第三章　线性表（Linear List）

- 线性表的定义
- 线性表的顺序存储
- 顺序表的基本算法实现
- 顺序表组织的查找表
- 顺序表的应用

3.1、线性表的定义

- **线性表(Linear List)：** 具有相同类型的n个元素组成的有限序列；
- **线性表定义：**
  - ***n*（≥0）个数据元素的有限序列，记作：**

    L=（$a_1, a_2, \ldots, a_n$）

    其中： a_i **是表中元素，** a_1 **是表头元素，** a_n **是表尾元素，** n **是表长度，当n=0时称为空表。**
- **线性结构的特点：**
  - 除第一个元素外，其他每一个元素有一个且仅有一个直接前驱。
  - 除最后一个元素外，其他每一个元素有且仅有一个直接后继。

线性表的数学定义

- **线性表数学定义：**
  - ***线性表是由两个集合构成的一个二元组。*记作：**

    L=（D, R）

  其中：**D=**{ a_i | $a_i \in$ ElementType, i=1,2,…,n n>=1 }叫数据集；

  R={ $<a_i, a_{i+1}>$ | $a_i, a_{i+1} \in$ **D**, i=1,2,…, n-1}叫关系集；

  ElementType为某一个特定的数据对象集合；

  n为线性表长度；n=0时，线性表为空表。

- **线性表L=（D, R）的图形表示：**

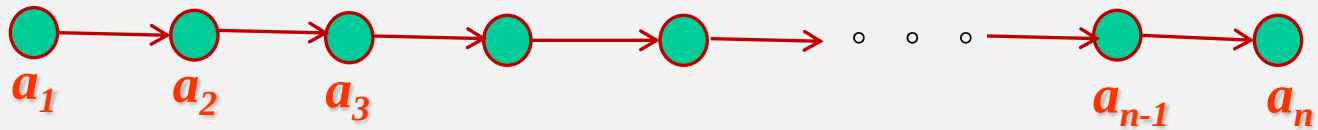


图3-1 线性表结构的逻辑图

线性表的基本操作

- ADT（List）

  数据对象：元素集合；**数据关系：**二元组集合

  数据操作： 基本操作如下

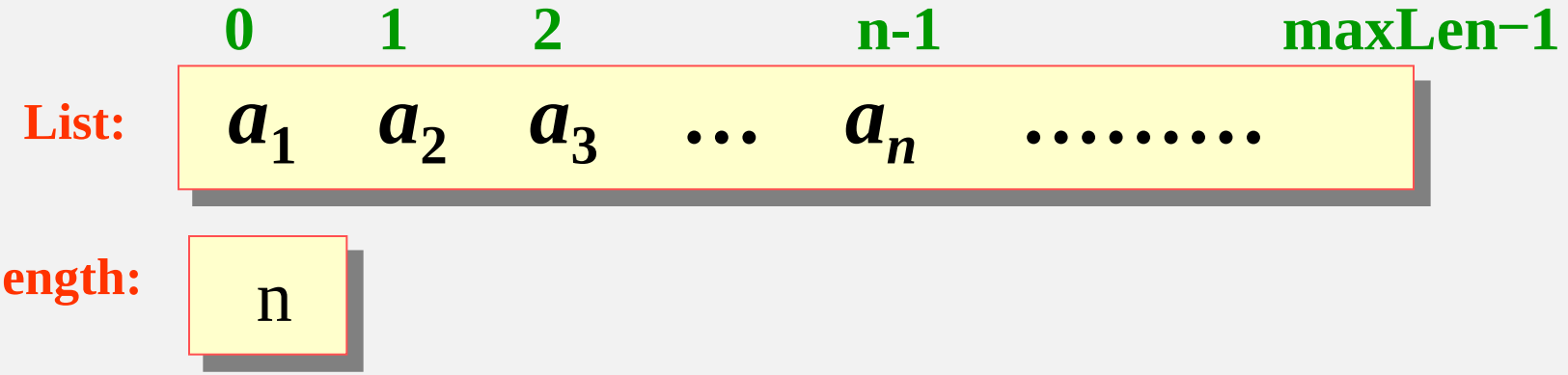
  - 初始化　Init_Sqlist
  - 求长　ListLength
  - 取元素　GetElem
  - 插入　ListInsert
  - 删除　ListDelete
  - 查找　LocateElem
  - 除了这些基本操作，还可以根据具体线性表的特性需要增加一些其他操作。比如：线性表的遍历、排序等。

