

# 算法与数据结构笔记

初旭

## 目录

▶ 绪论	
■ 需要记住的东西	3
▶ 线性表	
■ 基本概念	5
■ 算法	7
● 顺序表的定义与实现	7
● 单链表的定义与实现	10
● 静态链表的定义与实现	17
▶ 字符串	
■ 需要记住的东西	18
■ 算法	18
● 定长顺序串的实现	18
● 变长顺序串的实现	19
● 链串的实现	20
● 模式匹配	20
▶ 栈和队列	
■ 要记住的东西	23
■ 算法	23
● 栈的实现	23
● 队列的实现	25
▶ 树和二叉树	
■ 树的基本概念	29
■ 树的算法和数据结构	31
● 树的存储表示	31
● 树的周游	34
■ 二叉树的算法和数据结构	36
● 二叉树的存储表示	36
● 二叉树的周游	37
■ 哈夫曼树和哈夫曼	40
● 基本概念	40
● 哈夫曼树的数据结构和基本算法	41
▶ 字典和检索	
■ 字典的基本概念	43
■ 字典的存储表示方法	43
● 顺序表示	43
● 散列表示	45
● 二叉树表示	48

---

▶ 排序	
■ 基本概念	.....52
■ 基本算法	.....53
● 直接插入排序	.....53
● 直接选择排序	.....53
● 冒泡排序	.....54
▶ 图 55	
■ 基本概念	.....55
■ 图的存储结构	.....56
● 邻接矩阵表示法（数组表示法）	.....56
● 邻接表表示法（数组表示法）	.....57
■ 图的周游	.....59
● 深度优先搜索	.....59
● 广度优先搜索	.....60
■ 最小生成树	.....62
■ 最短路径	.....64
● 从某个源点到其余各顶点的最短路径（Dijkstra 算法）	.....64
● 每一对顶点之间的最短路径（Floyd 算法）	.....65

# 第一章 绪论

## 1.1 需要记住的东西

### 一、基本术语

- 1、数据在计算机科学中是指所有能输入到计算机中并被计算机程序处理的符号的总称。
- 2、数据元素：数据的基本单位，在计算机程序中通常作为一个整体进行考虑和处理。
- 3、数据对象：是性质相同的数元素的集合。
- 4、数据结构：是相互之间存在一种或多种特定关系的数据元素的集合。
- 5、数据结构的三要素：
  - 1) 逻辑结构：数据元素之间的逻辑关系
  - 2) 物理结构：数据结构在计算机中的表示（映象）
  - 3) 操作：抽象数据类型关心的各种行为在存储上的具体实现算法。
- 5、数据类型：是一个值的集合和定义在这个值集上的一组操作的总称。
- 6、数据类型的分类及作用：
  - 1) 原子类型
  - 2) 结构类型作用：实现信息的隐蔽

### 二、主要数据结构之逻辑结构

- 1、线性结构：每一个接点最多只有一个前驱和一个后继。
  - 1) 线性表：线性表中的元素之间是一种简单的线性关系。是零个或多个元素的有穷序列
  - 2) 顺序表：将线形表中的元素一个一个存储到一片相邻的区域中。
  - 3) 链表：线形表中的接点通过指针链接成为链表。
  - 4) 字符串：字符串是一种特殊的线性结构，它以字符为元素。
  - 5) 栈：栈元素的存入和取出按照后进先出原则，最先取出的总是在此之前最后放进去的那个元素；
  - 6) 队列：而队列实现先进先出的原则，最先到达的元素也最先离开队列。
- 2、树结构：每个节点只能有一个前驱，却可以有多个后继。
- 3、图形结构：
  - 1) 图：包括一个结点集合和一个边集合，边集合中每条边联系着两个结点。
  - 2) 字典：字典是一种二元组的集合，每个二元组包含着一个关键码和一个值。抽象地看，一个字典就是由关键码集合到值集合的一个映像。按关键码进行检索是字典中最常用的操作。

### 三、主要数据结构之存储结构

- 1、顺序表示：用一个连续的空间，顺序存放数据结构中的各个节点。
- 2、链接表示：接点的存放位置是任意的，节点之间的关系通过与节点关联的指标（或引用）方式表达出来。
- 3、散列表示：选择适当的散列函数，根据关键码的植将接点映射到给定的存储空间（散列表）中。
- 4、索引表示：同样也是给出一种从关键码到 地址的映像方法，不同的是：散列法的映像是通过函数定义；而索引法是通过建立辅助的索引结构解决。

### 四、算法

- 1、算法：算法是为实现某个计算过程而规定的基本动作的执行序列。
- 2、算法的性质：有穷性 确定性 可行性
- 3、算法的正确性：如果一个算法以一组满足出事条件的输入开始，那么该算法的执行一定终止，并且在终止时得到满足要求的输出结果。
- 4、可行解：满足安全性的普通解称为可行解。



5、最优解：分组数最少的解称为最优解。

6、课程中提到的算法：

1) 穷举法：逐个列举出所有可能的着色方案，检查这样的分组方案是否满足要求。首先慢子要求的分组，是问题的最优解。

2) 贪心法：当追求的目标是一个问题的最优解，设法把对整个问题的求解分成若干步骤来完成，在其中的某一段都选择从局部看来是最优的方案，以期通过咄咄段的局部最优选择达到整体的最优。

3) 分治法：把一个规模较大的问题分成两个或两个以上较小的相似子问题，首先对子问题求解，然后把各个问题的解合起来，得出整个问题的解。

4) 回溯法：采用一步一步镶嵌试探的方法，当某一步出现多种选择时，可以先任选一种，只要这种选择暂时可以继续镶嵌，一旦发现某步后无法向前，则后退回到上一步重新选择。

## 五、算法的分析

1、算法的空间代价：

当被解决问题的规模(以某种单位计算)由 1 增至  $n$  时，解该问题的算法所需占用的空间也以某种单位由  $f(1)$  增至  $f(n)$ ，这时我们称该算法的空间代价是  $f(n)$ 。

2、算法的时间代价：

当问题规模以某种单位由 1 增至  $n$  时，对应算法所耗费的时间也以某种单位由  $g(1)$  增至  $g(n)$ ，这时我们称该算法的时间代价是  $g(n)$ 。

3、空间单位：一般规定为一个简单变量(如整型、实型等)所占存储空间的大小。

4、时间单位：一般规定为执行一个简单语句(如赋值语句、判断语句等)所用时间。

5、“大 O”表示法：

如果存在正的常数  $c$  和  $n_0$ ，当问题的规模  $n \geq n_0$  后，该算法的时间(或空间)代价  $T(n) \leq c \cdot f(n)$ 。则称该算法的时间代价(或空间代价)为  $O(f(n))$ ，这时也称该算法的时间(或空间)代价的增长率为  $f(n)$ 。

大 O 表示法常用的计算规则：

6、“大 O”表示法的计算规则：

1) 加法准则

$$\begin{aligned} T(n) &= T_1(n) + T_2(n) \\ &= O(f_1(n)) + O(f_2(n)) = O(\max(f_1(n), f_2(n))) \end{aligned}$$

2) 乘法准则

$$\begin{aligned} T(n) &= T_1(n) \times T_2(n) \\ &= O(f_1(n)) \times O(f_2(n)) = O(f_1(n) \times f_2(n)) \end{aligned}$$

7、大 O 表示法的作用：

主要关注复杂性的量级，而忽略量级的系数，这使我们在分析算法的复杂度时，可以忽略零星变量的存储开销和循环外个别语句的执行时间，重点分析算法的主要代价。

## 六、抽象数据类型

1、抽象数据类型(ADT)：

可以看作是定义了一组操作的一个抽象(数学)模型。例如，集合与集合的并、交、差运算就可以定义为一个抽象数据类型。它不关心类型中值的具体表示方式和数据类型中定义的各种操作的具体实现方法，是所有可能的值的具体表示和各种操作的具体实现的抽象。

2、抽象数据类型的意义和作用

☞ 算法顶层设计与底层实现相分离

☞ 算法设计与数据结构设计相分离，运行数据结构自由选择，从中比较，优化算法效率

☞ 数据模型和该模型上的运算统一在抽象数据类型中，反映他们之间内在的相互依赖和相互制约的关系，便于空间和时间耗费的折中，灵活地满足用户需求

- ☛ 算法自然呈现模块化，抽象数据类型的表示和实现可以封装，便于移植和重用
- ☛ 为自顶向下逐步求精和模块化提供有效途径和工具
- ☛ 算法结构清晰，层次分明，便于算法正确性的说明和复杂性的分析

## 第二章 线性表

### 2.1 基本概念

#### 一、线性结构

##### 1、线性结构的特点：

在数据元素的非空有限集中，

- (1) 存在唯一的一个被称为“第一个”的数据元素；
- (2) 存在唯一的一个被称为“最后一个”的数据元素；
- (3) 除第一个之外，集合中的每个数据元素均只有一个前驱；
- (4) 除最后一个之外，集合中的每个数据元素均只有一个后继；

##### 2、线性表：简称为表，是零个或多个元素的有穷序列。

##### 3、表长：线性表中所含元素的个数。

##### 4、空表：长度为零的线性表。

##### 5、表目：线性表中的元素。

#### 二、线性表的顺序表示

##### 1、线性表的顺序表示：

以元素在计算机内存中的“物理位置相邻”来表示线性表中数据元素之间的逻辑关系。只要确定了首地址，线性表中任意数据元素都可以随机存取。

##### 2、顺序表：将线形表中的元素一个一个存储到一片相邻的区域中。

#### 三、线性表的链接表示

##### 1、链接表示：

一种实现顺序表的结构，不要求落家馆系项链的两个元素在物理位置上也相邻存储，而是通过指针来表示元素之间的逻辑关系

##### 2、结点：数据元素的存储映象，由数据域和指针域构成。

##### 3、结点的域：

- 1) 数据域：存放元素本身的信息。
- 2) 指针域：存放其后继结点的存储位置。

##### 4、链表：假设线性表中有 $n$ 个元素，那么这 $n$ 个元素所对应的 $n$ 个结点就通过指针针链接成为链表。

##### 5、单链表与顺序表的比较：

- (1) 单链表的存储密度比顺序表低；但在许多情况下链式的分配比顺序分配有效；
- (2) 在单链表里进行插入、删除运算比在顺序表里容易得多；
- (3) 对于顺序表，可随机访问任一个元素，而在单链表中，需要顺着链逐个进行查找。

##### 6、循环链表：单链表中，不让最后一个结点的指标为 NULL，而让它指向头结点，得到循环链表。

### 2.2 算法

#### 2.2.1 顺序表的定义与实现

##### 一、顺序表的定义

```
typedef struct SeqList
{
    ElemType elems[MAXSIZE];           //数据元素类型约定为 ElemType
    int      nCount;                   //这里还可以包含其它信息
}SeqList, *PSeqList;
```



## 二、顺序表的实现

### 1、在指定位置插入一个元素

```
BOOL Insert(PSeqList pList, ElemType x, int pos)
```

//在 pList 所指的顺序表 pos 位置上插入一个值为 x 的元素，并返回成功与否的标志。

```
{
    int    q;
    assert(0<=pos && pos<=pList->nCount);    //断言函数，用来判断 pos 是否合法

    if ( pList->nCount >= MAXSIZE )            //防止插入后溢出
    {
        return ( FALSE );
    }
    for(q = pList->nCount - 1; q>=pos; q--)        //从后向前循环把 pos 后面的元素后移一位
    {
        pList->elems[q+1] = pList->elems[q];
    }

    pList->elems[pos] = x;
    ++pList->nCount;                                //同时长度增加

    return ( TRUE );
}
```

### 2、在表头插入一个元素

```
BOOL InsertHead(PSeqList pList, ElemType x)
```

```
{
    return Insert(pList, x, 0);                    //要尽量利用已有的函数
}
```

### 3、在表尾增加一个元素

```
BOOL InsertTail(PSeqList pList, ElemType x)
```

```
{
    if(pList->nCount < MAXSIZE)
    {
        pList->elems[pList->nCount++] = x;            //相当于：
                                                        pList->nCount++;
        return TRUE;                                pList->elems[pList->nCount]=x
    }
    else
    {
        return FALSE;
    }
}
```

#### 4、删除指定位置上的元素

```
BOOL Delete(PSeqList pList, int pos)
{
    int q;
    assert(0<=pos && pos < pList->nCount);

    if(pList->nCount <= 0)                //如果是空表
        return FALSE;

    for(q=pos; q<pList->nCount-1; q++)    //从 pos 处开始循环, 后面的向前移动一位并覆盖前面的内容
    {
        pList->elems[q] = pList->elems[q+1];
    }
    --pList->nCount;                    //表长减去 1, 最后的一个元素实质上被屏蔽了

    return ( TRUE );
}
```

#### 5、求第一个元素

```
ElemType GetHead(PSeqList pList)
{
    assert(pList->nCount > 0);
    return pList->elems[0];
}
```

#### 6、寻找某个值的元素的下标, 不存在返回-1

```
int Locate(PSeqList pList, ElemType x)
{
    int q;
    for ( q=0; q<pList->nCount; q++)
    {
        if( Compare(pList->elems[q], x) == 0)    //Compare 函数根据实际需要来做
            return (q);
    }
    return (-1);
}
```

#### 7、寻找某个位置上的元素

```
ElemType RetriveElem(PSeqList pList, int pos)
{
    assert(0<=pos && pos<pList->nCount);

    return ( pList->elems[pos] );
}
```



## 8、寻找某个位置上的元素的后继值，不存在返回特殊值

```
ElemType GetNext(PSeqList pList, int pos)
{
    assert(0<=pos && pos<pList->nCount-1);
    if(pos < pList->nCount - 1 )
        return pList->elems[pos+1];
    else
        return SPECIAL;                //SPECIAL 根据需要定义
}
```

## 9、寻找某个位置前驱的值，不存在返回特殊值

```
ElemType GetPrevious(PSeqList pList, int pos)
{
    assert(0<=pos && pos<pList->nCount);
    if ( pos>0 && pos<pList->nCount )
        return pList->elems[pos-1];
    else
        // return SPECIAL;
}
```

## 10、创建一个空顺序表，返回指针

```
PSeqList CreateNullList(void)
{
    PSeqList pList;
    pList = (PSeqList)malloc(sizeof(SeqList));
    pList->nCount = 0;
    return pList;
}
```

## 11、初始化一个已经存在的顺序表

```
void InitList(PSeqList pList)
{
    pList->nCount = 0;                //后面的随机得到的数可以通过长度来屏蔽不用全置零
}
```

## 12、判断是否为空

```
BOOL IsListEmpty(PSeqList pList)
{
    return ( pList->nCount == 0 );
}
```

## 13、销毁顺序表(对于用 CreateNullList 创建的顺序表)

```
void DestroyList(PSeqList pList)
```

```
{
    free(pList);
    pList=NULL;
}
```

#### 14、求表长

```
int GetSize(PSeqList pList)
{
    return pList->nCount;
}
```

## 2. 2. 2 单链表的定义与实现

### 一、链表结点及链表的定义

```
Typedef struct
{
    ElemType elems[MAXSIZE];
    Pnode    next;
}Node,*PNode;
Typedef struct
{
    Pnode Head;
    Pnode Tail;
}SeqLink,*PseqLink;
```

