第1章 绪论

第1.1节 基本概念

数据

数据是描述客观事物的数值、字符以及所有能输入到计算机中并被计算机程序处理的符号的集合。数据可分为两类：数值数据，包括整数、实数和复数等，另一类是非数值数据，主要包括字符、图形、语音等。

数据元素

数据元素是数据处理的基本单位。是数据集合中的一个个体，在计算机中通常作为一个整体进行考虑和处理。数据元素也称为元素、结点和记录。

数据项

数据元素是由若干个数据项组成，数据项是数据处理的最小单位，也称为数据域。

数据结构

数据结构是指相互之间存在着某种特定关系的数据元素的集合，描述了按照一定关系组织起来的待处理数据元素的表示及相关操作，涉及数据的逻辑结构、存储结构和运算。

姓名	性别	年龄	专业
刘小平	男	21	计算机
王红	女	20	数学
吕军	男	20	经济
⋮	⋮	⋮	⋮
马文华	女	19	管理

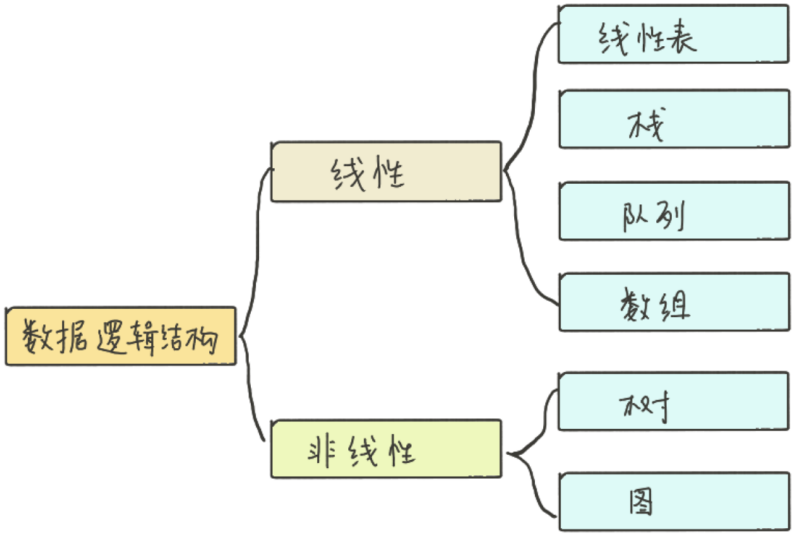
数据元素

数据项

第1.2节 逻辑结构与存储结构

逻辑结构

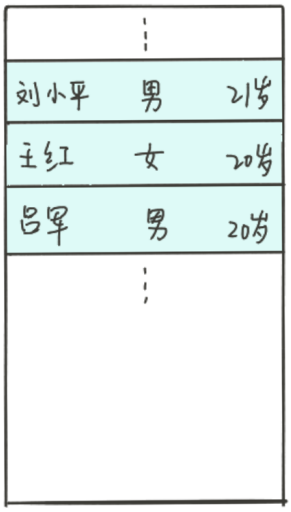
数据的逻辑结构是对数据元素之间逻辑关系的描述，数据逻辑结构的种类如下：



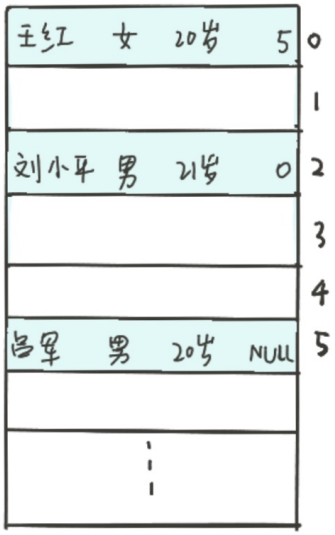
存储结构

数据的存储结构是数据结构在计算机中的表示方法，是数据的逻辑结构到计算机存储器的映像。有以下四种基本存储结构：

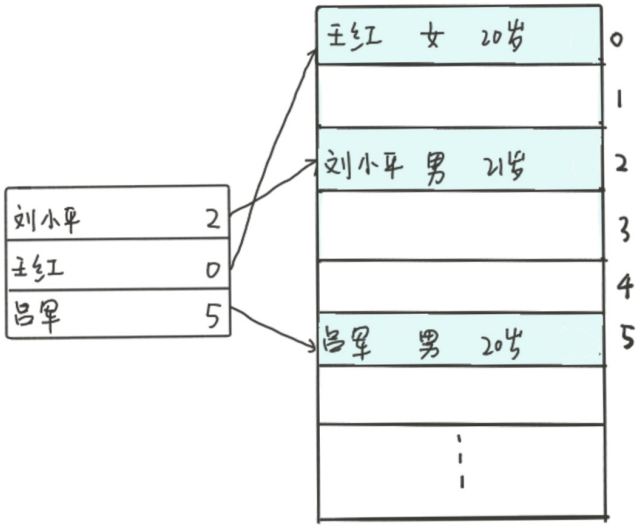
- 顺序存储结构：利用数据元素在存储器中的相对位置来表示数据元素之间的逻辑关系。数据元素必须利用连续的存储空间存放。主要优点是空间利用率高，可随机存取，缺点是不便于插入删除元素。（随机存取指存取的时间与元素位置无关）



- 链式存储结构：在数据元素上附加指针域，并借助指针来指示数据元素之间的逻辑关系。数据元素可以离散地存放，不需要分配连续的存储空间。主要优点是便于插入删除，缺点是不能随机存取。



- 索引存储结构：为数据元素建立索引表，索引表项为：（关键字，地址）。由关键字可立即通过地址找到该元素。数据元素可以离散存放，索引表需要采用顺序存储。优点是可随机存取，通过关键字查找速度快，缺点是增加了索引表，降低了存储空间利用率。



- 哈希存储结构：将关键字通过哈希函数计算出数据元素的存储地址。优点是查找插入速度快，但只能存储数据元素，不能保存数据元素之间的逻辑关系。

逻辑结构与存储结构的关系

