二、线性表

2.1线性表的定义

线性表是具有**相同数据类型**的n(n>0)个数据元素的**有限序列**，其中n为表长，当n=0时线性表是一个**空表**。

2.2顺序表的定义

2.2.1. 顺序表的基本概念

线性表的顺序存储又称 **顺序表** 。它是用一组地址连续的存储单元依次存储线性表中的数据元素，从而使得**逻辑上相邻的两个元素在物理上也相邻。**顺序表的特点是 **表中元素的逻辑顺序与其物理顺序相同。**

特点：

- **随机访问**，即可以在 O(1)时间内找到第 i 个元素。

> 顺序表最大的特点是随机访问，可通过首地址和元素序号在O(1)的时间复杂度内找到指定的元素，因为顺序表是连续存放的。

- 存储密度高，每个节点只存储数据元素。
- 拓展容量不方便（即使使用动态分配的方式实现，**拓展长度的时间复杂度也比较高**，因为需要把数据复制到新的区域）。
- 插入删除操作不方便，需移动大量元素：O(n)

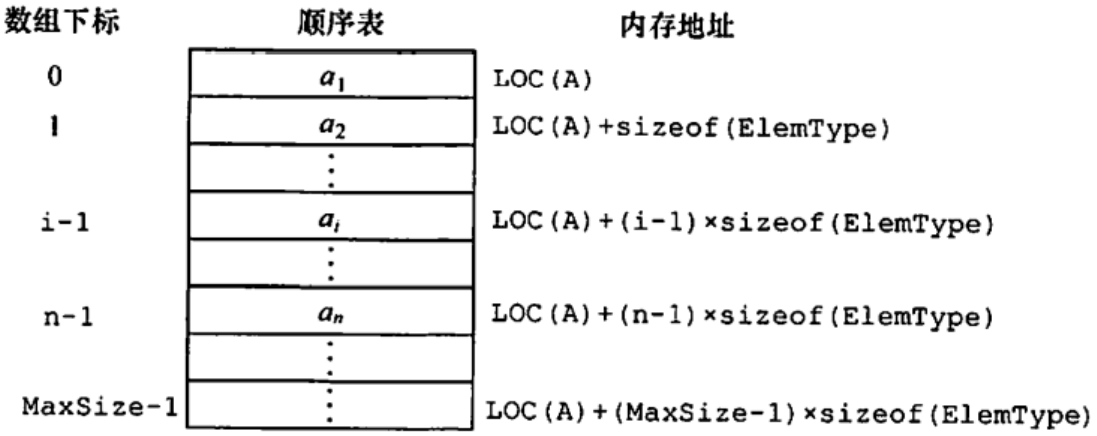


图 2.1　线性表的顺序存储结构

2.2.2. 顺序表的实现

静态实现：

```
// 顺序表实现（静态分配）
#define MaxSize 10

#include <stdio.h>
#include <stdlib.h>

typedef struct {
    int data[MaxSize];
```

```
    int length;
} SqList;

void InitList(SqList &L) {
    L.length = 0;
}

int main() {
    SqList L;
    InitList(L);
    for (int i = 0; i < MaxSize; i++) {
        printf("data[%d]=%d\n", i, L.data[i]);
    }
    printf("%d", L.length);
    return 0;
}
```

动态实现：

```
//顺序表实现（动态分配）
#define InitSize 10     // 初始化顺序表的长度

#include <stdio.h>
#include <stdlib.h>

typedef struct {
    int *data;      // 声明动态分配数组的指针
    int MaxSize;    // 最大长度
    int length;     // 当前长度
} SeqList;

// 初始化顺序表
void InitList(SeqList &L) {
    // 用malloc函数申请一片连续的存储空间
    L.data = (int *) malloc(sizeof(int) * InitSize);
    L.length = 0;
    L.MaxSize = InitSize;
}

// 增加动态数组的长度，本质上是将数据从旧的区域复制到新区域
void IncreaseSize(SeqList &L, int len) {
    // 使p指针和data指向同一目标
    int *p = L.data;
    L.data = (int *) malloc(sizeof(int) * (L.MaxSize + len));  // 申请一块新的连续空间用于存放新表，并将data指针指向新区域
    for (int i = 0; i < L.length; i++) {
        L.data[i] = p[i];  //将数据复制到新区域
    }
    L.MaxSize += len;
    free(p);  //释放原区域的内存空间
}

// 打印顺序表
void printList(SeqList L) {
    for (int i = 0; i < L.length; i++) {
```

```
        printf("%d, ", L.data[i]);
    }
}

int main() {
    SeqList L;
    InitList(L);
    printf("增加前顺序表的长度：%d \n", L.MaxSize);
    printList(L);
    IncreaseSize(L, 5);
    printf("增加后顺序表的长度：%d \n", L.MaxSize);
    return 0;
}
```

