qq群：136808580

数据结构与算法（代码）

第一章 线性表

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**。

线性表的表示：若用L命名，表示：L=(a1,a2,a3,a4,a5,.....,an)

线性表的逻辑特性：

- a1：唯一的表头元素
- an：唯一的表尾元素
- 除去a1：每个元素有且仅有一个直接前驱
- 除去an：每个元素有且仅有一个直接后继

线性表的特点：

- 表中元素是有限个
- 表中元素具有逻辑上的顺序性，各个元素有先后次序
- 表中元素都是数据元素，每一个元素都是单个元素
- 表中元素的数据类型都相同
- 表中每个元素占用相同大小的存储空间
- 表中元素具有抽象性。线性表 **仅讨论元素间的逻辑关系**，不讨论元素的具体内容

线性表、顺序表、链表

- 线性表是 **逻辑结构**，表示一对一的相邻关系
- 顺序表是 **采用顺序存储** 对线性表的实现，是指存储（物理）结构
- 链表是采用 **链式存储** 对线性表的实现，是指存储（物理）结构

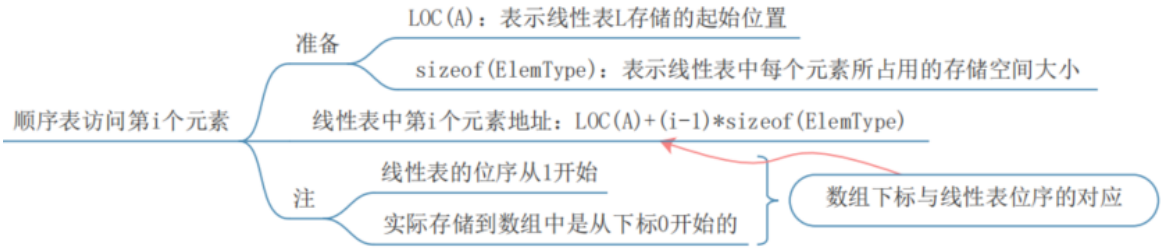
线性表的基本操作

1.1 顺序表

把线性表中所有元素按照逻辑顺序，依次存储到从指定位置开始的一块 **连续存储空间** ，线性表的顺序存储叫作顺序表。

特点：

- 第一个元素的存储位置就是指定的存储位置
- 第i+1个元素的存储位置紧接在第i个元素的存储位置后面
- 逻辑相邻的两个元素物理也相邻
- 顺序表可以实现随机存取
- 存储密度高（不用额外存储指示信息）
- 逻辑相邻在物理上也相邻，插删需要移动大量元素



1.1.1 顺序表结构

线性表的顺序存储（即顺序表）的存储类型描述

一维数组静态分配

```
#define MaxSize 128     //定义顺序表的最大长度
typedef int ElemType;   //理解成给int起个别名,以后用ElemType代表int，体现抽象数据类型
//顺序表的顺序存储结构
typedef struct {
    ElemType data[MaxSize]; //顺序表的主体用来存储数据元素
    int length;             //顺序的当前长度
}SqList;            //同样给顺序表struct类型起个别名，以后用SqList代表顺序表struct
```

静态存储的特点

- 数组的大小空间已经固定
- 空间满时，再加入新数据会导致溢出

一维数组的动态分配

```
#define InitSize 16     //顺序表的初始容量大小
#define MaxSize 128     //顺序表的最大容量大小

typedef int ElemType;
typedef struct {
    ElemType* data;     //动态分配数组的指针
    int length;         //顺序表的当前长度
    int capacity;       //记录当前容量
}DySqList;
```

动态存储的特点

- 在执行过程中根据需要，动态分配。
- 空间满时，可以开辟另外一块更大空间，达到扩充目的

动态分配不是链式存储，分配的空间依然是连续的，依然采用随机存取方式

动态存储的实现

与静态存储方式的实现基本一致，只是需要手动申请空间以及释放空间（这里给出代码不再细讲）

```
//初始化
void initList(DySqList& L) {
    L.data = (ElemType*)malloc(InitSize * sizeof(ElemType));
    if (L.data == NULL) {
        exit(-1);   //内存分配失败
    }
    L.length = 0;   //顺序表长置0
    L.capacity = InitSize;  //更开始申请时顺序表的容量即为初始化容量大小
}
//扩容
void expansion(DySqList& L) {
    L.capacity += (L.capacity >> 1);
    if (L.capacity >= MaxSize) {
        L.capacity = MaxSize;
        exit(-1);
    }
    //先判断当前的指针是否有足够的连续空间，如果有，扩大mem_address指向的地址，并且将
mem_address返回，如果空间不够，先按照newsize指定的大小分配空间，将原有数据从头到尾拷贝到新分
配的内存区域，而后释放原来mem_address所指内存区域
    L.data = (ElemType*)realloc(L.data, L.capacity * sizeof(ElemType));
    if (!L.data) {
        exit(-1);//分配内存失败
    }
}

//创建顺序表
void createList(DySqList& L) {
    ElemType x;
    scanf("%d", &x);
    int i = 0;
    while (x != 999) {
        if (i >= L.capacity) {
            expansion(L);
        }
        //*(L.data + i) = x;
        L.data[i++] = x;
        scanf("%d", &x);
    }
    L.length = i;
}

//插入操作。在表L中第pos个位置上插入指定元素e
bool insertList(DySqList& L, int pos, ElemType e) {
    //如果顺序表已满返回false
    if (L.length >= L.capacity) {
```

```
        exit(-1);
    }
    //检查插入位置pos是否合法，不合法返回false
    else if (pos<1 || pos>L.length + 1) {
        return false;
    }
    //将最后一个元素开始直到第pos个位置处的元素依次后移
    for (int i = L.length - 1; i >= pos - 1; i--) {
        L.data[i + 1] = L.data[i];
    }
    L.data[pos - 1] = e;    //将待插入的元素插入到第pos个位置上
    L.length++;             //维持表长正确，表长+1
    return true;
}

bool deleteList(DySqList& L, int pos, ElemType& e) {
    //如果顺序表为空，返回false
    if (L.length == 0) {
        return false;
    }
    //检查pos位置的合法性
    else if (pos<1 || pos>L.length) {
        return false;
    }
    else {
        //将待删除元素用e接收
        e = L.data[pos - 1];
        //将第pos+1个位置处的元素直到最后一个元素依次前移
        for (int i = pos; i < L.length; i++) {
            L.data[i - 1] = L.data[i];
        }
        //维持顺序表长度的正确性
        L.length--;
        return true;
    }
}

//按值查找操作。在表L中查找具有给定关键字值的元素，若存在返回第一个值为e的所在位置，不存在返回0
int locateElem(DySqList L, ElemType e) {
    int ans = 0;    //用来记录最终返回的结果
    for (int i = 0; i < L.length; i++) {
        if (L.data[i] == e) {
            ans = i + 1;    //返回所在位置，下标要+1，如果找到退出查找
            break;
        }
    }
    return ans;
}

//按位查找操作。获取表L第pos个位置的元素的值
ElemType getElem(DySqList L, int pos) {
    //检查pos位置的合法性，不合法直接退出程序
    if (pos <1 || pos>L.length) {
        exit(0);
    }
```

```
    return L.data[pos - 1];
}

//判空操作。若L为空表，则返回true，否则返回false
bool isEmpty(DySqList L) {
    return L.length == 0;
}

//销毁顺序表
void destroyList(DySqList& L) {
    free(L.data);   //释放内存空间
    L.length = 0;
    L.capacity = 0;
    L.data = NULL;
}

void printList(DySqList L) {
    for (int i = 0; i < L.length; i++) {
        printf("%d ", L.data[i]);
    }
    printf("\n");
}
```

