第2章 线性表

- **主要内容：**
- 线性表的概念和逻辑结构
- 线性表的顺序表示和实现
- 线性表的链式表示和实现

线性表概述

- 线性表是一种线性结构。
- 线性结构的特点是数据元素之间是一种线性关系，数据元素“一个接一个的排列”。
- 在一个线性表中数据元素的类型是相同的，或者说线性表是由同一类型的数据元素构成的线性结构。

线性表是最简单、最常用的一种数据结构

§2.1 线性表的概念和逻辑结构

- 1. 线性表的定义
- **线性表(Linear List)**：由$n(n\geqslant 0)$个数据元素(结点)a_1，a_2，…，a_n组成的有限序列。
  - 数据元素的个数n为表的长度。当$n=0$时为空表，非空的线性表$(n>0)$记为：$(a_1, a_2, \ldots, a_n)$
  - $a_i(1\leqslant i\leqslant n)$只是一个抽象的符号，在同一表中特性相同，且相邻数据元素之间具有**序偶**关系。
- 例1：英文字母表：(A, B, C, …, Z)
- 例2：一周的组成：(周日, 周1, 周2, …, 周六)
- 例3：一周金价：(300, 302,…, 300, 301)

2. 形式定义

- **形式定义**：**Linear—List** = (D, R)

  $D = \{ a_1, a_2, \ldots\ldots, a_n \}$

  $R = \{<a_{i-1}, a_i> \mid i = 2\ldots..n\}$
  - n 称为线性表长度，$n = 0$ 称空表；
  - a_i为数据元素；
  - 当线性表非空时，i为数据元素a_i在表中的位序。
    - a_1是第1个数据元素，a_2是第2个数据元素，a_n是第n个数据元素，a_i是第i个数据元素。

非空线性表的逻辑特征

- 有且仅有一个被称为“第1个”的首元素a_1，它没有直接前趋，至多有一个直接后继
- 有且仅有一个被称为“最后1个”的尾元素a_n，它没有直接后继，至多有一个直接前趋
- 除首元素外，任何元素有且仅有一个前驱
- 除尾元素外，任何元素有且仅有一个后继
- 每个元素有一个位序

一个图书馆里的书目数据是否线性表?

3. 抽象数据类型定义

```
ADT  List{
   数据对象：D
   数据关系：R
   基本操作：
     Initlist(&L);              Destroylist(&L);
     Clearlist(&L);             Listempty(L);
     Listlength(L);             Getelem(L, i, &e);
     Locatelem(L,e);            Priorelem(L,e,&pe);
     Nextelem(L,e,&ne);         Listinset(&L,i,e);
     Listdelete(&L,i,&e);       Listtraverse(L);
} ADT  List
```

§2.2 线性表的顺序表示和实现

- 1. 顺序表
- **顺序表**：把线性表的结点按逻辑顺序依次存放在一组地址连续的存储单元里。用这种方法存储的线性表简称**顺序表**。
  - 特点：逻辑上相邻←→物理地址相邻
- 顺序表存储结构的描述
  - 存储空间的起始地址
  - 顺序表的容量(最大长度)
  - 顺序表的当前长度
- 顺序表的内存分配：一维数组

顺序存储结构也称为向量存储

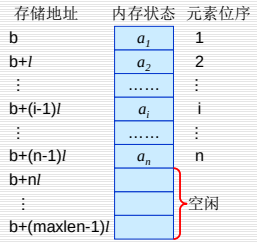
顺序表存储示意图

- 设：
  - ***b***为存储空间的首地址
  - ***l***为数据元素长度
- 第***i+1***个数据元素的存储位置和第***i***个数据元素的存储位置之间满足下列关系：

$LOC(a_{i+1})=LOC(a_i)+l$

- 线性表的第***i***个数据元素a_i的存储位置为：

$LOC(a_i)=LOC(a_1)+(i-1)*l$



任何数据元素均可在O(1)时间内存取

存储结构和存取结构

- 存储结构：数据及其逻辑结构在计算机中的表示
- 存取结构：在一个数据结构上对查找操作的时间性能的一种描述
  - 顺序存取：从第一个数据元素开始顺序进行访问，直到需要的数据元素
  - 随机存取：通过某种方式，直接找到需要的数据元素
- “顺序表是一种随机存取的存储结构”的含义为：在顺序表这种存储结构上进行的查找操作，其时间性能为O(1)。