同一种逻辑结构可以采用不同的存储结构，要根据需求确定存储结构，比如线性表要求随机存取可采用顺序存取结构，需要频繁进行插入删除可采用链式存储结构，需要按关键字快速查询时可采用索引存储结构。

第1.3节 算法分析

空间复杂度

程序运行从开始到结束所需的存储量。

时间复杂度

程序运行从开始到结束所需要的时间，用语句执行次数之和的数量级表示。

例1

```
for(i=0;i<n;i++)//n+1
    for(j=0;j<n;j++)//n(n+1)
    {
        c[i][j] = 0;//n^2
        for(k=0;k<n;k++)//n^2(n+1)
            c[i][j] = c[i][j] + a[i][k]*b[k][j];//n^3
    }
```

$T(n) = n+1+n(n+1)+n^2+n^2(n+1)+n^3 = 2n^3+3n^2+2n+1$，$T(n)=O(n^3)$。

例2

```
for(i=1;i<=n;i++)
    for(j=1;j<=i;j++)
        for(k=1;k<=j;k++)
            x++;
```

$$\sum_{i=1}^{n}\sum_{j=1}^{i}\sum_{k=1}^{j}1 = \sum_{i=1}^{n}\sum_{j=1}^{i}j = \sum_{i=1}^{n}(1+2+3+\cdots+i)$$

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

$$= \frac{1}{2}\cdot\frac{1}{6}n(n+1)(2n+1) + \frac{1}{2}\cdot\frac{1}{2}n(n+1)$$

$$= O(n^3)$$

例3

```
int i = 1;
while(i<=n)
    i = i*2;
```

假设循环了k次，那么$2^k <= n$，则k<=logn，时间复杂度为T(n)=O(logn)。

第2章 线性结构

第2.1节 线性表

线性表的逻辑结构

定义

具有相同数据类型的n(n>=0)个数据元素的有限序列。记作

$$(a_1, a_2, \ldots, a_{i-1}, a_i, a_{i+1}, \ldots, a_n)$$

a_i是表中数据元素， n 是表长度。

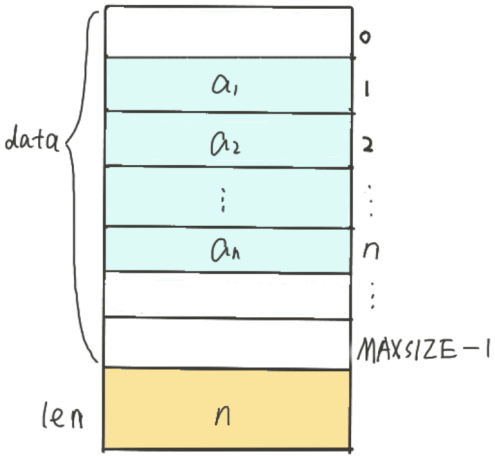
特点

- 除第一个元素外，其它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，其它每一个元素有一个且仅有一个直接后继。