1.1.2 顺序表的初始化

```
void initList(SqList& L) {
    //对数据进行初始化，防止不当使用时出现脏数据
    for (int i = 0; i < MaxSize; i++) {
        L.data[i] = 0;
    }
    L.length = 0;   //表长初始化为0
}
```

1.1.3 在指定位置插入元素

位置	1	2	3	4	5	6
下标	0	1	2	3	4	5
元素	10	20	30	40	50	

在第二个位置插入元素100

位置	1	2	3	4	5	6
下标	0	1	2	3	4	5
元素	10	100	20	30	40	50

从最后一个位置到第pos个位置上的元素依次后移一位，再将100插到第pos个位置上，表长+1

```
//插入操作。在表L中第pos个位置上插入指定元素e
bool insertList(SqList& L, int pos, ElemType e);
```

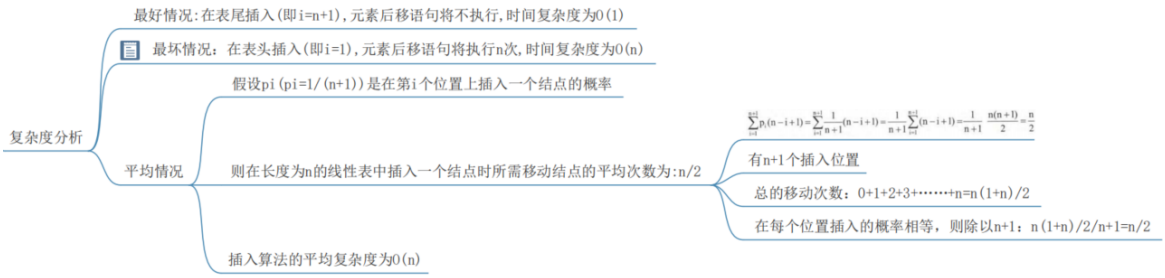
算法步骤

1. 检查顺序表是否已满，如果已满返回false表示插入失败
2. 检查pos位置是否合法（1<=pos<=L.length+1，如果pos不合法返回false，表示插入失败
3. 从最后一个位置开始直到第pos个位置上的元素依次后移
4. 将待插入的元素插入到pos个位置处
5. 维持表长的正确性（即将表长加一）返回true

算法实现

```
bool insertList(SqList& L, int pos, ElemType e) {
    //如果顺序表已满返回false
    if (L.length >= MaxSize) {
        return false;
    }
    //检查插入位置pos是否合法，不合法返回false
    else if (pos<1 || pos>L.length + 1) {
        return false;
    }
    else {
        //将最后一个元素开始直到第pos个位置处的元素依次后移
        for (int i = L.length - 1; i >= pos - 1; i--) {
            L.data[i + 1] = L.data[i];
        }
        L.data[pos - 1] = e;     //将待插入的元素插入到第pos个位置上
        L.length++;              //维持表长正确，表长+1
        return true;
    }
}
```

复杂度分析



区别顺序表的位序（1开始）和数组下标（0开始）

1.1.4 删除指定位置的元素

位置	1	2	3	4	5	6
下标	0	1	2	3	4	5
元素	10	20	30	40	50	

删除第二个位置的元素值

位置	1	2	3	4	5	6
下标	0	1	2	3	4	5
元素	10	30	40	50	50	

从pos后一个位置到顺序表中最后一个位置依次前移，最后表长-1。此时原表中的最后一个元素还存在在表中，但是表长-1后我们可以理解顺序表中已经被删除了要删除的元素了。或者最后可以手动将最后一个元素值设为0。

```
//删除操作。删除表L中第pos个位置的元素,并用e返回删除元素的值
bool deleteList(SqList& L, int pos, ElemType& e);
```

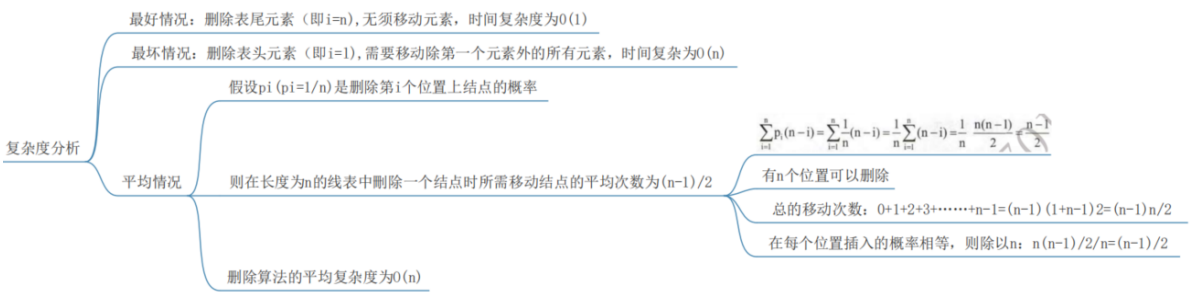
算法步骤

1. 如果表为空返回false
2. 检查pos位置是否合法（1<=pos<=L.length）,如果不合法返回false
3. 用e接收被删除的元素值
4. 将第pos+1位置上的元素直到表的最后元素依次前移
5. 维持表长的正确性（表长减一）返回true

代码实现

```
bool deleteList(SqList& L, int pos, ElemType& e) {
    //如果顺序表为空，返回false
    if (L.length == 0) {
        return false;
    }
    //检查pos位置的合法性
    else if (pos<1 || pos>L.length) {
        return false;
    }
    else {
        //将待删除元素用e接收
        e = L.data[pos - 1];
        //将第pos+1个位置处的元素直到最后一个元素依次前移
        for (int i = pos; i < L.length; i++) {
            L.data[i - 1] = L.data[i];
        }
        //维持顺序表长度的正确性
        L.length--;
        return true;
    }
}
```

复杂度分析



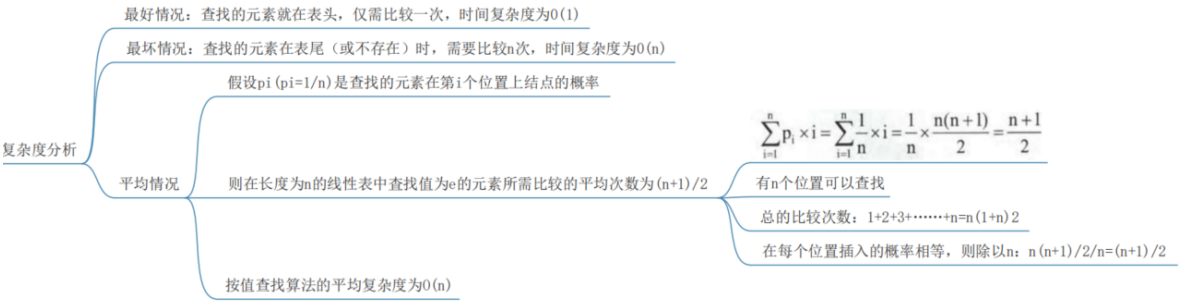
1.1.5 查找指定位置的元素（顺序查找）

```
//按值查找操作，在表L中查找具有给定关键字值的元素，若存在返回第一个值为e的所在位置，不存在返回0
int locateElem(SqList L, ElemType e);
```

算法实现

```
//按值查找操作，在表L中查找具有给定关键字值的元素，若存在返回第一个值为e的所在位置，不存在返回0
int locateElem(SqList L, ElemType e) {

    int ans = 0;    //用来记录最终返回的结果
    for (int i = 0; i < L.length; i++) {
        if (L.data[i] == e) {
            ans = i + 1;    //返回所在位置，下标要+1，如果找到退出查找
            break;
        }
    }
    return ans;
}
```

复杂度分析



1.1.6 查找元素所在位置

```
ElemType getElem(SqList L, int pos) {
    //检查pos位置的合法性，不合法直接退出程序
    if (pos <1 || pos>L.length) {
        exit(0);
    }
    return L.data[pos - 1];
}
```