2. 顺序结构的实现

- 组成

elem[ ]	length	listsize

  - 一组连续存储空间
  - 当前长度
  - 空间大小(当前分配的存储容量)
- 存储结构定义的C语言实现

```
#define  List-Init-Size   100
#define  Listincrement    10
typedef  struct  {
      Elemtype  *elem；//基地址
      int     length；//数据个数
      int     listsize；//总容量
} Sqlist；
```


3. 顺序表的基本操作

- (1) 初始化

```
status Initlist_Sq(Sqlist &L)
{
  L.elem=(ElemType*)malloc(LIST_INIT_SIZE* sizeof(ElemType));
  if(!L.elem)
    exit(OVERFLOW);   //存储分配失败
  L.length=0;              //空表长度为0，满为List-Init-Size
  L.listsize= List-Init-Size; //初始存储容量
  return  OK;
}
```


(2) 查找

- 按位序

```
void GetElem_Sq(L,i,&e)
{  e=L.elem[i-1];  }
```

注意区分数据元素的序号与数组元素下标之间的关系

- 按数据元素值

```
int LocateElem_Sq(SqList L,ElemType e)
{  int i=0;
   while(i < L.length && L.elem[i] != e) i++;
   if(i<L.length) return(i+1);
   else return(0);
}
```

平均比较次数为：
(1+2+3+…+n)/n=(n+1)/2
算法时间复杂度为O(n)

(3) 插入

- 插入是指在表的第i $(1\leqslant i\leqslant n+1)$个位置上，插入一个新结点$x$，使长度为$n$的线性表：$(a_1, \ldots, a_{i-1}, a_i, \ldots, a_n)$，变成长度为$n+1$的线性表：$(a_1, \ldots, a_{i-1}, x, a_i, \ldots, a_n)$。

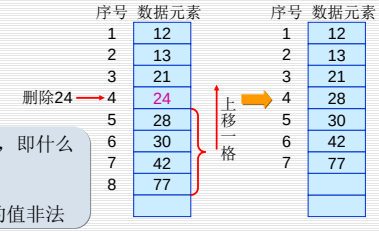
注意边界条件，即什么时候不能插入
①表空间已满
②插入位置i的值非法


```
status Listinsert_Sq(Sqlist &L,int i, Elemtype e)
{  if (i<1 ‖ i >L.length+1) return ERROR;
   if (L.length>=L.listsize) //判断是否有空闲的存储空间
   {  newbase=(ElemType *)realloc(L.elem,
            (L.listsize+Listincrement)*sizeof(ElemType));
      if(!newbase) exit(OVERFLOW);
      L.elem=newbase;
      L.listsize+=Listincrement;
   }
   q=&(L.elem[i-1]);  //q为插入位置
   for (p=&(L.elem[L.length-1]); p>=q; --p)
      *(p+1)=*p;        //插入位置及之后的元素后移
   *q=e;   //插入
   ++L.length;
   return OK;  }
```

for(j = L.length; j >= i; j --)
L.elem[j] = L.elem[j-1];
L.elem[j] = e;

(4) 删除

- 线性表的删除运算是指将表的第i $(1\leqslant i\leqslant n)$结点删除，使长度为$n$的线性表：$(a_1, \ldots, a_{i-1}, a_i, \ldots, a_n)$变成长度为$n-1$的线性表：$(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n)$



注意边界条件，即什么时候不能删除
①表空
②删除位置i的值非法

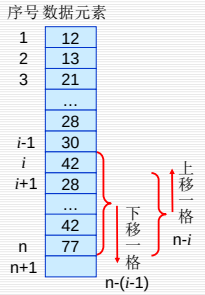
删除程序

```
status Listdelete_Sq(Sqlist &L,int i, Elemtype &e)
{   if (i<1 ‖ i >L.length) return ERROR;
    p=&(L.elem[i-1]);   //被删除元素的位置
    e=*p;   //取出被删结点元素
    q=L.elem+L.length-1; //表尾位置
    for (++p;p<=q;++p)
       *(p-1)=*p;     //被删除元素之后的元素左移
    --L.length;
    return OK;
}
```

for(j=i;j<L.length;j++)
L.lem[j-1]=L.lem[j];