> malloc() 函数的作用：会申请一片存储空间，并返回存储空间第一个位置的地址，也就是该位置的指针。

2.2.3. 顺序表的基本操作

插入

```
// 将元素e插入到顺序表L的第i个位置
bool ListInsert(SqList& L, int i, int e) {
    if (i < 1 || i > L.length + 1)
        return false;
    if (L.length >= MaxSize)
        return false;
    for (int j = L.length; j >= i; j--) {  // 将第i个元素及之后的元素后移
        L.data[j] = L.data[j - 1];
    }
    L.data[i - 1] = e;  // 在位置i处插入元素e
    L.length++;
    return true;
}
```

> 最好时间复杂度：O(1)（插入在表尾）
>
> 最坏时间复杂度：O(n)（插入在表头）
>
> 平均时间复杂度：O(n)

删除

```
bool ListDelete(SqList& L, int i, int& e) {
    if (i < 1 || i > L.length) {  //判断i的范围是否有效
        return false;
    }
    e = L.data[i - 1];
    for (int j = i; j < L.length; j++) {  //将第i个位置后的元素前移
        L.data[j - 1] = L.data[j];
    }
    L.length--;
    return true;
}
```

> 最好时间复杂度：O(1)（删除表尾元素）
>
> 最坏时间复杂度：O(n)（删除表头元素）
>
> 平均时间复杂度：O(n)

查找

- 按位查找

```
// 按位查找
int getElemByLoc(SqList L, int i) {
    return L.data[i - 1];
}
```

> 时间复杂度：O(1)

因为顺序表是连续存放的，故可以在O(1)的时间复杂度内通过下标找到元素。

- 按值查找

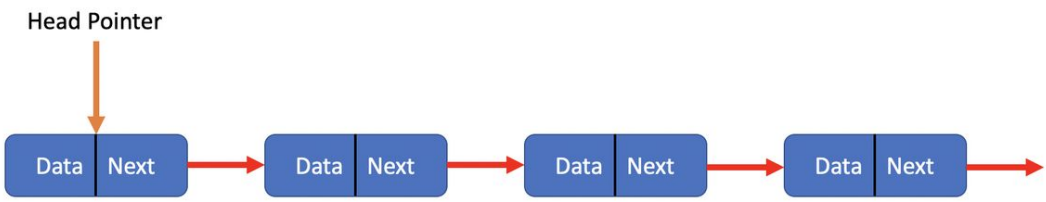
```
int getElemByValue(SqList L, int value) {
    for (int i = 0; i < L.length; i++) {
        if (L.data[i] == value) {
            return i + 1;
        }
    }
    return 0;
}
```

> 注意：在数据结构初试中，手写代码可以直接使用“==”，无论Elemtype是基本数据类型还是结构体类型。

> 最好时间复杂度：O(1)
>
> 最坏时间复杂度：O(n)
>
> 平均时间复杂度：O(n)

2.3 链表

2.3.1 单链表



1. 单链表：用**链式存储**实现了线性结构。一个结点存储一个数据元素，各结点间的前后关系用一个**指针**表示。
2. 特点：

1. **优点：不要求大片连续空间，改变容量方便。插入和删除操作不需要移动大量元素**
2. **缺点：不可随机存取，要耗费一定空间存放指针。**

3. 两种实现方式：
    1. 带头结点，写代码更方便。头结点不存储数据，头结点指向的下一个结点才存放实际数据。(L = NULL;)
    2. 不带头结点，麻烦。对第一个数据结点与后续数据结点的处理需要用不同的代码逻辑，对空表和非空表的处理需要用不同的代码逻辑。(L->next = NULL)