- 线性表例子：

  比如学生成绩列表、学生花名册、通信录、图书目录等

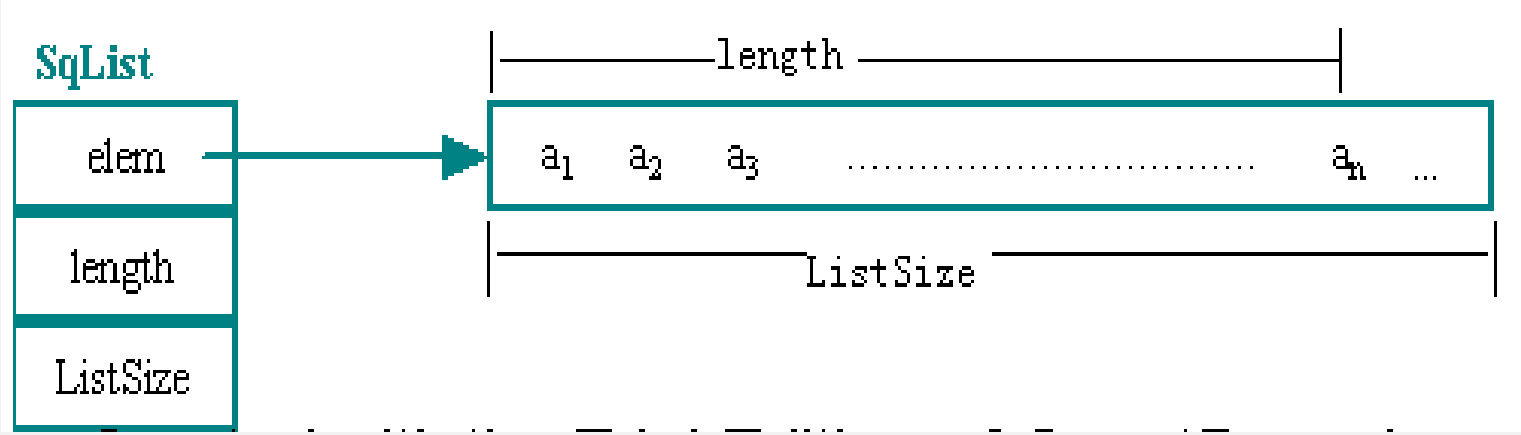
3.2、线性表的顺序存储实现

- 1、使用顺序存储结构——顺序表（Sequential List）
  - 利用顺序存顺结构实现逻辑结构，即利用元素之间的相对位置表示他们之间的次序关系。相邻元素的物理位置相邻。



  - 元素的位置关系：$LOC(a_i)=LOC(a_1) + (i-1)*L$　L为元素大小
  - 可以利用数组存储

- 2、顺序表的C语言定义实现




```
#define  DefaultSize=20;
typedef int ElemType;        /* 线性表元素类型定义，元素也可是结构型 */
typedef struct               /* 线性表结构定义 */
{   ElemType  elem[];        /* 线性表空间数组定义 */
    int length;              /* 线性表长度变量定义 */
    int ListSize;            /* 线性表空间大小变量定义 */
}SqList;                     /* 线性表类型标示符定义 */
SqList  List;                /* 线性表结构变量List定义 */
```