### 二、带头结点的单链表的实现

#### 1、在指定位置插入一个元素

```
BOOL Insert(PLinkList pList, ElemType x, int pos)    //pos 从 0 开始数
{
    int i;
    PNode pre, q;    //q 是要插入的, pre 是要插入的前一结点
    assert(pos >= 0);

    pre = pList->head;
    for(i=0; i<pos; i++)
    {
        if(pre != NULL)
            pre = pre->next;
        else    //这时 pre 已经移到链表的外面
            return FALSE;
    }

    q = (PNode)malloc( sizeof(Node ) );    //申请空间
    assert(q != NULL);
    q->elem = x;
```

```
q->next = pre->next;
pre->next = q;

if(q->next == NULL)
    pList->tail = q;           //要更改尾结点!! (头结点为空结点)

return TRUE;
}
```

## 2、在单链表尾增加一个元素

BOOL InsertTail(PLinkList pList, ElemType x)

```
{
    PNode    pNode;
    pNode = (PNode)malloc(sizeof(Node));
    assert(pNode);
    pNode->elem = x;

    pNode->next = pList->tail->next;
    pList->tail->next = pNode;
    pList->tail = pNode;

    return TRUE;
}
```

## 3、在单链表头部增加一个元素

BOOL InsertHead(PLinkList pList, ElemType x)

```
{
    PNode    pNode;
    pNode = (PNode)malloc(sizeof(Node));
    pNode->elem = x;

    if(pList->head->next == NULL)           //如果是空表，那么就插入了尾部
    {
        pList->tail = pNode;
    }
    pNode->next = pList->head->next;
    pList->head->next = pNode;

    return TRUE;
}
```

## 4、在 pNode 所指结点的后面插入一个元素

void InsertNode(PLinkList pList, ElemType x, PNode pNode)

```
{
```



```

PNode      q;
q = (PNode)malloc( sizeof(Node ) );
assert(q);
q->elem = x;
assert(pNode != NULL);
q->next = pNode->next;
pNode->next = q;

if(q->next == NULL)                //如果插入到了尾部
    pList->tail = q;
}

```

### 5、删除某个位置的结点

```

BOOL Delete(PLinkList pList, int pos)
{
    PNode    pre, q;                //删除 q, pre 为欲删除的结点之前的结点
    assert(pos >= 0);
    pre = pList->head;

    for(int i=0; i<pos; i++)        //通过计数找到位置
    {
        if(pre != NULL)
            pre = pre->next;
        else
            return FALSE;          //这时已经走出了链表
    }

    if( pre->next == NULL )        //这样的话上面不会返回 FALSE，不过也已经走出了链表
        return FALSE;
    else
    {
        q = pre->next;
        pre->next = q->next;
        free( q );                //勿忘!!!

        if(pre->next==NULL)        //如果删除的是尾结点
            pLink->Tail = pre;
    }
    return TRUE;
}

```

### 6、返回第一个元素的值

```

ElemType GetHead(PLinkList pList)
{

```

```
PNode pNode = pList->head->next;           //头结点为空结点!!
assert(pNode != NULL);
return pNode->elem;
}
```

#### 7、寻找值为 x 的元素的下标，不存在返回特殊值

```
int Locate(PLinkList pList, ElemType x)
{
    PNode      p;
    int         pos = -1;                     //因为 pos 是从 0 开始数的
    p = pList->head->next;
    while( p != NULL)
    {
        ++pos;
        if(Compare(p->elem, x) == 0)         // Compare 根据实际需要而定
            break;
        else
            p = p->next;
    }
    if (p == NULL)
        return (SPECIAL);
    else
        return pos;
}
```

#### 8、寻找位置为 pos 的元素的值，不存在返回特殊值

```
ElemType RetriveElem(PLinkList pList, int pos)
{
    int         i;
    assert(pos >= 0);
    PNode      p = pList->head->next;

    for(i=0; i<pos; i++)
    {
        if(p != NULL)
            p = p->next;
        else
            return(SPECIAL);
    }
    if (p == NULL)
        return (SPECIAL);
    else

        return p->elem;
}
```

```
}
```

#### 9、获得 pos 位置的后继的值，不存在返回特殊值

```
ElemType GetNext(PLinkList pList, int pos)
```

```
{
    Return RetriveElem(pList, pos+1);
}
```

或者：ElemType GetNext(PLinkList pList, int pos)

```
{
    int i;
    assert(pos >= 0);
    PNode p = pList->head->next;
    for(i=0; i<=pos; i++) //与上面的区别：这里是 i<=pos. (上面是 i<pos)
    {
        if(p != NULL)
            p = p->next;
    }
    assert(p != NULL);
    return p->elem;
}
```

#### 10、取得 pos 位置元素的前驱的值，不存在返回特殊值

```
ElemType GetPrevious(PLinkList pList, int pos)
```

```
{
    PNode pre;
    pre = pList->head; //与上面的区别：是 head 而不是 head->next
    for(int i=0; i<pos; i++)
    {
        if(pre != NULL)
            pre = pre->next;
    }
    if(pre == pList->head) //如果前驱结点是头结点（即没有前驱结点）
        return(SPECIAL);
    else
        return pre->elem;
}
```

#### 11、创建空链表，返回指针

```
PLinkList CreateNullList(void)
```

```
{
    PLinkList pList;
    PNode pNode;
```



```
pList = (PLinkList)malloc( sizeof(LinkList) );           //为链表申请空间
assert(pList);

pNode = (PNode)malloc(sizeof(Node));                     //为头结点申请空间
assert(pNode);

pNode->next = NULL;                                       //初始化
pList->head = pNode;
pList->tail = pNode;
return pList;
}
```

## 12、判断单链表是否为空

```
BOOL IsListEmpty(PLinkList pList)
{
    return (pList->head->next == NULL);
}
```

## 13、销毁单链表

```
void DestroyList(PLinkList pList)
{
    PNode    p = pList->head;
    while(p)
    {
        pList->head = pList->head->next;
        free(p);
        p = pList->head;
        p = NULL;
    }
    free (pList);
}
```

## 14、找到单链表中 pos 位置的元素，返回指针

```
PNode Find(PLinkList pList, int pos )
{
    PNode    p;
    int    j;
    p = pList->head;
    for(j=0; j<=pos; j++)
    {
        p = p->next;
        if( p == NULL )
            return NULL;
    }
}
```

```
    return p;
}
```

#### 15、求单链表的长度（实际存在的元素的个数，不包括头结点）

```
int GetSize(PLinkList pList)
{
    PNode    pNode = pList->head->next;
    int      count;
    count = 0;
    while(pNode != NULL)
    {
        ++count;
        pNode = pNode->next;
    }
    return count;
}
```

#### 16、两个链表相连，连接后仍保持有序

```
void MergeList (LinkList la, LinkList lb, PLinkList plc)
```

（//这是带头结点的情况，如果要处理不带头结点的，可以编一个简单的函数将其转化为带头结点的链表）

```
    PNode    pa, pb, pc;
    pa = la.head->next;
    pb = lb.head->next;
    pc = plc->head = la.head;                //用 la 的头结点做 lc 的头结点
    while(pa && pb)
    {
        if(Compare(pa->elem, pb->elem) <= 0) //这里偷懒了，其实还应该考虑合并的问题
        {
            pc->next = pa;
            pc = pa;
            pa = pa->next;
        }
        else
        {
            pc->next = pb;
            pc = pb;
            pb = pb->next;
        }
    }

    pc->next = pa?pa:pb;                      //将另一个尚未结束的链表中剩下的元素全加到 lc 后
    free(lb.head);                           //注意：只释放头结点，其它结点已经接在 lc 后面了！
}
```

## 2. 2. 3 静态链表的定义与实现

### 一、静态链表的定义

```
typedef struct
{
    Element    data;
    int        cursor;
}Component, SLinkList[MaxSize];
```

### 二、静态链表的实现

#### 1、初始化静态链表

```
void InitList(SLinkList list)
{
    int i;
    for(i = 0; i < MaxSize - 1; i++)
        list[i].cursor = i + 1;
    list[MaxSize-1].cursor = 0;
}
```

#### 2、申请空间，返回申请到的空间的下标，申请不到返回 0

```
int Malloc(SLinkList list)
{
    int i;
    i = list[0].cursor;
    if(list[0].cursor != 0)
        list[0].cursor = list[i].cursor;
    return i;
}
```

#### 3、释放 ‘k’ 指向的空间

```
void Free(SLinkList list, int k)
{
    list[k].cursor = list[0].cursor;
    list[0].cursor = k;
}
```



## 第三章 字符串

### 3.1 需要记住的东西

#### 一、字符串的基本概念

1、串：是由零个或多个字符组成的有限序列，一般记为

$$s = \text{"a1a2...an"} \quad (n \geq 0)$$

其中  $a_i$  可以是字母、数字或其它任何字符。字符串是表中元素为字符的线性表

2、串的长度：串中字符的个数  $n$  称为串的长度。

3、子串：串中任意个连续的字符组成的子序列。

4、主串：包含子串的串相应地称为主串。

5、位置：字符在序列中的序号。子串在主串中的位置则以子串的第一个字符在主串中的位置来表示。

6、相等：两个串的长度相等，并且对应位置的字符都相同。

7、模式匹配：子串（又称*模式串*）在主串中的定位操作。

#### 二、串的逻辑结构和线性表的区别：

数据对象不同

☞ 串的数据对象仅局限于字符；

☞ 线性表的数据对象可以为任何类型的数据；

线性表的基本操作的操作对象不同

☞ 串的操作对象以“串的整体”为主；

☞ 线性表的操作对象以“单个元素”为主；

### 3.2 算法

#### 3.2.1 定长顺序串的实现

1、求从  $s$  所指的字符串中第  $i$  个字符开始连续  $j$  个字符构成的字符串

`PSeqString SubString(PSeqString s, int i, int j)`

```
{
    PSeqString    s1;                                // 指向新串
    int           k, m;
    s1 = CreateNullString( );                        // 创建一空串
    if(s1==NULL)
        return NULL;
    if( i>0 && i<=s->n && j>0 )
    {
        if ( s->n < i+j-1 )
            j = s->n-i+1;                            // 若从 i 开始取不了 j 个字符, 则能取几个就取几个
        for (k=0; k<j; k++)
        {
            s1->c[k]=s->c[i+k-1];
            s1->c[j]='\0';
            s1->n=j;
        }
    }
}
```

```
else
    s1->c[0]='\0';
return(s1);
}
```

- 2、置串为一个空串、判断一个串是否为空串、求一个串的长度、将两个串拼接在一起构成一个新串（同顺序表）

### 3. 2. 2 变长顺序串的实现

#### 1、数据结构的定义

```
typedef struct _string
{
    char *ch;
    int length;
}HString;
```

#### 2、赋值的实现

```
void StrAssign(HString *str, char *chars)
{
    char *p = chars;
    int length, i;

    if(str->ch) // 如果空串，释放已有空间
    {
        free(str->ch);
        str->ch = NULL;
    }

    while(*p++) ; // 求串长，chars 的最后一位是\0!!，不算
    length = p - chars - 1;

    if(length == 0)
        str->length = 0;
    else
    { // 重新申请空间
        str->length = length;
        str->ch = (char *)malloc(sizeof(char)*length);
        for(i=0; i<length; i++)
            str->ch[i] = chars[i];
    }
}
```

#### 3、字符串复制的实现

```
void StrCopy(HString *dest, HString src) //将 src 中的内容复制到 dest 中
{
    char *p;
    int i;
    p = (char *)malloc(sizeof(char) * src.length); //申请空间
```

```

dest->ch = p;
dest->length = src.length;
for(i=0; i<src.length; i++)
    p[i] = src.ch[i];
}

```

### 3. 2. 3 链串的实现

1、求从  $s$  所指的链串中第  $i$  ( $i>0$ ) 个字符开始连续  $j$  个字符所构成的子串

```

PLinkString SubString(PLinkString s, int i, int j)
{
    PLinkString    s1;
    PstrNode        p, q, t;
    int             k;

    s1 = CreateNullString( );                // 创建空链串
    if (i<1 || j<1)
        return(s1);                        // i, j 值不合适, 返回空串

    p = s->head;
    for (k=1; k<i; k++)                    // 找开始字符 (使 p 指向第 i 个结点)
        if ( p != NULL)
            p = p->link;
        else
            return(s1);
    if (p==NULL)                            // 虽然符合上面的条件, 但是 p=p->link, 仍然可能 p=NULL
        return(s1);
    t = (PstrNode)malloc(sizeof(struct StrNode));
    s1->head = t;
    for (k=1; k<=j; k++)                    // 连续取 j 个字符
    {
        t->c = p->c;
        if (p->link!=NULL && k<j)          // 注意: 这里是 k<j, 而不是 k<=j!!
        {
            q = (PstrNode)malloc(sizeof(struct StrNode));
            p = p->link;
            t->link = q;
            t = q;
        }
        else
            break;
    }
    t->link = NULL;
    return(s1);
}

```

### 3. 2. 4 模式匹配



## 一、朴素的模式匹配算法

该算法的时间效率：一般情况： $O(n)$ ；最坏时： $O(n*m)$

求 p 所指的串在 t 所指的串中第一次出现时，所指串的第一个元素在 t 所指的串中的序号

```
int index(PSeqString t,PSeqString p)
{
    int i=0, j=0; //初始化
    while (i<p->n && j<t->n) //反复比较
        if (p->c[i]==t->c[j])
        {
            i++;
            j++;
        }
        else
        {
            j = j - i + 1; //主串、子串回溯，重新开始下一次匹配
            i = 0;
        }

    if (i>=p->n)
        return( j - p->n + 1); //匹配成功
    else
        return( -1 ); //匹配失败
}
```

## 二、KMP 算法

1、KMP 算法（从 pos 位置开始找 p 在 t 中第一次出现时的 t 串中的序号）

```
int Index_KMP(PSeqString t, PSeqString p, int pos)
{
    int i, j, *next;
    next = (int *)malloc(sizeof(int)*(p->n));
    Next(p, next);

    i = pos; j = 0;
    while(i < t->n && j < p->n)
    {
        if(j == -1 || t->c[i] == p->c[j]) // 'j== -1' 意味着失配且无两个相同子串
        {
            ++i; ++j;
        }
        else
            j = next[j];
    }

    free(next);
    if(j >= p->n) return (i - p->n + 1); // 找到
    else return -1; // 没有找到
}
```

```
}
```

## 2、NEXT 数组的求法

```
void GetNext(PSeqString p, int *next)
{
    int    j, k;
    j = 0;   k = -1;
    next[j] = -1;
    while(j < p->n)
    {
        if(k == -1 || p->c[j] == p->c[k])
        {
            ++j; ++k;
            next[j] = k;          /* next[++j] = ++k; */
        }
        else k = next[k];
    }
}
```