时间复杂度：O(1)。顺序表支持随机存取

1.1.7 顺序表的判空

```
bool isEmpty(SqList L) {
    return L.length == 0;
}
```

1.1.8 销毁顺序表

```
void destroyList(SqList& L) {
    for (int i = 0; i < L.length; i++) {
        L.data[i] = 0;
    }
    L.length = 0;
}
```

静态数组的内存由编译器在栈上管理，而不需要手动释放

1.1.9 顺序表的打印

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

1.1.10 刷题

01.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，则显示出错信息并退出运行。

遍历顺序表，找到值最小的元素下标

下标	0	1	2	3	4	5
元素	3	5	2	1	4	6
	①min minIndex		②min minIndex	③min minIndex		

将最后一个元素放到最小值所在下标处。

下标	0	1	2	3	4	5
元素	3	5	2	6	4	6
				③min minIndex		

算法思路

遍历顺序表，找到值最小的元素所在下标，将最后一个元素放到最小值所在下标处。

算法步骤

1. 判断表是否为空，如果为空返回false
2. 定义两个变量min及minIndex分别用来表示最小值和最小值所在位置下标，默认0位置处元素是最小值

3. 从第二个元素开始遍历顺序表如果当前遍历的元素值比min小，则将当前元素赋值给min，当前下标赋值给minIndex
4. 遍历结束后，minIndex位置即为最小值所在位置下标，用最后一个元素覆盖最小值
5. 维持表长正确性（表长减一），返回true

算法实现

```
bool delMin(SqList& L) {
    //空表直接返回false
    if (L.length == 0) {
        printf("表空无法操作\n");
        return false;
    }
    int min = L.data[0];//记录最小值
    int minIndex = 0;   //记录最小值所在位置下标
    for (int i = 1; i < L.length; i++) {
        if (L.data[i] < min) {
            min = L.data[i];
            minIndex = i;
        }
    }
    L.data[minIndex] = L.data[L.length - 1];     //用最后一个位置元素覆盖被删除元素
    L.length--; //维持表长正确性
    return true;
}
```

02.设计一个高效算法，将顺序表L的所有元素逆置，要求算法的空间复杂度为0(1)。

算法思路

将第一个元素与最后一个进行交换，第二个与倒数第二个交换，依次类推，直到交换表长的一半次即可

算法步骤

1. 记录应交换的次数（即L.length/2次）
2. 进行L.length/2次元素交换即可

算法实现

```
//实现方法1
void reverseList(SqList& L) {
    int midCount = L.length / 2;     //记录应交换的次数
    int i = 0;
    while (i < midCount){
        //首尾交换元素
        int temp = L.data[i];
        L.data[i] = L.data[L.length - 1 - i];
        L.data[L.length - 1 - i] = temp;
        i++;
    }
}
```

```
//实现方法2
void reverseList2(SqList& L) {
    int low = 0, high = L.length - 1;//当前两个"指针"分别指向第一个元素和最后一个元素
    //当前半部分指针和下半部分指针没有相遇时交换low指针和high指针所指向的元素
    while (low < high) {
        int temp = L.data[low];
        L.data[low] = L.data[high];
        L.data[high] = temp;
        low++;
        high--;
    }
}
```

03.对长度为n的顺序表L,编写一个时间复杂度为0(n)、空间复杂度为0(1)的算法，该算法删除线性表中所有值为x的数据元素。

算法思路

边遍历边统计边移动。遍历顺序表，统计值为x的个数count，将不是值为x的元素前移count位，维护表长正确性，将表长减count即可

算法实现

```
void delX(SqList& L,ElemType x) {
    int count = 0;  //用来记录值为x的元素个数
    for (int i = 0; i < L.length; i++) {
        if (L.data[i] == x) {
            count++;
        }
        else {
            L.data[i - count] = L.data[i];  //如果当前元素值不为x，将其前移count位
        }
    }
    L.length -= count;  //维护表长
}
```

04.从顺序表中删除其值在给定值s与t之间(包含s和t, 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。

算法思路1

与03题思路相同，只不过此时在遍历顺序表时统计的是介于s与t之间的元素个数count，将值不在s、t之间的元素前移count位，并维护表长。

算法实现1

```
void delS2T(SqList& L, ElemType s, ElemType t) {
    if (L.length == 0) {
        printf("顺序表为空,无法操作\n");
        exit(0);
    }else if (s >= t) {
```

```
        printf("s、t输入不合法\n");
        exit(0);
    }else {
        int count = 0;  //用来统计介于s与t之间的元素个数
        for (int i = 0; i < L.length; i++) {
            if (L.data[i] >= s && L.data[i] <= t) {
                count++;
            }
            else {
                L.data[i - count] = L.data[i];
            }
        }
        L.length -= count;
    }
}
```

算法思路2

利用双指针，i作为遍历指针，j每次指向满足不在s、t之间要插入的位置，遍历完成后，j即为删除介于s、t之间元素后的新表长。

算法实现2

```
void delS2T_2(SqList& L, ElemType s, ElemType t) {
    if (L.length == 0) {
        printf("顺序表为空,无法操作\n");
        exit(0);
    }
    else if (s >= t) {
        printf("s、t输入不合法\n");
        exit(0);
    }
    else {
        int i = 0, j = 0;//i作为遍历指针，j每次指向满足不在s、t之间要插入的位置
        while (i < L.length) {
            if (L.data[i]<s || L.data[i]>t) {
                L.data[j++] = L.data[i];    //将需要保留的元素插入到j所指位置
            }
            i++;
        }
        L.length = j;   //j即为新表长
    }
}
```

05.从有序顺序表中删除其值在给定值s与t之间(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.

算法思路1

找到第一个大于等于s的元素以及大于t的元素，将第一个比t大的元素直到最后一个元素前移到第一个大于等于s的位置及其后续位置。

算法实现1

```
void delStoT(SqList& L, ElemType s, ElemType t) {
    if (L.length == 0) {
        printf("顺序表为空,无法操作\n");
        exit(0);
    }
    else if (s >= t) {
        printf("s、t输入不合法\n");
        exit(0);
    }
    else {
        int i = 0, j;
        //查找第一个>=s的元素下标
        while (i < L.length && L.data[i] < s) {
            i++;
        }
        //如果不存在大于等于s的元素直接退出，没必要在删除了
        if (i >= L.length) {
            exit(0);
        }
        //找第一个>t的元素下标
        for (j = i; j < L.length && L.data[j] <= t; j++);
        //第一个比t大的元素直到最后一个元素前移到第一个大于等于s的位置及其后续位置。
        while (j < L.length) {
            L.data[i++] = L.data[j++];
        }
        L.length = i;//此时i即为新表长
    }
}
```

算法思路2

方法1中我们在进行查找第一个s和第一个大于t的元素时使用的顺序查找，时间复杂度为O(n)，为此我们可以利用折半查找来降低查找的时间复杂度使之变为O(logn)，当然后续移动的操作不变，由于在顺序表删除元素的时间复杂度扔为O(n)，故整体时间复杂度依然为O(n)，只不过对方法1的查找操作进行了代码优化。

```
//用二分法找第一个比s大的元素所在位置下标
int findBigEqualLeft(SqList L, ElemType target) {
    int left = 0, right = L.length - 1;
    int ans = -1;
    while (left <= right) {
        int mid = (left + right) >> 1;
        if (L.data[mid] >= target) {
            ans = mid;
            right = mid - 1;
        }
        else {
            left = mid + 1;
        }
    }
    return ans;
}
//用二分法找第一个比t大的元素所在位置下标
int findSmallEqualRight(SqList L, ElemType target) {
    int left = 0, right = L.length - 1;
```

```
    int ans = -1;
    while (left <= right) {
        int mid = (left + right) >> 1;
        if (L.data[mid] <= target) {
            ans = mid;
            left = mid + 1;
        }
        else {
            right = mid - 1;
        }
    }
    return (ans == -1 ? ans : ans + 1);
}

void delStoT_3(SqList& L, ElemType s, ElemType t) {
    if (L.length == 0) {
        printf("顺序表为空,无法操作\n");
        exit(0);
    }
    else if (s >= t) {
        printf("s、t输入不合法\n");
        exit(0);
    }
    else {
        int findBigSFirst = findBigEqualLeft(L, s);
        if (findBigSFirst == -1) {
            return;
        }
        int findBigTFirst = findSmallEqualRight(L, t);
        if (findBigTFirst == -1) {
            return;
        }
        //没有比t大的元素
        if (findBigTFirst >= L.length) {
            L.length -= (findBigTFirst - findBigSFirst);
            return;
        }
        else {
            for (int i = findBigTFirst; i < L.length; i++) {
                L.data[findBigSFirst++] = L.data[i];
            }
            L.length = findBigSFirst;
        }
    }
}
```