线性表的顺序存储结构——顺序表

线性表的顺序存储是指在内存中用地址连续的一块存储空间对线性表中的各元素按其逻辑顺序依次存放，用这种存储形式存储的线性表称为顺序表。设a_1的存储地址为Loc(a_1)，每个数据元素占d个存储位置，则第i个元素的地址为：Loc(a_i)=Loc(a_1)+(i-1)*d

顺序表示意图：



顺序表的类型定义

```
typedef struct
{
    datatype data[MAXSIZE];//存储顺序表中的元素
    int len;//顺序表的表长
}SeqList;//顺序表类型
```

- MAXSIZE是足够大的整数，代表最大表长
- datatype为元素类型，实际中可以为int、float、char等
- data数组从下标为1存放元素
- len为顺序表的表长

关于顺序表的基本运算

顺序表初始化：构造一个空表，给顺序表分配存储空间，并设表长len=0

```
SeqList *Init_SeqList()
{
    SeqList *L;//定义顺序表指针变量
    L = (SeqList*)malloc(sizeof(SeqList));//给顺序表分配存储空间，sizeof函数返回类型所占的字节数
    L->len = 0;//表长设置为0
    return L;//返回顺序表指针
}
```

建立顺序表：输入顺序表的长度n和n个元素即可建立顺序表

```
void CreatList(SeqList *L)
{
    int i,n;
    printf("输入顺序表的长度：");
    scanf("%d",&n);//输入顺序表的表长n
    printf("输入顺序表元素");
    for(i=1;i<=n;i++)
        scanf("%d",&(L->data[i]));//依次输入n个元素
    L->len = n;//设置表长
}
```

插入运算：在顺序表的第i个位置插入新元素x

```
void Insert_SeqList(SeqList *L, int i, datatype x)
{
    int j;
    if(L->len == MAXSIZE)//判断表满
        printf("表满！");
    else
        if(i<1 || i>L->len+1)//判断插入位置非法
            printf("插入位置非法！");
        else
        {
            for(j=L->len;j>=i;j--)//将a[n]到a[i]顺序后移一个位置 ，使第i个位置空出来
                L->data[j+1] = L->data[j];
            L->data[i] = x;//新元素放入第i个位置
            L->len++;//表长加1
        }
}
```

插入运算时间消耗主要在移动元素上，时间复杂度分析过程：

- 可插入位置从1到n+1共有n+1个位置
- 在第i个位置上插入新元素时需要移动的元素个数为：n-(i-1)=n-i+1
- 设每个位置上插入元素的概率相同，则第i个位置插入的概率为p_i=1/(n+1)
- 元素的平均移动次数为：

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

- 时间复杂度为O(n)

删除运算：将顺序表中第i个元素删去

```
void Delete_SeqList(SeqList *L, int i)
{
    int j;
```

```
    if(L->len == 0)//判断表为空
        printf("表为空! ");
    else
        if(i<1 || i>L->len)//判断删除位置非法
            printf("删除位置非法! ");
        else
        {
            for(j=i+1;j<=L->len;j++)//将a[i+1]到a[n]向前移一位，a[i]就被覆盖掉了
                L->data[j-1] = L->data[j];
            L->len--;//表长减一
         }
}
```

删除运算时间消耗主要也是在移动元素上，时间复杂度分析过程：

- 可删除的位置从1到n共有n个
- 删除第i个元素时需要移动的元素的个数为：n-i
- 设每个位置上的元素被删除的概率相同，则第删除第i个位置的元素概率为$p_i=1/n$
- 元素的平均移动次数为：

$$E=\sum_{i=1}^{n}p_i(n-i)=\frac{1}{n}\sum_{i=1}^{n}(n-i)=\frac{n-1}{2}$$

- 时间复杂度为O(n)

查找运算：查找与给定值x相等的元素，找到就返回下标，否则返回0

```
int Location_SeqList(SeqList *L, datatype x)
{
    int i=1;//从第一个元素开始查找
    while(i<L->len && L->data[i]!=x)//顺序表未查找完且当前元素不是要找的元素
        i++;
    if(L->data[i]==x)//找到返回其下标
        return i;
    else
        return 0;//未找到返回0
}
```

查找运算时间消耗主要在比较元素上，时间复杂度分析过程：

- 比较的次数最少为1，最多为n，共有n种可能，设每种可能等概率$p_i=1/n$
- 在第i个位置查找到元素时，需要比较的次数为：i
- 平均比较的次数为：

$$E=\sum_{i=1}^{n}p_i i=\frac{1}{n}\sum_{i=1}^{n}i=\frac{n+1}{2}$$

- 时间复杂度为O(n)