3.3、顺序表的基本算法实现

1、初始化---构建一个空表（Init_Sqlist）

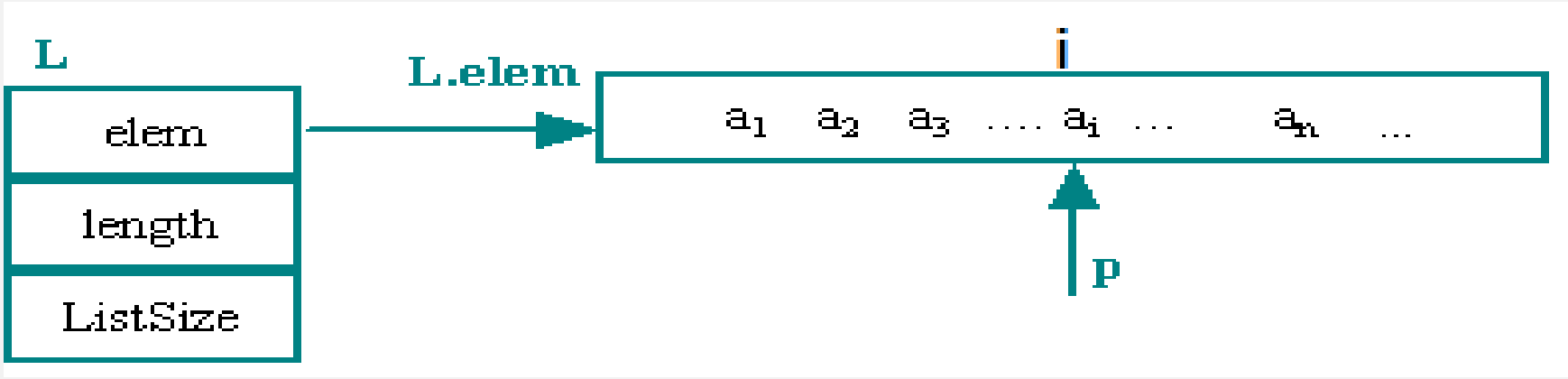
- 需要预先分配空间 **malloc**

```
void Init_Sqlist(SqList  &L,  int Lsize=DefaultSize)   /*  线性表指针，分配大小 */
{
    L.ListSize = Lsize+1;                  /*  线性表空间大小赋值，下标0不存元素 */
    L.length = 0;                          /*  线性表长度初始化为0 */
    /*  为线性表动态申请空间 */
    L.elem = (ElemType *)malloc(L.ListSize * sizeof(ElemType) );
    if (L.elem==NULL){                     /*  判断线性表空间分配是否成功 */
          printf("\n存储分配失败错误！ \n");
          exit(1);                         /*  分配失败。终止程序运行 */
    }
}
```

顺序表插入操作实现思想

2、插入操作---- ListInsert把新元素插入到第i号位置

- 判断空间是否溢出（可考虑扩增空间**realloc**）
- 移动数据
- 放入新元素，并修改顺序表大小



顺序表插入算法

```
bool  ListInsert (SqList  &L,  int i, ElemType x)  /*顺序表地址、位置、新元素 */
{    /* 函数返回插入是否成功的消息，成功返回ture，否则返回falase  */
     if ( L.length == L.ListSize ) return false;   /*  检查顺序表溢出 */
     if ( i<1 || i>L.length+1 ) return false;        /*  检查插入位置合理性 */
     for（int j= L.length; j>=i; j--)                 /*  元素依次后移，空出第i号位置*/
            L.elem[j] = L.elem[j-1];
     L.elem[i-1] = x;                                          /*  插入新元素*/
     L.length++;                                               /*  调整顺序表大小*/
     return true;                                              /*  返回插入成功*/
}
```

顺序表插入算法复杂度分析

- 插入算法的性能分析---时间复杂度
  - 插入 – 算法的执行时间与插入位置***i***有关
    - Best Case　　***i = List.length***
    - Worst Case　　***i = 0***
    - Average Case　***i = 0*** <—> ***List.Length***

    假定每种情况等概率发生：

$$T(n) = \sum_{i=0}^{n} p_i \times c_i = \sum_{i=0}^{n} \frac{1}{n+1} \times (n - i)$$

