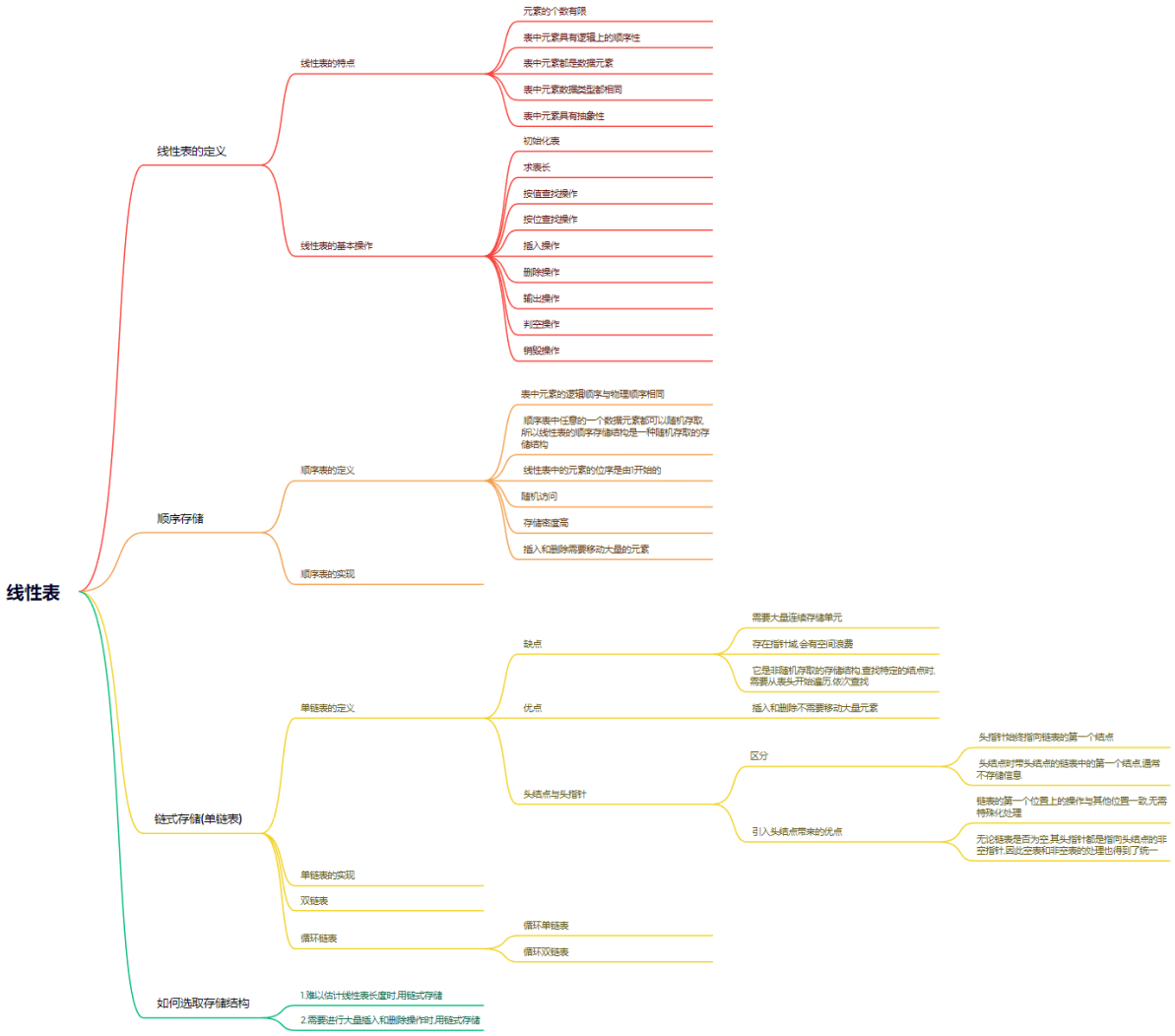
第二章 线性表



1.线性表的定义和基本操作

1.1线性表的定义

线性表:具有相同数据类型的n个数据元素的有限序列

表头元素:线性表中的"第一个"数据元素

表尾元素:线性表中"最后一个"数据元素

直接前驱(前驱):线性表中一个数据元素相邻的前一项数据元素

直接后继(后继):线性表中一个数据元素相邻的后一项数据元素

线性表的**特点**:

- 元素个数有限
- 有先后次序
- 每个元素都是单个元素(由于元素都是数据元素)
- 每个元素占有相同的存储空间(由于数据类型相同)
- 抽象性,只讨论元素间的逻辑关系,不考虑元素的具体内容

1.2例题

1.线性表是具有n个()的有限序列

答:数据元素

2.以下()是一个数据元素

A.由n个实数组成的集合

B.由100个字符组成的序列

C.所有整数组成的序列

D.邻接表

答:B,注意线性表的要求特点为:相同的数据类型,有限序列

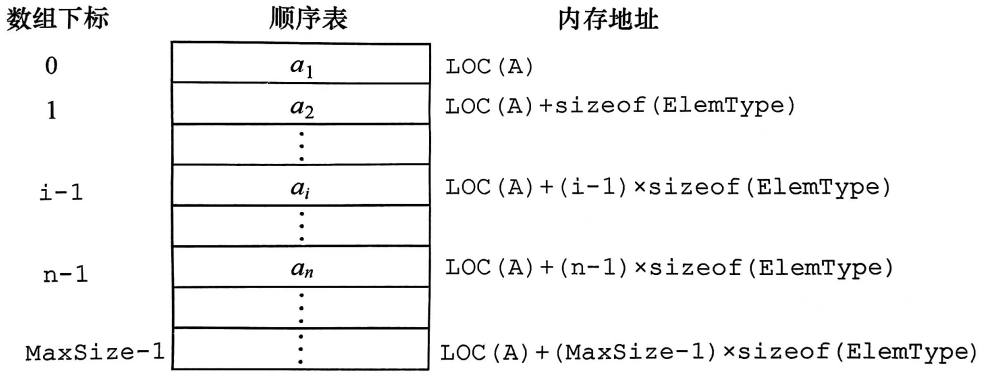
2.线性表的顺序表示

2.1顺序表的定义

顺序表:线性表的顺序存储

顺序表的特点:表中的逻辑顺序与其物理顺序相同

假设线性表L存储的起始位置为LOC(A),sizeof(ElemType)是每个数据元素所占用的存储空间大小,如图所示



顺序表任意一个数据元素都可以**随机存取**,所以线性表的顺序存储结构是一种**随机存取的存储结构**(详见第一章的存储结构)

注:**线性表中的元素位序是从1开始的**,而**数组中的元素下标是从0开始的**

一维数组有两种分配方法:

1. 静态分配:一般是直接定义一个固定长度大小的数组,由于数组大小和空间已经固定,一旦空间占满,新的数据便会溢出导致程序崩溃,

   例如:

   ```
   int arr[MaxSize];
   ```

2. 动态分配:用动态存储分配语句(malloc或new)分配,可以动态的变化数组的大小,避免了静态分配的缺点,

   例如:

   ```
   #define InitSize 100 //表初始长度的定义
   typedef struct{
       ElemType* data;      //指示动态分配数组的指针
       int MaxSize,length; //数组的最大容量和当前个数
   }SeqList;

   //C动态分配语句
   L.data=(ElemType*)malloc(sizeof(ElemType)*InitSize)

   //c++动态分配语句
   L.data=new ElemType[InitSize]
   ```

   注:使用**动态分配并不意味着链式存储**,它仍然**属于顺序存储结构**,物理结构并没有发生变化,**依然是随机存储方式**,只是分配的空间在运行时动态决定

顺序表的特点:

1. **最主要的特点:随机访问**,时间复杂度为O(1)
2. **存储密度高**,每个结点只存储数据元素
3. **插入和删除需要移动大量元**素,例如:在一个数组中删除一个元素,需要将该元素后面的所有元素向前移动占取删除元素的位置

2.2顺序表的基本操作的实现

1.**插入操作**

```
bool ListInsert(SqList &L,int i,ElemType e){
    if(i<1||i>L.length+1){ //如果插入的位置不正确,注意线性表的元素位序是从1开始的
        return false;//直接返回失败
    }
    if(L.length>=MaxSize){ //如果存储空间已满
        return false;//直接返回失败
    }
    for(int j=L.length;j>=i;j--){
        L.data[j]=L.data[j-1];//将插入元素位置的元素及后续元素全部向后移动一个元素大小
    }
    L.data[i-1]=e;//注意线性表的元素位序,与数组不同
    L.length++;
    return true;
}
```