顺序表的应用

例1：已知顺序表A的长度为n，写出将该顺序表逆置的算法

思路：第一个元素与倒数第一个元素交换，第二个元素与倒数第二个元素交换……

```
void Coverts(SeqList *L)
{
    int i, n;
    datatype tmp;//临时变量
    n = L->len;
    for(i=1;i<=n/2;i++)//只用遍历前一半元素，后一半自然就交换了
    {
        tmp = L->data[i];
        L->data[i] = L->data[n-i+1];
        L->data[n-i+1] = tmp;
    }
}
```

例2:顺序表A和B中元素均按由小到大顺序排列，写一个算法将它们合并为一个顺序表C，C中元素也按从小到大排列

思路：从两个顺序表起始位置开始比较，将较小的元素放入C中，后移一位继续比较，直到有一个顺序表已空，将另一个表中剩余元素直接复制到C中

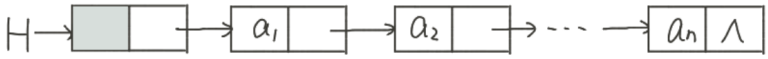
```
void Merge(SeqList *A, SeqList *B, SeqList *C)
{
    int i,j,k;
    i=j=k=1;
    if(A->len+B->len>=MAXSIZE)//合并之后元素个数不能超出最大范围
        printf("元素个数超出范围！");
    else
    {
        while(i<=A->len && j<=B->len)//将A和B中元素逐一比较，直到有一个顺序表已空
            if(A->data[i]<B->data[j])
                C->data[k++] = A->data[i++];
            else
                C->data[k++] = B->data[j++];
        if(j<=B->len)//B中有剩余元素，即A先遍历完，就将指针A指向B
        {
            A = B;
            i = j;
        }
        while(i<=A->len)//将剩下的元素直接复制到C中
            C->data[k++] = A->data[i++];
        C->len = k-1;//设置表长
    }
}
```

线性表的链式存储结构——链表

链表采用连续或不连续的存储单元来存储线性表中的元素，此时元素的先后关系不能用物理位置上的邻接关系来表示，需要用“指针”将元素“串”起来，每个元素除了存储自身的信息外还需保存直接前驱元素或直接后继元素的存储位置，我们称这种元素为结点，结点中存放数据信息的部分称为数据域，存放指向前驱结点或后继结点地址的部分称为指针域。

单链表

每个结点中只有一个指向后继结点的指针，头结点中数据域不存放数据，示意图如下：



(a)带头结点的单链表

(b)带头结点的空单链表

结点类型定义

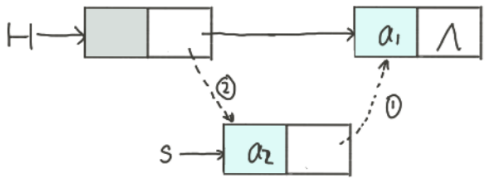
```
typedef struct node
{
    datatype data;//数据域，保存元素内容
    struct node *next;//指针域，保存下一个元素地址
}LNode;//单链表结点类型
```

关于单链表的基本运算

单链表初始化：创建一个带头结点的空单链表


```
LNode *Create_head( )
{
    LNode *head;//头指针，指向头结点的指针
    head = (LNode *)malloc(sizeof(LNode));//生成头结点
    head->next = NULL;//指针域为空
    return head;
}
```

头插法建立单链表：将新结点插入头结点之后，输入元素的顺序和输出顺序相反



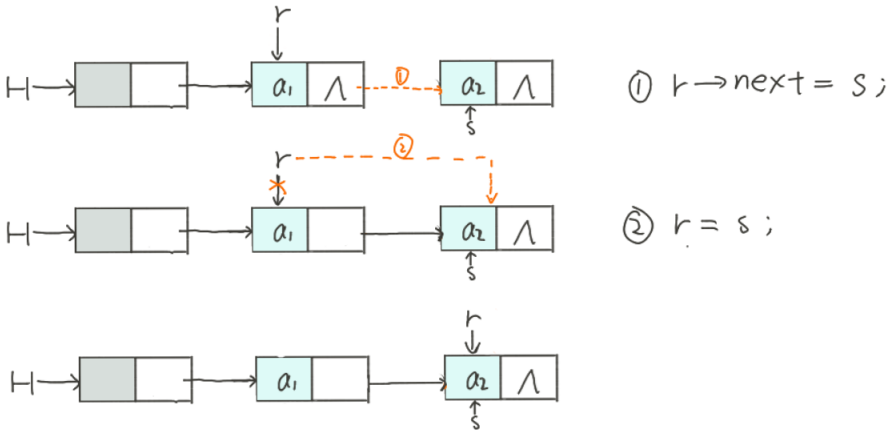
① s→next = H→next;