单链表中结点类型的描述如下：

```
typedef struct LNode {
    int data;
    struct LNode* next;  // 由于指针域中的指针要指向的也是一个节点，因此要声明为 LNode 类型
} LNode, *LinkList;      //这里的*LinkList强调元素是一个单链表.LNode强调元素是一个节点。本质上是同一个结构体
```

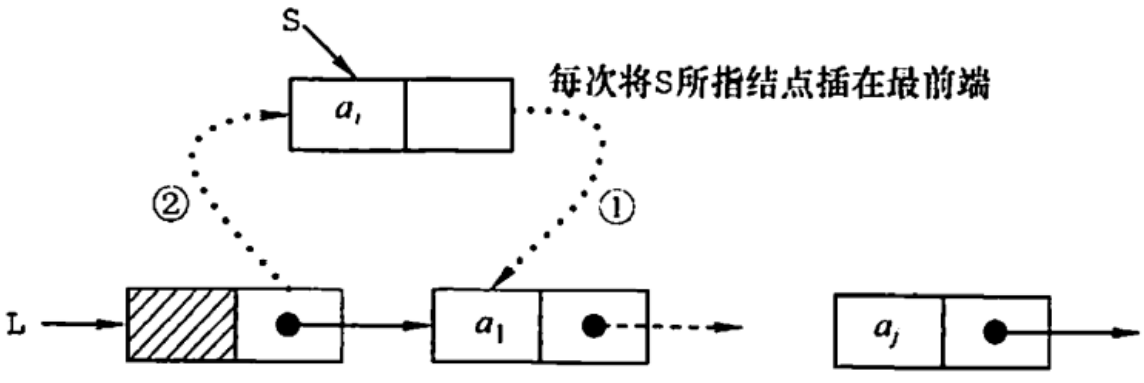
单链表的元素离散分布在各个存储空间中，是非随机存取的存储结构，不能直接找到表中每个特定的结点。查找结点时，需要从头往后依次遍历。

2.3.2 单链表的基本操作

1、单链表的初始化

```
// 不带头结点
bool InitList(LinkList &L) {
    L = NULL;   // 空表，不含任何结点
    return true;
}

// 带头结点
bool InitListWithHeadNode(LinkList &L) {
    L = (LNode *) malloc(sizeof(LNode)); // 分配一个结点
    if (L == NULL) {    // 内存不足，分配失败
        return false;
    }
    L->next = NULL; // 单链表后面还没有结点
    return true;
}
```

2、头插法建立单链表



> 头插法的一个重要应用：单链表的逆置

```
// 头插法建立单链表
LinkList List_HeadInsert(LinkList &L) {
    LNode *s;
    int x;
    L = (LinkList) malloc(sizeof(LNode));
    L->next = NULL;
    cout << "请输入结点的值，输入9999结束：" << endl;
    cin >> x;
    while (x != 9999) {
        s = (LNode *) malloc(sizeof(LNode)); // 创建新结点
        s->data = x;
        s->next = L->next;
        L->next = s;
        cin >> x;
    }
    return L;
}
```

3、尾插法建立单链表

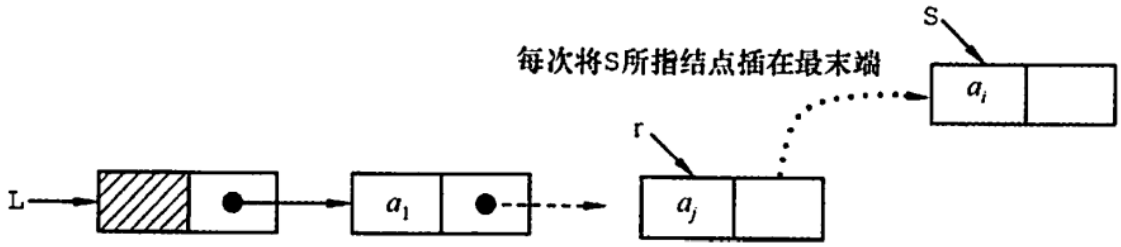


图 2.6　尾插法建立单链表

```
// 尾插法建立单链表
LinkList List_TailInsert(LinkList &L) {
    int x;
    L = (LinkList) malloc(sizeof(LNode));
    LNode *s;
    LNode *r = L;   // 尾指针
    cout << "请输入结点的值，输入9999结束：" << endl;
    cin >> x;
    while (x != 9999) {
        s = (LNode *) malloc(sizeof(LNode));
        s->data = x;
        r->next = s;
        r = s;
        cin >> x;
    }
    r->next = NULL;
    return L;
}
```