算法思路3

利用双指针，i为工作指针，j为满足不介于s、t之间的元素所指位置，当遍历到的当前元素<s，i和j指针同时后移，当元素值介于s、t之间时，只移动工作指针i，当遍历到大于t时，将此时及后面的所有元素覆盖从i开始的元素值，最后j即为表长。

算法实现3

```
void delStoT_2(SqList& L, ElemType s, ElemType t) {
```

```
    if (L.length == 0) {
        printf("顺序表为空,无法操作\n");
        exit(0);
    }
    else if (s >= t) {
        printf("s、t输入不合法\n");
        exit(0);
    }
    else {
        int i = 0, j = 0;    //i作为遍历指针，j指向满足要保存的元素位置
        for (; i < L.length; i++) {
            if (L.data[i] < s) {
                j++;
            }
            else if (L.data[i] > t) {
                L.data[j++] = L.data[i];
            }
        }
        L.length = j;
    }
}
```

06.从有序顺序表中删除所有其值重复的元素，使表中所有元素的值均不同。

算法思路1

从第二个元素依次和前一个元素比较值是否相等，如果相等记录此时出现相同元素的个数count，如果不等将当前元素前移count位，维护表长的正确性。

算法实现1

```
void delRepeat(SqList& L) {

    int count = 0; //统计重复元素的个数
    for (int i = 1; i < L.length; i++) {
        if (L.data[i] == L.data[i - 1]) {
            count++;
        }
        else {
            L.data[i - count] = L.data[i];  //将不重复的元素前移count位
        }
    }
    L.length -= count;
}
```

算法思路2

利用双指针，i和j初始都指针第二个元素，因为第一个元素自己的话一定不重复，i作为工作指针遍历顺序表，j每次指向满足不重复元素应在的位置，如果i和i-1处的值相同，只移动i指针，如果不等说明其是应保留的非重复元素将其放到j所指位置，最终j即为表长。

算法实现2

```
void delRepeat2(SqList &L) {
    int i = 1, j = 1;
    while (i < L.length) {
        if (L.data[i] != L.data[i - 1]) {
            L.data[j++] = L.data[i];
        }
        i++;
    }
    L.length = j;
}
```

07.将两个有序顺序表合并为一个新的有序顺序表，并由函数返回结果顺序表。

算法思路

同时遍历LA和LB表中的元素，比较当前两个表中的哪个元素小，谁小将其拷贝到LC中，如果相等规定先拷贝LA中的，当然也可以规定相等时先拷贝LB中的。

算法实现

```
bool mergeList(SqList LA, SqList LB,SqList &LC) {
    if (LA.length + LB.length > MaxSize) {
        return false;
    }
    int i = 0, j = 0, k = 0;    //i,j,k分别作为LA,LB,LC的遍历指针
    while (i < LA.length && j < LB.length) {
        if (LA.data[i] <= LB.data[j]) {     //谁小将其拷贝到LC中，相等默认先拷贝LA中的
            LC.data[k++] = LA.data[i++];
        }
        else {
            LC.data[k++] = LB.data[j++];
        }
    }
    //如果LA,LB有一个还没遍历完，将其剩下元素拷贝到LC中。两个while只会执行一个
    while (i < LA.length) {
        LC.data[k++] = LA.data[i++];
    }
    while (j < LB.length) {
        LC.data[k++] = LB.data[j++];
    }
    LC.length = LA.length + LB.length;
    return true;
}
```

08.已知在一维数组A[m + n]中依次存放两个线性表(a1,a2,a3,...,am)和(b1,b2,b3,...,bn).编写一个函数，将数组中两个顺序表的位置互换，即将(b1,b2,b3,...,bn)放在(a1,a2,a3,...,am)的前面。

算法思路

先逆置前m个元素，在逆置m-m+n位置的元素，最后将整个表逆置即可。

算法实现

```
//将顺序表的第start到end位置上的元素进行逆置
void inverList(SqList& L, int start, int end) {
    int low = start - 1, high = end - 1;
    while (low < high) {
        int temp = L.data[low];
        L.data[low] = L.data[high];
        L.data[high] = temp;
        low++;
        high--;
    }
}
void exchangeListElem(SqList& L, int m, int n) {
    inverList(L, 1, m); //逆置前m个元素
    inverList(L,m + 1, m + n);  //逆置后n个元素
    inverList(L, 1, m + n);     //将顺序表整体进行逆置
}
```

1.2单链表

1.2.1定义

通过一组 **任意的存储单元** 来存储线性表中的数据元素，是线性表的 **链式存储。**

单链表节点类型描述

1. 因为逻辑相邻不一定物理相邻，所以需要额外的指针来存储后继信息。
2. 单链表节点的组成
    1. data：数据域，存放数据元素
    2. next：指针域，存放后继节点的地址
3. 代码定义

```
typedef struct {
    ElemType data;  //数据域
    LNode* next;    //指针域，指向后继结点
}LNode,*LinkedList;
```

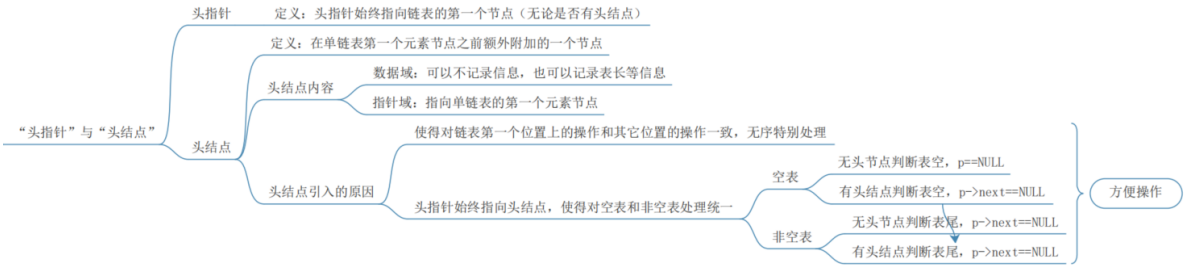
```
typedef struct {
    ElemType data;
    LNode* next;
}LNode;

typedef struct {
    LNode* head;    //头指针
    int length;     //记录表长
}LinkedList;
```

单链表特点

1. 解决了顺序表插删需要大量移动元素的缺点
2. 引入了额外的指针域，浪费了空间
3. 单链表是非随机存取的存储结构，查找需要从头遍历

“头指针”与“头结点”

``



1.2.2头插法创建单链表（不带头结点）

```
void createListByHead(LinkedList& L) {
    ElemType x;
    scanf("%d", &x);
    while (x != 999) {
        LNode* node = (LNode*)malloc(sizeof(LNode));    //申请结点内存空间
        node->data = x; //为新结点赋值
        if (L == NULL) {
            node->next = NULL;  //如果是插入的第一个结点。用头插法的话最后他是尾结点将其
next指空
        }
        else {
            node->next = L; //如果不是插入的第一个结点，将当前结点直接之前的头指针
        }
        L = node;   //更新最新的结点作为头指针
        scanf("%d", &x);
    }
}
```