最好的情况:在表尾插入,不需要移动元素,时间复杂度为O(1)

最差的情况:在表头插入,需要移动所有元素,时间复杂度为O(n)

平均情况:平均次数为n/2,时间复杂度为O(n)

2.**删除操作**

```
bool ListDelete(SqList &L,int i,ElemType &e){
    if(i<1||i>L.length) return false;
    e=L.data[i-1]; //注意线性表的元素位序与数组不同
    for(int j=i;j<L.length;j++){
        L.data[j-1]=L.data[j];//把第i个元素后的元素前移
    }
    L.length--;
    return true;
}
```

最好情况:删除表尾元素,无序移动元素,时间复杂度为O(1)

最差情况:删除表头元素,需要移动其他所有元素,时间复杂度为O(n)

平均情况: 平均次数(n-1)/2,时间复杂度为O(n)

由上述情况可知:**线性表插入和删除的时间主要浪费在移动元素上**

3.**查找操作**(按值查找,顺序查找)

```
int LocateElem(SqList L,ElemType e){
    int i;
    for(int i=0;i<L.length;i++){
        if(L.data[i]==e){
            return i+1;
        }
    }
    return 0;
}
```

最好情况:查找的元素在表头,时间复杂度为O(1)

最差情况:查找的元素在表尾,需要比较n次,时间复杂度为O(n)

平均情况:平均次数为(n+1)/2,时间复杂度为O(n)

2.3例题

1.顺序存储的优点有那些?

答:随机访问,存储密度大

2.线性表的顺序存储结构是一种()存取的存储结构

答:随机存取

3.一个顺序表所占用的存储空间大小与()无关

A.表的长度

B.元素的存放顺序

C.元素的类型

D.元素中各字段的类型

答:B,由线性表的动态初始化公式得:

```
L.data=(ElemType*)malloc(sizeof(ElemType)*InitSize)

sizeof(ElemType)*InitSize//意味着表的长度
(ElemType*)//意味着元素的类型,元素的类型又与各字段的类型相关
```

因此顺序表的存储空间大小为:表长*元素的类型

4.在一个长度为n的顺序表中删除第i个元素时,需要移动()个元素

答:需要移动n-i个元素,因为顺序表长为n,因此在第i个元素后面的元素总和为n-i个,向前移动一个位置即可,因此是n-i

5.在一个长度为n的顺序表中第i个位置插入新的元素,需要移动()个元素

答:需要移动n-i+1个元素,要在第i个位置插入新的元素,意味着要把第i个后续的所有元素(n-i)包含i(1)向后移动一个单位(n-i+1)

6.对于顺序表,访问第i个位置的元素和在第i个位置插入一个元素的时间复杂度为()

答:O(1),O(n),访问是随机存取,时间复杂度为O(1),插入和删除的时间复杂度为O(n)

7.若长度为n的非空线性表采用顺序存储结构,在表的第i个位置插入一个数据元素,则i的合法值为()

A.1<=i<=n

B.1<=i<=n+1

C.0<=i<=n-1

D.0<=i<=n

答:B,注意线性表的次序是从1开始的,长度为n,及位置有n+1个

8.从顺序表中删除具有最小值的元素(假设唯一)并由函数返回被删元素的值,空出的位置由最后一个元素填补,若顺序表为空,则显示错误并退出

```
bool DelMin(Sqlist &L,ElemType &value){
    if(L.length==0) return false; //如果表为空
```

```
    value=L.data[0]; //假设0号位元素最小
    int pos=0; //记录最小值的下标
    for(int i=0;i<L.length;i++){//遍历整个顺序表
        if(L.data[i]<value){//如果第i个元素比之前记录的最小元素小
            value=L.data[i];//更新最小的元素
            pos=i;//记录最新最小元素的位置
        }
    }
    L.data[pos]=L.data[L.length-1];//填补最后一个位置
    L.length--;
    return true;
}
```

9.设计一个高效算法,将顺序表L所有的元素逆置,要求算法的空间复杂度为O(1)

```
void Reverse(Sqlist &L){
    ElemType temp;//利用一个中间值交换
    for(int i=0;i<L.length/2;i++){
        temp=L.data[i];
        L.data[i]=L.data[L.length-i-1];
        L.data[L.length-i-1]=temp;
    }
}
```