4、按序号查找结点

在单链表中从第一个结点出发，顺指针next 域逐个往下搜索，直到找到第i个结点为止,否则返回最后一个结点指针域NULL。

```
// 从单链表中查找指定位序的结点，并返回
LNode *getElem(LinkList L, int i) {
    if (i < 0) {
        return NULL;
    }
    LNode *p;
    int j = 0;  // 定义一个j指针标记下标
    p = L;      // p指针用来标记第i个结点
    while (p != NULL && j < i) {  // p==NULL 超出表长,返回空值  j>i 超出所查询元素的下标
        p = p->next;
        j++;        // j指针后移
    }
    return p;
}
```

5、按值查找结点

从单链表的第一个结点开始，由前往后依次比较表中各结点数据域的值若某结点数据域的值等于给定值e，则返回该结点的指针;若整个单链表中没有这样的结点，则返回NULL。

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

6、头插法插入元素

插入结点操作将值为x的新结点插入到单链表的第i个位置上。先检查插入位置的合法性，然后找到待插入位置的前驱结点，即第i-1个结点，再在其后插入新结点。

算法首先调用按序号查找算法GetElem(L,i-1)，查找第i-1个结点。假设返回的第i-1个结点为*p，然后令新结点*s的指针域指向*p 的后继结点，再令结点*p 的指针域指向新插入的结点*s。

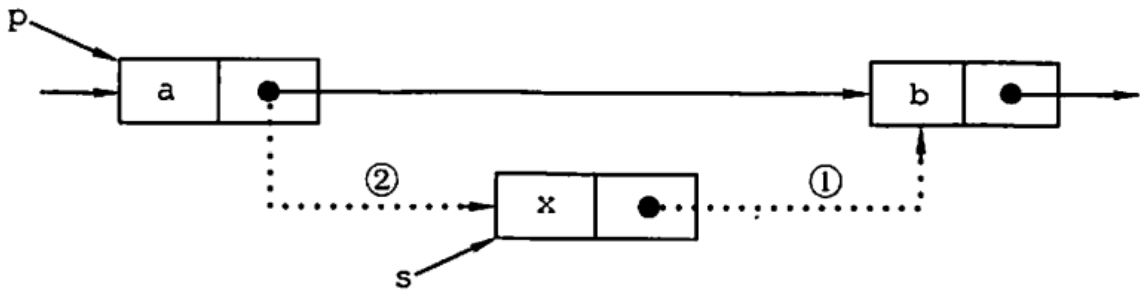


图 2.7　单链表的插入操作

```
// 在第i个元素前插入元素value
```

```
bool ListInsert(LinkList &L, int i, int e) {
    if (i < 1)
        return false;
    LNode *p;            //指针p指向当前扫描到的结点
    int j = 0;             //当前p指向的是第几个结点
    p = L;               //循环找到第i-1个结点
    while (p != NULL && j < i - 1) {       //如果i>lengh，p最后会等于NULL
        p = p->next;
        j++;
    }
    //p值为NULL说明i值不合法
    if (p == NULL)
        return false;
    //在第i-1个结点后插入新结点
    LNode *s = (LNode *) malloc(sizeof(LNode));
    s->data = e;
    s->next = p->next;
    p->next = s;
    //将结点s连到p后
    return true;
}
```

