遍历 i 的左子树
        else
            seqPreOrder(0);
        if(2*i+1<=node[0])
```

```
            seqPreOrder(2*i+1);//先序遍历 i 的右子树
        else
            seqPreOrder(0);
    }
}
```

顺序存储的完全二叉树递归后序遍历算法描述（C 语言）如下：

```
void seqPostOrder(int i)
{
    if(i==0)                    //递归调用的结束条件
        return;
    else
    {
        if(2*i<=node[0])    //后序遍历 i 的左子树
            seqPostOrder(2*i);
        else
            seqPostOrder(0);
        if(2*i+1<=node[0])      //后序遍历 i 的右子树
            seqPostOrder(2*i+1);
        else
            seqPostOrder(0);
        printf("%2c",node[i]); //输出根结点
    }
}
```

实验五　哈夫曼树

1．实验目的

（1）理解哈夫曼树的概念、结构特性和哈夫曼编码原理；

（2）掌握构造哈夫曼树的基本方法；

（3）掌握运用哈夫曼树进行哈夫曼编码的方法。

2．实验内容

根据哈夫曼(Huffman)编码的原理，编写一个程序，在用户输入节点权重的基础上建立它的哈夫曼编码。

3．解题思路

这涉及**哈夫曼树**。首先回顾几个术语：叶子结点的权值：指对叶子结点赋予的一个有意义的数值量；二叉树的带权路径长度：从根结点到各个叶子结点的路径长度与相应叶子结点权值的乘积之和叫做二叉树的带权路径长度。哈夫曼树：给定一组具有确定权值的叶子结点，可以构造出不同的二叉树，将其中带权路径长度最小的二叉树称为哈夫曼树。

哈夫曼树可用于构造最短的不等长编码方案，称为哈夫曼编码树。规定哈夫曼编码树的左分支代表 0，右分支代表 1，则从根结点到每个叶子结点所经过的路径组成的 0 和 1 的序列便为该叶子结点对应字符的编码，称为**哈夫曼编码**。

哈夫曼算法的基本思想是：

（a）初始化：由给定的 n 个权值$\{w_1, w_2, ..., w_n\}$构造 n 棵只有一个根结点的二叉树，从而得到一个二叉树集合$F = \{T_1, T_2, ..., T_n\}$；

（b）选取与合并：在 F 中选取根结点的权值最小的两棵二叉树分别作为左、右子树，构造一棵新的二叉树，这棵新二叉树的根结点的权值为其左、右子树根结点的权值之和。

（c）删除与加入：在 F 中删除作为左、右子树的两棵二叉树，并将新建立的二叉树加入到 F 中。

（d）重复（b）和（c）两步，当集合 F 中只剩下一棵二叉树时，这棵二叉树便

是哈夫曼树。

例：对于给定权值集合 W={2,3,4,5}，图 5-1 给出了哈夫曼树的构造过程。

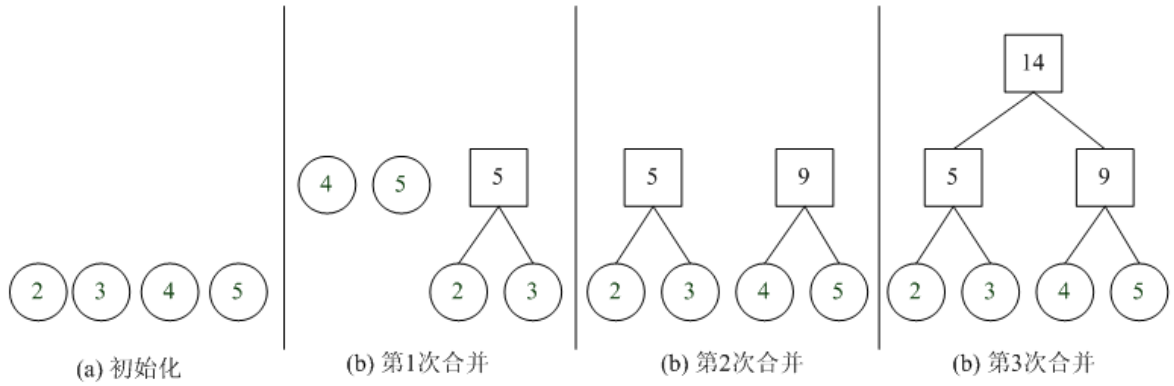

图 5-1

具有 n 个叶子结点的哈夫曼树共有 2n-1 个结点，其中有 n-1 个非叶子结点，它们是在 n-1 次合并过程中生成的。定义一个数组 huffTree[2n-1]保存哈夫曼树中各结点的信息，数组元素的结点结构如下：