算法复杂度分析

- 问题规模为表的长度，设为n
- 移动结点的次数由表长n和插入位置i决定
  - 当$i=n+1$(插入)，或$i=n$(删除)时，由于操作都在尾部，将不移动语句。这是最好情况，其时间复杂度$O(1)$。
  - 当$i=1$时，移动语句将循环执行n次(插入)或$n-1$次(删除)，需移动表中所有结点，这是最坏情况，其时间复杂度为$O(n)$。



算法复杂度分析(.续)

- 平均时间复杂度：
  - 插入：在长度为n的线性表中第i个位置之前插入一个结点，则在第i个位置之前插入一个结点的移动次数为$n-i+1$。令$E_{is}(n)$表示移动结点的期望值(即移动的平均次数)，故：

$$E_{is}(n)=\sum_{i=1}^{n+1}p_i(n-i+1)$$

  - 删除：在长度为n的线性表中删除第i个元素所需移动的元素次数为为$n-i$，则其数学期望值为：

$$E_{dl}(n)=\sum_{i=1}^{n}q_i(n-i)$$

算法复杂度分析(..续)

- 假设在表中任何位置($1 \leqslant i \leqslant n+1$)上插入或删除结点的机会是均等的，则：

$p_{i+1}=1/(n+1)$，$q_i=1/n$

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

$$E_{dl}(n)=\frac{1}{n}\sum_{i=1}^{n}(n-i)=\frac{n-1}{2}$$

- 在等概率插入的情况下$O(n)$。

(5) 其它操作

```
void DestroyList_Sq(&L)
{  free(L.elem); L.elem=NULL;  }
```

```
void ClearList_Sq(&L)
{  L.length=0;  }
```

```
int ListLength_Sq(L)
{  return(L.length);  }
```

```
status ListEmpty_Sq(SqList L)
{  if(L.length==0)  return(TRUE);
   else return(FALSE);  }
```


顺序表总结

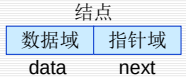
- 优点
  - 存储空间使用紧凑
  - 可随机存取任一元素
- 缺点
  - 预先分配空间需按最大空间分配，利用不充分
  - 表容量扩充麻烦
  - 插入、删除操作需要移动大量元素

§2.3 线性表的链式表示和实现

- 1. 线性链表(单链表)
- 链式存储结构特点
  - 用一组**任意**的存储单元存储线性表的数据元素
  - 利用**指针**实现用物理上不相邻的存储单元存放逻辑上相邻的元素
  - 每个数据元素a_i，除存储本身信息外，还需存储包含其相邻数据元素的位置信息的指针
- 线性链表(单链表)特点
  - 每个数据元素结点只包括一个指针域，存放直接后继的地址信息

线性链表的表示

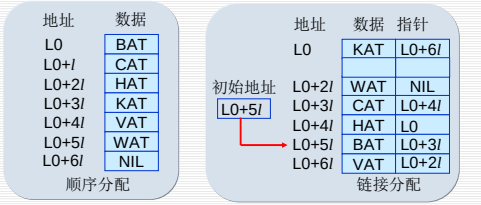
- 数据元素/结点

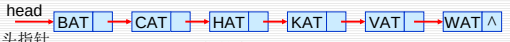


  - data域是数据域存放元素本身信息
  - next是指针域(链域)，存放直接后续的存储位置
- 线性链表
  - 每个结点的存储地址都由前一结点的指针域指示
  - 设头指针head，指出第1个结点的地址
  - 最后一个结点无后继结点，其指针域为空