7、删除结点

删除结点操作是将单链表的第i个结点删除。先检查删除位置的合法性,后查找表中第i-1个结点，即被删结点的前驱结点，再将其删除。

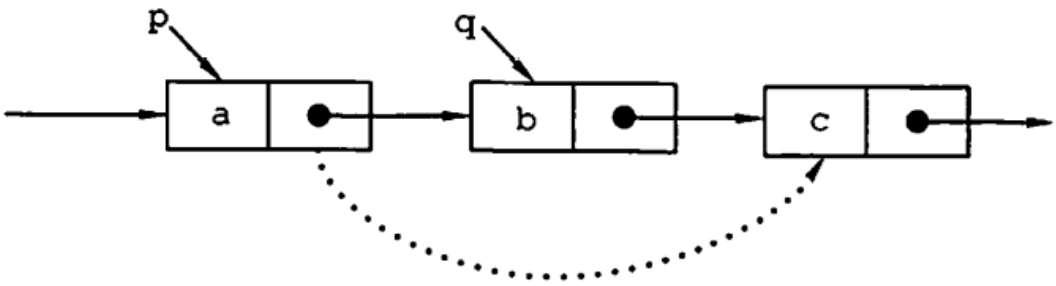


图 2.8　单链表结点的删除

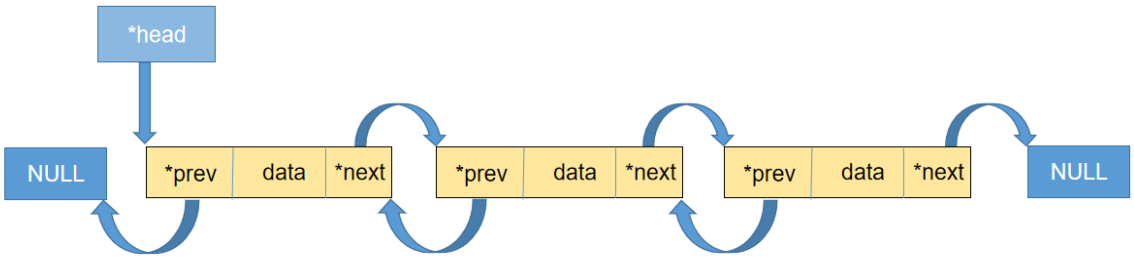
```
// 删除第i个结点并将其所保存的数据存入value
bool ListDelete(LinkList &L, int i, int &value) {
    if (i < 1)
        return false;
    LNode *p;        //指针p指向当前扫描到的结点
    int j = 0;         //当前p指向的是第几个结点
    p = L;
    //循环找到第i-1个结点
    while (p != NULL && j < i - 1) {
        //如果i>lengh，p和p的后继结点会等于NULL
        p = p->next;
        j++;
    }
    if (p == NULL)
        return false;
    if (p->next == NULL)
        return false;
```

```
    //令q暂时保存被删除的结点
    LNode *q = p->next;
    value = q->data;
    p->next = q->next;
    free(q);
    return true;
}
```

2.3.3 双链表

单链表结点中只有一个指向其后继的指针，使得单链表只能从头结点依次顺序地向后遍历。要访问某个节点的前驱结点（插入、删除操作）时，只能从头开始遍历，访问后继节点的时间复杂度为O(1)，访问前驱结点的时间复杂度为O(n)。

非循环带头双链表结构示意图



为了解决如上问题，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。

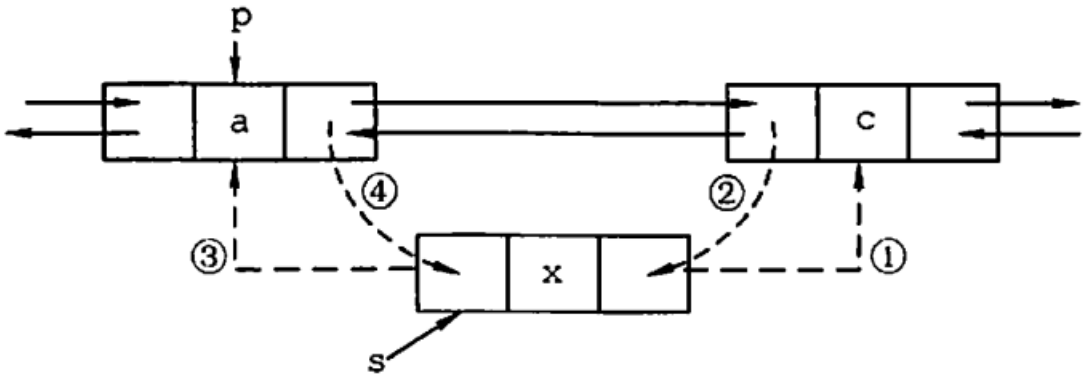
双链表的类型描述如下：

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

1、双链表的初始化

```
bool InitDLinkList(DLinkList &L) {
    L = (DNode *) malloc(sizeof(DNode));
    if (L == NULL) {
        return false;
    }
    L->prior = NULL; // 头结点的前驱指针永远指向NULL
    L->next = NULL; // 后继指针暂时为空
    return true;
}
```