```
#define Max      20   //最大叶子结点数
struct huffNode
{
    char ch;       //结点字符 A-Z
    int weight;    //结点权值域
    int lChild;    //该结点左孩子结点在数组中的下标
    int rChild;    //该结点右孩子结点在数组中的下标
    int parent;    //该结点双亲结点在数组中的下标
    char hCode[Max]; //该结点字符的哈夫曼编码
};
struct huffNode huffTree[2*Max-1];
```

哈夫曼树算法描述（伪代码）如下：

1．数组 huffTree 初始化，所有数组元素的双亲、左右孩子都置为-1，hCode 置空；

2．数组 huffTree 的前 n 个元素的字符和权值置给定值；

3．进行 n-1 次合并

3.1　在二叉树集合中选取两个权值最小的根结点，其下标分别为 i1，i2；

3.2　将二叉树 i1，i2 合并为一棵新的二叉树 k；

建立哈夫曼树之后，就可对相应字符进行编码，编码的基本思想是从叶子结点开

始沿着双亲结点，直到根结点，这样形成从该叶子结点到根结点的一条路径，将这条路径上的左孩子编码为 0，右孩子编码 1，再从根结点到叶子结点的顺序将编码组合。重复上述操作直到为每个叶子完成编码。

哈夫曼编码描述（C 语言）如下：

```
void huffmanTreeCode(int n)
{
    char hc[MaxSize];
    int hcLen;
    int i,j,k,parent,p;
    for(i=0;i<n;i++)
    {
        hcLen=0;
        parent=huffmanTree[i].parent;//待编码字符的双亲结点下标
        p=i;
        while(parent!=-1)//未到达根结点
        {
            if(huffmanTree[parent].lChild==p)//是左孩子
                hc[hcLen]='0',hcLen++;
            else if(huffmanTree[parent].rChild==p)//是右孩子
                hc[hcLen]='1',hcLen++;
            p=parent;
            parent=huffmanTree[parent].parent;//继续向根结点查找
        }
        for(j=0,k=hcLen-1;j<hcLen;j++,k--)//将编码写入相应字符数组
            huffmanTree[i].hCode[j]=hc[k];
        huffmanTree[i].hCode[j]='\0';//加上字符串结束符
    }
    return;
}
```

实验六　图的基本存储

1．实验目的

（1）理解图的基本概念、图的结构特性；

（2）掌握邻接表表示图的基本方法；

（3）掌握连通图遍历的基本的方法。

2．实验内容

（1）用邻接表存储一个图 G。分别设计实现下列要求的算法：

- ✧ 求出图 G 中每个顶点的出度；
- ✧ 求出图 G 中出度最大的一个顶点，输出该顶点的编号；
- ✧ 计算图 G 中出度为 0 的顶点数。

（2）编写一个实现连通图 G 的深度优先遍历（从顶点 V_0 出发，以邻接矩阵为存储结构）的递归和非递归函数，分别用这两种方法对图进行深度优先遍历，比较遍历结果。

3．解题思路

（1）邻接表（Adjacency List）是一种顺序存储与链接存储相结合的存储方法，类似于树的孩子链表表示法。对于图的每个顶点 v_i，将所有邻接于 v_i 的顶点链成一个单链表，称为顶点 v_i 的边表（有向图中称为出边表）。所有边表的头指针采取顺序存储，构成邻接表的表头数组，称为顶点表。顶点表结点和边表结点的结构如下：