② H→next = s;

```
LNode *Create_LinkList1()
{
    LNode *head, *s;
    int x;
    head = Create_head(); //创建头结点
    scanf("%d",&x);
    while(x!=-1)//-1为结束标志
    {
        s = (LNode*)malloc(sizeof(LNode));//为新结点申请空间
        s->data = x;//新结点数据域赋值
```

```
        s->next = head->next;//将头结点后面的结点链接到新结点之后
        head->next = s;//插入头结点之后
        scanf("%d",&x);
    }
    return head;
}
```

尾插法建立单链表：在单链表尾结点后插入新结点，输入元素顺序和输出元素顺序相同




```
LNode *Create_LinkList2()
{
    LNode *head, *r, *s;
    int x;
    head = Create_head(); //创建头结点
    r=head;//尾指针指最后一个结点即头结点
    scanf("%d",&x);
    while(x!=-1)//-1为结束标志
    {
        s = (LNode*)malloc(sizeof(LNode));//为新结点申请空间
        s->data = x;//新结点数据域赋值
        s->next = NULL; //新结点要插入尾部，所以指针域为空
        r->next = s;
        r = s;
        scanf("%d",&x);
    }
    return head;
}
```

求带头结点单链表表长：从第二个结点开始向后遍历，每遍历一个结点计数器加1，直到指针域为NULL

```
int Length_LinkList(LNode *head)
{
    LNode *p=head->next;//从头结点的直接后继结点开始
    int i=0;
    while(p!=NULL)//当未到链尾时继续遍历
    {
        i++;
        p=p->next;
    }
    return i;
}
```

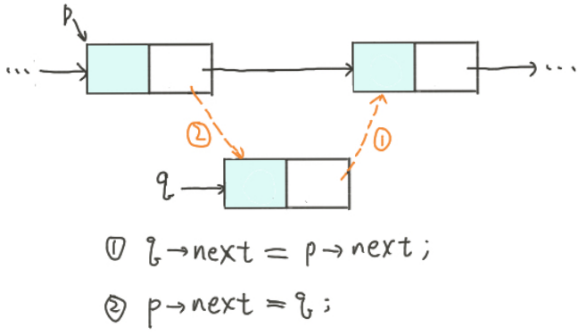
按序号查找：找到第i个数据结点并返回其指针，没找到则返回空值

```
LNode *Get_LinkList(LNode *head, int i)
{
    LNode *p=head;//由第一个结点开始查找，i==0则返回头结点指针
    int j=0;
    while(p!=NULL && j<i)//当未找到链尾且没到第i个时继续查找
    {
        p=p->next;
        j++;
    }
    return p;//找到则返回指向i结点的指针，找不到则p已为空返回空值
}
```

按值查找：找到值等于x的结点，没找到则返回空值

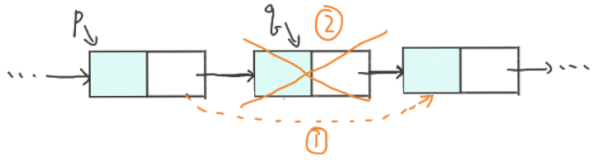
```
LNode *Locate_LinkList(LNode *head, int x)
{
    LNode *p=head->next;//从第一个数据结点开始查找
    while(p!=NULL && p->data!=x)//当未查到链尾且当前结点不等于x时继续查找
        p=p->next;
    return p;//找到则返回指向值为x的结点的指针值，找不到则p已为空返回空值
}
```

插入结点：在单链表第i个位置上插入值为x的元素




```
void Insert_LinkList(LNode *head, int i, datatype x)
{
    LNode *p,*q;
    p=Get_LinkList(head, i-1);//查找第i-1个元素
    if(p==NULL)
        printf("插入位置非法！"); //第i-1个位置不存在
    else
    {
        q=(LNode *)malloc(sizeof(LNode));//申请结点空间
        q->data=x;
        q->next=p->next;//先连
        p->next=q;//后断
    }
}
```

删除结点：删除单链表上第i个数据结点



① p→next = q→next;