2、双链表的后插操作




```
// 将节点s插入到结点p之后
bool InsertNextDNode(DNode *p, DNode *s) {
    if (p == NULL || s == NULL) {
        return false;
    }
    s->next = p->next;  // 将p的后继赋给s的后继
    // 判断p之后是否还有前驱节点
    if (p->next != NULL) {
        p->next->prior = s;
    }
    s->prior = p;
    p->next = s;
    return true;
}
```

> 双链表的前插操作、按位序插入操作都可以转换成后插操作

3、双链表的删除操作

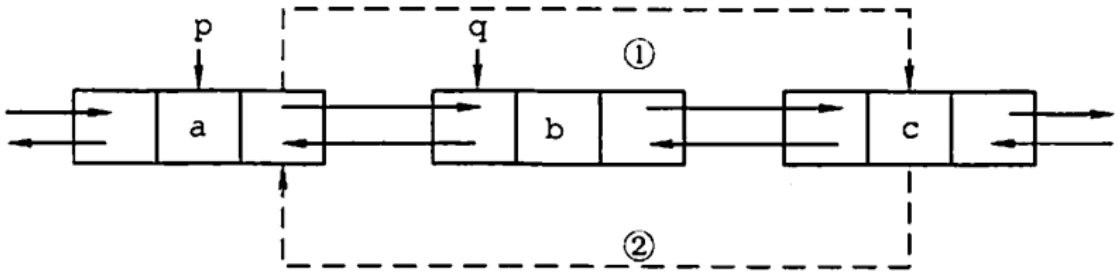


图 2.11　双链表删除结点过程

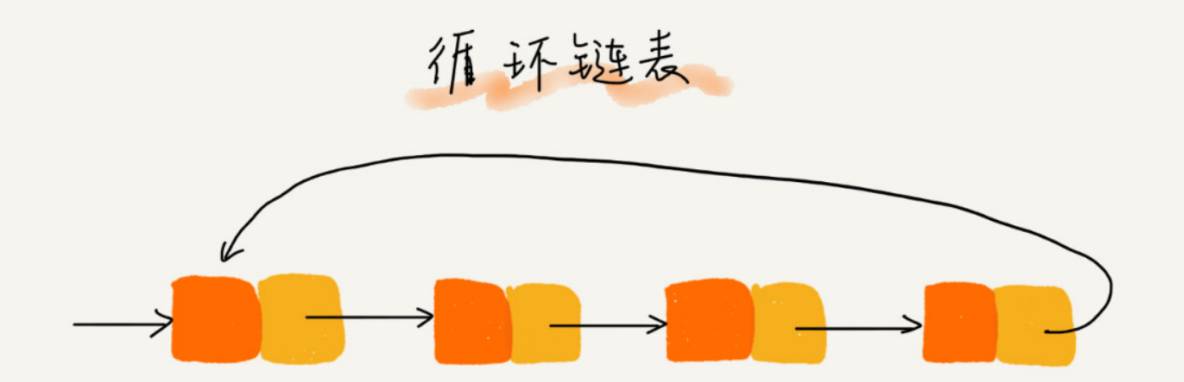
```
// 删除p结点的后续结点
bool DeleteNextDNode(DNode *p) {
    if (p == NULL) {
        return false;
    }
    // 找到p的后继结点q
    DNode *q = p->next;
    if (q == NULL) {
        return false;
    }
    p->next = q->next;  // 将q的后继赋给p的后继
    if (q->next != NULL) {  // 若q的后继结点不为空
        q->next->prior = p; // 将p赋给q的后继节点的前驱节点
```