$$= 1/(n+1) * (n*(n+1)/2)$$

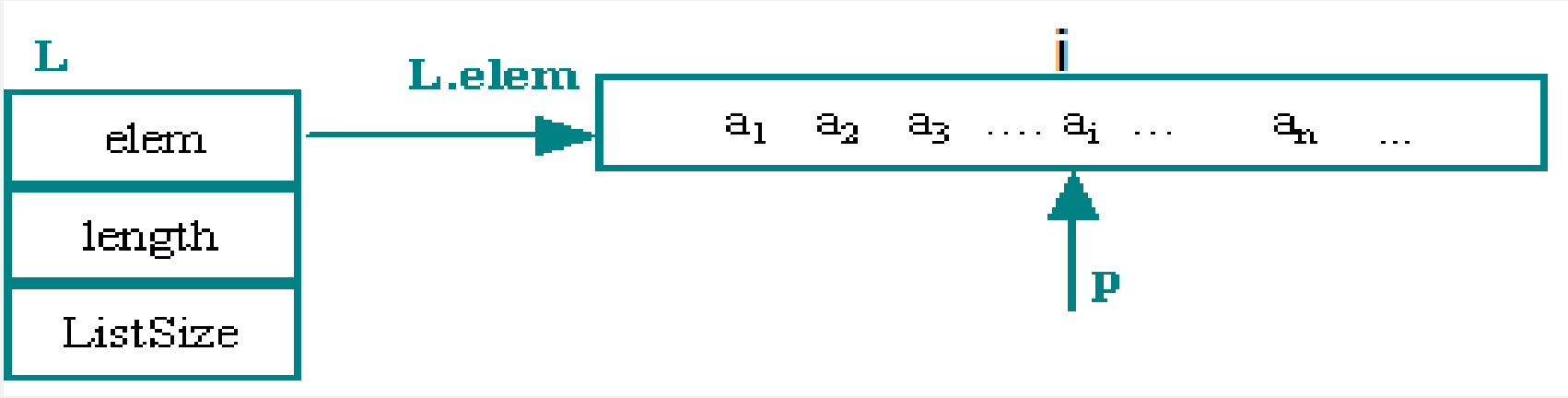
$$= n/2$$

$$= O(n)$$

顺序表删除操作实现思想

3、删除操作---- ListDelete删除第i号位置的元素

- 判断顺序表是否为空表
- 移动数据（被删除元素之后前移一位）
- 调整顺序表大小



顺序表删除算法

```
bool  ListDelete (SqList  &L,int i,ElemType &x)  /*顺序表地址、位置、旧元素 */
{    /* 函数返回删除是否成功的消息，成功返回ture，否则返回falase  */
     if ( L.length == 0 ) return false;              /*  检查顺序表是否为空表 */
     if (i<1 || i>L.length) return false;/*  检查删除位置合理性 */
     x = L.elem[i-1];                                        /*  存储被删除元素的值*/
     for（int j=i; j<L.length; j++)                  /*  元素依次前移，填补第i位置*/
          L.elem[j-1] = L.elem[j];
     L.length--;                                      /*  调整顺序表大小*/
     return true;                                     /*  返回删除成功*/
}
```

顺序表删除算法复杂度分析

- 删除算法的性能分析---时间复杂度
  - 删除 – 算法的执行时间与删除位置***i***有关
    - Best Case　　***i = List.length-1***
    - Worst Case　***i = 0***
    - Average Case　***i = 0 <—> List.Length-1***

    假定每种情况等概率发生：

$$T(n) = \sum_{i=0}^{n-1} p_i \times c_i = \sum_{i=0}^{n-1} \frac{1}{n} \times (n-i-1)$$

$$= 1/n * (n*(n-1)/2)$$

$$= (n-1)/2$$

$$= O(n)$$

顺序表查找操作算法

4、查找操作---- LocateElem查找与值x匹配的元素

```
int  LocateElem(SqList  &L,  ElemType x)  /*顺序表地址、待查找值 */
{    /* 函数返回查找到的位置，若返回0，则意味着查找失败 */
     for（int j=1; j<=L.length; j++)                /*  元素依次比较，返回查找位置号*/
            if（ L.elem[j-1] == x ) return j;
     return 0;                                      /*  返回查找失败*/
}
```

顺序表查找算法复杂度分析

- 查找算法的性能分析---时间复杂度
  - 查找 – 算法的执行时间与删除位置***i***有关
    - Best Case　　***i = 0***
    - Worst Case　***i = List.length-1***
    - Average Case　***i = 0 <—> List.Length-1***

    假定每种情况等概率发生：

$$T(n) = \sum_{i=0}^{n-1} p_i \times c_i = \sum_{i=0}^{n-1} \frac{1}{n} \times (i+1)$$

$$= 1/n * (n*(n+1)/2)$$

$$= (n+1)/2$$

$$= O(n)$$

顺序表存储总结

- 顺序表的特点：
  - 各元素的逻辑顺序与屋里顺序一致；无需存储关系集；
  - 顺序表可以进行顺序访问，也可进行随机访问；
  - 顺序表可以用C语言的一维数组来简单实现；即可静态也可动态；
  - 存储数组的元素类型就是线性表元素类型；
  - 顺序表中第i个元素被存储在数组下标为i-1的位置上。
- 优点：
  - 数据读取方便，可实现随机存取。
  - 操作简单、易于编程实现
- 缺点：
  - 插入、删除时需要移动数据，效率低
  - 线性表空间有限制，而自动扩展的代价高