10.对长度为n的顺序表L,编写一个时间复杂度为O(n),空间复杂度为O(1)的算法,该算法删除线性表中所有值为x的数据元素

```
//利用两个快慢双指针解决
void delX(Sqlist &L,ElemType x){
    int k=0,i;
    for(i=0;i<L.length;i++){
        if(L.data[i]!=x){//如果快指针所指向的元素不等于x
            L.data[k]=L.data[i];//把快指针指向的元素赋值给慢指针
            k++;//移动慢指针
        }
    }
    L.length=k;//更新线性表的长度
}
```

11.从有序顺序表中删除其值在定制s到t之间(包含t和s,要求s<t)的所有元素,若s或t不合理或顺序表为空,则显示出错信息并退出运行

```
bool Del_s_t(SqList &L,ElemType s,ElemType t){
    if(s>=t||L.length==0){//如果s大于t,表长为0返回false
        return false;
```

```
    }
    int i,j;
    for(i=0;i<L.length&&L.data[i]<s;i++);//找到第一个值大于或等于s的元素
    if(i>=L.length) return false;//如果在表长中没有找到大于s的值,返回false
    for(j=i;j<L.length&&L.data[j]<t;j++);//找到大于t的第一个元素
    for(;j<L.length;i++,j++){
        L.data[i]=L.data[j];//让大于t的元素占取删除元素的位置
    }
    L.length=i;//刷新表长
    return true;

}
```

12.从有序顺序表中删除所有值重复的值,使表中所有的值均不同

```
//利用快慢指针解决
bool Delete_same(SqList &L){
    if(L.length==0) return false;//判断线性表的长度是否正确
    int i,j=0;//j为慢指针,i为快指针
    for(i=1;i<L.length;i++){//遍历
        if(L.data[i]!=L.data[j]){//如果快指针的值不等于慢指针的值
            L.data[++j]=L.data[i];//移动快指针,并将快指针对应的元素赋值给慢指针
        }
    }
    L.length=j+1;//更新线性表的长度
    return true;
}
```

13.将两个有序顺序表合并为一个新的有序顺序表,并返回结果顺序表(**重点**)

```
bool Merge(SqList A,SqList B,SqList &C){
    if(A.length+B.length>C.maxSize) return false;//大于顺序表的最大长度
    int i=0,j=0,k=0;//三个指针,对应三个线性表
    while(i<A.length&&j<B.length){
        if(A.data[i]<=B.data[j]){
            C.data[k++]=A.data[i++];
        }else{
            C.data[k++]=b.data[j++];
        }
    }
    while(i<A.length) C.data[k++]=A.data[i++];
    while(j<B.length) C.data[k++]=A.data[j++];
    C.length=k;
    return ture;

}
```

3,线性表的链式表示

3,1单链表的定义

单列表:线性表的链式存储,指通过一组任意的存储单元来存储线性表中的数据元素

```
typedef struct LNode{
    ElemType data;
    Struct LNode *next;
}LNode,*LinkList;
```

结点结构:

其中data为数据域,存放数据元素

next为指针域,存放后继结点的地址

特点:

- **优点**:
    1. **解决了顺序表的需要大量连续存储单元的缺点**
    2. **在插入和删除时不需要移动大量元素**
- **缺点**:
    1. **单链表附加指针域,存在空间浪费**
    2. **单链表是非随机存取的存储结构,不能直接找到表中某个特定的结点,找查某一特定结点时需要从表头开始遍历**

注:根据顺序表和链表的优缺点我们可以选择在特定情况下合适的结构

通常情况下,我们会选择同**头指针来标识一个单链表**.此外,为了操作方便(一般情况下是为了避免表头的特殊情况),在单链表第一个结点之前附加一个结点,称为**头结点**.