## 3、NEXT 数组的改进

```
void GetNext(PSeqString p, int *next)
{
    int    j, k;

    j = 0;   k = -1;
    next[j] = -1;
    while(j < p->n)
    {
        if(k == -1 || p->c[j] == p->c[k])
        {
            ++j; ++k;
            if(p->c[j] != p->c[k])
                next[j] = k;
            else
                next[j] = next[k]; //如果 next[j]=k, 而 pj=pk, 则当 si≠pj 时, 显然也不再需要
                                   //让 si 与 pnext[j], 即 pk 进行比较了, 而只需让 si 与 pnext[k]
                                   //进行比较。
        }
        else k = next[k];
    }
}
```

## 第四章 栈和队列

### 4. 1 需要记住的东西

#### 一、栈

- 1、栈：是一种特殊的线性表，对于它的所有插入和删除都限制在表的同一端（表尾）
- 2、栈顶：表尾端，元素从这儿插入和删除。
- 3、栈底：表头端。
- 4、栈的特点：先进先出

#### 二、栈的应用

##### 1、函数的调用过程

调用前：

- (1) 将所有的实参、返回地址传递给被调用函数保存
- (2) 为被调用函数的局部变量分配存储区
- (3) 将控制转移到被调用函数入口。

调用后：

- (1) 保存被调用函数的计算结果。
- (2) 释放被调用函数的数据区
- (3) 依照被调用函数保存的返回地址将控制转移到调用函数

多个函数嵌套调用时，按照“后调用先返回”的原则进行

2、递归定义：用自身的简单情况来定义自己的方式，称为“递归定义”。

3、静态分配：在非递归调用的情况下，数据区的分配可以在程序运行前进行，一直到整个程序运行结束才释放，这种分配称为静态分配

动态分配：在递归调用的情况下，被调用的函数的局部变量不能分配给固定的某个单元，而必须每调用一次就分配一份，当前程序使用的所有量都必须是最近一次递归调用时所分配的数据区中的量。所以，存储分配只能在执行递归调用时进行。

##### 4、动态分配的通常方法：

- 1) 在内存中开辟一个足够大的动态区，成为运行栈。栈中元素的类型就是被调用函数需要的数据区类型
- 2) 每次调用时，对栈使用 push 操作，分配被调用函数所需的数据区类型  
每次返回时，执行 pop 操作，释放本次调用所分配的数据区，恢复到上次调用所分配的数据区中。其中，被调用函数中变量的地址全部采用相对与栈顶的位移量来表示。

### 4. 2 算法

#### 4. 2. 1 栈的实现

##### 一、顺序表示

```
typedef struct SeqStack
{
    Element    elems[MAXNUMS];    //存储栈中所有的元素
    int         top;                //记录栈顶的位置
}SeqStack;
typedef SeqStack  Stack;
```

##### 二、顺序实现

###### 1、初始化一个栈

```
void InitStack(Stack *st)
```



```
{
    st->top = -1;           // 判断栈是空栈 ⇔ top = -1
}

2、判断一个栈是否是空栈
bool IsStackEmpty(Stack st)
{
    return (st.top < 0);    // 如果是一个空栈, 那么 top<0, 返回 1
}

3、向栈顶插入一个元素
void Push(Stack *st, Element elem)
{
    if(st->top >= MAXNUMS - 1)
        printf("Stack overflow!\n"); // 如果栈已经满了, 就没有地方再插入了
    else
    {
        st->top += 1;           // 栈顶的位置向后移动一位
        st->elems[st->top] = elem; // 将元素插到栈顶
    }
}

4、删除栈中的一个元素 (必然是栈顶的元素)
void Pop(Stack *st)
{
    if( st->top < 0 )
        printf("Stack underflow!\n"); // 如果本来就是空栈, 那么就不用删除了
    else
        st->top -= 1;           // 如果不是空栈, 只用把栈顶的位置移动就可以了, 不用
                                // 改动元素
}

5、取得栈顶元素
Element GetTop(Stack st)
{
    return (st.elems[st.top]);
}

6、清空栈
void DestroyStack(Stack *st)
{
    st->top = -1;           // 这里注意, 不用改动元素, 只要移动栈顶, 自然就屏蔽
                            // 掉了所有元素
}
```

### 三、链接表示

```
struct Node // 单链表结点结构
{
    ElemType info;
    struct Node *link;
}StackNode, *PStackNode;
```

```

struct LinkStack          // 链栈结构类型
{
    PStackNode top;       // 指向栈顶的结点
}LinkStack, *PLinkStack;

```

#### 四、链接表示的实现

##### 1、初始化栈

```

void InitStack(Stack *st)
{
    st->top = NULL;        // 只需将栈顶指针设为空
}

```

##### 2、判断一个栈是否为空

```

bool IsStackEmpty(Stack st)
{
    return (st.top == NULL); // 也就是判断栈顶是否为空
}

```

##### 3、插入一个元素

```

void Push(Stack *st, Element elem)
{
    PNode pNode;           // 指向新结点的指针
    pNode = (PNode)malloc(sizeof(Node)); // 申请空间
    pNode->elem = elem;

    pNode->link = st->top;   // 插入，调整指针顺序
    st->top = pNode;
}

```

##### 4、从栈中删除一个元素

```

void Pop(Stack *st)
{
    PNode pNode;           // 这个指针指向待删除元素
    if(st->top)
    {
        pNode = st->top;    // 删除的肯定是栈顶的指针
        st->top = st->top->link; // top 指针向后移动一位
        free(pNode);        // 释放空间
    }
}

```

## 4. 2. 2 链式队列的实现

### 一、队列的链式表示定义

```

typedef struct _QNode
{
    ElemType info;

```

```

    struct _QNode *link;
}Node, *PNode;
typedef struct
{
    PNode      f;
    PNode      r;
}LinkQueue, *PLinkQueue;

```

## 二、队列的链式表示实现

### 1、创建空队列

```

PLinkQueue CreateEmptyQueue( )
{
    PLinkQueue plqu;
    plqu = (struct LinkQueue *)malloc(sizeof(struct LinkQueue));
    if (plqu!=NULL)
    {
        plqu->f = NULL;
        plqu->r = NULL;
    }
    else
        printf("Out space!! \n");
    return plqu;
}

```

### 2、判断队列是否为空

```

BOOL IsQueueEmpty( PLinkQueue plqu )
{
    return (plqu->f == NULL);
}

```

### 3、向队列中增加一个元素

```

void InQueue( PLinkQueue plqu, Datatype x )
{
    PNode    p;
    p = (PNode)malloc( sizeof( struct Node ) );
    if ( p == NULL )
        printf("out of space!");
    else
    {
        p->info = x;
        p->link = NULL;
        if (plqu->f == NULL)
        {
            plqu->f = p;
            plqu->r = p;
        }
        else

```



```

        {
            plqu->r->link = p;
            plqu->r = p;
        }
    }
}

```

#### 4、出队列

```

void OutQueue( PLinkQueue plqu )
{
    PNode    p;
    if( plqu->f == NULL )
        printf( "Empty queue.\n " );
    else
    {
        p = plqu->f;
        plqu->f = plqu->f->link;
        free(p);
    }
}

```

#### 5、取得队首元素

```

Datatype GetFront( PLinkQueue plqu )
{
    return plqu->f->info;
}

```

### 三、循环队列的定义（同顺序队列）

#### 四、循环队列的实现

##### 1、创建空队列

```

PSeqQueue CreateEmptyQueue( void )
{
    PSeqQueue paqu;
    paqu = (struct SeqQueue *)malloc(sizeof(struct SeqQueue));
    if (paqu==NULL)
        printf("Out space!! \n");
    else
    {
        paqu->f = 0;
        paqu->r = 0;
    }
    return paqu;
}

```

##### 2、判断队列是否为空

```

BOOL IsQueueEmpty( PSeqQueue paqu )
{
    return (paqu->f == paqu->r);
}

```

```
}
```

### 3、入队列

```
void InQueue( PSeqQueue paqu, DataType x )
{
    if( (paqu->r + 1) % MAXNUM == paqu->f )
        printf( "Full queue.\n" );
    else
    {
        paqu->q[paqu->r] = x;
        paqu->r = (paqu->r + 1) % MAXNUM;
    }
}
```

### 4、出队列

```
void OutQueue( PSeqQueue paqu )
{
    if( paqu->f == paqu->r )
        printf( "Empty Queue.\n" );
    else
        paqu->f = (paqu->f + 1) % MAXNUM;
}
```

### 5、取得队首元素

```
DataType GetFront( PSeqQueue paqu )
{
    return paqu->q[paqu->f];
}
```

## 第五章 树和二叉树

### 5.1 树的基本概念

#### 一、树的定义和基本术语

1、**树**：是  $n(n \geq 0)$  个结点的有限集。具有如下特征：

在任意一棵非空树中：

(1) 有且仅有一个特定的称为**根**的结点；

(2) 除根结点外，其余结点可分为  $m(m \geq 0)$  个互不相交的有限集  $T_1, T_2, \dots, T_m$ ，其中每一个集合本身又是一棵树，称为根的**子树(Subtree)**。

2、**空树**：具有 0 个结点的树。

3、**结点**：包含一个数据元素及若干指向其子树的分支。

4、**结点的度**：结点的子女个数

**树的度**：树中度数最大的结点的度数

5、**树叶结点**：终端结点，度数为 0

**分枝结点**：度数大于 0 的节点

6、**边**：父节点  $x$  到子节点  $y$  的有序对  $(x, y)$

7、**结点的层数**：规定根节点的层数为 0，其余节点的层数等于其父节点的层数加 1。

**树的层数**：树中层数最大的结点的层数

**树的高度或深度**：树中结点的最大层数

8、**路径**：如果  $x$  是  $y$  的一个祖先，又有  $x = x_1, x_2, \dots, x_n = y$ ，满足  $x_i (i=0, 1, \dots, n-1)$  为  $x_{i+1}$  的父节点，则称  $x_1, x_2, \dots, x_n$  是从  $x$  到  $y$  的一条路径

**路径长度**：路径中的边数

9、**有序树**：对子树之间的次序加以区别的树

**无序树**：对子树的次序不加区别的树

10、**树的周游**：树的周游是按某种方式系统的访问树中的所有节点的过程，它是每个节点都被访问一次并且只被访问一次。通过一次周游，可以使树中的所有节点，按照某种线形序列进行一次处理。

#### 按深度优先周游

● **先根次序**：首先访问根节点；然后从左到有按先根次序周游根节点的每棵子树。

● **中根次序**：

● **后根次序**：首先按照从左至右的顺序周游根节点的每棵自述；最后访问根节点。

#### 按宽度优先周游

先访问层数为 0 的节点，在从左至右逐个访问层数为 1 的节点；……依此类推，直到访问完树种的全部节点。

11、**树林**：是  $m(m \geq 0)$  棵互不相交的树的集合。对树中每个结点而言，其子树的集合即为树林。

#### 二、二叉树的基本概念

1、**二叉树**：它的每一个结点至多有两棵子树，并且子树有左右之分，不能随意颠倒。

二叉树不是树的特殊情形，它们是两个概念。

树和二叉树之间最主要的差别是：二叉树中结点的子树要区分为左子树和右子树，即使在结点只有一棵子树的情况下也要明确指出该子树是左子树还是右子树。

2、**空二叉树**：具有 0 个结点的二叉树。

3、二叉树的一些概念（同“树”）

4、特殊的二叉树

● **满二叉树**：如果一棵二叉树的任何结点或者是树叶，或者有两棵非空子树，则此二叉树称作“满



二叉树”。

- **完全二叉树**：如果一棵二叉树至多只有最下面的两层结点度数可以小于 2，并且最下面一层的结点都集中在该层最左边的若干位置上，则此二叉树称为“完全二叉树”。完全二叉树不一定是满二叉树。

- **扩充二叉树**：把原二叉树的结点都变为度数为 2 的分支结点，也就是说，如果原结点的度数为 2，则不变，度数为 1，则增加一个分支，度数为 0（树叶）增加两个分支。

- **外部节点**：新增加的节点。

**内部节点**：树中原有的节点。

性质：在扩充的二叉树里，外部结点的个数比内部结点个数多 1。

- **外部路径长度 E**：在扩充二叉树中从根到每个外部节点的路径长度之和。

**内部路径长度 I**：在扩充二叉树中从根到每个内部节点的路径长度之和。

性质： $E = I + 2n$ （ $n$  为扩充二叉树的内部结点个数）

## 5、二叉树的周游：

二叉树的周游是按某种方式系统的访问二叉树中的所有节点的过程，它是每个节点都被访问一次并且只被访问一次。通过一次周游，可以使树中的所有节点，按照某种线形序列进行一次处理。

- 深度优先

- **先根次序**：若二叉树不空，则先访问根；然后按先根次序周游左子树，最后按先根次序周游右子树
- **后根次序**：若二叉树不空，先按后根次序周游左子树，然后按后根次序周游右子树，最后访问根；
- **中根次序（对称序）**：若二叉树不空，先按中根次序周游左子树，然后访问根，最后按中根次序周游右子树；

- 宽度优先

## 三、二叉树的性质

### 1、一般二叉树的性质

性质 1：在二叉树的第  $i$  层上至多有  $2^i$  个结点（ $i \geq 0$ ）。

性质 2：深度为  $k$  的二叉树至多有  $2^{k+1} - 1$  个结点，（ $k \geq 0$ ）。

**证明**：
$$M = \sum_{i=0}^k m_i = \sum_{i=0}^k 2^i = 2^{k+1} - 1$$

性质 3：对任何一棵二叉树  $T$ ，如果其终端结点数为  $n_0$ ，度为 2 的结点数为  $n_2$ ，则  $n_0 = n_2 + 1$ 。

**证明**：设二叉树的结点数为  $n$ ，度数为 1 的结点数为  $n_1$ ，则有： $n = n_0 + n_1 + n_2$

二叉树除根结点外，每一个结点都有一个前驱，设  $B$  为总的边数，则显然有  $B = n - 1$ 。

而每一个分支都是从度数为 1 或者 2 的结点发出的，因此有： $B = 2n_2 + n_1 = n - 1$ ，

即  $n = 2n_2 + n_1 + 1 = n_0 + n_1 + n_2$

由此即得  $n_0 = n_2 + 1$

### 2、完全二叉树的性质：

性质 4：具有  $n$  个结点的完全二叉树的深度  $k$  为  $\lfloor \log_2 n \rfloor$

**证明**：根据性质 2 和完全二叉树的定义有： $2^{k-1} < n \leq 2^{k+1} - 1$ ，即： $2^k \leq n < 2^{k+1}$

对不等式取对数，得： $k \leq \log_2 n < k+1$

由于  $k$  是整数，因此有  $k = \lfloor \log_2 n \rfloor$

性质 5：如果对一棵有  $n$  个结点的完全二叉树按层序从 1 开始编号，则对任一结点  $i$ （ $1 \leq i \leq n$ ），有：

- a)  $i=1$ , 结点  $i$  是根;  $i>1$ , 其双亲结点是  $[i/2]$ 。
- b)  $i*2>n$ , 结点  $i$  无左子节点; 否则其左子节点是结点  $2i$ 。
- c)  $i*2+1>n$ , 结点  $i$  无右子节点; 否则其右子节点是结点  $2i+1$ 。

