# 数据结构-第二章-线性表

Created by Yusheng Huang




**数据结构-第二章-线性表**

1. 线性表的概念
   线性表的两种存储
2. 顺序表
   2.1 顺序表实现--C语言(struct)
      1）结构体
      2）初始化
      3）增(插入)
      4）删
      5) 查和改
   2.2 应用举例
3.链表--基于C语言(struct)实现
   3.1 单链表
      1）单链表概述
      2）数据结构
      3）初始化与链表的创建
      4）增（插入）
      5）删
      6）查/改
   3.2 循环单链表

# 1. 线性表的概念

- 线性表 ——是n个相同类型数据元素组成的有限序列。
- 相邻元素间是**前驱和后继的顺序关系**，对于任一对相邻结点< ai，ai+1 >，ai称为ai+1的前驱，ai+1称为ai的后继

特点：

- 表中元素都是同一数据类型
- 元素个数有限，表长就是个数
- 线性表是有顺序的，有唯一的第一和最后一个元素

## 线性表的两种存储

- **顺序存储--叫顺序表**
  - 操作时间复杂度：
    - 查/改--知道序号就可以直接访问(叫随机存取)，时间复杂度O(1)
    - 插入/删除--要移动插入/删除位置后面的所有元素，平均时间复杂度O(n)
  - 优点：
    - 实现简单，可以基于数组去实现
    - 无需额外存储空间，空间利用率高
    - 方便存取任一元素
  - 缺点：
    - 插入/删除要移动大量元素
    - 难以判断存储空间的大小，存在空间的浪费
  - **频繁的访问，使用顺序表**
- **链式存储--叫链表**
  - 操作时间复杂度：
    - 查/改--需要遍历整个链表，平均时间复杂度O(n)
    - 插入/删除--插入删除给定元素，只需要改变指针域就行，时间复杂度O(1)
  - 优点：
    - 插入删除简单，不需要移动元素
    - 不需要考虑预分配多少空间，空间是动态分配的
  - 缺点：
    - 需要额外的存储空间存放逻辑关系，空间利用率小
    - 访问某个元素时，需要从头指针开始找，不具有按序号随机访问的特点
  - **频繁的插入删除，选择链表**

# 2. 顺序表

- 可以使用数组也可以使用vector实现。
- 记得顺序存储即可
- 设计前：
    - 需要确定数组的0号位置是元素还是表长
        - 是元素，需要额外的变量保存表长
        - 是表长，数组大小要预估多1
    - 需要预估数组的大小
    - 该数据结构操作的时候是传指针还是传引用

## 2.1 顺序表实现--C语言(struct)

- **一个数据结构基本操作就是：初始化+增删查改**
- 一个数据结构=数据+操作，即 结构体+操作的函数
- 此处的结构是，使用一个额外的变量表示表中最后一个元素的index，而表从数组0号位置开始
- 此处的结构，数据结构操作的时候是传指针

### 1）结构体

```
#define MAXSIZE 100
typedef struct Linear_list
{datatype  elem[MAXSIZE]; //数组
     int  last;//某尾元素的index
} SeqList;

SeqList  L ;//使用时
```

### 2）初始化

```
SeqList *InitList()//返回一个List
{
    SeqList *L;
    L = (SeqList *) malloc(sizeof(SeqList));
    L->last=-1;
    return L;
}

//调用初始化的函数
SeqList *L = InitList();
```

### 3）增(插入)

- 操作概述
    - 在某个位置增加元素，就是把该位置后面的元素向后移动一位
    - 需要把某尾插入和中间插入统一
- **异常情况：**
    - 表满不能增
    - 给定的index要在表内
- **代码框架：**
    - 判断异常
    - 中间插入和末尾插入统一