```
struct Edge
{
    int adjvex;//邻接点以数字编号，如 0,1,2,...,9
    int weight;
    struct Edge *nextEdge;
};
struct Node
{
    int vertex;//顶点以数字编号，如 0,1,2,...,9
    struct Edge *firstEdge;
};
struct Node graphNode[MaxVertexNum];//图顶点数组
```

采取邻接表存储的有向图，每个顶点的边表结点个数即为该顶点的出度。循环扫描顶点表中各顶点的边表结点个数，即可相应的求出图中每个顶点的出度、最大出度和出度为 0 的顶点。

以图 6-1 为例，各顶点出度如下：V_0：2，V_1：2，V_2：1，V_3：1，V_4：1，出度最大的顶点为 V_0 和 V_1，出度为 0 的顶点个数为 0。

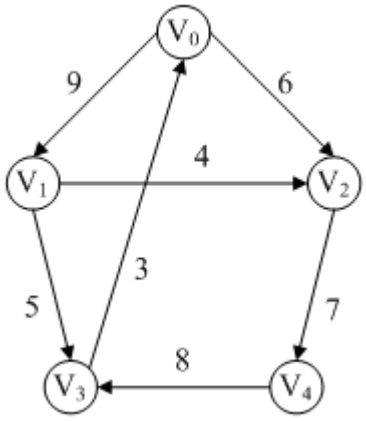

图 6-1

（2）**深度优先遍历**算法的基本思想是：a. 访问顶点 v；b. 从 v 的未被访问的邻接点中选取一个顶点 w，从 w 出发进行深度优先遍历；c. 重复 a 和 b，直至图所有和 v 有路径相通的顶点都被访问到。

广度优先遍历算法的基本思想是：a. 访问顶点 v；b. 依次访问 v 的各个未被访问的邻接点 $v_1,v_2,\cdots,v_k$；c. 分别从 $v_1,v_2,\cdots,v_k$ 出发依次访问它们未被访问的邻接点，并使“先被访问顶点的邻接点”先于“后被访问顶点的邻接点”被访问，直至图所有和 v 有路径相通的顶点都被访问到。

以图 6-1 为例，采用邻接表存储。

深度优先遍历递归算法描述（C 语言）如下：

```
void DFS(int i)
{
    struct Edge *p;
    printf("V%d->",graphNode[i].adjvex);
    visited[i]=1;   //顶点 Vi 已访问
    p=graphNode[i].firstEdge;
    while(p)
    {
```

```
            if (!visited[p->vexdata])
                DFS(p->vexdata);
            p=p->nextEdge;
        }
}

void DFSTraverse(void)
{
    int i;
    for(i=0;i<vertexNum;i++)     //重置顶点访问数组
        visited[i]=0;
    printf("图的深度优先遍历序列为：");
    for(i=0;i<vertexNum;i++)     //从第 1 个顶点开始依次遍历
        if(!visited[i])
            DFS(i);
   return;
}
```

广度优先遍历算法描述（C 语言）如下：

```
void BFSTraverse(int i)
{
    int j;
    struct Edge *p;
    front=rear=-1;              //初始化队列
    for(j=0;j<vertexNum;j++)//重置顶点访问数组
        visited[j]=0;
    printf("图的广度优先遍历序列为：");
    printf("V%d->",graphNode[i].vertex);
    visited[i]=1;
    seqQueue[++rear]=i;     //被访问顶点入队
    while(front!=rear)
    {
        i=seqQueue[++front]; //顶点出队
        p=graphNode[i].firstEdge; //工作指针指向顶点 i 的边表
        while(p!=NULL)
        {
            j=p->adjvex;
            if(!visited[j])/     /顶点 j 未被访问过
            {
                printf("V%d->",j);
                visited[j]=1;
                seqQueue[++rear]=j;
            }
```

```
            p=p->nextEdge;
        }
    }
    return;
}
```

深度优先遍历算法的非递归实现需要了解深度优先遍历的执行过程，设计一个栈来模拟递归实现中系统设置的工作栈，同时图需要采取邻接矩阵来存储。图 6-1 的邻接矩阵为：

```
int vertex[5]={0,1,2,3,4};
int arc[5][5]={0,9,6,-1,-1,-1,0,4,5,-1,-1,-1,0,-1,7,3,-1,-1,0,-1,-1,-1,-1,8,0};
//邻接矩阵中，-1 表示不邻接
```

深度优先遍历算法非递归算法描述（C 语言）如下：

```
//------------深度优先遍历(非递归)------------
//图以邻接矩阵存储，-1 表示不可达，0 表示自己，>0 表示权值
void depthFirstTraverse(int v)
{
    int i,j;
    for(i=0;i<MaxVertexNum;i++)//初始化顶点访问数组
        visited[i]=0;
    top=-1;//初始化栈,顺序栈
    printf("图的深度优先遍历(非递归)序列为：");
    printf("V%d->",v);//访问顶点 v
    visited[v]=1;
    seqStack[++top]=v;//顶点入栈
    while(top!=-1)
    {
        v=seqStack[top];//取栈顶顶点
        for(j=0;j<vertexNum;j++)
            if(arc[v][j]>0 && visited[j]==0)
            {
                printf("V%d->",j);//访问顶点 j
                visited[j]=1;
                seqStack[++top]=j;
                break;
            }
        if(j==vertexNum)
            top--;
    }
    return;
```