3.4、顺序表组织的查找表

- 计算机存储信息的目的是要使用这些信息，必须能对这些信息进行查找，从中找到所需的数据。
- 如何实现对信息的有效查找？针对信息的不同特点和不同的存储方法，有不同的查找方法。

■查找表：同一类型数据组成的集合，每个数据在表中又称为一个记录。

■查找是作用在同一个类型的数据元素集合上的，将这样的集合称为查找表。

■关键字：是数据元素中某个数据项的值。

■主关键字：可以唯一地标识一个记录。

■例如：学生记录中的学号

学号	姓名	年龄	籍贯

所谓查找，就是在数据集合中按关键字寻找满足某种条件的数据对象。

静态查找——查找表的内容不变（即元素都已知）

1、顺序查找

2、折半查找

1、顺序表组织的顺序查找表（1）

1、顺序查找---- found_elem查找与值x匹配的元素

待查找数据元素存放在顺序表中，在顺序表中查找一个元素的基本思想是逐个比较，直到找到为止，或者失败。

1) 查找表大小为N+1，查找的元素存放在elem[1-n]。

调用程序形式：found_elem1(List, 50)

```
int  found_elem1(SqList  &L,  ElemType x)  /*顺序表地址、待查找值 */
{    /* 函数返回查找到的位置号，若返回0，则意味着查找失败 */
     L.elem[0] = x;
     for（int j= L.length; j>=0; j--)              /*  反向依次比较，返回查找位置号*/
          if（ L.elem[j] == x ) return j;          /* 当返回0时，意味着失败*/
}
```

1、顺序表组织的顺序查找表（2）

2)查找表大小为N+1，查找的元素存放在elem[1-n]，顺序表采用指针传递参数。

调用程序形式：found_elem2(&List，50)

```
int  found_elem2(SqList  *L，ElemType x)  /*顺序表指针、待查找值 */
{    /* 函数返回查找到的位置号，若返回0，则意味着查找失败 */
     L->elem[0] = x;
     for（int j= L->length; j>=0; j--)            /*  反向依次比较，返回查找位置号*/
          if（ L->elem[j] == x ) return j;        /* 当返回0时，意味着失败*/
}
```

1、顺序表组织的顺序查找表（3）

3)查找表中元素类型为结构型，查找的元素存放在elem[1-n]，顺序表采用指针传递参数。

调用程序形式：found_elem3(&List, 50)

```
typedef  int  KeyType;
typedef  struct ElemType {
    KeyType   key;
    /*  其他成员的字段定义 */
}  ElemType;
/*   原来的SqList结构定义 */
int  found_elem3(SqList  *L,  KeyType x)  /*顺序表指针、待查找值 */
{    /* 函数返回查找到的位置号，若返回0，则意味着查找失败 */
    L->elem[0].key = x;
    for（int j= L->length; j>=0; j--)              /*  反向依次比较，返回查找位置号*/
          if（ L->elem[j].key == x ) return j; /* 当返回0时，意味着失败*/
}
```

2、顺序表组织的折半查找表

- 2、有序表的**折半查找**(**Binary Search也称二分法)**

  折半查找表的要求：

  – 静态数据且各个记录按关键字大小顺序排列(有序表)

  – 查找表是可以随机检索的（即随机存取，如数组等）

  折半查找的思想：

  - 原理：通过与中间位置上的元素进行比较，每次可以将比较的元素范围缩小一半。

  例如：在下列表中查找：key=19

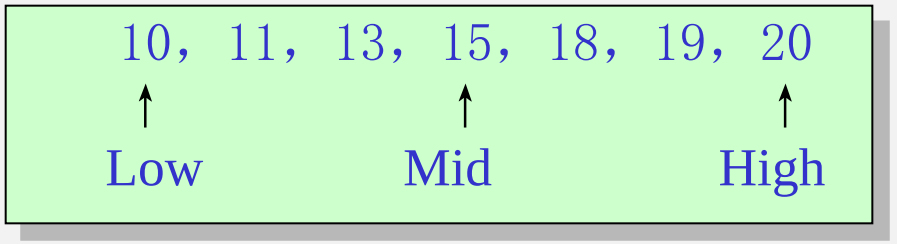
  (8，15，18，23，41，56，63)