        - 移动元素（某尾插入不需要）
        - 插入值
        - 个数统计++

```
int List_Insert(SeqList* L, int index, int value)
{
    //index是插入位置(这里假定index是从0开始的数组序号，因为last也是从0开始的数组序号，因此统一一下)
    //value是插入的值
    if (L->last==MAXSIZE-1||index<0||index>MAXSIZE-1)//异常情况
        //对应 表满  和 index非法
        return -1;
    else//这里需要合并一下在某尾/中间插入的情况
    {
        /*
        某尾插入时，index为last+1，for循环（即移动元素）不会被执行
        需要执行的只是 插入和个数统计增加
        --从后往前向后移，不需要额外存储空间
        */
        for(int tmp_index=L->last+1; tmp_index>index; tmp_last--)
        {
            L->elem[tmp_index]=L->elem[tmp_index-1];
        }
        L->elem[index]=value;
        L->last+=1;
    }
    return 0;
}
```

### 4）删

- 操作概述
    - 在某个位置删除元素，就把某个位置之后的元素向前移动就好
    - 需要把末尾删除和中间删除统一
- **异常情况：**
    - 表空不能删除
    - 给定的删除index要在表内
- 代码框架：
    - 判断异常情况
    - 某尾删除和中间删除统一
        - 移动元素
        - 个数统计++

```
int List_Delet(SeqList* L, int index)
{
    //index是删除位置(这里假定index是从0开始的数组序号，因为last也是从0开始的数组序号，因此统一一下)
    if (L->last==-1||index<0||index>MAXSIZE-1)//异常情况
        //对应 表空  和 index非法
        return -1;
    else//删除元素
    {
        /*
```

```
        假设元素的非法值是0
        为了使得中间删除和某尾删除相统一，我们先将需要删除位置的元素置为非法值，然后再移动元素
        这样，如果是末尾删除，元素是不需要被移动的，然后被删除的元素也被置为非法值了
        如果被删除的元素是在中间，那么会被后面的覆盖掉，无所谓
        --从前往后向前移，不需要额外存储空间
        */
        L->elem[index]=0;//要删除的元素设计为非法值 这样就可以和末尾删除兼容
        for(int tmp_index=index; tmp_index<L->last; tmp_last++)
        {
            //末尾删除时，tmp_index==L->last循环不会被执行
            L->elem[tmp_index]=L->elem[tmp_index+1];
        }
        L->last-=1;
    }
    return 0;
}
```

### 5) 查和改

- 给定序号直接查和改就行。
- 按值查找直接遍历即可

## 2.2 应用举例

**例2-5 有顺序表A和B，其元素均按从小到大的升序排列，编写一个算法将它们合并成一个顺序表C，要求C的元素也是从小到大的升序排列。例如A = (2, 2, 3), B = (1, 3, 3, 4)，则C = (1, 2, 2, 3, 3, 3, 4)。**

**算法思路：**

**依次扫描A和B的元素，比较当前的元素的值，将较小值的元素赋给C，如此直到一个线性表扫描完毕，然后将未完的那个顺序表中余下部分赋给C即可。C的容量要能够容纳A、B两个线性表。设表C是一个空表，设两个指针i、j分别指向表A和B中的元素，若A.elem[i] > B.elem[j]，则将B.elem[j]插入到表C中；若A.elem[i]≤B.elem[j]，则当前先将A.elem[i]插入到表C中，如此进行下去，直到其中一个表被扫描完毕，然后再将未扫描完的表中剩余的所有元素放到表C中。**

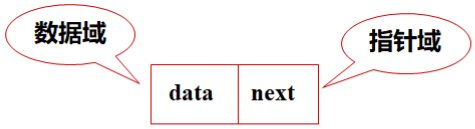
```
1  void merge(SeqList A,  SeqList B,  SeqList  *C)
2  {   int  i, j, k;
3      i=0;  j=0;  k=0;
4      while(i<=A.last && j<=B.last)     /* A表、B表都不为空*/
5        if(A.elem[i]<=B.elem[j])
6           C->elem[k++]=A.elem[i++];   /*将A表中i指针指向记录插入到C中*/
7        else
8           C->elem[k++]=B.elem[j++];  /*将B表中i指针指向记录插入到C中*/
9     while(i<=A.last)                  /*将A表剩余部分插入到C中*/
10       C-> elem[k++]=A.elem[i++];
11    while(j<=B.last)                  /*将B表剩余部分插入到C中*/
12       C->elem[k++]=B.elem[j++];
13    C->last=k-1;
14  }
```