举例：单链表存储示意

- 例如：线性表[BAT, CAT, HAT, KAT, VAT, WAT]的顺序存储结构和链式存储结构



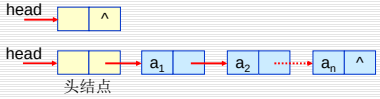


头结点

- 空表和非空表
  - 空表：head=Null
  - 非空表：

如何让空表和非空表统一？

- 头结点：为链表设置一相同类型的表头结点，一般无数据信息，只含首元素的地址信息
  - 空表：
  - 非空表：



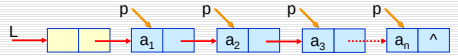
2. 线性链表的实现

- 存储结构：非顺序映象或链式映象
- 存储结构的C语言定义

```
typedef  struct LNode
{
   ElemType data;     /*定义链表中数据类型*/
   struct LNode *next;  /*定义链表中下一个数据地址*/
} Lnode, *LinkList;   /*结点类型名称与该链表的名称*/
```


3. 线性链表的基本操作

- (1) 查找
- 按序号查找/按值查找
- 例如：访问以L为头结点的链表中第 i个结点，且将此结点的地址存在p中。



- 核心操作：遍历
  - 从头结点出发，通过工作指针的反复**后移**而将整个单链表“审视”一遍的方法称为扫描或遍历

如何让工作指针后移？

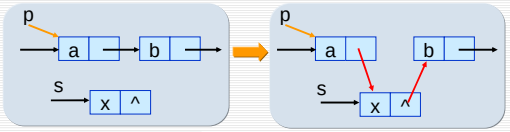
```
p=p->next;
```


查找程序

```
status Get_Elem(LinkList L, int i, ElemType &e)
{    p=L->next;        /*将p指向链表的第一个结点*/
     j=1;              /*结点计数器j置1*/
     while(p && j<i)   /*查找第i个元素*/
     {  p=p->next;     /* p指针后移，同时计数器加1*/
        j++; }
     if(!p || j>i)  return  ERROR; //表空和i>表长||i<1非法
     e=p->data;
     return OK; }
```


(2) 插入

- 在第i个元素前插入一个新结点
  - 找到插入位置，一般是第i个结点的前驱结点
  - 构造新结点
  - 插入
- 插入过程




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

这两条指令可否互换？

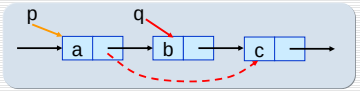
算法程序

```
status Listinsert_L(Linklist &L,int i,Elemtype e)
{    p=L;  j=0;  //插入位置可为1，因此其前一位置j从0开始
     while (p && j<i-1)  //查找第i-1个结点
     {  p=p->next; ++j;  }
     if ( !p || j>i-1) return ERROR; //表空和i>表长||i<1操作非法
     s=(Linklist)malloc(sizeof(LNodes)); //或new(s);
     s->data=e;
     s->next=p->next; //插入
     p->next=s;
     return OK;   }
```

- 时间复杂度：O(n)

(3) 删除

- 删除第i个元素
  - 找到删除位置
  - 删除
  - 释放结点a_i空间
- 删除过程




```
q=p->next;
p->next=q->next;
```

```
q=p->next;
p->next=p->next->next;
```


算法程序

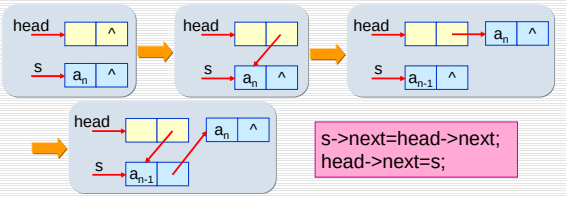
```
status Listdelete_L(Linklist &L,int i,Elemtype &e)
{   p=L;  j=0;
    while (p->next && j<i-1)  //查找第i-1个结点
    { p=p->next;  ++j; }
    if ( !(p->next) || j>i-1)  return ERROR; //删除位置非法
    q= p->next ; p->next=q ->next;        //删除结点
    e=q ->data;
    free(q);
    return OK;  }
```

注意：当i=n+1(被删结点不存在)，其前趋结点存在，为尾结点，此时p不为空。因此仅当*p存在，且*p不是尾结点时(p->next!=NULL)，才能确定被删结点存在。

- 时间复杂度：O(n)