```
}
```


五邑大学
WUYI UNIVERSITY

实验七　图的应用

1．实验目的

（1）理解拓扑排序的基本概念和拓扑排序的实用意义；

（2）掌握 AOV-网解决拓扑排序问题的基本方法。

（3）理解最小生成树的基本概念和实用意义；

（4）掌握利用 MST 性质构造最小生成树的 prim 算法和 Kruskal 算法。

2．实验内容

（1）教学计划编制问题。按先修课程和后续课程关系，编制一个简单的排课计划程序。

（2）在 n 个城市之间架设 n-1 条线路，要求建设代价最小。编写完整算法构造最小生成树。

3．解题思路

（1）教学计划编制问题其本质为 AOV 网的拓扑排序问题。AOV 网：用顶点表示活动，用弧表示活动之间的优先关系。AOV 网中的弧表示了活动之间存在的某种制约关系。排课计划需要考虑先修课程和后续课程的关系，AOV 网正好能反映这种关系。

下表为一个课程修读计划，用 AOV 网表示如图 7-1 所示。

编号	课程名称	先修课
C1	高等数学	无
C2	计算机导论	无
C3	离散数学	C1
C4	程序设计	C1，C2
C5	数据结构	C3，C4
C6	计算机原理	C2，C4
C7	数据库原理	C4，C5，C6

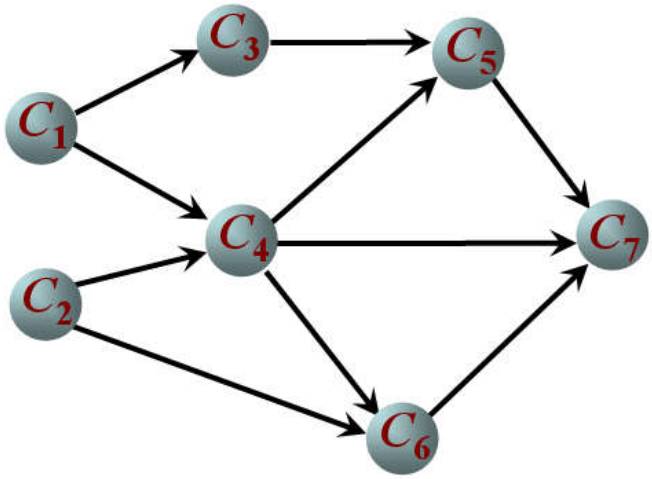

图 7-1 课程修读 AOV 网

对图 7-1 进行拓扑排序，所得结果即为排课的计划顺序。图采用邻接表存储，但需要对顶点表结点作一点调整，增加一个入度域，记录该顶点的入度。

```
struct vertexNode
{
    int inDegree;//顶点入度
    int vertex;//顶点
    struct arcNode *firstArc;//第一条弧
};
```

拓扑排序算法描述（伪代码）如下：

1．栈 S 初始化；累加器 count 初始化；

2．扫描顶点表，将没有前驱（即入度为 0）的顶点压栈；

3．当栈 S 不空时循环

3.1 j=栈顶元素出栈；输出顶点 j；count++；

3.2 对顶点的每一个邻接点 k 执行下述操作；

3.2.1 将顶点 k 的入度减 1；

3.2.2 如果顶点 k 的入度为 0，则将顶点 k 入栈；

4．if(count<vertexNum) 输出有回路信息。

（2）这个问题可以转化为最小生成树的求解。最小生成树的经典算法 Prim 算法和 Kruskal 算法。以下以 Prim 算法为例（Kruskal 算法请自行实现）。

Prim 算法的基本思想是：设 $G=(V,E)$ 是一个无向连通网，令 $T=(U,TE)$ 是 G 的

最小生成树。T 的初始状态为$U=\{v_0\}(v_0\in V), TE=\{\}$，然后重复执行下述操作：在所有$u\in U, v\in V-U$的边中找一条代价最小的边$(u,v)$并入集合 TE，同时 v 并入 U，直到 U=V 为止。此时 TE 中必有 n-1 条边，则 T 就是一棵最小生成树。

Prim 算法描述（伪代码）如下：

1．初始化：$U=\{v_0\}, TE=\{\}$；

2．重复下述操作直到 U=V

2.1 在 E 中寻找最短边(u,v)，且满足$u\in U, v\in V-U$；

2.2 U=U+{v}；

2.3 TE=TE+{(u,v)}；

Prim 算法的存储结构：

（1）图采用邻接矩阵存储；

（2）侯选最短边集：设置一个数组 shortEdge[n]表示侯选最短边集合，数组元素包括 adjvex 和 lowcost 两个域，分别表示候选最短边的邻接点和权值。定义如下：