② free(q);

```
void Del_LinkList(LNode *head, int i)
{
    LNode *p,*q;
    p=Get_LinkList(head, i-1);//查找第i-1个元素
    if(p==NULL)
        printf("第i-1个结点不存在！");//待删除结点的前一个结点不存在
    else
        if(p->next==NULL)
            printf("第i个结点不存在！");//待删除结点不存在
        else
        {
            q=p->next;
            p->next=q->next;
            free(q);//回收第i个结点的空间
        }
}
```

单链表应用

例1：将带头单链表H逆置

思路：头插法创建单链表时，输入顺序和结点序列顺序相反，可将数据结点采用头插法依次插入头结点后实现链表逆置

```
void Convert(LNode *H)
{
    LNode *p,*q;
    p=H->next;//p指向剩余结点链表的第一个数据结点
    H->next=NULL;//新链表初始为空
    while(p!=NULL)
    {
        q=p;//从剩余结点中取第一个结点
        p=p->next;//p继续指向剩余结点的第一个数据结点
        q->next=H->next;//头插法将q插入到新链表H中
        H->next=q;
    }
}
```

例2：将两个递增有序的单链表A和B合并为一个递减有序的单链表C，不允许增加新结点

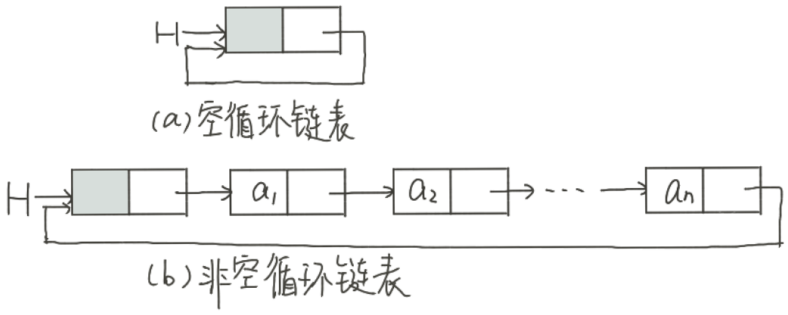
思路：将两链表元素逐一比较，采用头插法将值较小的结点插入头结点后

```
void Merge(LNode *A,LNode *B,LNode *C)
{
    LNode *p,*q,*s;
    p=A->next;//p指向链表A的第一个数据结点
    q=B->next;//q指向链表B的第一个数据结点
    C=A;//使用链表A的头结点当链表C的头结点
```

```
    C->next=NULL;//切断头结点和后面的数据结点
    free(B);//回收链表B的头结点
    while(p!=NULL && q!=NULL)//将链表A,B中当前比较结点中值小者赋给*s
    {
        if(p->data<q->data)
        {
            s=p;
            p=p->next;
        }
        else
        {
            s=q;
            q=q->next;
        }
        s->next=C->next;//头插法将结点*s查到链表C中
        C->next=s;
    }
    if(p==NULL)
        p=q;
    while(p!=NULL)//将剩余结点依次插入链表C中
    {
        s=p;
        p=p->next;
        s->next=C->next;
        C->next=s;
    }
}
```

循环链表

将单链表最后一个结点的指针值由空改为指向单链表的头结点，形成一个环，特点是从任一结点出发都能找到其他结点



带头循环单链表的操作与带头单链表相同，将原来判断指针是否为NULL改为判断是否为头指针

例如带头循环单链表的查找算法实现如下：

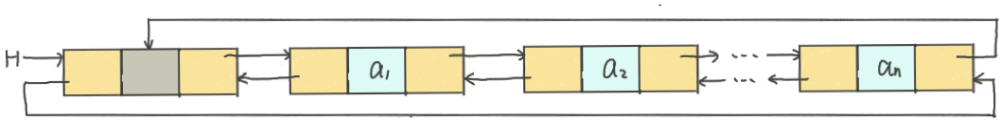
```
LNode *Locate_CycLinkList(LNode *head, int x)
{
    LNode *p=head->next;//从第一个数据结点开始查找
    while(p!=head && p->data!=x)//当未查完循环链表且当前结点不等于x时继续查找
        p=p->next;
    if(p!=head)
        return p;//找到值为x的结点，返回其指针值
    else
        return NULL;//说明p又循环到了头结点，即没查到，返回空值
}
```

双向链表

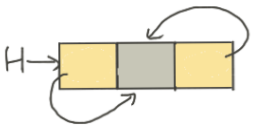
循环单链表只能顺着指针方向向后寻找其他结点，而无法直接到达该结点的前驱结点，因此可在每个结点中设置两个指针域，一个指向直接前驱结点，一个指向直接后继结点，便可以沿两个方向遍历链表