  Low(下界)　mid(比较)　high(上界)

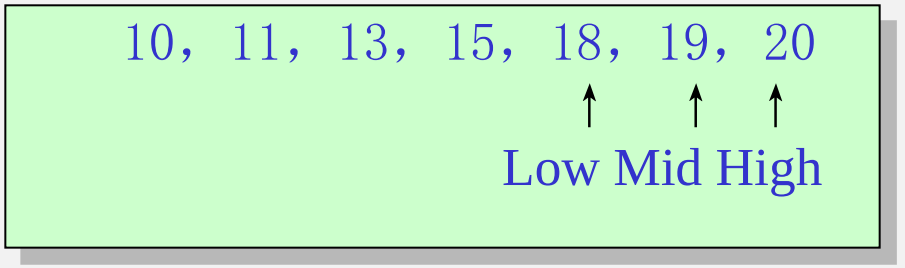
折半查找过程（举例）

折半查找过程：(举例)

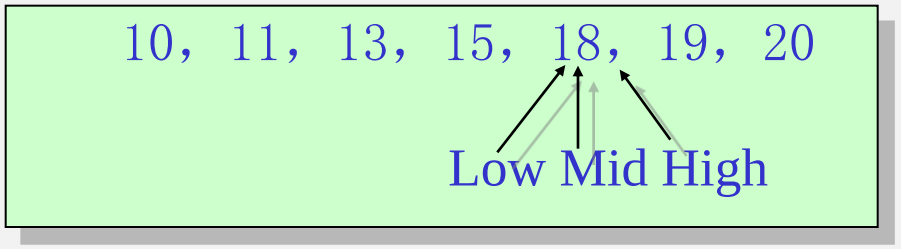
第1次： Key=18
Low =1, high=7,
mid=(low+high)/2=4



第2次： 由于key>B[4]
Low=mid+1=5,
high=7,
mid=(low+high)/2=6



第3次： 由于key<B[6]
Low=5
high =mid-1=5,
mid=(low+high)/2=5



折半查找过程（算法描述）

折半查找方法Binary search method:

给定一个关键字值K. 我们在rec[low..high] 中查找一个记录R，要求满足rec[i]==K(low=<i<=high)。折半查找算法要把中间位置的记录rm与待查找键值K进行比较，有以下三种情况：

1）如果K==rm, 则rm是要查找的记录，成功结束;

2）如果K<rm，则继续在rec[low..m-1]中搜索;

3）如果K>rm，则继续在rec[m+1..high]中搜索;

折半查找算法1（递归程序）

折半查找算法1(recursive algorithm)

```
int binsearch( int  rec[], int k, int low, int high)
{
    if  ( low > high ) return -1;    /* 查询失败 */
    int mid = (low + high) / 2;
    if  ( k == rec[mid] ) return mid+1;
    else if  ( k < rec[mid] )
            binsearch(rec, k, low, mid-1);
    else
            binsearch(rec, k, mid+1, high);
}
```

调用语句如下：

```
        binsearch(List.elem, key, 0, List.length-1)
```

折半查找算法2（非递归程序）

折半查找算法很容易写成非递归程序如下：

```
int binsearch( int  rec[], int k, int n)
{
    int  mid, low = 0, high = n -1;
    while ( low <= high ) {
        mid = (low+high) / 2;
        if  ( k == rec[mid] ) return mid+1;
        else if  ( k < rec[mid] )  high = mid-1;
              else  low = mid + 1;
    }
    return -1;      /* 查询失败 */
}
```