1.2.3尾插法创建单链表（不带头结点）

```
void createListByTail(LinkedList& L) {
    LNode* tail = NULL; //尾指针
    ElemType x;
    scanf("%d", &x);
    while (x != 999) {
        LNode* node = (LNode*)malloc(sizeof(LNode));
        node->data = x;
        node->next = NULL;
        if (L == NULL) {    //如果是第一个结点，头尾都指向该结点
            L = node;
            tail = node;
        }
        else {
            tail->next = node;  //如果不是第一个结点，尾指针的next指向该结点
            tail = node;        //当前新添加结点作为尾结点
```

```
        }
        scanf("%d", &x);
    }
}
```

1.2.4返回第pos个位置上的结点（不带头结点）

```
LNode* getNode(LinkedList L, int pos) {
    //位置不对返回NULL
    if (pos < 0) {
        return NULL;
    }
    int i = 1;
    LNode* p = L;
    while (p != NULL && i < pos) {
        p = p->next;
        i++;
    }
    return p;
}
```

1.2.5按值查找结点（不带头结点）

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

1.2.6在第pos个位置插入元素e（不带头结点）

```
void insertElem(LinkedList& L, int pos, ElemType e) {
    //检查pos位置的合法性
    if (pos < 1 || pos>length(L) + 1) {
        return;
    }1
    //创建新结点
    LNode* node = (LNode*)malloc(sizeof(LNode));
    node->data = e;
    //如果是在第一个位置插入元素，新结点作为头指针
    if (pos == 1) {
        node->next = L;
        L = node;
    }
    else {
        LNode *pre = getNode(L, pos - 1);    //找到插入位置的前一个结点
        node->next = pre->next;
        pre->next = node;
    }
}
```

1.2.7删除第pos个位置的元素结点，并用e返回被删除结点的元素值（不带头结点）

```
void deleteElem(LinkedList& L, int pos, ElemType& e) {
    //检查pos位置的合法性
    if (pos < 1 || pos>length(L)) {
        return;
    }
    //查找被删除结点
    LNode *removed = getNode(L, pos);
    //如果要删除第一个结点元素，直接将要删除的下一个元素作为头指针
    if (pos == 1) {
        L = removed->next;
    }
    else {
        LNode* pre = getNode(L, pos - 1);    //找到被删除元素结点的前一个
        pre->next = removed->next;
    }
    e = removed->data;  //将要删除结点的值赋值给e接收
    free(removed);      //释放被删除元素的空间内存
}
```

1.2.8求表长（不带头结点）

```
int length(LinkedList L) {
    LNode* p = L;
    if (p == NULL) {
        return 0;
    }
    int ans = 0;
    while (p != NULL) {
        p = p->next;
        ans++;
    }
    return ans;
}
```

1.2.9单链表的基本操作（带头结点）

```
typedef int ElemType;

typedef struct LNode {
    ElemType data;
    LNode* next;
}LNode, * LinkedList;
//初始化单链表
void initLinkedList(LinkedList& L) {
    L = (LNode*)malloc(sizeof(LNode));  //申请头结点
    if (L) {
        L->next = NULL;
        L->data = 0;                          //头结点的数据域保存链表长度
```

```
    }
}
//头插法创建单链表
void createLinkedListWithHead(LinkedList& L) {
    ElemType x;
    scanf("%d", &x);
    while (x != 999) {
        LNode* cur = (LNode*)malloc(sizeof(LNode)); //申请新结点空间
        cur->data = x;
        cur->next = L->next;    //新结点的next指向头结点的next
        L->next = cur;          //头结点的next指向当前新结点
        L->data += 1;           //链表长度+1
        scanf("%d", &x);
    }
}
//尾插法创建单链表
void createLinkedListWithTail(LinkedList& L) {
    ElemType x;
    LNode* tail = L;
    scanf("%d", &x);
    while (x != 999) {
        LNode* cur = (LNode*)malloc(sizeof(LNode));
        cur->data = x;
        cur->next = NULL;
        tail->next = cur;   //尾指针的next每次指向最新结点
        tail = cur;         //最新结点成为尾指针
        L->data += 1;       //表长+1
        scanf("%d", &x);
    }

}
//返回第pos个位置上的结点
LNode* getNode(LinkedList L, int pos) {
    //检查pos是否合法，data是链表的长度
    if (pos<0 || pos>L->data) {
        return NULL;
    }
    int i = 0;
    LNode* p = L;
    while (p && i < pos){
        p = p->next;
        i++;
    }
    return p;
}
//在第pos个位置插入元素e
void insertElem(LinkedList& L, int pos, ElemType e){
    if (pos<1 || pos>L->data + 1) {
        return;
    }
    LNode* cur = (LNode*)malloc(sizeof(LNode));
    cur->data = e;
    LNode * pre = getNode(L,pos - 1);   //获得要插入位置的前一个位置结点
    cur->next = pre->next;              //新结点的next指向前一个结点的next
    pre->next = cur;                    //前一个结点的next指向当前结点
```

```
    L->data += 1;                          //表长+1
}
//删除第pos个位置的元素结点，并用e返回被删除结点的元素值
void deleteElem(LinkedList& L, int pos, ElemType& e){
    //检查pos是否合法
    if (pos<1 || pos>L->data) {
        return;
    }
    LNode* removed = getNode(L, pos);    //记录要删除元素
    LNode* pre = getNode(L, pos - 1);    //记录要删除元素的前一个元素结点
    e = removed->data;                   //被删除元素用e接收返回
    pre->next = removed->next;           //被删除的前一个结点指向其后一个结点
    L->data--;                           //表长-1
    free(removed);                       //释放删除结点的空间
}
//打印链表
void printLinkedList(LinkedList L) {
    LNode* p = L->next;      //第一个数据元素是头结点的后继结点元素
    while (p) {
        printf("%d->", p->data);
        p = p->next;
    }
    printf("\n");
}
```

1.2.10刷题

01.设计一个递归算法，删除不带头结点的单链表L中所有值为x的结点。

```
//递归
void delX(LinkedList& L,ElemType x) {
    if (L == NULL) {    //终止条件
        return;
    }
    if (L->data == x) {      //如果是被删除元素结点
        LNode* p = L;        //防止断链
        L = L->next;         //记录被删除元素的下一个结点，
        free(p);             //释放被删除元素空间
        delX(L, x);          //递归下一个结点
    }
    else {                   //如果当前元素的值不是被删除元素，递归下一个元素
        delX(L->next, x);
    }
}
```

```
//非递归
void delX2(LinkedList& L, ElemType x) {
    if (L == NULL) {
        return;
    }
    LNode* removed;//用来记录被删除元素结点
    while (L!=NULL&&L->data == x) { //如果链表还没有遍历完找到第一个值不为x结点，同时将遍历过程中值为x的结点都删除
        removed = L;//如果当前结点值为x记录当前结点为待删除结点
```

```
        L = L->next;     //L后移
        free(removed);  //释放被删除结点的空间
    }
    if (L == NULL) {//如果L已经为NULL，证明结点值都为x且已被删除直接返回
        return;
    }
    LNode* p = L->next;      //此时L一定头指针，节点元素值不为x
    LNode* pre = L;          //用来指向不是值为x的元素的最后一个结点
    while (p){
        //如果当前节点的值为x，removed记录待删除节点，p后移，pre指向待删除结点的后继节点，释放removed空间
        if (p->data == x) {
            removed = p;
            p = p->next;
            pre->next = removed->next;
            free(removed);
        }
        //如果当前节点的值不为x，pre和p同时后移
        else {
            pre = pre->next;
            p = p->next;
        }
    }
    pre->next = NULL;   //删除结束后pre指向最后一个结点，将其next指空
}
```