**包含有三个并列的循环，语句4～语句8循环次数为表A的长度或者表B的长度，语句9～语句12是两个并列的循环，这两个中只有可能做一个，循环次数为表A或表B剩余的部分。因此，该算法的时间复杂度是O(A.last + B.last)。**

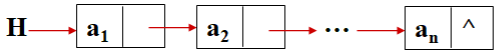
# 3.链表--基于C语言(struct)实现

## 3.1 单链表

### 1）单链表概述




**数据域——用来存储结点的值；**
**指针域——用来存储数据元素的直接后继的地址；**




**1) 链表每个结点只有一个指针域，这种链表又称为单链表。**

**2) 单链表第一个结点无前趋，设一个头指针H指向第一个结点。**

**3) 表中最后一个结点没有直接后继，则最后一个结点的指针域为“空”（NULL）。**

单链表建议带头结，这样可以实现插入删除的统一。下面给出的代码都是带头结点的。头结点的值是链表的长度。

**头指针和头结点的异同**

| 头 指 针 | 头 结 点 |
| --- | --- |
| ✧ 指向第一个结点的指针，若链表有头结点，则是指向头结点的指针<br>✧ 具有标示作用，常用头指针作为链表的名字 | ✧ 放在第一个元素之前，其数据域一般无意义(也可以放链表的长度)<br>✧ 有了头结点，在第一个元素结点之前插入和删除第一个结点，其操作与其他结点的操作就统一了<br>✧ 造成存储空间浪费 |

**不带头结点的链表和带头结点的链表的区别**

| 不带头结点 | 带 头 结 点 |
| --- | --- |
| ✧ 链表为空：L==NULL为真<br>✧ 链表的第一个数据元素由L指向<br>✧ 在第一个元素之前插入一个元素和删除第一个元素要单独处理，和在其他位置插入、删除操作不同 | ✧链表为空：L->next==NULL为真<br>✧链表的第一个数据元素由L->next指向<br>✧插入、删除操作统一 |

### 2）数据结构

```
typedef struct node
{   datatype   data;
     struct node  *next;
 }LNode, *LinkList; //LinkList 是指向该链表指针的类型别名
//定义一个链表
LinkList H; //一定要记得LinkList是一个指针
```

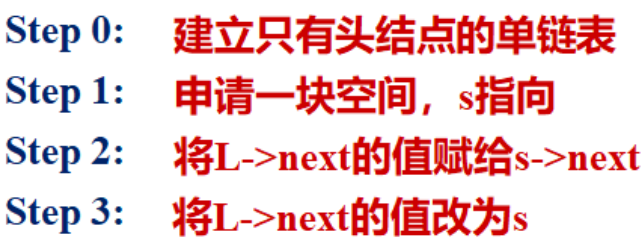
### 3）初始化与链表的创建

初始化

```
LinkList Init()//设置了一个头结点
{
    LinkList Node=(LinkList)malloc(sizeof(LNode));
    Node->data=0;
    Node->next=NULL;
    return Node;
}
//使用其进行初始化
LinkList H = Init(); //此时H指向一个结点，这个结点是头结点
```

创建

- 有两种方法
  - 头插法：**逆序创建链表**--




  - 尾插法：**顺序创建链表**

- **必须一直维护一个指向尾部的指针Tail**
- 算法：
  - 申请一个新空间，s指向
  - s->next=Null;
  - LLast->next=s;
  - LLast=s;更新尾结点的指向

头插法算法:

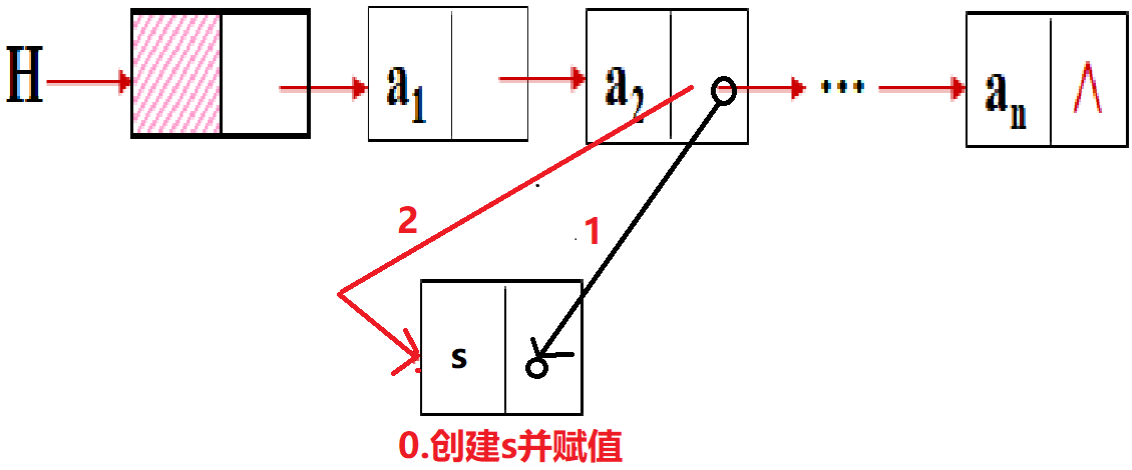
```
int CreatedFromHead(LinkList H, int data)
{
    LNode *s= (LNode *)malloc(sizeof(LNode));
    s->data=data;
    s->next=H->next;
    H->next=s;
    return 0;
}
```

尾插法算法:

```
//LNode *Tail是程序一直维护的指向表尾元素的指针
int CreatedFromTail(LNode *Tail, int data)
{
    LNode *s= (LNode *)malloc(sizeof(LNode));
    s->data=data;
    s->next=NULL;
    Tail->next=s;
    Tail=s;
    return 0;
}
```

### 4）增（插入）

插入图示--三步走：

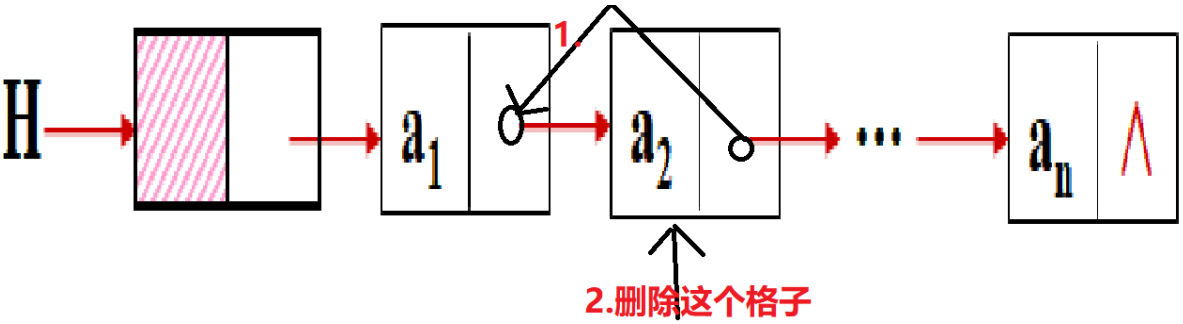



- 这里所示的插入是给一个指针，在该指针指向元素末尾插入,O(1)。**在某个位置插入，一定要知道该位置的前驱**
  - 这种指针使用过后需要销毁，因为不知道插入之后，这个指针指向哪里了
- 也有给定序号，找到该序号对应的元素，在该序号后插入，平均O(n)

```
int LinkList_insert(LinkList H, LNode* node, int value)
{
    LNode* nodetmp= (LNode*) malloc(sizeof(LNode));
    nodetmp->data=value;
    nodetmp->next=node->next;
    node->next=nodetmp;
}
```

### 5）删

删除图示--两步走




- 注意：所删除的元素要一定存在
- 这里举例的是给定指向元素的指针，删除后一个元素(即**删除某个结点一定要知道它的前驱**)
    - 这种指针使用过后需要销毁，因为不知道插入之后，这个指针指向哪里了
- 也有的是，给定序号，删除对应序号的元素

```
int LinkList_delet(LinkList H, LNode* node)
{
    LNode *tmpnode=node->next;//设置一个指向被删除结点的指针
    node->next=node->next->next;//从链表里面删除那个结点
    free(tmpnode);//释放掉那个被删除的结点
}
```