### 3、满二叉树的性质:

性质 6: 在满二叉树中, 叶子节点的个数比分支个数多 1。

**证明:** 在满二叉树中, 分支节点的度数全部为 2, 其它的节点都是叶子节点, 由性质 3 可得

### 4、扩充二叉树的性质

性质 7: 在扩充二叉树中, 外部节点的个数比内部节点多 1。

性质 8:  $E = I + 2n$  ( $E$  为外部路径长度,  $I$  为内部路径长度,  $n$  为扩充二叉树的内部结点个数)

## 5. 2 树的算法和数据结构

### 5. 2. 1 树的存储表示

#### 一、父指针表示法

1、描述: 用一组连续空间存储树的结点, 并附设一个指示器指示其父结点的位置。

#### 2、结构类型

节点的结构类型如下:

```
typedef struct ParTreeNode
{
```

```
    DataType info;           // 结点中的元素
```

```
    int parent;              // 结点的父结点位置, 根节点的父节点用-1表示
```

```
}ParTreeNode;
```

树的结构类型:

```
typedef struct ParTree
{
```

```
    ParTreeNode nodelist[MAXNUM]; // 节点连续存储
```

```
    int n;                       // 节点的个数
```

```
}ParTree, *PParTree;
```

#### 3、分析:

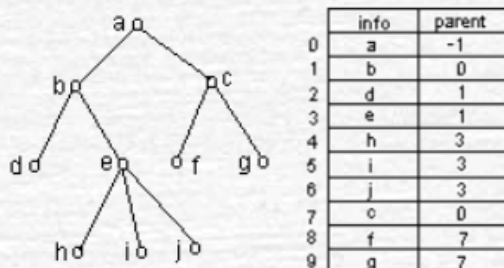
☞ 优点:

- 容易找到父结点及其所有的祖先;
- 能找到结点的子女和兄弟;

☞ 缺点:

- 没有表示出结点之间的左右次序;
- 找结点的子女和兄弟比较费事;

4、改进方法: 按一种周游次序在数组中存放结点。常见的一种方法是依次存放树的先根序列, 如下表



#### 5、基本实现 (按改进后)

## 1) 在父指针表示的树中求右兄弟节点的位置

```

int RightSibling_partree ( PParTree t , int p )
{
    int i;
    if( p>=0 && p< t->n )           // 判断 p 是树中的节点
        for ( i = p+1 ; i < t->n ; i++)
            if( t->nodelist[i].parent == t->nodelist[p].parent)    // 判断有相同的父节点
                return i ;           // 由于是按照改进后的，找到的第一个就是它的右兄弟
    return (-1);                     // 没有找到情况
}

```

## 2) 在父指针表示的树中求最左子节点节点的位置

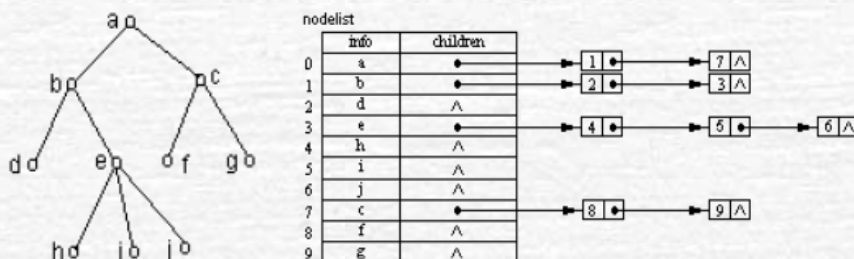
```

int leftChild_partree(PParTree t, int p)
{
    if( t->nodelist[ p+1 ].parent ==p )
        return (p+1);               // 因为是按照改进的情况，所以如果有子节点一定是表中下一个
    else return (-1);
}

```

## 二、子表表示法

1、描述：结点表中的每一元素又包含一个子表，存放该结点的所有子结点。结点表顺序存放，子表用链接表示。如图：



## 2、数据结构

子表中节点的结构：

```

typedef struct EdgeNode
{
    int node_position;           // 实际是边的结构
    struct EdgeNode* link;      // 子节点在节点表中的位置
                                // 指向下一个子表中的节点（右兄弟）
}EdgeNode;

```

结点表中节点的结构：

```

typedef struct ChiTreeNode
{
    DataType info;              // 节点本身的信息
    EdgeNode *children;         // 本节点子表的头指针，若无子节点，则为 NULL
}ChiTreeNode;

```

树结构

```

typedef struct ChiTree
{
    ChiTreeNode node_list[MAXNUM]; // 节点表
}

```



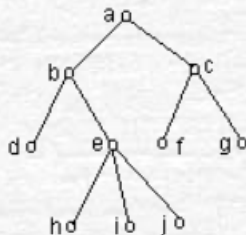
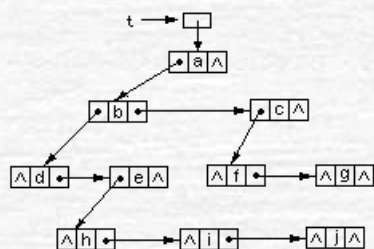
```

    int    root;                // 根结点的位置
    int    n;                   // 树中结点的个数
}ChiTree, *PChiTree;
3、评价
    优点
        ● 求某个节点的最左子节点运算容易实现
        ● 找到节点的全部子节点也很容易
    缺点
        ● 求某个节点的父节点费事
        ● 找到节点的右兄弟也很费事（要先找到父节点）
4、最基本的实现
1) 在树的字表表示中求父节点的位置
int parent_chitree(PChiTree t ,int p)
{
    int i;
    EdgeNode *v;
    for (i=0 ; i < t->n ; i++)                // 逐个检查树中的每一个节点，是不是父节点
    {
        v= t ->nodelist[i].children;          // 若检查的节点的子表中有 p，则返回值是该节点的位置
        while (v !=NULL)
            if (v ->nodeposition == p)
                return(i);
            else v = v -> link;
    }
    return (-1);                                // 无父节点，返回-1
}
2) 在树的子表表示中求右兄弟节点的位置
int rightSibling_chitree (PChiTree t, int p)
{
    int i;
    EdgeNode* v;                                // 用与记录中间变量
    for (i=0 ; i < t->n ; i++)                // 对节点表进行周游
    {
        v= t->nodelist[i].children;            // 对于确定的节点， 遍历它的子节点
        while (v!= NULL)
        {
            if ( v->nodeposition == p)
                if( v->link ==NULL )    return (-1)
                else    return (v ->link ->nodeposition);
            else v = v ->link;
        }
    }
}

```

### 三、长子兄弟表示法

1、描述：在树的每个节点中除其信息域，再增加两个指针，一个指向其最左子节点，一个指向右兄弟。



#### 2、数据结构

##### 节点的结构

```
typedef struct CSNode
```

```
{
```

```
    DataType info;                // 信息域
```

```
    struct CSNode *lchild;        // 指向最左子节点
```

```
    struct CSNode *rsibling;      // 指向右兄弟节点
```

```
}CSNode, *PCNode;
```

##### 树的结构类型

为了运算和参数传递的方便，直接将树定义为指向节点的指针类型

```
typedef struct CSNode * CTree;
```

#### 3、分析

##### 优点

- 在长子-兄弟表示法中，找节点的全部子节点容易
- 找右兄弟节点也容易

##### 缺点

- 找父节点困难（需要周游）
- 找左兄弟也困难

## 5. 2. 2 树的周游

### 一、先根次序周游算法

```
void preOrder(PNode p)
```

```
{
```

```
    PNode c;
```

```
    visit(p);
```

```
// visit 的具体实现不用考虑
```

```
    c=leftChild(p);
```

```
    while( c!=NULL )
```

```
    {
```

```
        preOrder(c );
```

```
        c = rightSibling(c);
```

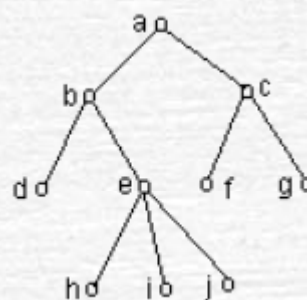
```
    }
```

```
}
```

#### 2、非递归算法

```
void preOrder( PNode p)
```

```
{
```



```

PNode    c;
Stack     s;
s = createEmptyStack();
c = root(p);
do
{
    while(c )
    {
        visit(c );
        push(s, c);
        c = leftChild(c );
    }
    while((c==NULL && !isEmptyStack(s)))
    {
        c = rightSibling(top(s));
        pop(s);
    }
}while(c!=NULL);
}

```

## 二、中根次序周游算法

// 向左走到底

```
void inOrder(PNode p)
```

```

{
    PNode    c;
    c = leftChild(p);
    if(c == NULL)
        visit(p);
    else
    {
        inOrder( c );

        visit(p);

        c = rightSibling(c);
        while(c != NULL)
        {
            inOrder(c);
            c = rightSibling(c);
        }
    }
}

```

// 走到底再访问

// 如果没有走到底

// 左

// 中

// 右

## 三、后根次序周游算法

```
void postOrder(PNode p)
```

```

{
    PNode    c;
    c = leftChild(p);

```

// 向左走到底



```

while(c != NULL)
{
    postOrder(c);
    c = rightSibling(c);           // 处理完左边处理右边
}
visit(p);                        // 最后是根
}

```

### 三、宽度方向周游

描述：首先把被周游的书送入队列，而后，每当从队首取出一棵树，访问其根节点之后，马上把它的子树按从左至右的次序送入队列尾端；重复此过程直到队列为空。

```

void levelOrder (PNode t)
{
    PNode c;
    Queue q;
    q= createEmptyQueue();
    c=t;
    InQueue(q, c);
    while(IsQueueEmpty(q) == FALSE)
    {
        c = GetFront(q);
        OutQueue(q);
        visit(c);
        c = leftChild(c);
        while(c != NULL)
        {
            InQueue(q, c);
            c = rightSibling(c);
        }
    }
}

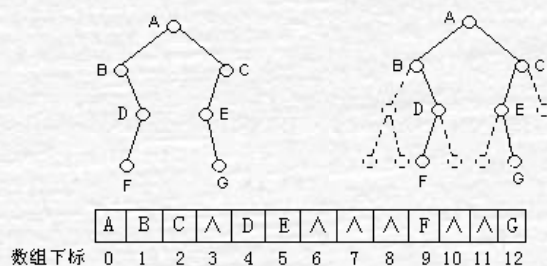
```

## 5. 3 二叉树的算法和数据结构

### 5. 3. 1 二叉树的存储表示

#### 一、顺序表示

1、描述：用一组地址连续的存储单元依次自上而下、自左而右存储完全二叉树的结点。对于一般树，则应将其每个结点与完全二叉树上的结点相对照，存储在一维数组的相应分量中（如下图）。



#### 2、数据结构

**节点的结构**

```
typedef struct SeqBTree
{
    DataType    nodelist[MAXNODE];
    int         n;                // 改造成完全二叉树后，结点的个数
}SeqBTree, *PSeqBTree;
```

**3、基本实现****1) 返回下标为 p 的节点的父节点的下标**

```
int parent_seq ( PseqBinTree t , int p )
{
    if (p<0 || p>=t->n )
        return -1;
    return (p-1)/2;                // 下标为 p 的节点的父节点的下标是 (p-1)/2
}
```

**2) 返回下标为 p 的节点的左子节点的下标**

```
int leftChild_seq ( PseqBinTree t , int p)
{
    if (p<0 || p>=t->n )
        return -1;
    return 2 * p + 1;                // 下标为 p 的节点的左子节点的下标是 2 * p + 1
}
```

**二、链接表示****1、二叉链表：数据域、左右指针**

```
typedef struct BinTreeNode
{
    DataType    info;
    struct BinTreeNode *llink;
    struct BinTreeNode *rlink;
}BinTreeNode, *PBinTreeNode, BinTree, *PBinTree;
```

**2、三叉链表：数据域、左右指针和父指针****5. 3. 2 二叉树的周游****一、先根次序****1) 递归算法**

```
void preOrder (BinTree t)
{
    if( t== NULL)
        return;
    visit(root (t));
    preOrder( leftChild(t));
    preOrder( rightChild(t));
}
```

**2) 非递归算法**

主要思路：从二叉树的根节点 p 开始，将 p 压入栈中；然后置 p 为当前二叉树的左子树，若左子树不空，

继续将左子树入栈，再进入其左子树，如此重复进行，知道 p 为空时，从栈中弹出栈顶元素赋给变量 p，访问后置 p 为它右子树。重复执行上述过程，直到 p 为空并且栈也为空时，周游结束

```
void preOrder (BinTree t)
{
    stack s
    BinTreeNode *c;           // 注意：栈中元素是 BinTree, 而不是节点
    if ( t==NULL )    return;

    s=createEmptyStack();
    pust (s,t);
    while (!isEmptyStack (s))
    {
        c = top(s);           // 取栈顶，出栈
        pop(s);
        if( c!=NULL )
        {
            visit (root(c));   // 访问
            push( s, rightChild(c)); // 右子树进栈
            push( s, leftChild(c));  // 左子树进栈
        }
    }
}
```

3) 算法改进：算法改进：访问一个结点后，仅当该结点左、右子树都不空时才把该结点的右子女入栈。以此减少入栈次数。

## 二、对称序

### 1) 递归算法

```
void inOrder ( BinTree t)
{
    if ( t==NULL )    return;
    inOrder (leftChild(t));
    visit(root(t));
    inOrder(rightChild(t));
}
```

### 2) 非递归算法

**主要思路：**若二叉树不为空时，则沿其左子树前进，在前进的过程中，把经过的二叉树逐个压入栈中，当左子树为空时，弹出栈顶元素，并访问该二叉树的根，如果它有右子树，在进入当前二叉树的右子树，从头执行上述过程；如果它没有右子树，则弹出栈顶元素，从前面继续执行。知道当前二叉树为空并且栈也为空时，周游结束。

```
void inOrder (BinTree t)
{
    stack s = createEmptyStack();
    BinTree c = t;           // 注意：栈中元素是 BinTree, 而不是节点
    if( c==NULL )    return;
    do
```



```

{
    while (c!=NULL)
    {
        push (s,c);
        c = leftChild(c);
    }
    c = top(s);
    visit (root(c));
    c= rightChild(c);
}while(c!=NULL || !isEmptyStack(s));
}

```

### 三、后根次序周游算法

#### 1) 递归算法

```

void postOrder(BinTree t)
{
    if(t==NULL) return;
    postOrder( leftChild(t));
    postOrder( rightChild(t));
    visit(root(t));
}

```

#### 2) 非递归算法

**主要思路：** 对一个二叉树的根节点访问前，要两次经过这个二叉树：首先是由该二叉树找到其左子树，周游其左子树，周游完返回到这个二叉树；然后由该二叉树找到其右子树，周游其右子树，周游完再次返回到这个二叉树，这时才能访问该二叉树的根节点。

注意，为了使每个二叉树都只进栈出栈一次，要对二叉树出栈时增加判断。如果从栈顶二叉树的右子树回来，就执行出栈，访问该二叉树的根节点。

```

void PostOrder( BinTree t )
{
    stack s = createEmptyStack();
    BinTree p,pr; // 注意：栈中元素是 BinTree,而不是节点
    BinTree p=t;

    while (p!=NULL ||!isEmptyStack(s)) // 注意，这里是“或”的关系，两样满足一样就继续循环
    {
        while( p!=NULL // 循环到当前需要处理的节点
        {
            push (s,p);
            pr = rightChild(p);
            p = leftChild (p);
            if ( p==NULL )
                p=pr;
        }
        p=top(s);
        pop(s);
    }
}