02.在带头结点的单链表L中，删除所有值为x的结点，并释放其空间，假设值为x的结点
不唯一，试编写算法以实现.上述操作。

```
void delX2(LinkedList& L, ElemType x) {
    LNode* p = L->next;//p作为遍历指针从首结点开始
    if (p == NULL) {//如果链表为空，返回
        return;
    }
    LNode* removed;//用来记录被删除元素结点
    LNode* pre = L;//pre始终指向最后一个值不为x的结点
    while (p) {
        //如果当前节点的值为x，removed记录待删除节点，p后移，pre指向待删除结点的后继节点，释放removed空间
        if (p->data == x) {
            removed = p;
            p = p->next;
            pre->next = removed->next;
            free(removed);
        }
        //如果当前节点的值不为x，pre和p同时后移
        else {
            pre = pre->next;
            p = p->next;
        }
    }
```

```
    pre->next = NULL;//删除结束后pre指向最后一个结点，将其next指空
}
```

03.设L为带头结点的单链表，编写算法实现从尾到头反向输出每个结点的值。

```
//递归
void inverPrintList(LinkedList L) {
    LNode* p = L->next;     //从第一个首节点开始
    if (p == NULL) {        //终止条件
        return;
    }
    inverPrintList(p);      //递归
    printf("%d->", p->data);
}
```

```
//非递归
void inverPrintList2(LinkedList L) {
    //申请动态数组用来保存链表中的元素值
    ElemType* set = (ElemType*)malloc(sizeof(ElemType) * L->data);
    int i = 0;
    LNode* p = L->next;
    while (p) {
        set[i++] = p->data;
        p = p->next;
    }
    for (int j = i - 1; j >= 0; j--) {
        printf("%d->", set[j]);
    }
    printf("\n");
}
```

04.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)。

```
void delMin(LinkedList& L) {
    if (L->next == NULL) {
        return;
    }
    LNode* p = L->next; //从首节点开始
    LNode* pre = L;     //始终是p的前一个位置的指针
    LNode* min = p;     //记录最小值指针，假设初始第一个元素就是最小值
    LNode* minpre = L;  //记录最小值的前一个元素指针
    while (p) {
        if (p->data < min->data) {  //如果当前元素的值比最小值小
            min = p;                //最小值指针指向当前结点
            minpre = pre;           //最小值的前一个指向pre
        }
        //每次p和pre都后移
        p = p->next;
        pre = pre->next;
    }
    minpre->next = min->next;   //删除min结点
    free(min);  //释放空间
```

```
}
```

05.试编写算法将带头结点的单链表就地逆置，所谓“就地”是指辅助空间复杂度为0(1)。

```
void reverse(LinkedList& L) {
    LNode* cur = L->next;
    LNode* next = NULL;
    L->next = NULL;
    while (cur) {
        next = cur->next;
        cur->next = L->next;
        L->next = cur;
        cur = next;
    }
}
```

06.有一个带头结点的单链表L,设计一个算法使其元素递增有序。

```
void sortListAsc(LinkedList& L) {
    //如果链表为空，或者链表只有一个结点直接返回不需要排序
    if (L->next == NULL || L->next->next == NULL) {
        return;
    }
    //从第二个结点开始
    LNode* p = L->next->next;
    L->next->next = NULL;
    while (p != NULL) {
        LNode* next = p->next;  //记录一下p的后继防止断链
        LNode* head = L->next;  //head作为每次已排好序的部分链表中的首节点同时作为遍历指
针
        LNode* pre = L; //遍历指针head的前一个结点
        while (head != NULL && p->data > head->data) {//如果不满足插入位置pre和head
后移
            pre = pre->next;
            head = head->next;
        }
        p->next = head; //在head和pre之间插入p
        pre->next = p;
        p = next;   //p后移重新遍历未进行排序的结点操作
    }
}
```

07.设在一个带表头结点的单链表中所有元素结点的数据值无序，试编写一个函数，删除表中所有介于给定的两个值(作为函数参数给出)之间的元素的元素(若存在)。

```
void delSToT(LinkedList& L, ElemType s, ElemType t){
    if (L->next == NULL) {
        return;
```

```
    }
    LNode* p = L->next;      //遍历指针从首节点开始
    LNode* pre = L;          //始终指向不是删除元素的最后位置
    while (p) {
        if (p->data >= s && p->data <= t) { //如果是被删除元素
            LNode* removed = p; //记录被删除元素
            p = p->next;        //后移
            pre->next = p;      //最后一个非删除元素指向当前删除元素的后继
            free(removed);      //释放删除结点空间
        }
        else {          //如果不在删除范围，p和pre后移
            p = p->next;
            pre = pre->next;
        }
    }
}
```

08.给定两个单链表，编写算法找出两个链表的公共结点。

```
LNode* findCommon(LinkedList L1, LinkedList L2){
    int len1 = length(L1);
    int len2 = length(L2);
    LNode* longhead = len1 > len2 ? L1 : L2;
    LNode* shorthead = longhead == L1 ? L2 : L1;
    //int diff = abs(len1 - len2);
    int diff = len1 - len2 > 0 ? len1 - len2 : len2 - len1;
    while (diff > 0) {
        longhead = longhead->next;
        diff--;
    }
    while (longhead) {
        if (longhead == shorthead) {
            return longhead;
        }
        longhead = longhead->next;
        shorthead = shorthead->next;
    }
    return NULL;
}
```

09.给定一个带表头结点的单链表，设head为头指针，结点结构为(data, next)，data为整型元素，next为指针，试写出算法:按递增次序输出单链表中各结点的数据元素，并释放结点所占的存储空间(要求：不允许使用数组作为辅助空间)。

```
LNode* delMin2(LinkedList& L) {
    if (L->next == NULL) {
        return L;
    }
    LNode* p = L->next; //从首节点开始
    LNode* pre = L;     //始终是p的前一个位置的指针
    LNode* min = p;     //记录最小值指针
    LNode* minpre = L;  //记录最小值的前一个元素指针
    while (p) {
```

```
        if (p->data < min->data) {
            min = p;
            minpre = pre;
        }
        p = p->next;
        pre = pre->next;
    }
    minpre->next = min->next;
    return min;
}

void sortListAndDelete(LinkedList& L) {
    if (L->next == NULL) {
        return;
    }
    LNode *min = delMin2(L);
    printf("%d->", min->data);
    free(min);
    if (L->next != NULL) {
        sortListAndDelete(L);
    }
}
```

10. 将一个带头结点的单链表A分解为两个带头结点的单链表A和B,使得A表中含有原表中序号为奇数的元素, 而B表中含有原表中序号为偶数的元素，且保持其相对顺序不变。

```
LinkedList resolveList(LinkedList& LA) {
    LNode* p = LA->next;
    if (p == NULL) {
        return NULL;
    }
    LinkedList LB = (LNode*)malloc(sizeof(LNode));  //申请LB的头结点
    LNode* tailA = LA;  //LA的尾指针
    LNode* tailB = LB;  //LB的尾指针
    int i = 1;          //用来标记遍历的结点的序号是奇序号还是偶序号
    while (p) {
        //奇数时
        if (i % 2 == 1) {
            tailA->next = p;
            tailA = p;
        }
        //偶数时
        else {
            tailB->next = p;
            tailB = p;
        }
        p = p->next;
        i++;
    }
    tailA->next = NULL;     //最后不要忘了把LA和LB的尾指针的next指向空
    tailB->next = NULL;
    return LB;
}
```