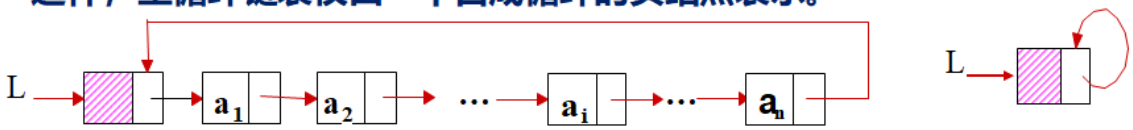
### 6）查/改

- 一直遍历，到当前结点->next==NULL结束
- 异常情况：表不能为空
- 查找的时间复杂度因为是遍历，所以平均复杂度为O(n)

## 3.2 循环单链表

**将单链表最后一个结点的指针域由NULL改为指向头结点或线性表中的第一个结点，就得到了单链形式的循环链表，并称为循环链表。**

**为了使某些操作实现起来方便，在循环单链表中也可设置一个头结点。这样，空循环链表仅由一个自成循环的头结点表示。**




**假设 P指向链表最后一个结点。**

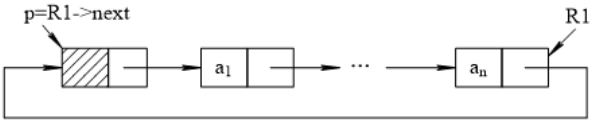
**单链表： p->next= =NULL为真。循环单链表： p->next== L为真。**

**判断链表是否为空的条件：**

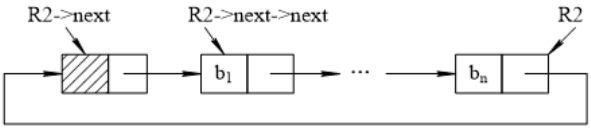
**单链表： p! = L　　循环单链表： p->next! = L。**

- 单链表--只能从**头结点**开始遍历整个链表
- 循环单链表则可以从表中**任意结点**开始遍历整个链表。
- 对链表常做的操作是在表尾、表头进行时，**设置一个指向尾结点的指针**，可以提高操作效率。

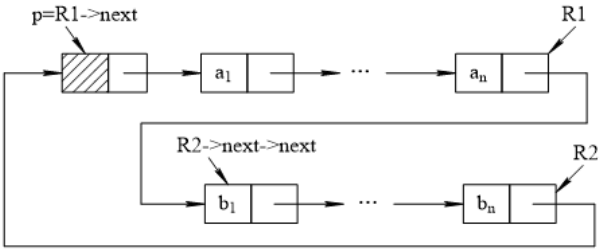
简单应用：有两个带头结点的循环单链表LA、LB，编写一个算法，将两个循环单链表合并为一个循环单链表，其头指针为LA




(a) 循环链表R1



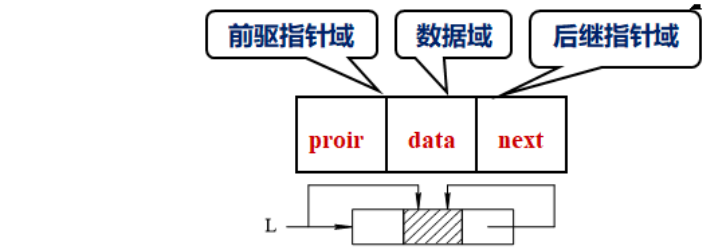

(b) 循环链表R2



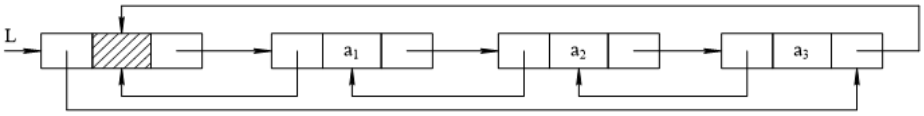

(c) R1、R2合并之后

## 3.3 双向链表

- 在单链表的每个结点里再增加一个指向其前驱的指针域prior。
- 这样形成的链表中就有两条方向不同的链，我们可称之为**双(向)链表(Double Linked List)。**