```
struct sEdge
{
    int adjvex;//最短边邻接点
    int lowcost;//权值
};
struct sEdgeshortEdge[N];//候选最短边集
```

Prim 算法描述（C 语言）如下：

```
void MST_Prim(void)
{
    int i,j,k,totalCost=0;
    for(i=1;i<N;i++)//初始化辅助数组 shortEdge
    {
        shortEdge[i].lowcost=arc[0][i];//其他顶点到顶点 V0 的权值
        shortEdge[i].adjvex=0;
    }
    shortEdge[0].lowcost=0;//将顶点 V0 加入集合 U
    printf("最小生成树的边集和权值如下：\n");
    for(i=1;i<N;i++)
    {
```

```
        k=minEdge();
        printf("(V%d,V%d)-cost:%d\n",shortEdge[k].adjvex,k,shortEdge[k].lowcost);
        //输出最短边及权值
        totalCost+=shortEdge[k].lowcost;
        shortEdge[k].lowcost=0;  //将顶点 k 加入集合 U 中
        for(j=1;j<N;j++) //调整数组 shortEdge[N]
        {
            if(arc[k][j]<shortEdge[j].lowcost)//arc[k][j]的值为 Max 表示不可达
            {
                shortEdge[j].lowcost=arc[k][j];
                shortEdge[j].adjvex=k;
            }
        }
    }
    printf("最小生成树的总代价为：%d\n",totalCost);
    return;
}
```

MinEdge 算法描述（C 语言）如下：

```
int minEdge(void)
{
    int i,j,min;
    for(i=1;i<N;i++)
        if(shortEdge[i].lowcost>0)//未加入 U 的顶点
        {
            min=i;//先设为最小权值
            break;
        }
    //在余下的侯选边中查找最小权值邻接点, lowcost==0 表示该顶点已选
     for(j=i+1;j<N;j++)
        if(shortEdge[j].lowcost>0 && shortEdge[j].lowcost<shortEdge[min].lowcost)
            min=j;
    return min;
}
```

实验八 查找

1．实验目的

（1）理解查找表的基本概念，定义、分类和各类的特点；

（2）掌握哈希表的构造方法以及解决冲突的方法；

（3）掌握哈希存储和哈希查找的基本思想及有关方法。

2．实验内容

使用哈希函数：H（K）=3K MOD 11

并采用链地址法处理冲突。试对关键字序列（22，41，53，46，30，13，01，67）构造哈希表，设计构造哈希表的完整算法。

3．解题思路

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

链表结点及头指针数组定义如下：

```
#define MaxSize 11
//链表结点
struct linkNode
{
    int data;//数据域
    struct linkNode *next;//指针域
};

struct linkNode head[MaxSize];//头指针数组
```

构造散列表算法描述（C 语言）如下：

```
//构造散列表，n 为数据个数，散列函数：H(K)=3*k MOD MaxSize
void createHashTable(int a[],int n)
{
    struct linkNode *p;
    int i,j;
    for(i=0;i<MaxSize;i++)//初始化为空
```

```
        head[i]=NULL;
    for(i=0;i<n;i++)
    {
        j=3*a[i] % MaxSize;
        p=(struct linkNode *)malloc(sizeof(struct linkNode));
        p->data=a[i];
        p->next=head[j];//头插法
        head[j]=p;
    }
}
```

散列表查找算法描述（C 语言）如下：