在头结点中,头结点的数据域不设任何信息,也可以用于记录表长,头结点的指针域指向线性表的第一个元素结点

带头结点的单列表:

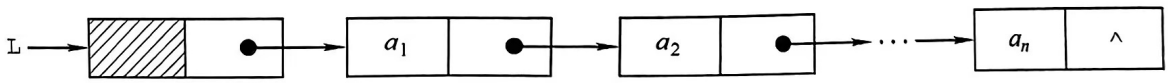


图 2.4　带头结点的单链表

头结点与头指针的区分:

- 头指针始终指向链表的第一个结点
- 头结点是带头结点链表的第一个结点,通常不存储信息

刚开始链表,头指针一般会指向头结点,

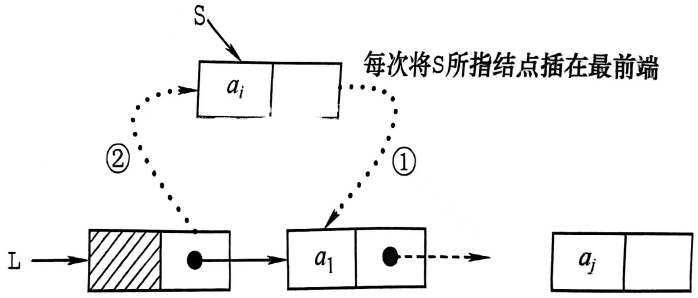
头结点可以不存在

头结点带来的优点:

1. 链表的第一个位置上的操作在表的其他位置上的操作一致,无需特殊化处理
2. 无论链表是否为空,其头指针都是指向头结点的非空指针,空表和非空表的处理得到了统一

3.2单链表基本操作的实现

1.**头插法建立单链表**

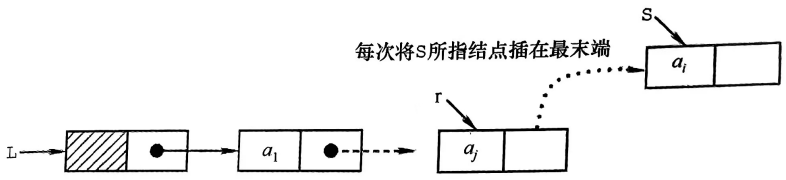



```
LinkList List_HeadInsert(LinkList &L){
    LNode* s;
    int x;
    L=(LinkList)malloc(sizeof(LNode));//创建头结点
    L->next=NULL;//初始为空链表
    scanf("%d",&x);//输入结点的值
    while(x!=9999){//输入9999表示结束
        s=(LNode*)malloc(sizeof(LNode));//创建新的结点
        s->data=x;
        s->next=L->next;
        L->next=s;
        scanf("%d",&x);
    }
    return L;
```

```
}
```

2.尾插法建立单链表

若需要结点的次序和输入数据的顺序一致,则使用尾插法




```
LinkList List_TailInsert(LinkList &L){//正向建立单链表
    int x;
    L=(LinkList)malloc(sizeof(LNode));
    LNode* s, *r=L;//r为表尾指针
    scanf("%d",&x);
    while(x!=9999){//输入9999表示结束
        s=(LNode*)malloc(sizeof(LNode));
        s->data=x;
        r->next=s;
        r=s;//表尾指针指向新的表尾结点
        scanf("%d",&x);
    }
    r->next=NULL;//尾结点置空
    return L;
}
```

3.按序号找查结点

从第一个结点触发,挨个向下搜索,直至找到第i个结点为止,否则返回最后一个指针域NULL

```
LNode *GetElem(LinkList L,int i){
    if(i<1){
        return NULL;
    }
    int j=1;//计数
    LNode *p=L->next;//第一个结点指针赋值给p
    while(p!=NULL&&j<i){//从第一个结点开始找查
        p=p->next;
        j++;
    }
    return p;
}
```

4.**按值查找表结点**

从第一个结点出发,依次比较表中各结点数据域中的值,若结点数据域中的值等于e,则返回该结点的指针,若没有该节点则返回NULL

```
LNode* LocateElem(LinkList L,ElemType e){
    LNode* p=L->next;
    while(p!=NULL&&p->data!=e){
        p=p->next;
    }
    return p;
}
```

5.**插入结点操作**

及其重要:下面的第二句,第三句位置不能调换

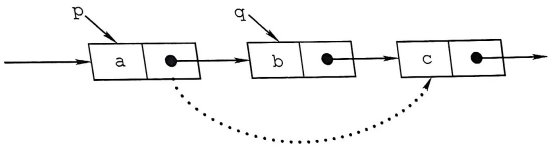
```
p->GetElem(L,i-1);
s->next=p->next;//先将新结点的指针指向原结点指针指向的位置
p->next=s;//再将原结点指针指向新结点
```