结点示意图：

带头结点的双向循环链表示意图：



(a) 非空带头双向循环链表



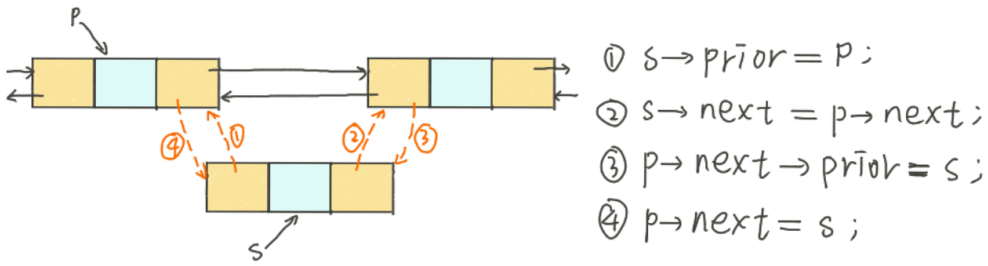
(b) 空带头双向循环链表

双向链表结点定义

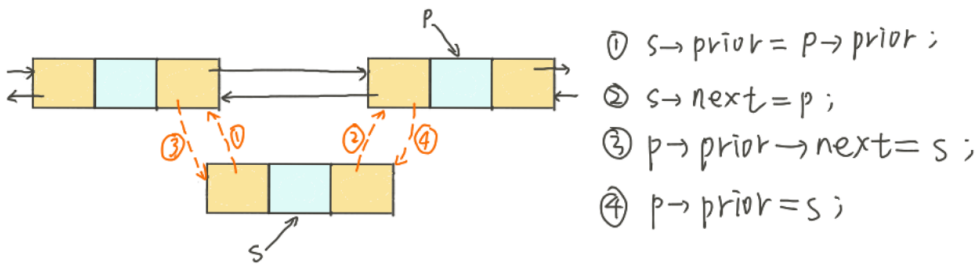
```
typedef struct dlnode
{
    datatype data;//数据域
    struct dlnode *prior,*next;//prior和next分别为指向直接前驱和直接后继结点的指针
}DLNode;//双向链表结点类型
```