```
//开散列表查找，k 为要查找的数，返回结点指针
struct linkNode *hashSearch(int k)
{
    struct linkNode *p;
    int i;
    i=3*k % MaxSize;
    p=head[i];
    while(p && p->data!=k)
        p=p->next;
    return p;
}
```

实验九　排序

1．实验目的

（1）理解各类内部排序的基本思想；

（2）掌握各类内部排序的基本方法；

（3）了解各类内部排序的优缺点。

2．实验内容

（1）编写一个双向冒泡排序算法，即在排序过程中以交替的正、反两个方向进行遍历。若第一趟把关键字最大的记录放到最末尾，则第二趟把关键字最小的记录放到最前端，如此反复进行之。

（2）设计一个进行堆排序的算法。

3．解题思路

（1）双向冒泡排序算法，在排序过程中以交替的正反两个方向进行遍历。把最大的放在末尾、最小的放在最前。假设第 1 趟先进行反向冒泡排序，最小数放到第 1 位，第 2 趟进行正向冒泡排序，最大值放到最末尾，交替进行，其排序过程如图 9-1 所示。每趟排序结束后，都会有 1 个数排好序，排序区间要缩减 1 位。

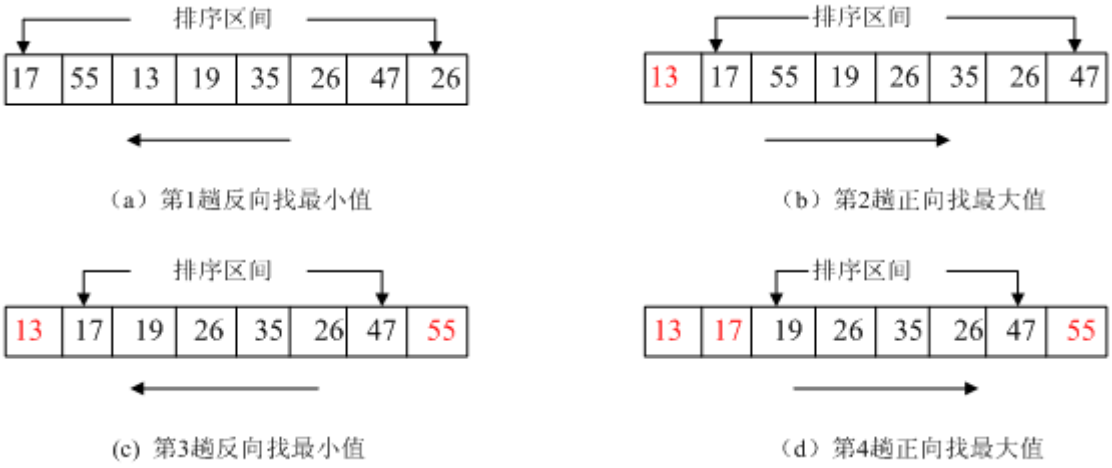

图 9-1　双向冒泡法排序过程

双向冒泡排序算法描述（C 语言）如下：