后插操作

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

6.**删除结点操作**

将单链表的第i个结点删除




```
p=GetElem(L,i-1);//找查删除位置的前驱节点
q=p->next;//令q指向被删除的结点
p->next=q->next;//将*p结点从链中断开
free(q);//释放存储空间
```

删除结点*p

将后继结点的值赋予自身,然后删除后继结点

```
q=p->next;//令q指向*p的后继结点
p->data=p->next->data;//用后继结点的数据域覆盖
p->next=q->next;//将*p结点从链中断开
free(p);//释放后继结点的存储空间
```

7.**求表长操作**

设置一个计数器,依次遍历

3.3双链表

在单链表中只有一个指向后继的指针,使得单链表只能从头结点依次顺序向后遍历.而当要访问某个结点的前驱结点时,只能从头开始遍历,为克服上述缺点,我们引入了双链表,双链表结点中有两个指针prior和next分别指向其前驱结点和后继结点.

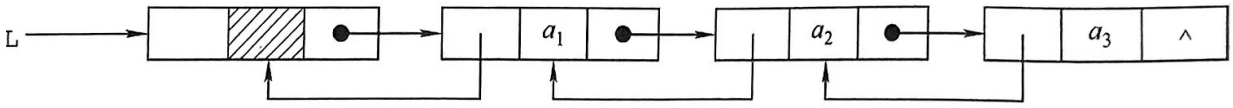


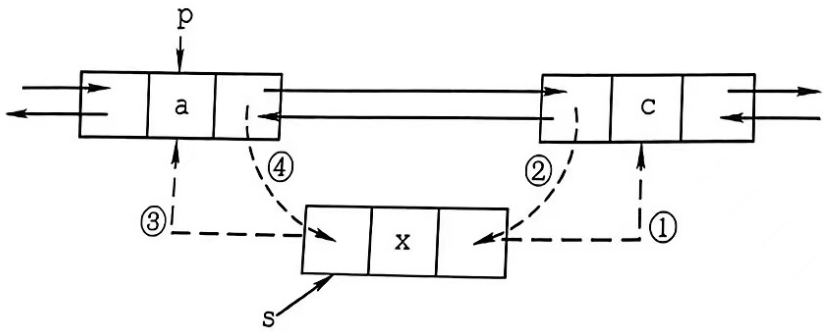
图 2.9　双链表示意图

```
typedef struct DNode{
    ElemType data;//数据域
    Struct DNode *prior,*next;//前驱和后继指针
}DNode,*NodeList
```

双链表的优点:相较于单链表双链表能够更快的找到前驱结点

缺点:会花费更多的空间大小

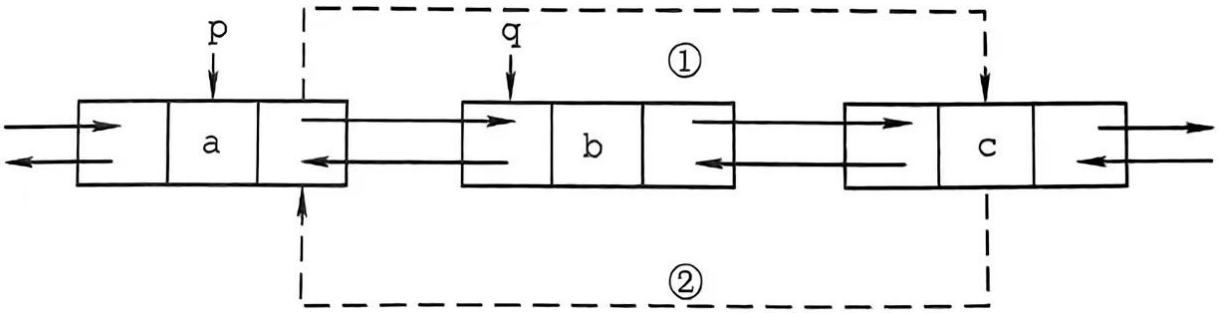
1.**双链表的插入操作**




```
1.s->next=p->next;
2.p->next->prior=s;
3.s->prior=p->next;
4.p->next=s;
```