1.3循环单链表

```
typedef int ElemType;

typedef struct LNode {
    ElemType data;
    LNode* next;
}LNode;
typedef struct LinkedList {
    LNode* head;    //头结点
    LNode* tail;    //尾结点
    int length;     //链表长度
}LinkedList;

//初始化单链表
void initLinkedList(LinkedList& L) {
    L.head = NULL;  //初始化头指针尾指针为空长度为0
    L.tail = NULL;
    L.length = 0;
}
//头插法创建单链表
void createLinkedListWithHead(LinkedList& L) {
    ElemType x;
    scanf("%d", &x);
    while (x != 999) {
        LNode* node = (LNode*)malloc(sizeof(LNode));
        node->data = x;
        if (L.head == NULL) {   //如果是创建的第一个元素，头指针尾指针都指向当前元素
            L.head = node;
            L.tail = node;
        }
        else {
            node->next = L.head;    //如果不是第一个元素和普通头插法一样
            L.head = node;
        }
        L.tail->next = L.head;      //每次要把尾指针的next指向头指针
        L.length++;                 //维护表长正确性
        scanf("%d", &x);
    }
}
//尾插法创建单链表
void createLinkedListWithTail(LinkedList& L) {
    ElemType x;
    scanf("%d", &x);
    while (x != 999) {
        LNode* node = (LNode*)malloc(sizeof(LNode));
        node->data = x;
        if (L.head == NULL) {       //如果是创建的第一个结点
            L.head = node;
            L.tail = node;
        }
        else {
            L.tail->next = node;    //尾指针的next指向新创建的结点
```

```
            L.tail = node;          //尾指针来到新结点处
        }
        L.tail->next = L.head;      //每次尾指针的next指向头指针
        L.length++;
        scanf("%d", &x);
    }
}
//返回第pos个位置上的结点
LNode* getNode(LinkedList L, int pos) {
    //位置不对或表位空返回NULL
    if (pos < 0 || L.head == NULL) {
        return NULL;
    }
    if (pos == 1) {     //如果要找第一个位置结点元素直接返回
        return L.head;
    }
    int i = 1;
    LNode* p = L.head->next;    //从第二个结点开始查找
    while (p != L.head && i < pos - 1) {
        p = p->next;
        i++;
    }
    return p == L.head ? NULL : p;  //如果再一次来到头指针即第一个元素位置证明没找到返回NULL，否则返回找到的p
}
//向表尾插入元素e
void insertElemToTail(LinkedList& L, ElemType e) {
    LNode* node = (LNode*)malloc(sizeof(LNode));
    node->data = e;
    if (L.head == NULL) {   //如果链表为空，创建新结点让头尾指针都指向新结点
        L.head = node;
        L.tail = node;
    }
    else {
        L.tail->next = node;
        L.tail = node;
    }
    L.tail->next = L.head;
    L.length++;
}
//打印链表
void printLinkedList(LinkedList L) {
    LNode* p = L.head;
    //分开打印是因为刚开始p的位置即为L.head，当p第二次来到L.head时说明已经遍历一轮了，为了避免下面while刚进去就退出条件的情况，如果是带头结点的循环单链表不需要单独分开打印
    if (p) {
        printf("%d->", p->data);
        p = p->next;
    }

    while (p != L.head) {
        printf("%d->", p->data);
        p = p->next;
    }
    printf("\n");
```

```
}
```

1.4双链表

```
typedef int ElemType;
typedef struct DNode {
    ElemType data;  //数据域
    DNode* prior;   //前驱指针域
    DNode* next;    //后继指针域
}DNode,*DLinkedList;

//双链表的初始化(带头结点)
void initDLinkedList(DLinkedList& L){
    L = (DNode*)malloc(sizeof(DNode));
    if (L) {
        L->next = NULL;
        L->prior = NULL;
        L->data = 0;
    }
}
//头插法创建单链表
void createDListWithHead(DLinkedList& L){
    ElemType x;
    scanf("%d", &x);
    while (x!=999){
        DNode* node = (DNode*)malloc(sizeof(DNode));
        if (node) {
            node->data = x;
            node->next = L->next;
            if (L->next != NULL) {
                L->next->prior = node;
            }
            L->next = node;
            node->prior = L;
            L->data++;
        }
        scanf("%d", &x);
    }
}

//尾插法创建单链表
void createDListWithTail(DLinkedList& L) {
    ElemType x;
    scanf("%d", &x);
    DNode* tail = L;
    while (x != 999) {
        DNode* node = (DNode*)malloc(sizeof(DNode));
        if (node) {
            node->data = x;
            node->next = NULL;
            tail->next = node;
            node->prior = tail;
            tail = node;
            L->data++;
        }
```

```
        scanf("%d", &x);
    }
}

//返回第pos个位置上的结点
DNode* getNode(DLinkedList L, int pos) {
    //检查pos是否合法，data是链表的长度
    if (pos<0 || pos>L->data) {
        return NULL;
    }
    int i = 0;
    DNode* p = L;
    while (p && i < pos) {
        p = p->next;
        i++;
    }
    return p;
}
//在第pos个位置插入元素e
void insertElem(DLinkedList& L, int pos, ElemType e) {
    if (pos<1 || pos>L->data + 1) {
        return;
    }
    DNode* node = (DNode*)malloc(sizeof(DNode));
    if (node) {
        node->data = e;
        DNode* pre = getNode(L, pos - 1);
        node->next = pre->next;
        if (pos != L->data + 1) {
            pre->next->prior = node;
        }
        pre->next = node;
        node->prior = pre;
        L->data++;
    }


}
//删除第pos个位置上的元素。并用e接收被删除元素值
void deleteElem(DLinkedList& L, int pos, ElemType& e) {
    if (pos<1 || pos>L->data) {
        return;
    }
    DNode* pre = getNode(L, pos - 1);
    DNode* removed = getNode(L, pos);
    if (pos == L->data) {
        pre->next = NULL;
    }
    else {
        pre->next = removed->next;
        removed->next->prior = pre;
    }
    e = removed->data;
    free(removed);
```

```
    L->data--;
}
//打印双链表
void printDList(DLinkedList L) {
    DNode* p = L->next;
    if (p == NULL) {
        return;
    }
    printf("%d->", p->data);
    printDList(p);
}
```

第二章 栈

2.1顺序栈

顺序栈的基本操作

```
#define MAXSIZE 128
typedef int ElemType;
typedef struct {
    ElemType data[MAXSIZE];     //用数组实现对栈中元素的存取
    int top;                    //栈顶指针
    int length;                 //栈的长度
}SqStack;

//初始化栈
void initStack(SqStack& S);

//判断栈是否x为空，为空返回true，否则返回false
bool isEmpty(SqStack S);

//如果栈S没有满，将x入栈
bool push(SqStack& S, ElemType x);

//如果栈不为空，则将栈顶元素出栈，并返回
ElemType pop(SqStack& S);

//如果栈顶不为空，返回栈顶元素
ElemType peek(SqStack S);

//销毁栈，释放栈所占的存储空间
void destroy(SqStack& S);
```