```

```

    visit(root(p)); // 走到头以后，p 树只有一个节点
    if(!isEmptyStack(s) && leftChild(top(s))==p) // 栈不空，且从左子树退回
        p=rightChild(top(s)); // 从右子树回来，退到上一层处理
    else p=NULL;
}
}

```

#### 四、广度优先周游算法

```

void levelOrder( BinTree t)
{
    BinTree c, cc;
    Queue q = createEmptyQueue();
    if(t==NULL) return;
    c=t;
    inQueue(q, c);
    while(!isEmptyQueue(q))
    {
        c=frontQueue(q);
        outQueue(q);
        visit(root(c));

        cc=leftChild(c);
        if(cc!=NULL);
        inQueue(q, cc);

        cc=rightChild(c);
        if(cc!=NULL);
        inQueue(q, cc);
    }
}

```

#### 五、二叉树周游的一个应用

对右图进行先根、后根和中根次序的周游得到如下的结点序列：

先根： $+-a/bcd$

后根： $abc/-d+$

对称序： $a-b/c+d$

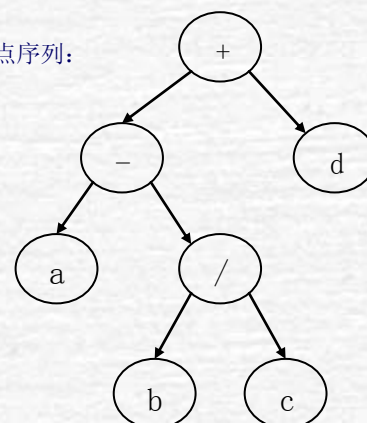
这些序列分别称为表达式

的前缀表示法(Polish Notation,

由波兰逻辑学家 Lukasiewicz 发明)、

后缀表示法（逆波兰表示法）和中缀法。

逆波兰表示法可以简化  
表达式的计算。



## 5. 4 哈夫曼树和哈夫曼

### 5. 4. 1 基本概念

#### 一、哈夫曼树

1、**带权的外部路径长度**:从根结点到各个外部结点的路径长度与相应结点权值的乘积的和。

$$WPL = \sum_{i=1}^m w_i l_i$$

其中  $w_i$  是第  $i$  个外部结点的权值,  $l_i$  是从根到第  $i$  个外部结点的路径长度,  $m$  为外部结点的个数。

2、**哈夫曼树**:带权的外部路径长度最小的扩充二叉树。

## 二、哈夫曼算法

算法的基本思想:

- 根据给定的  $n$  个权值  $\{w_1, w_2, \dots, w_n\}$ , 构成  $n$  棵二叉树的集合  $F = \{T_1, T_2, \dots, T_n\}$ , 其中每一棵二叉树  $T_i$  中只有一个带权为  $w_i$  的根结点, 其左右子树为空。
- 在  $F$  中选取两棵权值最小的二叉树作为左右子树以构造一棵新的二叉树, 且新二叉树的根结点的权值为其左右子树根结点权值之和。
- 在  $F$  中删除这两棵二叉树, 同时将新得到的二叉树加入  $F$  中。
- 重复 2 和 3, 直到  $F$  中只含一棵二叉树为止。

## 三、哈夫曼编码

1、**前缀编码**: 在一个字符集中, 任何一个字符的编码都不是另一个字符编码的前缀。

可以利用二叉树来设计二进制的前缀编码。约定左分支表示字符 ‘0’, 右分支表示字符 ‘1’, 则可以用从根结点到叶子结点的路径上的分支字符串作为该叶子结点字符的编码。如此得到的编码必是前缀编码。

2、**哈夫曼编码**: 用构造哈夫曼树的过程可以生成最短的二进制前缀编码, 该编码就是哈夫曼编码。

3、**哈夫曼编码的意义**: 出现频率大的字符的编码较短, 出现频率小的字符编码较大。有利于提高压缩比, 节省存储空间和传输带宽

## 5. 4. 2 哈夫曼树的数据结构和基本算法

### 一、数据结构

#### 1、哈夫曼树节点的数据结构

```
struct HtNode
{
    int    ww;                // 权值
    int    parent;           // 父结点指针
    int    llink, rlink;     // 左右子节点
}HTNode;
```

#### 2、哈夫曼树的数据结构

```
struct HuffTree
{
    HTNode  ht[MAXNUM];      // 存放 2*m-1 个节点的数组
    int     root;            // 哈夫曼树根在数组中的位置
    int     n;               // 外部节点的个数
}HuffTree, *PHuffTree;
```

### 二、基本的算法

1、构造具有  $m$  个叶结点的哈夫曼树 (数组  $w$  存放权值)

PHuffTree Huffman(int m, int \*w)

```
{
    PhuffTree  pHT;          // 这是要构造的树的指针
    int        i, j;
```



```

int x1, x2; // 存放权值最小的两个结点的位置
pHT = (PHuffTree)malloc(sizeof (HuffTree)); // 分配空间, 创建空哈夫曼树
for( i=0; i<2*m - 1; i++ ) // 置初态
{
    pHT->ht[i].llink = -1; // 指针指向空位置
    pHT->ht[i].rlink = -1;
    pHT->ht[i].parent = -1;
    if (i<m)
        pHT->ht[i].ww = w[i]; // 让数组中前 m 个节点做叶子节点
    else
        pHT->ht[i].ww = -1; // 剩下的是内部节点
}

for( i=0; i < m - 1; i++ ) // 每循环一次构造一个内部结点
{
    Select(pHT, m+i, x1, x2); // 作用: 从哈夫曼树中 pos 位置开始选取两个权值最小的
                                // 结点, 并把结点位置存放在 x1 和 x2 指向的空间中
    pHT->ht[x1].parent = m + i; // 构造一个内部结点
    pHT->ht[x2].parent = m + i;
    pHT->ht[m+i].ww = pHT->ht[x1].ww + pHT->ht[x2].ww;
    pHT->ht[m+i].llink = x1;
    pHT->ht[m+i].rlink = x2;
    pHT->root = m+i;
}
return pHT;
}

2、从哈夫曼树中 pos 位置开始选取两个权值最小的结点, 并把结点位置存放在 x1 和 x2 指向的空间中
void Select(PHuffTree pHT, int pos, int x1, int x2)
{
    int m1 = MAXINT, m2 = MAXINT; // 相关变量赋初值
    for(j=0; j<pos; j++) // 找两个最小权的无父结点的结点
    {
        if (pHT->ht[j].ww<m1 && pHT->ht[j].parent==-1) // x1 中存放最小权的无父节点节点下标
        {
            m2 = m1;
            x2 = x1;
            m1 = pHT->ht[j].ww;
            x1 = j;
        }
        else if(pHT->ht[j].ww<m2 && pHT->ht[j].parent==-1) // else if 使排除最小的权
        {
            // x2 中存放次小权的无父节点节点下标
            m2 = pHT->ht[j].ww;
            x2 = j;
        }
    }
}

```

## 第六章 字典和检索

### 6. 1 字典的基本概念

- 1、**字典**：元素的有穷集合，其中每个元素由两部分组成，分别称为元素的“关键码”和“属性”。字典中的两个元素能够根据其关键码进行比较。
- 2、**关联**：在字典的元素中，包含关键码和属性的二元组称为关联
- 3、**检索**：给定一个值 key，在字典中找到关键码等于 key 的元素。如果找到，检索成功；否则检索失败。
- 4、**静态字典**：一经建立就基本固定不变的字典  
**动态字典**：建立以后经常需要更新的字典。
- 5、**平均检索长度 ASL**：是衡量字典检索算法的检索效率主要标准，是检索过程中和关键码比较的次数

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

其中，n 是字典中元素的个数， $p_i$  是查找第 i 个元素的概率， $c_i$  是找到第 i 个元素的比较次数。

### 6. 2 字典的存储表示方法

#### 6. 2. 1 顺序表示

##### 一、顺序表示的数据结构

```
typedef int KeyType;
typedef int DataType;

typedef struct
{
    KeyType key;           // 字典元素的关键码字段
    DataType other;        // 字典元素的属性字段
}DicElement;

typedef struct
{
    DicElement element[MAXNUM]; // 存放字典中的元素
    int n;                     // 字典中实际元素的个数
}SeqDictionary;
```

##### 二、顺序检索

- 1、主要思路：从字典的一端开始顺序扫描，将字典中的元素的关键码与定值比较，如果相等，则检索成功；当扫描结束时，还未找到关键码等于给定元素，则检索失败。

##### 2、代码：

```
int seqSearch (SeqDictionary *pdic, KeyType key, int *position )
{
    int i;
    for(i=0;i< pdic->n ; i++) // 从头开始向后扫描
    {
        if(pdic->element[i].key ==key)
        {
            *position = i;
            return (i);
        }
    }
}
```

```

    }
    *position = pdic->n;           // 检索失败
    return(0);                     // *position 指向可以插入的位置
}

```

### 3、顺序检索的优缺点

☞ 优点：简单且适用面广，对表的结构没有要求，无论记录是否按关键字有序都可应用。

☞ 缺点：效率低。

### 4、算法分析

1) ASL 的分析（假设每个元素的检索概率相等）

$$ASL = \sum_{i=1}^n p_i c_i = \sum_{i=1}^n i/n = (n+1)/2$$

2) 平均检索时间： $O(n)$ 。

## 三、二分法检索

### 1、基本思想

这种方法检索时，要求字典的元素在顺序表中按关键码排序

首先将给定的值 key 与数组中间上元素的关键码比较，如果相等，则检索成功；否则，若 key 小，在数组前半部分进行二分法检索，否则在数组后半部分进行二分法检索。

### 2、代码

```

int binarySearch(SeqDictionary *pdic, KeyType key, int *position)
{
    int low, mid, high;
    low=0;   high= pdic->n-1;
    while(low<= high)           // 循环的条件
    {
        mid= (low+ high)/2;      // 中间位置（它自动就取整了）
        if ( pdic->element[mid].key ==key)
        {
            *position = mid;
            return (TRUE);       // 检索成功
        }
        else if(pdic->element[mid].key >key) // 如果比中间的值小
            high=mid-1;          // 在前半区寻找
        else                     // 如果比中间的值大
            low=mid+1;           // 在后半区寻找
    }
    return (FALSE);
}

```

### 3、二分法检索的优缺点：

☞ 优点：速度快；

☞ 缺点：只适用于顺序存储结构；且元素必须按关键码有序；

### 4、算法分析

1) ASL（假设每个元素被检索的概率相同）



$$ASL = \frac{1}{n} \times \left( \sum_{i=1}^j i \times 2^{j-i} \right) = \frac{1}{n} \times \sum_{i=1}^j \sum_{m=1}^j 2^{m-1} = \frac{1}{n} \times \sum_{i=1}^j (2^j - 2^{i-1}) = \frac{n+1}{n} \times \log_2(n+1) - 1$$

2) 时间复杂度  $O(\log_2 n)$

#### 四、斐波那契查找

- ☞ 利用斐波那契数列来对字典进行划分。
- ☞ 优点
  - 平均性能比二分法好
  - 分割时只需进行加减法
- ☞ 缺点：
  - 最坏情况下比二分法慢（但仍是  $O(\log_2 n)$ ）

#### 五、插值查找

$$i = \frac{key - ST.elem[l].key}{ST.elem[h].key - ST.elem[l].key} (h - l + 1)$$

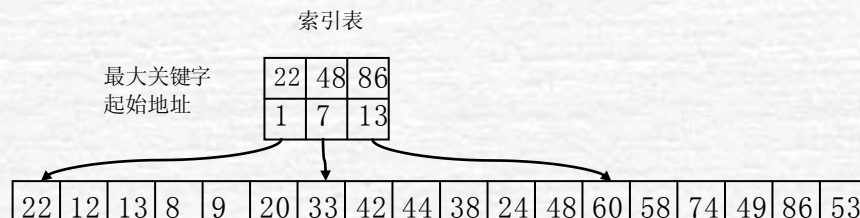
其中  $h, l$  为有序表中具有最小关键字和最大关键字的记录的位置。 $i$  为待比较的关键字的位置。

优点：

在关键字分布比较平均时平均性能比二分法高。

#### 六、索引顺序法

方法：顺序表+索引表。如下图：



整个表分成三个子表，对每个子表建立一个索引项，其中包括两项内容：关键字项（其值为该子表内的最大关键字）和指针项（指示该子表的第一个记录在表中的位置）。

## 6. 2. 2 散列表示

### 一、散列表定义

- 1、**散列法**：在记录的存储位置和关键字之间建立一个确定的对应关系，使每个关键字都和结构中一个唯一的存储位置对应。
- 2、**散列函数**：记录的存储位置和关键码之间的对应关系。
- 3、**碰撞**：对不同的关键码得到同一散列地址的情况。
- 4、**同义词**：发生碰撞的两个(或多个)关键码。
- 5、**散列表**：用散列法形成的字典。

散列表的查询与普通查询方法的区别：

- 普通查询方法基于元素之间的比较
- 散列表查询基于关键字和存储地址之间的对应关系。

6、**散列地址**：和元素的关键码对应的存储位置。

7、**构造散列函数的准则**：使关键字经过散列函数得到一个“随机的地址”，以便使一组关键字的散列地址

均匀地分布在逐个地址区间，从而减少冲突。

8、**负载因子** = 散列表中结点数 / 基本区域能容纳的结点数

## 二、散列函数的构造方法

- 1、直接地址法：取关键字或关键字的某个线性函数值为散列地址。如年龄。
- 2、数字分析法：假设关键字是以  $r$  为基的数，并且散列表中可能出现的关键字都是事先知道的，则可取关键字的若干数位组成散列地址
- 3、平方取中法：取关键字平方后的中间几位为散列地址。
- 4、折叠法：将关键字分割成位数相同的几部分，然后取这几部分的叠加和（舍去进位）作为哈希地址。
- 5、除留余数法：取关键字被某个不大于哈希表长度  $m$  的数  $p$  除后所得余数为哈希地址。数  $p$  的选取：一般可选它为质数或不包含小于 20 的质因素的和数。
- 6、随机数法：通常当关键字的长度不等时采用此法较恰当。
- 7、基数转换法：把关键码看作是另一个进制的表示，然后再转换成原来进制的数，最后用数字分析法取其中几位作为散列地址。

选取散列函数需考虑的因素：

- 计算哈希函数所需时间
- 关键字的长度
- 哈希表的大小
- 关键字的分布情况
- 记录的查找频率

## 三、碰撞处理方法

### 1、开地址法

算法基本思想：

在基本区域内形成一个同义词的探索序列，沿着探索序列逐个查找，直到找到查找的元素碰到一个未占用的地址为止。若插入元素，则碰到空的地址单元就存放要插入的同义词；若检索元素，则碰到空的地址单元后才能说明表中没有待查的元素。