2.**双链表的删除操作**

删除*p的后继结点*q



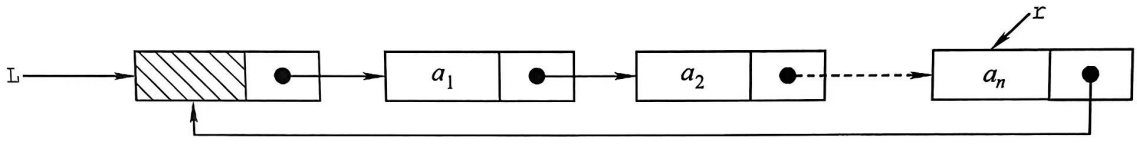

```
1.p->next=q->next;
2.q->next->prior=p;
free(q);
```

3,4循环链表

1.**循环单链表**

循环单链表与单链表的区别:

表中的最后一个结点的指针不是NULL,而改为指向头结点,使整个链表形成一个环

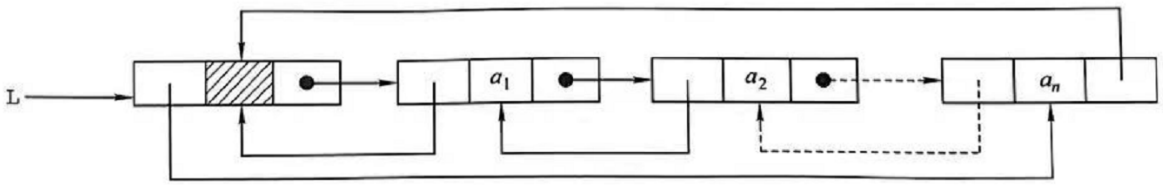


优点:可以从表中的任意一个结点开始遍历整个链表

有时对循环单链表不设头指针而仅设尾指针,以使得效率更高,例如:

如果只设头指针,那么在表尾插入元素的时间复杂度为O(n),但如果设的是尾指针,在表头表尾插入元素都只要O(n)个元素

2.**循环双链表**



如何判断*p是尾结点:p->next=L;

如何判断循环双链表为空表:头结点的prior域和next域都等于L

3.5静态链表

静态链表:利用数组来描述线性表的链式存储结构,结点也有data(数据域)和next(指针域)

指针式结点的相对地址(数组下标),和顺序表一样,静态链表也要预先分配一块连续的内存空间

数组下标	顺序表	内存地址
0	a_1	LOC(A)
1	a_2	LOC(A)+sizeof(ElemType)
	⋮	
i-1	a_i	LOC(A)+(i-1)×sizeof(ElemType)
	⋮	
n-1	a_n	LOC(A)+(n-1)×sizeof(ElemType)
	⋮	
MaxSize-1	⋮	LOC(A)+(MaxSize-1)×sizeof(ElemType)

```
#define MaxSize 50//静态链表的最大长度
typedef struct{//静态链表结构类型的定义
    ElemType data;//存储数据元素
    int next;//下一个元素的数组下标
}SLinkList[MaxSize];
```

静态链表以next==-1作为其结束的标志

4.例题

1,关于线性表的顺序存储结构和链式存储结构的描述中,正确的是()

1.线性表的顺序存储结构优于其链式存储结构

2.链式存储结构比顺序存储结构能更方便地表示各种逻辑结构

3.若频繁使用插入和删除结点操作,则顺序存储结构更优于链式存储结构

4.顺序存储结构和链式存储都可以进行顺序存取

A.1,2,3 B.2,4 C.2,3 D.3,4

答:B,1.存储结构有好有坏

2.链式结构中可以用指针表示逻辑结构,顺序结构只能用物理的邻接关系来表示逻辑结构

3.插入和删除链式结构更优,因为顺序结构插入和删除需要移动大量元素

4.顺序存储结构既能随机存取又能顺序存取,而链式结构只能顺序存取

2.下列关于线性表说法中,正确的是()

1.顺序存储方式只能用于存储线性结构

2.取线性表中的第i个元素的时间与i的大小有关

3.静态链表需要分配较大的连续空间,插入和删除不需要移动元素

4.在一个长度为n的有序单链表中插入一个新结点并保持有序的时间复杂度为O(n)

5.若用单链表来表示队列,则应该选用带尾指针的循环链表

A.1,2 B.1,3,4,5 C.4,5 D.3,4,5

答:D,

1.错,顺序存储方式也能同样适合图和树

2.错,当在顺序存储结构中的线性表随机存取元素,取得元素的时间复杂度为O(1)

3.正确

4.正确,要插入一个新结点并保持有序需要先遍历找到相应大小对应的位置,因此时间复杂度为O(n)

5.正确,队列需要在表头删除元素和表尾插入元素,因此仅带尾指针的循环链表会更加方便插入和删除的时间复杂度为O(n)

3.在一个单链表中,已知q所指结点是p所指结点的前驱结点,若在q和p之间插入点s,则该如何执行?