(a) 空的双向循环链表




(b) 非空的双向循环链表

1）插入

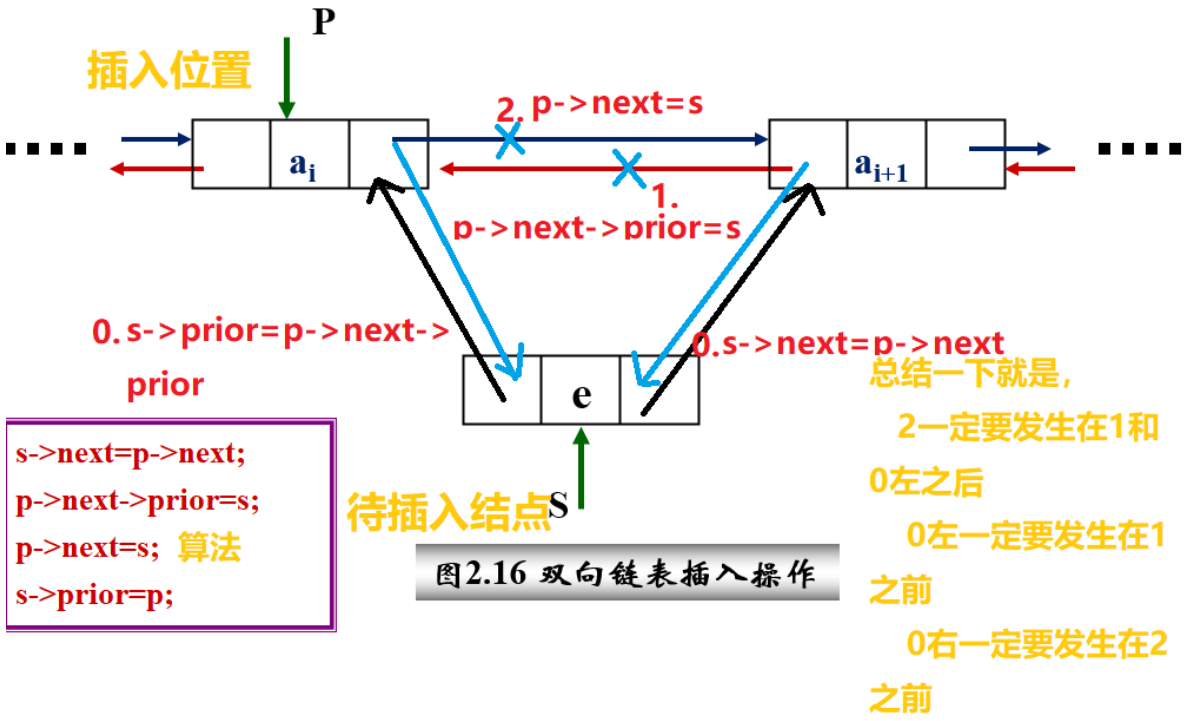
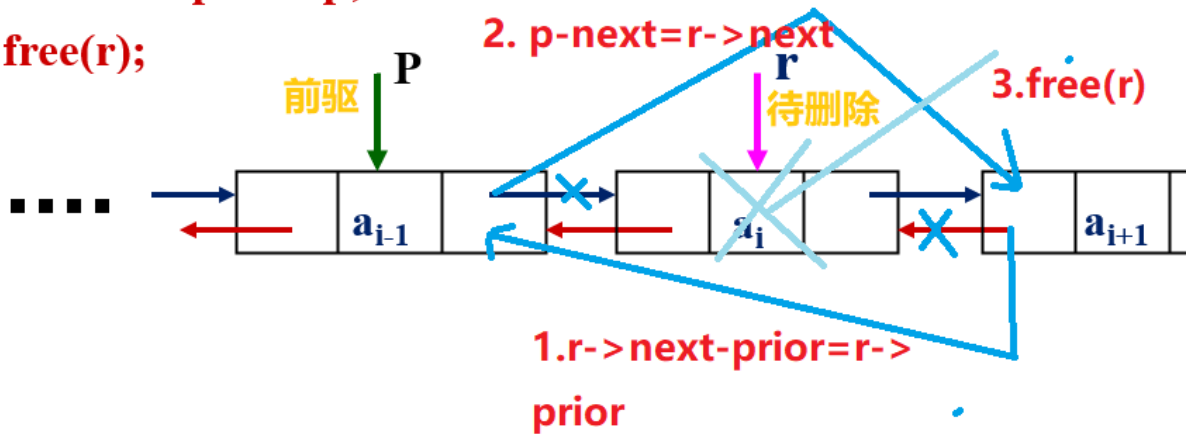