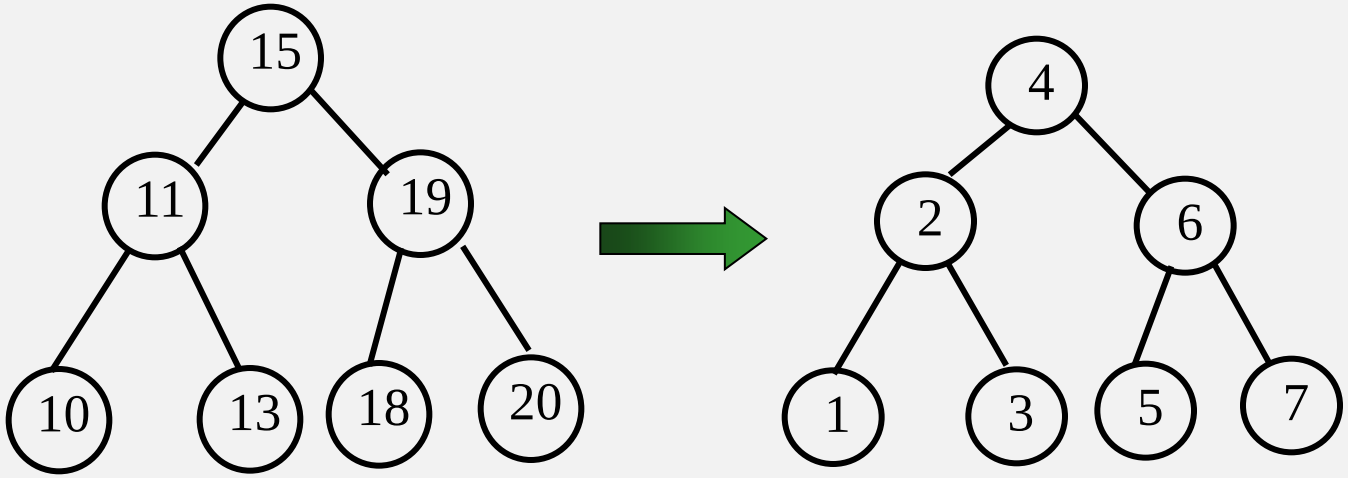
调用语句为：

```
binsearch(List.elem, key,  List.length)
```

折半查找算法的性能分析

折半查找性能分析： **(Binary Search performance)**

- 折半查找过程可以借助二叉树来描述、分析。这个二叉树称为**判定二叉树**或叫**判定树**。
- **构造过程**：以中间元素为根，所有小于中间元素的数据组成左子树，大于中间元素的数据组成右子树。



注意：在查找过程中，最大的比较次数就等于该判定树的高度。

折半查找算法的性能分析（续）

假设：$n=2^h-1$ (若不够，可增加一些节点)

那么：$h=\log(n+1)$

在第k层共有2^k个节点，他们分别需要做k+1次比较，如果每个节点查询概率是均等的，即1/n。则有：

$$ASL=\sum_{i=1}^{n} p_i C_i = \frac{1}{n}\sum_{i=1}^{n} C_i = \frac{1}{n}\sum_{i=0}^{h-1}(i+1)2^i$$

$$=\frac{1}{n}((h-1)\times 2^h+1)=\frac{1}{n}((n+1)\log_2(n+1)-n)$$

$$=\frac{n+1}{n}\log_2(n+1)-1\approx\log_2(n+1)-1$$

平均时间复杂度为：$O(\log_2 n)$

3.5、顺序表应用举例

1、串类型的定义

“串”是由零个或多个字符组成的有限序列，一般记为

$s=“a_1a_2…a_n”(n\geq 0)$

其中，s是**串名**，用单引号（也可以是用双引号括起来的）括起来的字符序列是**串的值**。a_i可以是字母、数字或其他字符；

串中字符的个数n称为串的**长度**。

子串：串中任意个**连续**的字符组成的子序列。

主串：包含子串的串相应地称为主串。

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。

相等：两个串的长度相等，并且对应位置的字符都相等。

注意区分**空串**与**空格串**的区别。

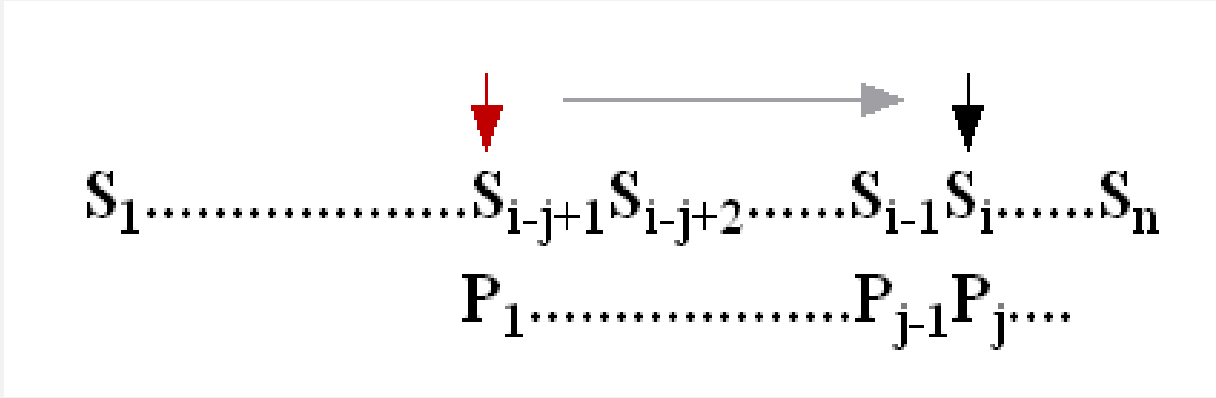
串的逻辑结构和线性表的区别:

1) 串的数据对象约束为**字符集**，而线性表的元素可任意。
2) 线性表的基本操作大多以“单个元素”为操作对象，而串的基本操作通常以“串的整体”（数据元素的集合）作为操作对象

2、模式匹配 --- 子串查找

模式匹配：子串（又称*模式串*）在主串中的定位操作

- 失配：匹配失败
- 主串：源串
- 模式串：要匹配的子串



一般做法(Brute-Force String Matching)

例3-4-2

NOBODY_NOTICED_HIE 中查找子串“NOT”

```
1  NOT
2     NOT
3        NOT
4           NOT
5              NOT
6                 NOT
7                    NOT
8                       NOT
```

设主串长n，模式串长为m

Worst Case: 每次匹配比较m次；需要比较n-m+1次。

T(n , m) = O(max(m*n,m*m))

例4-4-2

ABABCDABABE

ABABE

1 ABABCDABABE

ABABE

2 ABABCDABABE

ABABE

3 ABABCDABABE

ABABE

5 ABABCDABABE

ABABE

4 ABABCDABABE

ABABE

5 ABABCDABABE

ABABE

思想：指针在失配时，主串上的指针会回溯，而后继续比较，直到主串结束或中途匹配成功。

字符串匹配算法（朴素模式匹配）

其中：　typedef char ElemType；
　　SqList　List_main, List_sub;

调用语句为：

　　index(&List_main, &List_sub)

==========

```
int index( SqList  *S, SqList *P)
{    int  I,j,k;
     if  (P->length == 0) return 0;      //子串为空
     k=0;   i=k;   j=0;                                  //初始化：主串、子串下标
     while ( i<S->length && j<P->length ) {    //主串和子串都没有结束
          if  ( S->elem[i] == P->elem[j] )    // 主串与子串对应为比较
          { i++; j++ }                    //当前位置比较成功，调整两者后移一位
          else  {  k++;   i=k;  j=0;  } //主串原起点位失败，主串起点后移继续
     }
     if  ( j>= P->length)  return k+1; //子串到结束，比较成功，返回主串起点
     else return -1;              /* 匹配失败 */
}
```

求两个大整数的加法

3、大整数相加:

1) 任何机器上C语言的整数都有长度限制。比如32bit

2) 超过位数的大整数如何相加？

A=123456789234

B=9876543218767

A+B=10000000008001

基本思想：用三个顺序表分别存放两个被加数（list1和list2）和一个结果数(list3)，从两个被加数的低位开始，向高位按位相加，进位在下一个高位考虑加入。

由于结果表预先不知道位数，故现在结果list3中按倒序存放每一次的结果位（即从下标0开始存放），最后进行顺序表的逆转。

算法举例及算法程序见书p88-92 （略）

求两个大整数的加法(顺序表逆转)

基本思想：从两头开始向中间两两互换位置。

```
void  reverse(SqList *L3)      //逆转参数给出的字符串
{
    ElemType   temp, *p, *q;   //定义一个变量和两个指针变量
    p = L3->elem;              //取顺序表头元素的指针（地址）
    q = L3->elem + L3->length-1;  //取表尾元素的指针(指针加法)
    while( p < q )         //两头向中间对换是否结束
    {     temp = *p;       //存储p指向的对象（当前头）到临时变量
          *p = *q;         //把q指向对象(当前尾)复制到p指向位置
          *q = temp;       //把临时变量中原来p的值赋到q位置
          p++;             //当前头指针后移一位
          q--;             //当前尾指针前移一位
    }  //end of while(…)
}
```

《本章完》