(4) 建立单链表

- 尾插法建表
  - 构造结点，将输入数据以正序依次插入表尾
- 头插法建表
  - 构造结点，将输入数据以逆序依次插入表头



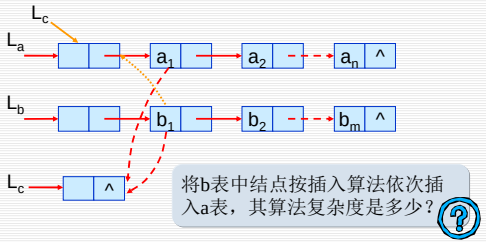

```
s->next=head->next;
head->next=s;
```


头插法算法程序

```
void CreateList_L(LinkList &L, int n)
{   L=(LinkList)malloc(sizeof(Lnode) );
    L->next=NULL;  //先建立带头结点的空链表
    for(i=n; i>0; --i)
    {   s=(LinkList)malloc(sizeof(Lnode));
        scanf(s->data);
        s->next=L->next;   //插入表头
        L->next=s;  }
}
```

- 时间复杂度O(n)

(5) 有序表的合并



将b表中结点按插入算法依次插入a表，其算法复杂度是多少？

- 合并方法：将两个有序表以遍历的方式依次插入到第3个表中。

算法程序

```
void Mergelist_L(Linklist &La, Linklist &Lb, Linklist &Lc)
{   pa=La->next;  pb=Lb->next;
    Lc=pc=La;  //用La的头结点作为Lc的头结点
    while (pa && pb)
    {   if ( pa->data<=pb->data)
            { pc->next=pa;pc=pa;pa=pa->next;}
        else { pc->next=pb;pc=pb;pb=pb->next;}
    }
    pc->next =pa? pa:pb;  //插入剩余段
    free(Lb);  //释放Lb的头结点
}
```

- 时间复杂度O(m+n)

(6) 其它操作

初始化

```
LinkList InitList_L(void)
{  head=(ListNode *)malloc(sizeof(ListNode));
   head->next=NULL;
   return(head);  }
```

撤销

```
void DestroyList_L(LinkList *L)
{   p=*L;
    *L=NULL;
    while(p)  { q=p->next;
                free(p);
                p=q;     }    }
```

清空

```
void ClearList_L(LinkList *L)
{   p=L->next;
    L->next=NULL;
    while(p)  { q=p->next;
                free(p);
                p=q;     }     }
```


单链表总结

- 优点
  - 它是一种动态结构，整个存储空间为多个链表共用，不需要预先分配空间
  - 插入删除不需要移动数据元素
- 缺点
  - 指针占用额外存储空间
  - 不能随机存取，查找速度慢

从单链表中某结点p出发：如何找到其他结点？
如何找到其前驱？

总结：顺序表和单链表比较

- 时间性能比较
  - 按位查找：
    - 顺序表的时间为O(1)，是随机存取
    - 单链表的时间为O(n)，是顺序存取
  - 插入和删除：
    - 顺序表需移动元素，时间为O(n)；
    - 单链表不需要移动元素，在给出某个合适位置的指针后，插入和删除操作所需的时间仅为O(1)。

顺序表和单链表比较(.续)

- 空间性能比较

$$\text{存储密度}=\frac{\text{数据域占用的存储量}}{\text{整个结点占用的存储量}}$$

  - 结点的存储密度：
    - 顺序表中每个结点的存储密度为1，没有浪费空间
    - 单链表的每个结点的存储密度<1，有指针的结构性开销。
  - 整体结构：
    - 顺序表需要预分配存储空间，如果预分配过大，造成浪费，若估计过小，又将发生上溢；
    - 单链表不需要预分配空间，元素个数没有限制。

顺序表和单链表比较(..续)

- 若线性表需频繁查找却很少进行插入和删除操作，或其操作和元素在表中的位置密切相关时，宜采用顺序表作为存储结构；若线性表需频繁插入和删除时，则宜采用单链表做存储结构。
- 当线性表中元素个数变化较大或者未知时，最好使用单链表实现；而如果用户事先知道线性表的大致长度，使用顺序表的空间效率会更高。