#### 1) 线性探查法

基本思想

将基本区域看作一个循环表，若在地址为  $d = h(\text{key})$  的单元发生碰撞，则依次探索下述地址单元：

$d+1, d+2, \dots, m-1, 0, 1, \dots, d-1$  ( $m$  为基本存储区长度)

直到查找到一个空单元或查找到关键码为  $\text{key}$  的元素为止。如果从单元  $d$  开始探索，查找一遍后有回到原地址，则说明存储区溢出。

算法

```
typedef SeqDictionary HashDictionary;           // 散列字典跟顺序字典在结构上是一样的
#define nil -1                                  // nil 为空结点标记
```

线性探查法解决碰撞的散列表的检索

```
BOOL LinearSearch(HashDictionary *pHash, KeyType key, int *position)
{
    int d;                                     // d 存放散列地址
    int inc;
    d = HashAddress(key);                     // d 为散列地址，散列函数为 HashAddress(key)
```



```

for(inc=0; inc<pHash->n; inc++)           // Hash 字典结构的成员 n 存放的是字典的大小
{
    if(pHash->element[d].key == key)
    {
        *position = d;                   // 检索成功
        return(TRUE);
    }
    else if(pHash->element[d].key == nil)
    {
        *position = d;                   // 检索失败，找到插入位置
        return(FALSE);
    }
    else                                  // 即这时发生了碰撞，散列地址上的 key 值不对
        d = (d+1) % pHash->n;
}
*position=-1;                             // 最终既没有检索到也没有找到插入位置，散列表溢出
return(FALSE);
}

```

散列表的插入算法(用线性探查法解决碰撞)

```

BOOL LinearInsert(HashDictionary *pHash, KeyType key)
{
    int    position;
    if(LinearSearch(pHash, key, &position) == TRUE ) // 散列表中已有关键码为 key 的结点
        printf( "Find\n" );
    else if(position != -1)
        pHash->element[position].key=key;           // 插入结点
    else
        return(FALSE);                             // 散列表溢出
    return(TRUE);
}

```

👉 评价：

- 优点：可以保证找到一个不发生冲突的地址
- 缺点：容易出现二次聚集

👉 **堆积**：用线性探索法解决碰撞的时候，两个同意子表结合在一起的现象称为堆积。

## 2) 双散列函数法（略）

### 2. 拉链法

1) 描述：将所有关键字为同义词的记录存储在同一线性链表中。最多可以建立  $m$  个链表。如果地址没有存放任何元素，则对应的链表为空链表。

2) 开链法存储结构

```

#define  MaxHashSize  200
typedef  int  KeyType;
typedef struct

```



```

{
    KeyType    key;
}ElemType;
哈希表的结点结构
typedef struct _Chain
{
    ElemType    data;
    Struct _Chain    *next;
}Chain;
哈希表的结构
typedef Chain *ChainHash[MaxHashSize];
3) 代码 (略)

```

## 6. 2. 3 二叉树表示

### 一、二叉排序树

1、**二叉排序树**：或者是一棵空二叉树；或者具有下列性质的二叉树：

- 若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值；
- 若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值；
- 它的左右子树也分别为二叉排序树。

二叉排序树的特点：

中根周游的结果就是从小到大排序的序列

2、二叉排序树的存储结构

```

typedef struct BinNode
{
    KeyType    key;                // 结点的关键码字段
    DataType    other;            // 结点的属性字段
    struct BinNode    *llink;    // 二叉树的左指针
    struct BinNode    *rlink;    // 二叉树的右指针
}BinTreeNode, *PBinTreeNode, BinTree, *PBinTree;

```

### 3、检索

二叉排序数结点的搜索（非递归算法）没有搜索到时，返回待插入的位置

```

BOOL SearchNode(PBinTree pTree, KeyType key, PBinTreeNode *position)
{
    PBinTreeNode pNode, pPreNode;
    pNode = pTree;
    pPreNode = pNode;                // pPreNode 用来记录先前位置
    while(pNode != NULL)
    {
        pPreNode = pNode;
        if(pNode->key == key)        // 检索成功
        {
            *position = pNode;
            return(TRUE);
        }
    }
}

```

```

        else if(pNode->key > key)           // 进入左子树继续检索
            pNode = pNode->llink;
        else                               // 进入右子树继续检索
            pNode = pNode->rlink;
    }
    *position = pPreNode;
    return(FALSE);                       // 检索失败
}

```

#### 4、插入节点

```

int InsertNode(PBinTree *ppTree, KeyType key) // 返回插入成功与否的标志
{
    PBinTreeNode pNode, pTempNode;
    if(!SearchBSTForIns(*ppTree, key, &pNode)) // 如果原来的树中没有这样的节点(可以插入)
    {
        pTempNode = (PBinTree)malloc(sizeof(BinTreeNode)); // 待插节点
        pTempNode->key = key;
        pTempNode->llink = pTempNode->rlink = NULL;

        if(!pNode) // 如果原来是空树
            *ppTree = pTempNode;
        else if(Compare(key, pNode->key) < 0) // 比直接的根节点小
            pNode->llink = pTempNode;
        else
            pNode->rlink = pTempNode;
        return OK;
    }
    else // 已经存在这样的节点
        return ERROR;
}

```

#### 5、二叉排序树的构造

```

void CreateSortTree(PBinTree *ppTree, SeqDictionary dic) // 来自字典 dic
{
    int i;
    for(i=0; i<dic.n; i++)
        InsertNode(ppTree, dic.element[i].key); // 将新结点插入树中
}

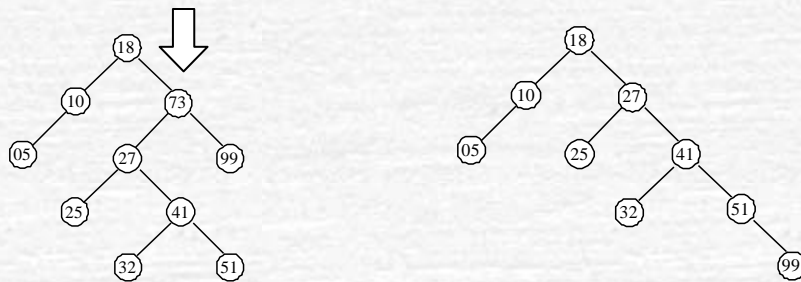
```

#### 6、二叉排序树结点的删除

##### 方法一

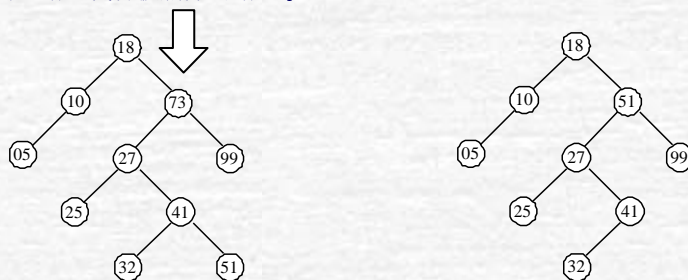
- 如果被删除结点 p 没有左子树，则用 p 的右子女代替 p 即可。
- 否则，在 p 的左子树中，对称序周游找出最后一个结点 r (r 一定无右子女)，将 r 的右指针指向 p 的右子女，用 p 的左子女代替 p 结点。

如图：



### 方法 2:

- 如果被删除结点  $p$  没有左子树，则用  $p$  的右子女代替  $p$  即可；
- 否则，在  $p$  的左子树中，对称序周游找出最后一个结点  $r$  后，将  $r$  删除（ $r$  一定无右子女，用  $r$  的左子女代替  $r$  结点即可）；
- 用  $r$  结点代替被删除的结点  $p$ 。




### 删除关键字为 key 的节点（方法一）

```
void DeleteNode(PBinTree *ppTree, KeyType key)
{
    PBinTreeNode pParent, pNode, pRNode;
    pNode=*ppTree;      pParent=NULL;
    while(pNode != NULL)    // ①找到这个节点
    {
        if(pNode->key == key)
        {
            break;          // 找到了关键字为 key 的节点
        }
        pParent = pNode;    // 用于临时记录，便于回溯
        if(pNode->key>key)
            pNode=pNode->llink;    // 进入左子树
        else
            pNode=pNode->rlink;    // 进入右子树
    }
    if(pNode==NULL)
        return;            // 二叉排序树中无关键字为 key 的节点
    if(pNode->llink==NULL)  // 考察②结点*p 无左子树
    {
        if(pParent==NULL)  // 被删除的节点是原二叉排序树的根节点
            *ppTree=pNode->rlink;
        else if(pParent->llink==pNode) // *pNode 是其父结点的左子女
            pParent->llink=pNode->rlink; // 将*p 的右子树链到其父结点的左链上
        else
            pParent->rlink=pNode->rlink; // 将*p 的右子树链到其父结点的右链上
    }
}
```



```
}
else // ③结点*pNode 有左子树
{
    pRNode=pNode->llink;
    while(pRNode->rlink!=NULL)
        pRNode=pRNode->rlink;
    pRNode->rlink=pNode->rlink;
    // 在*p 的左子树中找最右下结点*pRNode
    // 用最右下结点*pRNode 的右指针指向*p 的右子女

    if(pParent==NULL) // 被删除的结点是原二叉排序树的根结点
        *ppTree=pNode->llink;
    else if(pParent->llink==pNode) // *pNode 是其父结点的左子女
        pParent->llink=pNode->llink;
    else // *pNode 是其父结点的右子女
        pParent->rlink=pNode->llink;
}
free(pNode); // 释放被删除结点
// ④*p 的左子女代替*pNode
```



## 第七章 排序

### 7. 1 基本概念

1、**排序**：假设含  $n$  个记录的序列为  $\{R_1, R_2, \dots, R_n\}$ ，其相应的关键字序列为  $\{K_1, K_2, \dots, K_n\}$ ，需确定  $1, 2, \dots, n$  的一种排列  $p_1, p_2, \dots, p_n$ ，使其相应的关键字满足如下的非递减（或非递增）关系

$K_{p_1} \leq K_{p_2} \leq \dots \leq K_{p_n}$ ，即使序列成为一个按关键字有序的序列  $R_{p_1} \leq R_{p_2} \leq \dots \leq R_{p_n}$ ，这样一种操作称为**排序**。

2、**稳定排序法**和**不稳定排序法**：

在待排序的文件中，如果存在多个排序码相同的记录，经过排序后记录的相对次序保持不变，则这种排序方法称为是“**稳定的**”，否则是“**不稳定的**”。

3、**内排序**和**外排序**

待排序的记录在排序过程中全部存放在内存中的排序方法称**内排序**，而排序过程中需要使用外存的排序方法称**外排序**。

4、几种排序的基本思想方法：

- ▶ **插入排序**：每步将一个待排序的记录按其关键字大小插入到前面已排序表中的适当位置，直到全部插入完为止。
- ▶ **选择排序**：每一趟在  $n-i+1$  个记录中选取关键字最小（大）的记录作为有序序列中第  $i$  个记录。
- ▶ **交换排序法**：两两比较待排序记录的排序码，交换不满足顺序要求的偶对，直到全部满足为止。
- ▶ **（一路）归并排序**：把初始的  $n$  个记录看成是  $n$  个有序的子序列，每个子序列的长度为 1，然后两两归并，得到  $\lceil n/2 \rceil$  个长度为 2 或 1 的有序子序列；再两两归并，如此重复直到得到一个长度为  $n$  的有序序列为止。
- ▶ **分配排序**：分配排序是一种借助多关键码排序思想对单逻辑关键码排序的方法。

排序方法	最坏时间复杂度	平均时间复杂度	辅助空间	稳定性
直接插入排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定的
二分法插入排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定的
表插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	稳定的
Shell排序	$O(n^{1.3})$	$O(n^{1.3})$	$O(1)$	不稳定的
直接选择排序	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定的
堆排序	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(1)$	不稳定的
起泡排序	$O(n^2)$	$O(n^2)$	$O(1)$	稳定的
快速排序	$O(n^2)$	$O(n \log_2 n)$	$O(\log_2 n)$	不稳定的
基数排序	$O(d(n+r))$	$O(d(n+r))$	$O(r+n)$	稳定的
归并排序	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n)$	稳定的

## 7. 2 基本算法

### 7. 2. 1 直接插入排序

#### 1、算法基本思想:

假设待排序的  $n$  个记录  $\{R_0, R_1, \dots, R_{n-1}\}$  存放在数组中，插入记录  $R_i$  时，记录集合被划分为两个区间  $[R_0, R_{i-1}]$  和  $[R_i, R_{n-1}]$ ，其中，前一个子区间已经排好序，后一个子区间是当前未排序的部分，将排序码  $K_i$  与  $K_{i-1}, K_{i-2}, \dots, K_0$  依次比较，找出应该插入的位置，将记录  $R_i$  插入，原位置的记录向后顺移。

#### 2、代码

```
void insertSort(SortObject * pvector)
{
    int i, j;
    RecordNode temp;
    for( i = 1; i < pvector->n; i++ )
    {
        temp = pvector->record[i];
        j = i-1;
        while ((temp.key < pvector->record[j].key)&&(j>=0) )
        {
            pvector->record[j+1] = pvector->record[j];
            j--;
        }
        if( j!=(i -1) )
            pvector->record[j+1] = temp;
    }
}
```

#### 3、算法分析

空间效率：只需要一个记录的辅助空间。 $O(1)$

时间效率： $O(n^2)$

基本操作有：关键字比较和记录移动

关键字比较的次数为：最小： $n-1$  次；

最大： $n(n-1)/2$  次

记录移动的次数：最小： $n$

最大： $(n+2)(n-1)/2$

平均情况：比较  $O(n^2)$ ，移动  $O(n^2)$

#### 4、算法稳定性：稳定

## 7. 2. 2 直接选择排序

#### 1、算法基本思想:

首先在所有记录中选出排序码最小的记录，与第一个记录交换，然后在其余的记录中再选出排序码最小的记录与第二个记录交换，以此类推，直到所有记录排好序。

#### 2、代码:

```
void selectSort(SortObject *pvector)
{
    int i, j, k;
```



```

RecordNode    temp;
for( i = 0; i < pvector->n-1; i++ )
{
    k=i;
    for(j=i+1; j<pvector->n; j++)
        if(pvector->record[j].key<pvector->record[k].key)
            k=j;
    if(k != i)
    {
        temp=pvector->record[i];
        pvector->record[i]= pvector->record [k];
        pvector->record[k]=temp;
    }
}

```

3、**效率分析**：空间复杂度： $O(1)$  时间复杂度： $O(n^2)$

4、**算法稳定性**：不稳定

### 7. 2. 3 冒泡排序

1、**算法基本思想**：相邻数据比较，如果后者大于前者，则交换位置。否则不交换。

2、**代码**：

```

void bubbleSort(SortObject * pvector)
{
    int        i, j, noswap;
    RecordNode  temp;
    for(i=0; i<pvector->n-1; i++)                // 做 n-1 次起泡
    {
        noswap=TRUE;
        for(j=0; j<pvector->n-i; j++)
            if(pvector->record[j+1].key<pvector->record[j].key)
            {
                temp=pvector->record[j];
                pvector->record[j]=pvector->record[j+1];
                pvector->record[j+1]=temp;
                noswap=FALSE;
            }
        if(noswap)
            break;
    }
}