图2.16 双向链表插入操作

2）删除

算法

```
p->next=r->next;
r->next->prior=p;
free(r);
```



## 3.4 静态链表

**借用一维数组来存储线性链表。 定义一个较大的结构数组作为备用结点空间(即存储池)，每个结点应包含两个域data域和next域，data域用来存放结点的数据信息，而next域不再是指针而是指示其后继结点在结构数组中的相对位置(即数组下标)，通常称为游标。我们把用这种方式实现的单链表叫做静态链表(Static Linked List)。**

```
# define  Maxsize = 链表可能达到的最大长度
  typedef  struct
   {  datatype  data;
      int   next;
   } Component,  StaticList[Maxsize];
  StaticList  S;
  int SL, AV;                    /*两个头指针变量*/
```

SL=0

AV=6

| | data | next |
|---|---|---|
| 0 | | 4 |
| 1 | $a_4$ | 5 |
| 2 | $a_2$ | 3 |
| 3 | $a_3$ | 1 |
| 4 | $a_1$ | 2 |
| 5 | $a_5$ | −1 |
| 6 | | 7 |
| 7 | | 8 |
| 8 | | 9 |
| 9 | | 10 |
| 10 | | 11 |
| 11 | | −1 |

- SL和AV可以看作是两个带头结点的链表。SL是被使用的链表，而AV是空闲的链表。
- 要往静态链表中插入结点时，使用头插法往SL中插入一个结点（该结点是从AV中使用头删除法删除的结点）
- 要删除静态链表中结点时，整理SL中的结点，并往AV中使用头插法插入被SL删除的结点

## 3.5 经典应用

### 1）倒置单向链表

思路：扫描链表一遍并且用头插法，O(n)

```
void reverse(LinkList H)//设链表是带头结点的链表
{
    LinkList tmpnode=H->next;
    H->next=NULL;//为了保证程序可以一个循环，不需要判断，第一次时将H->next为NULL
    //因为头号元素就要变成最后一个啦，这样可以确保头号元素成为末尾时,某尾->next=NULL
    while(tmpnode!=NULL)
    {
        LinkList tmpnode2=tmpnode->next;
        tmpnode->next=H->next;
        H->next=tmpnode;
        tmpnode=tmpnode2;
    }
}
```

### 2）两个递增链表合并成一个递减链表

设有两个单链表A、B，其中元素递增有序，编写算法将A、B归并成一个按元素值递减（允许有相同值）有序的链表C，要求用A、B中的原结点形成，不能重新申请结点。

思路：

- 两个链表递增，因此选AB的头中最小的，即为最小的

- 利用头插法，插入之后导致链表的特点
- 即可以完成合并和倒顺序
- 因为也只是扫描完A和B，时间复杂度为A和B的长度之和

```
LinkList reverseABtoC(LinkList A, LinkList B)//设A和B和C都是带头结点的单向链表
{
    LinkList C;
    C->data=0;
    C->next=NULL;
    LinkList pointerA=A;
    LinkList pointerB=B;
    while(pointerA->next!=NULL&&pinterB->next!=NULL)
    {
        if (pointerA->next.data>pointerB->next.data)
        {
            LinkList tmp=C->next;
            C->next=pointerB->next;
            C->next->next=tmp;
            pointerB->next=pointerB->next->next;
        }
        else
        {
            LinkList tmp=C->next;
            C->next=pointerA->next;
            C->next->next=tmp;
            pointerA->next=pointerA->next->next;
        }
    }
    if (pointerA->next==NULL)//下面统一操作pointerA
        pointerA=pointerB;
    while(pointerA->next!=NULL)
    {
        LinkList tmp=C->next;
        C->next=pointerA->next;
        C->next->next=tmp;
    }
    return C;
}
```