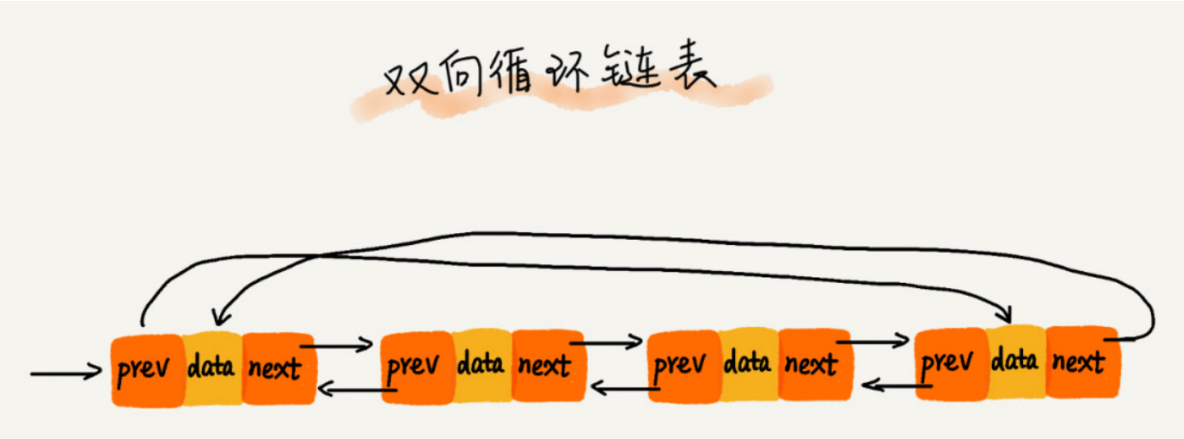
```
    }
    free(q);    // 释放q
    return true;
}

// 销毁一个双链表
bool DestoryList(DLinkList &L) {
    // 循环释放各个数据结点
    while (L->next != NULL) {
        DeleteNextDNode(L);
        free(L);
        // 头指针置空
        L = NULL;
    }
}
```

> 双链表不可随机存取，按位查找、按值查找操作都只能用遍历的方式实现。

2.3.4 循环链表

循环链表是另一种形式的链式存储结构。它的特点是表中最后一个结点的指针域指向头结点，整个链表形成一个环。





> 判断循环单链表是否为空：
>
> ```
> if (L->next == L){
>     return true;
> } else {
>     return false;
> }
> ```

循环单链表可以从任意结点开始往后遍历整个链表。

2.3.5 静态链表

用数组的方式实现的链表。分配一整片连续的内存空间，各个结点集中安置，每个结点包括了数据元素和下一个结点的数组下标。

- 优点：增、删操作不需要大量移动元素。

- 缺点：不能随机存取，只能从头结点开始依次往后查找，**容量固定不变！**

定义1：

```
#define MaxSize 10          //静态链表的最大长度
struct Node{                //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
};
```

定义2：

```
#define MaxSize 10          //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];

void testSLinkList(){
    SLinkList a;
}
```

> 第一种是我们更加熟悉的写法，第二种写法则更加侧重于强调 a 是一个静态链表而非数组。

4. **静态链表的注意点：**

    1. 初始化静态链表时，需要把 `a[0]` 的 `next` 设为-1，并将空闲结点的 `next` 设置为某个特殊值，比如-2。
    2. 按位序查找结点时，从头结点出发挨个往后遍历结点，时间复杂度 $O=(n)$。
    3. 按位序插入结点的步骤：①找到一个空的结点，存入数据元素；②从头结点出发找到位序为 i-1 的结点；③修改新结点的 `next` 为 -1；④修改 i-1 号结点的 `next` 为新结点的下标；

2.4 顺序表vs链表

	顺序表	**链表**
逻辑结构	属于线性表，都是线性结构	属于线性表，都是线性结构
存储结构	顺序存储 优点：支持随机存取，存储密度高 缺点：大片连续空间分配不方便，改变容量不方便	链式存储 优点：离散的小空间分配方便，改变容量方便 缺点：不可随机存取，存储密度低
基本操作——创建	需要预分配大片连续空间。若分配空间过小，则之后不方便拓展容量；若分配空间过大，则浪费内存资源。 静态分配：静态数组，容量不可改变。 动态分配：动态数组，容量可以改变，但是需要移动大量元素，时间代价高（使用 `malloc()`、`free()`）。	只需要分配一个头结点或者只声明一个头指针。
基本操作——	修改 `Length` = 0 静态分配：静态数组——系统自动回收空间。 动态分配：动态数组——需要手动 `free()`。	依次删除各个结点 `free()`。

销毁	顺序表	链表
基本操作——增删	插入 / 删除元素要将后续元素后移 / 前移； 时间复杂度：O(n)，时间开销主要来自于移动元素。	插入 / 删除元素只需要修改指针； 时间复杂度：O(n)，时间开销主要来自查找目标元素。
基本操作——查找	按位查找：O(1) 按值查找：O(n)，若表内元素有序，可在O(log2n) 时间内找到（二分法）	按位查找：O(n) 按值查找：O(n)