```

3、**效率分析**：空间复杂度： $O(1)$  时间复杂度： $O(n^2)$

4、**算法稳定性**：不稳定

## 第八章 图

### 8. 1 基本概念

- 1、**图**：是一种网状数据结构，其中结点之间的关系是任意的，即图中任何两个结点之间都可能直接相关。
- 2、**顶点**：图中的数据元素。
- 3、**边**：图中两个顶点之间的关系。  
**有向边**：由**始点**指向**终点**的一条边。
- 4、**有向图**：图中的每一条边都是有方向的。  
**无向图**：图中的每一条边都是无向的。
- 5、**邻接点**：若顶点  $(V_i, V_j)$  是一条边，则  $V_i$  和  $V_j$  互为邻接点。
- 6、**顶点的度**：与顶点相关联的边的数目。  
**出度**：在有向图中，以顶点  $v$  为始点的边的数目。  
**入度**：在有向图中，以顶点  $v$  为终点的边的数目。
- 7、**无向完全图**：有  $n(n-1)/2$  条边的无向图。  
**有向完全图**：有  $n(n-1)$  条边的有向图。
- 8、**稀疏图**：有很少条边或弧的图( $e < n \log n$ )  
**稠密图**： $(e) \geq n \log n$
- 9、**子图**：设有图  $G=(V, E)$  和  $G'=(V', E')$ ，如果  $V'$  是  $V$  的子集， $E'$  是  $E$  的子集，则称  $G'$  是  $G$  的子图。
- 10、**路径**：在图中从顶点  $v$  到顶点  $v'$  所经过的所有顶点的序列。  
**简单路径**：序列中顶点不重复出现的路径。
- 11、**回路或环**：第一个顶点和最后一个顶点相同的路径。  
**简单回路或环**：除第一个和最后一个顶点，其余顶点不重复出现的路径。
- 12、**有根图**：有向图中，若存在一顶点  $v$ ，从该顶点有路径可以到图中其它所有顶点，则称此有向图为有根图， $v$  称为图的根。
- 13、**连通**：在无向图中，如果从  $v$  到  $v'$  存在路径，则称  $v$  和  $v'$  是连通的。  
**连通图**：无向图  $G$  中如果任意两个顶点  $v_i, v_j$  之间都是连通的，则称图  $G$  是**连通图**。
- 14、**连通分量**：无向图中的极大连通子图。
- 15、**强连通图**：在**有向图** $G$ 中，如果对于每一对  $v_i, v_j \in V, v_i \neq v_j$ ，从  $v_i$  到  $v_j$  和从  $v_j$  到  $v_i$  都存在路径，则称  $G$  是**强连通图**。
- 16、**强连通分量**：有向图中的极大强连通子图。
- 17、**带权图**：图的每条边都赋上一个权值。
- 18、**网络**：带权的连通图。
- 19、**连通图的生成树**：是**连通图**的一个**极小**连通子图，它含有图中的全部  $n$  个顶点，但只有足以构成一棵树的  $n-1$  条边。
- 20、**有向树**：如果一个有向图恰有一个顶点入度为 0，其余顶点入度均为 1，则该图必定是一棵有向树。
- 21、**有向图的生成森林**：由若干棵有向树组成，含有图中全部顶点，但只有构成若干棵不相交的有向树的弧。
- 22、**图的周游**：从图中某一顶点出发按照某种方式系统地访问图中所有结点，并且使每一个结点被访问且仅被访问一次。
- 23、**路径**：如果图中从一个顶点可以到达另一个顶点，则称这两个顶点间存在一条路径。
- 24、**最短路径**：如果图是一个带权图，则路径长度为路径上各边的权值的总和，两个顶点间路径长度最短的那条路径称为两个顶点间的最短路径，其路径长度称为最短路径长度。

## 8. 2 图的存储结构

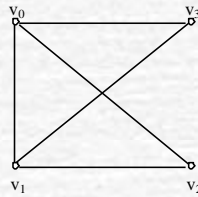
## 8. 2. 1 邻接矩阵表示法（数组表示法）

## 一、举例

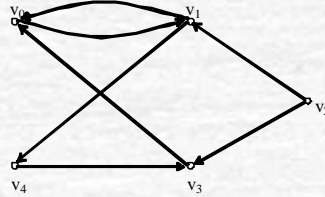
## 1、无向图及有向图

■ 顶点信息的存储方法：数组

■ 弧或边的存储：二维数组  $A[i, j] = \begin{cases} 1, & \text{若}(v_i, v_j) \text{或} \langle v_i, v_j \rangle \text{是图} G \text{的边} \\ 0, & \text{若}(v_i, v_j) \text{或} \langle v_i, v_j \rangle \text{不是图} G \text{的边} \end{cases}$



$$A_1 = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$



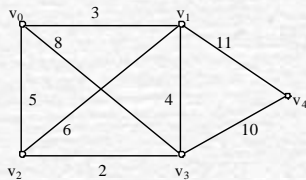
$$A_2 = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

## 2、网络

如果  $G$  是网络， $w_{ij}$  是边  $(v_i, v_j)$  或  $\langle v_i, v_j \rangle$  的权，则其邻接矩阵定义为：

$$A[i, j] = \begin{cases} w_{ij}, & \text{若}(v_i, v_j) \text{或} \langle v_i, v_j \rangle \text{是图} G \text{的边} \\ 0 \text{ 或 } \infty, & \text{若}(v_i, v_j) \text{或} \langle v_i, v_j \rangle \text{不是图} G \text{的边} \end{cases}$$

下面网络的邻接矩阵如下：



$$A_3 = \begin{bmatrix} 0 & 3 & 5 & 8 & 0 \\ 3 & 0 & 6 & 4 & 11 \\ 5 & 6 & 0 & 2 & 0 \\ 8 & 4 & 2 & 0 & 10 \\ 0 & 11 & 0 & 10 & 0 \end{bmatrix}$$

## 二、图的邻接矩阵存储结构

```
typedef struct
```

```
{
```

```
    VexType vexs[MAXVEX];
```

```
// 顶点信息
```

```
    AdjType arcs[MAXVEX][MAXVEX];
```

```
// 邻接矩阵信息
```

```
    int arcCount, vexCount;
```

```
// 图的顶点个数
```

```
}Graph, *PGraph;
```

## 三、邻接矩阵表示法的评价

1、优点：各种基本操作都易于实现。

缺点：空间浪费严重。某些算法时间效率低。

2、邻接矩阵的特点：

- 无向图的邻接矩阵一定是一个对称矩阵。
- 无向图的邻接矩阵的第  $i$  行 (或第  $i$  列) 非零元素 (或非  $\infty$  元素) 个数为第  $i$  个顶点的度  $D(v_i)$ 。



- 有向图的邻接矩阵的第  $i$  行非零元素(或非 $\infty$ 元素)个数为第  $i$  个顶点的出度  $OD(v_i)$ ，第  $i$  列非零元素(或非 $\infty$ 元素)个数就是第  $i$  个顶点的入度  $ID(v_i)$ 。
- 邻接矩阵表示图，很容易确定图中任意两个顶点之间是否有边相连。

#### 四、基本算法

##### 1、创建无向网

```
Status CreateUDN(Graph *G)
```

```
{
    int i, j;
    float w;
    // 输入顶点数、边数和是否输入边的相关信息的标志
    scanf("%d%d", &G->vexCount, &G->arcCount);
    for(i = 0; i < G->vexCount; i++) // 读入所有的顶点
        scanf("%d", &G->vexs[i]);
    for(i = 0; i < G->vexCount; i++) // 初始化邻接矩阵
        for(j = 0; j < G->vexCount; j++) // 初始假设所有顶点都互不邻接
        {
            G->arcs[i][j].adj = INFINITY;
        }
    for(k=0; k<G->arcCount; k++) // 输入所有的边
    {
        scanf("%d%d%f", &v1, &v2, &w); // 输入一条边依附的两个顶点和边上的权值
        i = LocateVex(G, v1); // 查询两个顶点在图中存储的位置
        j = LocateVex(G, v2);
        G->arcs[i][j].adj = w;
        G->arcs[j][i].adj = G->arcs[i][j].adj; // 根据无向图的对称性填充矩阵的对称部分
    }
    return OK;
}
```

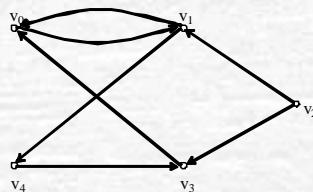
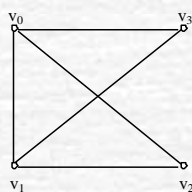
##### 2、查找某个顶点在图中的存储位置（下标），找不到返回-1

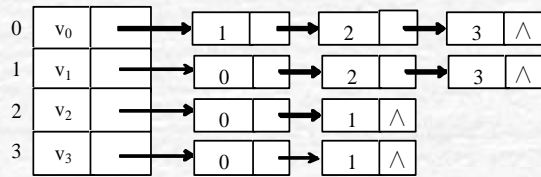
```
int LocateVex(Graph g, VexType vert)
```

```
{
    int i;
    for(i=0; i<g.vexCount; i++)
        if(g.vexs[i] == vert)
            return i;
    return -1;
}
```

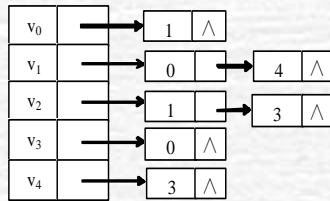
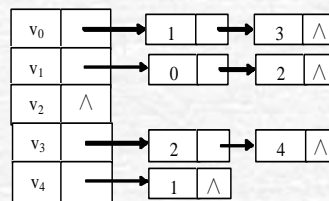
### 8. 2. 2 邻接表表示法（数组表示法）

#### 一、举例





(无向图的邻接表)

有向图 $G_2$ 的邻接表有向图 $G_2$ 的逆邻接表

**逆邻接表：**所有顶点的边表中的边都是以该顶点为终点的边。

## 二、存储结构

### 1、边表

```
typedef struct EdgeNode
{
    int      endvex;           // 相邻顶点字段
    AdjType  weight;          // 边的权
    struct EdgeNode  nextedge; // 链字段
}EdgeList, *PEdgeList, *PEdgeNode, EdgeNode; // 边表
```

### 2、顶点表

```
typedef struct
{
    VexType vertex;           // 顶点信息
    PEdgeList edgelist;       // 边表头指针
} VexNode;                   // 顶点表
```

### 3、图

```
typedef struct
{
    VexNode vexs[MAXVEX];
    int vexNum, edgeNum;      // 图的顶点个数
}GraphList;
```

## 三、邻接表的优缺点

- 优点：容易找任一结点的第一邻接点和下一个邻接点；存储量小。
- 缺点：判定任意两个结点之间是否有边或弧不方便。

## 四、基本算法

### 1、创建有向图

```
Status CreatedG(GraphList *g)
{
    int      i, j, k,  v1, v2;
    EdgeNode *p;
    printf("Input the numbers of vertexes, arcs:\n");
```

```

scanf("%d %d", &g->vexNum, &g->edgeNum);
printf("Input vertexes:\n");
for(i=0; i<g->vexNum; i++)
{
    scanf("%d", &g->vexs[i].vertex);
    g->vexs[i].edgelist = NULL;
}
for(k=0; k<g->arcNum; k++)
{
    printf("Input the start and end vertexes of an arc:\n");
    scanf("%d %d", &v1, &v2);    // Get the location of vertex 'v1' and 'v2' in the graph
    i = LocateVex(*g, v1);
    j = LocateVex(*g, v2);
    p = (EdgeNode *)malloc(sizeof(EdgeNode));
    p->endvex = j;
    if(g->vexs[i].edge == NULL)
    {
        g->vexs[i].edgelist = p;
        p->nextedge = NULL;
    }
    else
    {
        p->nextedge = g->vexs[i].edgelist->nextedge;
        g->vexs[i].edgelist->nextedge = p;
    }
}
return OK;
}

```

## 8. 3 图的周游

### 8. 3. 1 深度优先搜索

#### 一、周游规则

从图的指定顶点  $v$  出发，先访问顶点  $v$ ，并将其标记为已访问过，然后依次从  $v$  的未被访问过的邻接顶点  $w$  出发进行深度优先搜索，直到图中与  $v$  相连的所有顶点都被访问过。如果图中还有未被访问的顶点，则从另一未被访问过的顶点出发重复上述过程，直到图中所有顶点都被访问过为止。

#### 二、基本算法

1、从  $v_i$  出发进行深度优先搜索，图采用邻接表表示法

```

void DFSInList(GraphList * pgraphlist, int visited[], int i)
{
    int j;
    PEdgeNode p;
    printf("node: %c\n", pgraphlist->vexs[i].vertex);
    visited[i]=TRUE;
    p=pgraphlist->vexs[i].edgelist;    // 取边表中的第一个边结点
    while(p!=NULL)

```



```

    {
        if(visited[p->endvex]==FALSE)           // 该顶点的相邻顶点未被访问
            DFSInList(pgraphlist, visited, p->endvex); // 继续进行深度优先搜索
        p=p->nextedge;                           // 取边表中的下一个边结点
    }
}

```

2、从  $v_i$  出发进行深度优先搜索，图采用邻接矩阵表示法

```

void DFSInMatrix(Graph * pGraph, int visited[], int i)
{
    int    j;
    printf( "node: %c\n", pGraph->vexs[i]); // 访问出发点  $v_i$ 
    visited[i]=TRUE;
    for(j=0; j<pGraph->n; j++)
        if((pGraph->arcs[i][j]==1) && (visited[j]==FALSE) )
            DFSInMatrix(pGraph, visited, j);
}

```

3、图的深度优先周游（邻接矩阵）

```

void traverDFS(Graph * pGraph)
{
    int visited[MAXVEX];
    for(i=0; i<pGraph->n; i++)           // 初始化数组 visited
        visited[i]=FALSE;
    for(i=0; i<n; i++)
        if(visited[i]==FALSE)
            DFSInMatrix(pGraph, visited, i);
}

```

**效率分析：**

- 空间复杂度：标志数组和栈， $O(n)$
- 时间复杂度：对于邻接矩阵表示法为  $O(n^2)$   
对于邻接表表示法为  $O(n+e)$

## 8. 3. 2 广度优先搜索

### 一、周游规则

从图的指定顶点  $v$  出发，先访问顶点  $v$ ，接着依次访问  $v$  的所有邻接点  $w_1, w_2, \dots, w_x$ ，然后，再依次访问与  $w_1, w_2, \dots, w_x$  邻接的所有未被访问过的顶点，以此类推，直到所有已访问顶点的邻接点都被访问过为止。如果图中还有未被访问过的顶点，则从另一未被访问过的顶点出发进行广度优先搜索，直到所有顶点都被访问过为止。