```
//双向冒泡排序,数据从数组 r 的 0 号单元开始存放
```

```
void doubleBubble(int r[],int n)
{
    int flag=1;
    int i=0,j;
    int temp;
    while (flag==1)
    {
        flag=0;
        for(j=n-i-1;j>=i+1;j--)//反向遍历找最小值
            if(r[j]<r[j-1])
            {
                flag=1; //能交换说明没有排好序，要继续
                temp=r[j];
                r[j]=r[j-1];
                r[j-1]=temp;
            }
        for(j=i+1;j<n-1;j++)//正向遍历找最大值
            if(r[j]>r[j+1])
            {
                flag=1; //能交换说明没有排好序，要继续
                temp=r[j];
                r[j]=r[j+1];
                r[j+1]=temp;
            }
            i++;
    }
    return;
}
```

（2）堆排序（Heap Sort）是简单选择排序的一种改进，改进的着眼点是：如何减少关键码的比较次数。

堆（heap）是具有下列性质的**完全二叉树**：每个结点的值都小于或等于其左右孩子结点的值（称为**小根堆**）；或者每个结点的值都大于或等于其左右孩子结点的值（称为**大根堆**）。

如果将堆按层序从 1 开始编号，则结点之间满足如下关系：

$$\begin{cases} k_i \le k_{2i} \\ k_i \le k_{2i+1} \end{cases} \text{或} \begin{cases} k_i \ge k_{2i} \\ k_i \ge k_{2i+1} \end{cases} \qquad 1 \le i \le \lfloor n/2 \rfloor$$

从堆的定义可以看出，一个完全二叉树如果是堆，则要结点（称为堆顶）一定是当前堆中所有结点的最大者（大根堆）或最小者（小根堆）。以结点的编号为下标，

将堆用顺序存储结构存储，则堆对应于一组序列。如图 9-2 所示。

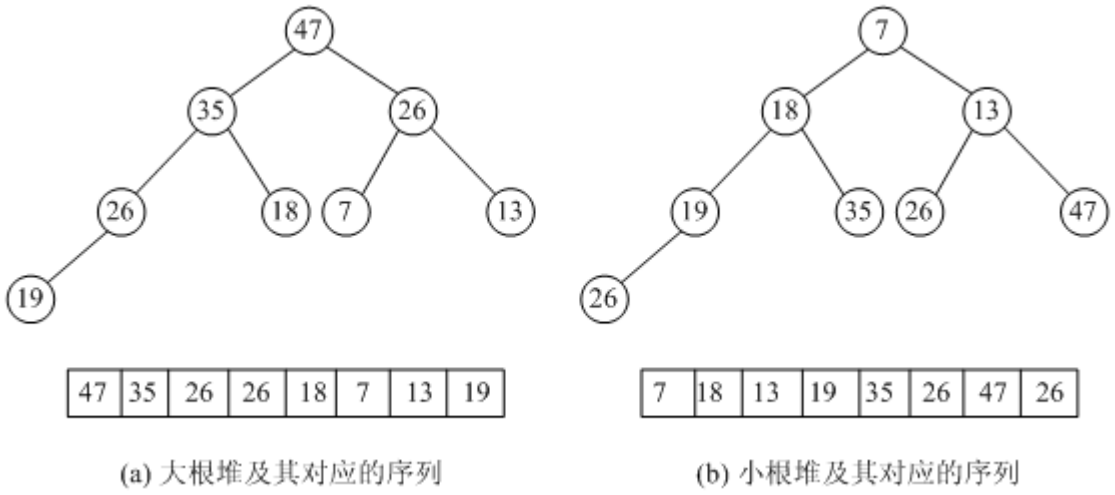

图 9-2　堆的示例

在调整堆的过程中，总是将根结点（即被调整结点）与左右孩子结点进行比较。若不满足堆的条件，则将根结点与左右孩子的较大者（或较小者）进行交换，这个调整过程一直到所有子树均为堆或将被调整结点交换到叶子为止。这个自堆顶至叶子的调整过程称为“筛选”。

筛选法调整堆算法描述（C 语言）如下：

```
//筛选算法调整堆,k 为当前要筛选的结点编号,m 为堆中最后一个结点的编号,
//且结点 k 的左右子树均是堆,数组 r 的 0 号单元作为交换暂存单元
void sift(int r[],int k,int m)
{
    int i,j;
    i=k;//i 指向被筛选结点
    j=2*i;// j 指向结点 i 的左孩子
    while(j<=m)
    {
        if(j<m && r[j]<r[j+1])//比较 i 的左右孩子, j 指向较大者
            j++;
        if(r[i]>r[j])//根结点已经大于左右孩子中的较大者
            break;
        else
        {
            r[0]=r[i];//交换根结点与结点 j
            r[i]=r[j];
            r[j]=r[0];
            i=j;//被筛选结点位于原来结点的 j 的位置
            j=2*i;
        }
```

```
    }
    return;
}
```

堆排序是利用堆（假设利用大根堆）的特性进行排序的方法，其基本思想是：首先将待排序的记录序列构造成一个堆，此时，选出了堆中所有记录的最大者即堆项记录。然后将堆顶记录移走，并将剩余的记录再调整成堆，这样又找出了次大的记录。以此类推，直到堆中只有一个记录为止。

堆排序算法描述（C 语言）如下：

```
//堆排序,数组 r 的 0 号单元作为交换暂存单元
void heapSort(int r[],int n)
{
    int i;
    for(i=n/2;i>=1;i--)//初始建堆,从最后一个分支结点往根结点调整
        sift(r,i,n);
    for(i=1;i<=n;i++)//重复执行移走堆顶及重建堆的操作
    {
        r[0]=r[1];
        r[1]=r[n-i+1];
        r[n-i+1]=r[0];
        sift(r,1,n-i);
    }
    return;
}
```

附录　实验报告模板

五邑大学实验报告

数据结构课程实验报告

2016~2017 学年第 2 学期

院系 计算机学院

班级 _____

学号 _____

姓名 _____

任课教师：刘兴林　成绩评定：_____

实验一题目：

完成日期：2017 年 月 日

1、实验目的

2.、实验内容（**实验题目与说明**）

3、算法设计（**核心代码或全部代码**）

4、运行与测试（**测试数据和实验结果分析**）

5、总结与心得