双向链表的运算

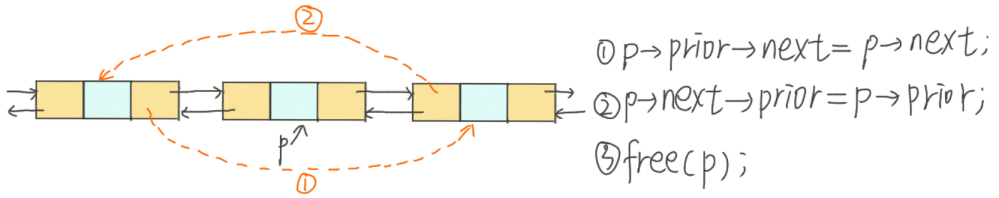
在结点*p之后插入结点*s



在结点*p之前插入结点*s



删除结点*p



静态链表

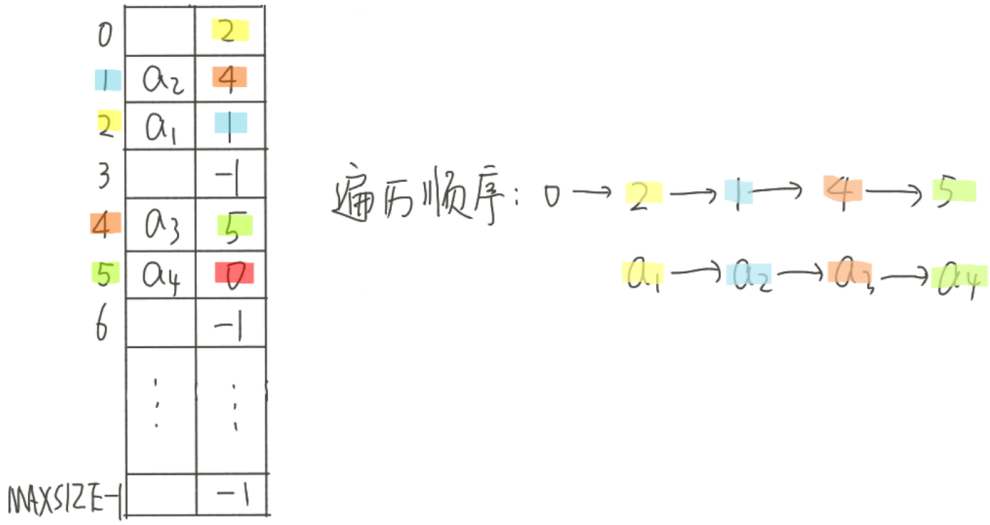
有的高级语言没有指针数据类型，因此可以用一维数组模拟链表结构，成为静态链表。

数组中每一个元素是一个结构体，结构体中包含一个数据域用来存放数据，还包含一个下标域用来指示直接后继结点在数组中存放的下标。

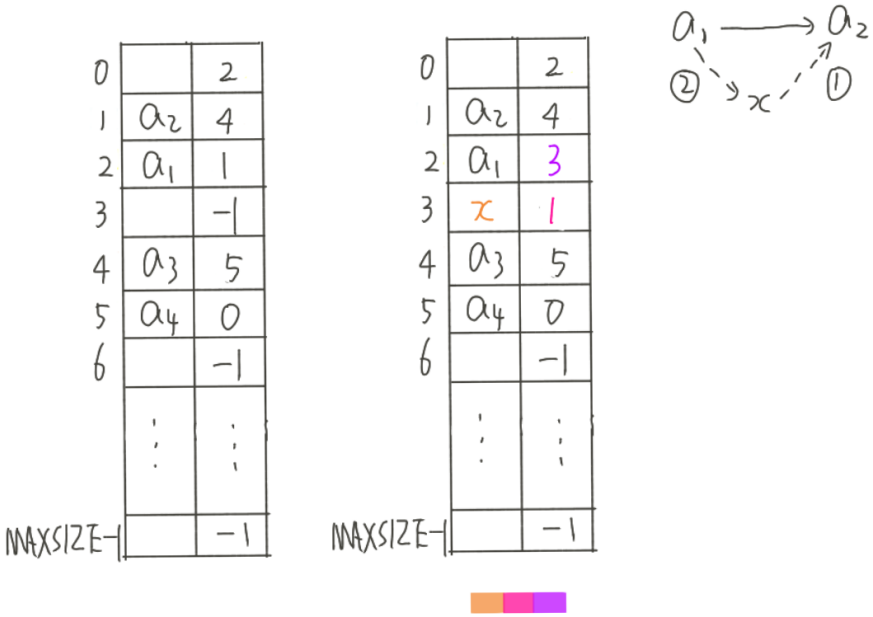
数组中0号元素的数据域为空，下标域存放第一个数据结点存放的下标，最后一个结点的下标域为0，标记链表结尾。

下标域为-1说明此结点还未使用。

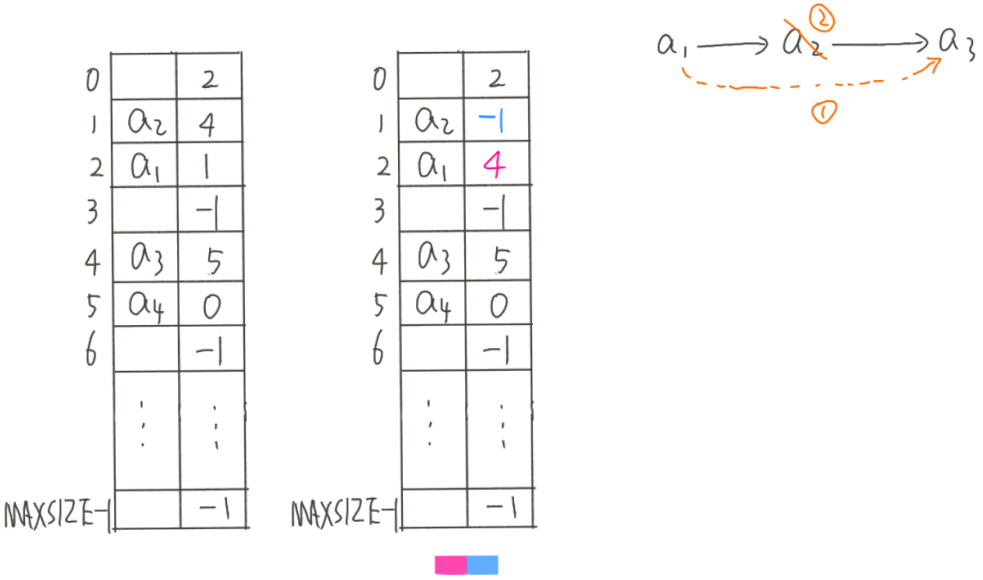
静态链表示意图与遍历：



在第二个位置上（即a_1之后）插入一个值为x的结点：



删除a_2结点：



第2.2节 栈

第2.3节 队列

第2.3节 数组与广义表