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

4.给定有n个元素的一维数组,建立一个有序单链表的最低时间复杂度是()

答:先建立好链表(O(n))再对数组进行排序(O(nlogn)

5.将长度为n的单链表链接在长度为m的单链表后面,其算法的时间复杂度为?

答:O(m)

6.在一个长度为n的带头结点的单链表h上,设有尾指针r,则执行()操作与链表的表长有关

A.删除单链表的第一个元素

B.删除单链表的最后一个元素

C.在单链表第一个元素前插入一个新元素

D.在单链表最后一个元素后插入一个新元素

答:B,删除单链表的最后一个元素需要将前置结点的指针域设为空,要找到前置结点需要从头开始依次遍历,因此与链表的长度有关

7.对于一个头指针为head的带头结点的单链表,判定该表为空表的条件是(),对于不带头结点的单链表,判定空表的条件是()

```
head->next==NULL;
head==NULL
```

8.在长度为n的有序单链表中插入一个新结点,并仍然保持有序的时间复杂度是()

答:O(n),线性表需要遍历寻找到正确的位置插入,因此时间复杂度为O(n)

9.已知一个带有表头结构的双向循环链表L,其中prev和next分别是指向其直接前驱和直接后继结点的指针,先要删除指针p所指的结点,正确的语句操作是()

```
p->next->prev=p->prev;
p->prev->next=p->next;
free(p);
```

10.设计一个递归算法,删除不带头结点的单链表L中所有值

```
void Del_X_3(LinkList &L,ElemType x){
    LNOde* p;
    if(L==NULL) return;
    if(L->data==x){
        P=L;
        L=L->next;
        free(p);
        Del_X_3(L,x);
    }else{
          Del_X_3(L->next,x);
    }
}
```

11.在带头结点的单链表L中,删除所有值为x的 结点,并释放空间,假设值为x的结点不唯一

```
void Del_X_1(LinkList &L,ElemType e){
    LNode* p=L->next,*pre=L,*q;//p表示当前指针位置,pre表示前驱的指针,q用于记录当前指针
    while(p!=NULL){
        if(p->data==x){
            q=p;//让q记录p当前的位置
            p=p->next;//更新p的位置
            pre->next=p;//更新前驱指向的结点
            free(q);
        }else{
            pre=p;//更新前驱结点
            p=p->next;//向后遍历线性表
        }
    }
}
```

12.试编写在带头结点的单链表L中删除一个最小值结点的高效算法(假设最小值结点是唯一的)

```
LinkList DeleteMin(LinkList &L){
    LinkList p=L->next,pre=L,minPre=pre,min=p;
    while(p!=NULL){
        if(p->data<min->data){
            min=p;
            minPre=pre;
        }
```

```
        pre=p;
        p=p->next;
    }
    minPre->next=min->next;
    free(min);
    return L;
}
```

13.试编写算法将带头结点的单链表就地(指辅助空间复杂度为O(1))逆置

```
//将头结点摘下,再按头插法顺序插入
LinkList Reverse(LinkList &L){
    LinkList p=L->next,r;//p表示当前指针,r表示后继
    L->next=NULL;//将链表的头结点断开
    while(p!=NULL){
        r=p->next;
        p->next=L->next;//插入操作
        L->next=p;
        p=r;//更新结点
    }
    return L;
}
```

14.给定两个单链表,编写算法找出两个链表的公共结点

```
//当两个链表有公共结点时,意味着链表的最后一个元素是相等的,因此我们只需要分别遍历两个单链表记录下最后一个结点,比较是否相等即可
//遍历
LinkList Ergodic(LinkList L){
    LinkList p=L->next,pre=L;
    while(p!=NULL){
        pre=p;
        p=p->next;
    }
    return pre;
}

bool SearchCommon(LinkList L1,LinkList L2){
    LinkList p1=Ergodic(L1);
    LinkList p2=Ergodic(L2);
    if(p1==p2){
        return true;
    }
    return false;

}
```