对于广度优先周游，关键在于怎么保证“先被访问的顶点的邻接点”要先于“后被访问的顶点的邻接点”被访问。

### 二、基本算法

1、从  $v_i$  出发进行广度优先搜索，图采用邻接矩阵表示法

```

void bFSInMatrix(Graph * pGraph, int visited[], int i)
{
    PLinkQueue    pq;

```

```

int      j, k;
pq=createEmptyQueue_link();           // 置队列为空
printf( "node:%c\n", pGraph->vexs[i]);
visited[i]=TRUE;
enqueue_link(pq, i);                  // 将顶点序号进队
while( !isEmptyQueue_link(pq) )       // 队列非空时执行
{
    k=deQueue_link(pq);               // 队头顶点出队
    for(j=0; j<pGraph->n; j++)
        if( (pGraph->arcs[k][j]==1) && (!visited[j]) ) // 访问相邻接的未被访问过的顶点
        {
            printf( "node:%c\n", pGraph->vexs[j]);
            visited[j]=TRUE;
            enqueue_link(pq, j);       // 新访问的顶点入队
        }
    }
}

```

## 2、从 vi 出发进行广度优先搜索，图采用邻接表表示法

```
void bFSInList(GraphList *pgraphlist, int visited[], int i)
```

```

{
    PLinkQueue    pq;
    PEdgeNode     p;
    int           j;
    pq=createEmptyQueue_link();           // 置队列为空
    printf( "node:%c\n", pgraphlist->vexs[i].vertex);
    visited[i]=TRUE;
    enqueue_link(pq, i);                  // 将顶点序号进队
    while (!isEmptyQueue_link(pq) )       // 队列非空时执行
    {
        j=deQueue_link(pq);              // 队头顶点出队
        p=pgraphlist->vexs[j].edgelist;
        while( p!=NULL)
        {
            if (!visited[p->endvex]) /*访问相邻接的未被访问过的顶点 */
            {
                printf( "node:%c\n", pgraphlist->vexs[p->endvex].vertex);
                visited[p->endvex] = TRUE;
                enqueue_link(pq, p->endvex); // 新访问的顶点入队
            }
            p=p->nextedge;
        }
    }
}

```

## 3、图的广度优先周游（邻接矩阵）

```

void traverBFS(Graph *pGraph)
{
    int    visited[MAXVEX];
    int    i,n;
    n=pGraph->n;
    for(i=0;i<n;i++)
        visited[i]=FALSE;
    for(i=0; i<n; i++)
        if(visited[i]==FALSE)
            BFSInMatrix(pGraph, visited, i);
}

```

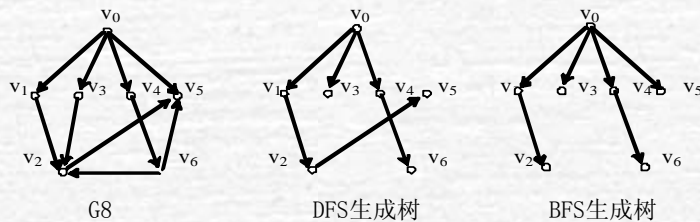
### 三、效率分析：

- 空间复杂度：标志数组和队列， $O(n)$
- 时间复杂度：对于邻接矩阵表示法为  $O(n^2)$   
对于邻接表表示法为  $O(n+e)$

## 8. 4 最小生成树

### 一、概述

- 1、**DFS 生成树**：从连通图的任一顶点出发，进行深度优先周游，记录周游中访问的所有顶点及经过的边，便得到深度优先生成树。
- 2、**BFS 生成树**：从连通图的任一顶点出发，进行广度优先周游，记录周游中访问的所有顶点及经过的边，便得到广度优先生成树。



### 3、生成树的特点：

- 包含连通图的全部  $n$  个顶点和其中  $n-1$  条边；
- 不存在回路。
- 如果在其中任意添加一条边，必定会构成回路。
- 如果去掉其中任意一条边，则会成为非连通图。

### 4、生成森林：由非连通图得到。

### 5、生成树的权：生成树各边的权值的总和。

### 6、最小生成树：在网络中，具有最小权值的生成树。

## 二、Prim 算法

### 1、基本思想

设  $G=(V, E)$  是具有  $n$  个顶点的网络， $T=(U, TE)$  为  $G$  的最小生成树， $U$  是  $T$  的顶点集合， $TE$  是  $T$  的边集合。

- (1) 从集合  $V$  中任取一顶点  $v_0$  放入集合  $U$  中，这时  $U=\{v_0\}$ ， $TE=NULL$ ；
- (2) 在所有一个顶点在集合  $U$  里，另一个顶点在集合  $V-U$  里的边中，找出权值最小的边  $(u, v)$  ( $u \in U, v \in V-U$ )，将边加入  $TE$ ，并将顶点  $v$  加入集合  $U$ ；
- (3) 重复上述操作直到  $U=V$  为止。



## 2、效率分析

空间复杂度:  $O(n)$ : 存放挑选出的边

时间复杂度:  $O(n^2)$ : 与边数无关, 适用于稠密图

## 4、代码

```
void prim(Graph * pgraph, Edge mst[])
{
    int i, j, min, vx, vy;
    float weight, minweight;
    Edge edge;
    for(i=0; i<pgraph->n-1; i++)
    {
        mst[i].start_vex=0;
        mst[i].stop_vex=i+1;
        mst[i].weight=pgraph->arcs[0][i+1];
    }
    for(i=0; i<pgraph->n-1; i++) // 共 n-1 条边
    {
        minweight=MAX; min=i;
        for(j=i; j<pgraph->n-1; j++) // 从所有边 (vx, vy) ( $vx \in U, vy \in V-U$ ) 中选出最短的边
            if(mst[j].weight<minweight)
            {
                minweight=mst[j].weight; min=j;
            }
        // mst[min]是最短的边 (vx, vy) ( $vx \in U, vy \in V-U$ ), 将 mst[min]加入最小生成树
        edge=mst[min];
        mst[min]=mst[i];
        mst[i]=edge;
        // vx 为刚加入最小生成树的顶点的下标
        vx=mst[i].stop_vex;
        for(j=i+1; j<pgraph->n-1; j++) // 调整 mst[i+1]到 mst[n-1]
        {
            vy=mst[j].stop_vex;
            weight=pgraph->arcs[vx][vy];
            if(weight<mst[j].weight)
            {
                mst[j].weight=weight; mst[j].start_vex=vx;
            }
        }
    }
}
```

## 二、Kruskal 算法

### 1、算法描述

设连通网  $N=(V, \{E\})$ 。令最小生成树的初始状态为只有  $n$  个顶点而无边的非连通图  $T=(V, \{\})$ , 图中每个顶点自成一个连通分量。在  $E$  中选择代价最小的边, 若该边依附的顶点落在  $T$  中不同的连通分量

上，则将此边加入  $T$  中，否则舍去此边而选择下一条代价最小的边，依次类推，直到  $T$  中所有顶点都在同一连通分量上为止。

## 8. 5 最短路径

### 8. 5. 1 从某个源点到其余各顶点的最短路径 (Dijkstra 算法)

设图  $G=(V, E)$ ， $v_0 \in V$ ，求从  $v_0$  出发到其他顶点的最短路径。

Dijkstra 提出了按路径长度递增的次序产生最短路径的算法。

#### 一、基本思想：

设  $U$  存放已求出最短路径的顶点， $V-U$  是尚未确定最短路径的顶点集合； $U$  中顶点的距离值是从  $v_0$  到该顶点的最短路径长度， $V-U$  中顶点的距离值是从  $v_0$  到该顶点的只包括  $U$  中顶点为中间顶点的最短路径长度。

- (1) 初始时  $U$  中只有  $v_0$ ，集合  $V-U$  中顶点  $v_i$  的距离值为边  $(v_0, v_i)$  的权值；
- (2) 在  $V-U$  中选择距离值最小的顶点  $v_{min}$  加入  $U$ ；然后对  $V-U$  中各顶点的距离值修正，如加入  $v_{min}$  为中间顶点后，使  $v_0$  到  $v_i$  的距离值比原距离值小，则修改  $v_i$  的距离值。
- (3) 反复操作，直到从  $v_0$  出发可以到达的所有顶点都在  $U$  中为止。

#### 二、实现方法：

- (1) 初始时，集合  $U$  中只有顶点  $v_0$ ，从顶点  $v_0$  到其它顶点  $v_i$  ( $i=1, 2, \dots, n-1$ ) 的最短路径长度为边  $(v_0, v_i)$  的权值。若顶点  $v_0$  和  $v_i$  不相邻，则设其权值为无穷大。
- (2) 在集合  $V-U$  中找出距离值最小的顶点  $v_{min}$ ，将其加入集合  $U$ ，从顶点  $v_0$  到顶点  $v_{min}$  的最短路径长度就是  $v_{min}$  的距离值。
- (3) 调整集合  $V-U$  中顶点的距离值。

如果将新加入的顶点  $v_{min}$  作为中间顶点后， $v_0$  到  $v_i$  ( $v_i \in V-U$ ) 的距离值更小，则应修改  $v_i$  的距离值。即：

```
if(dist[i].length > dist[min].length+graph.arcs[min][i])
    dist[i].length = dist[min].length+graph.arcs[min][i]
将路径上  $v_i$  的前趋顶点改为  $v_{min}$ ，即：dist[i].prevex=min。
```

- (4) 重复(2)，(3)操作，直到集合  $V-U$  中的顶点都加入到集合  $U$  中为止。

#### 三、效率分析：

空间复杂度： $O(n)$

时间复杂度： $O(n^2)$

#### 四、最短路径及长度存储结构：

```
typedef struct
{
    VexType    vertex;        // 顶点信息
    AdjType    length;        // 最短路径长度
    int        prevex;        // 从  $v_0$  到达  $v_i$  的最短路径上  $v_i$  的前趋顶点
}Path;
```

使用一个 Path 型数组存放即可。

图用邻接矩阵表示法存储。

#### 五、代码

```
void dijkstra(Graph graph, Path dist[])
{
    int        i, j, minvex;
    AdjType    min;
```

```

dist[0].length=0;
dist[0].prevex=0;
dist[0].vertex=graph.vexs[0];
graph.arcs[0][0]=1;           // 表示顶点 v0 在集合 U 中
for(i=1; i<graph.n; i++)      // 初始化集合 V-U 中顶点的距离值
{
    dist[i].length=graph.arcs[0][i];
    dist[i].vertex=graph.vexs[i];
    if(dist[i].length!=MAX)
        dist[i].prevex=0;
    else
        dist[i].prevex= -1;
}
for(i=1; i<graph.n; i++)
{
    min=MAX;
    minvex=0;
    for(j=1; j<graph.n; j++)    // 在 V-U 中选出距离值最小顶点
        if( (graph.arcs[j][j]==0) && (dist[j].length<min) )
        {
            min=dist[j].length;
            minvex=j;
        }
    if(minvex==0)                // 从 v0 没有路径可以通往集合 V-U 中的顶点
        break;
    graph.arcs[minvex][minvex]=1; // 集合 V-U 中路径最小的顶点为 minvex
    for(j=1; j<graph.n; j++)     // 调整集合 V-U 中的顶点的最短路径
    {
        if(graph.arcs[j][j]==1)
            continue;
        if(dist[j].length>dist[minvex].length+graph.arcs[minvex][j])
        {
            dist[j].length = dist[minvex].length+graph.arcs[minvex][j];
            dist[j].prevex = minvex;
        }
    }
}
}

```

### 8. 5. 2 每一对顶点之间的最短路径 (Floyd 算法)

#### 一、Floyd 算法的基本思想:

设图  $G=(V, E)$ , 有  $n$  个顶点, 采用邻接矩阵存储。如果边  $(v_i, v_j) \in E$ , 则从  $v_i$  到  $v_j$  存在一条长度为  $\text{arcs}[i][j]$  的路径, 该路径不一定是从  $v_i$  到  $v_j$  的最短路径, 因为可能存在从  $v_i$  到  $v_j$  并且包含其它中间顶点的路径。因此, 应该在所有从  $v_i$  到  $v_j$  并允许其它顶点为中间顶点的路径中, 找出长度最短的路径。



(1) 首先加入一个中间顶点  $v_0$ ，考虑路径  $(v_i, v_0)$ ， $(v_0, v_j)$  是否存在，如存在，则比较  $(v_i, v_j)$  和  $(v_i, v_0) + (v_0, v_j)$  的长度，取其较短者为当前从  $v_i$  到  $v_j$  的最短路径，该路径是从  $v_i$  到  $v_j$  允许一个中间顶点  $v_0$  的最短路径。

(2) 其次在从  $v_i$  到  $v_j$  并且考虑了  $v_0$  的路径中加入一个中间顶点  $v_1$ 。把  $(v_i, \dots, v_1) + (v_1, \dots, v_j)$  与上一步求出的  $v_i$  到  $v_j$  允许  $v_0$  为中间顶点的最短路径比较，取其较短者为当前从  $v_i$  到  $v_j$  的最短路径，该路径是从  $v_i$  到  $v_j$  允许两个顶点  $v_0$  和  $v_1$  为中间顶点的最短路径。

(3) 然后，再加入顶点  $v_2$ ，等等。

如果  $(v_i, \dots, v_k)$  和  $(v_k, \dots, v_j)$  分别是  $v_i$  到  $v_k$  和从  $v_k$  到  $v_j$  允许  $k$  个顶点  $v_0, v_1, \dots, v_{k-1}$  为中间顶点的最短路径，则将  $(v_i, \dots, v_k, \dots, v_j)$  和已经得到的从  $v_i$  到  $v_j$  允许  $k$  个顶点  $v_0, v_1, \dots, v_{k-1}$  为中间顶点的最短路径进行比较，取其中较短的路径为从  $v_i$  到  $v_j$  允许  $k+1$  个顶点  $v_0, v_1, \dots, v_k$  为中间顶点的最短路径。

(4) 以此类推，直到加入顶点  $v_{n-1}$  为止，则得到的是从  $v_i$  到  $v_j$  允许  $n$  个顶点  $v_0, v_1, \dots, v_{n-1}$  为中间顶点的最短路径。

由于已经考虑了所有顶点作为中间顶点的可能性，因此得到的最终结果就是从  $v_i$  到  $v_j$  的最短路径。

## 二、代码

```
#define MAX          1e+38

void floyd(Graph * pgraph, ShortPath * ppath)
{
    int      i, j, k;
    for(i=0; i<pgraph->n; i++)
        for(j=0; j<pgraph->n; j++)
        {
            if(pgraph->arcs[i][j]!=MAX)
                ppath->nextvex[i][j]=j;
            else
                ppath->nextvex[i][j]= -1;
            ppath->a[i][j]=pgraph->arcs[i][j];
        }
    for(k=0; k<pgraph->n; k++)
        for(i=0; i<pgraph->n; i++)
            for(j=0; j<pgraph->n; j++)
            {
                if( (ppath->a[i][k]>=MAX) || (ppath->a[k][j]>=MAX) )
                    continue;
                if(ppath->a[i][j]>(ppath->a[i][k]+ ppath->a[k][j]) )
                {
                    ppath->a[i][j]= ppath->a[i][k]+ ppath->a[k][j];
                    ppath->nextvex[i][j]=ppath->nextvex[i][k];
                }
            }
}
```