顺序栈的代码实现

```
#define MAXSIZE 128
typedef int ElemType;
```

```
typedef struct {
    ElemType data[MAXSIZE];     //用数组实现对栈中元素的存取
    int top;                    //栈顶指针
    int length;                 //栈的长度
}SqStack;
//初始化栈
void initStack(SqStack& S) {
    //对申请的数组空间依次进行初始化防止不当操作出现脏数据
    for (int i = 0; i < MAXSIZE; i++) {
        S.data[i] = 0;
    }

    S.top = 0;  //默认栈顶指向0下标
    S.length = 0;//初始化长度为0
}

//判断栈是否x为空，为空返回true，否则返回false
bool isEmpty(SqStack S) {
    return S.length == 0;
}

//如果栈S没有满，将x入栈
bool push(SqStack& S, ElemType x) {
    //栈已满
    if (S.top >= MAXSIZE) {
        return false;
    }
    S.data[S.top++] = x;    //先赋值后移动top索引
    S.length++;             //栈长+1
}

//如果栈不为空，则将栈顶元素出栈，并由x返回
ElemType pop(SqStack& S) {
    if (isEmpty(S)) {       //规定-999为空，如果栈空返回空
        return -999;
    }
    S.length--;     //表长-1
    return S.data[--S.top]; //删除栈顶时，先将栈顶指针下移
}

//如果栈顶不为空，返回栈顶元素
ElemType peek(SqStack S) {
    if (!isEmpty(S)) {
        return S.data[S.top - 1];   //如果栈不为空，返回栈顶元素。栈顶下标-1为第一个栈顶
元素
    }
    return -999;
ee
//销毁栈，释放栈所占的存储空间
void destroy(SqStack& S) {
    //销毁栈手动将数据置为0。栈顶指针指向0，表长设为0
    for (int i = 0; i < S.length; i++) {
        S.data[i] = 0;
    }
    S.top = 0;
```

```
    S.length = 0;
}

void printStack(SqStack S) {
    while (S.top > 0) {
        printf("%d ", S.data[--S.top]);
    }
    printf("\n");
}
```

2.2链栈

```
typedef int ElemType;
typedef struct LNode{
    ElemType data;
    LNode* next;
}LNode,*LinkedStack;

//初始化链栈
void initLinkedStack(LinkedStack& L) {
    L = (LNode*)malloc(sizeof(LNode));  //建立头结点
    L->next = NULL;      //头结点的next指空
    L->data = 0;         //头结点的data保存栈长度
}

//判断栈空
bool isEmpty(LinkedStack L) {
    return L->next == NULL;      //或return L->data == 0
}

//进栈（入栈）
void push(LinkedStack& L, ElemType x) {
    LNode* node = (LNode*)malloc(sizeof(LNode));    //创建新结点
    node->data = x; //为新结点赋值
    node->next = L->next;   //头插法插入新结点
    L->next = node;
    L->data++;  //栈长+1
}

//出栈（弹栈）
ElemType pop(LinkedStack& L) {
    if (L->next == NULL) {
        return -999;    //规定-999为空，若栈空返回空
    }
    LNode* removed = L->next;        //记录待删除元素，以便释放空间
    L->next = removed->next;    //跳过被删除元素结点，头结点直接指向被删除元素的后继
    ElemType x = removed->data; //x接收被删除元素的值以便返回
    free(removed);       //释放被删除节点空间
    L->data--;           //表长-1
    return x;
}

//取栈顶
ElemType peek(LinkedStack L) {
```

```
    return L->next == NULL ? -999 : L->next->data;  //如果表为空返回-999,否则返回首节点的元素值
}

//销毁栈
void destroy(LinkedStack& L) {
    LNode* p = L->next;
    LNode* tail;
    while (p) {     //依次释放栈除头结点外其他元素结点的空间
        tail = p->next;
        free(p);
        p = tail;
    }
    L->data = 0;        //栈空置为0
}

//求栈长
int length(LinkedStack L) {
    return L->data;
}

//打印栈元素
void printStack(LinkedStack L) {
    if (L == NULL||L->next == NULL) {
        return;
    }
    LNode* p = L->next;
    while (p){
        printf("%d->", p->data);
        p = p->next;
    }
    printf("\n");
}
```

2.3共享栈

```
#define MAXSIZE 16
typedef int ElemType;
typedef struct {
    int data[MAXSIZE];  //共享栈的数据
    int size;           //共享栈的总大小
    int top1;           //第一个栈的栈顶
    int top2;           //第二个栈的栈顶
}SharedStack;

//初始化共享栈
void initStack(SharedStack& S) {
    for (int i = 0; i < MAXSIZE; i++) {
        S.data[i] = 0;
    }
    S.size = 0;     //初始化栈的总大小为0
    S.top1 = 0;     //第一个栈的栈顶指针初始化指向0
    S.top2 = MAXSIZE - 1;//第一个栈的栈顶指针初始化指向MAXSIZE-1
}
//判断第一个栈是否已满
```

```
bool isFull1(SharedStack S) {
    return S.top1 > S.top2; //当第一个栈的栈顶超过了第二个栈的栈顶证明栈已满
}
//判断第二个栈是否已满
bool isFull2(SharedStack S) {
    return S.top2 < S.top1; //当第二个栈的栈顶小于了第一个栈的栈顶证明栈已满
}

//判断第一个栈是否为空
bool isEmpty1(SharedStack S) {
    return S.top1 == 0;
}

//判断第二个栈是否为空
bool isEmpty2(SharedStack S) {
    return S.top2 == MAXSIZE - 1;
}

// 在第一个栈中入栈元素e
void push1(SharedStack& S, ElemType e) {
    //如果栈1不满在执行添加操作，添加时先添加元素后移动栈顶指针，别忘了维持共享栈的总大小
    if (!isFull1(S)) {
        S.data[S.top1++] = e;
        S.size++;
    }
}

// 在第二个栈中入栈元素e
void push2(SharedStack& S, ElemType e) {
    if (!isFull2(S)) {
        S.data[S.top2--] = e;
        S.size++;
    }
}

// 在第一个栈中出栈
ElemType pop1(SharedStack& S) {
    //如果栈不空才执行删除操作，删除时栈顶指针先减在移除
    if (!isEmpty1(S)) {
        S.size--;
        ElemType x = S.data[--S.top1];
        S.data[S.top1] = 0; //删除完给他置为0
        return x;
    }
    return -999;//代表空栈返回空
}

// 在第二个栈中出栈
ElemType pop2(SharedStack& S) {
    if (!isEmpty2(S)) {
        S.size--;
        ElemType x = S.data[++S.top2];
        S.data[S.top2] = 0;
        return x;
    }
```

```
    return -999;
}

// 查看第一个栈栈顶元素
ElemType peek1(SharedStack S) {
    if (!isEmpty1(S)) {
        return S.data[--S.top1];
    }
    return -999;//代表空栈返回空
}

// 查看第二个栈栈顶
ElemType peek2(SharedStack S) {
    if (!isEmpty2(S)) {
        return S.data[++S.top2];
    }
    return -999;
}

//获取第一个栈的长度
int getLength1(SharedStack S) {
    return S.top1;
}

//获取第二个栈的长度
int getLength2(SharedStack S) {
    return MAXSIZE - 1 - S.top2;
}
//打印第一个栈中的元素值
void printStack1(SharedStack S) {
    for (int i = S.top1 - 1; i >= 0; i--) {
        printf("%d ", S.data[i]);
    }
    printf("\n");
}

//打印第二个栈中的元素值
void printStack2(SharedStack S) {
    for (int i = S.top2 + 1; i < MAXSIZE; i++) {
        printf("%d ", S.data[i]);
    }
    printf("\n");
}
//打印共享栈中所有的元素值
void printStack(SharedStack S) {
    for (int i = 0; i < MAXSIZE; i++) {
        printf("%d ", S.data[i]);
    }
    printf("\n");
}
```

第三章 队列

3.1顺序队列

```
#define MAXSIZE 64
typedef int ElemType;
typedef struct {
    ElemType data[MAXSIZE];
    int front;  //队头指针
    int rear;   //队尾指针
    int size;   //队列大小
}SeQueue;

//初始化队列
void initQueue(SeQueue& Q) {
    //对数据元素进行初始化，防止出现脏数据
    for (int i = 0; i < MAXSIZE; i++) {
        Q.data[i] = 0;
    }
    Q.front = 0;    //初始化队头指针指向0索引
    Q.rear = 0;     //队尾指针也指向0索引
    Q.size = 0;     //队列大小为0
}

//判断队列是否为空
bool isEmpty(SeQueue Q) {
    //return Q.front == Q.rear; //如果front和rear指向同一个位置则队列为空
    return Q.size == 0;         //或直接用size是不是0来判断
}

//元素x入队
void enQueue(SeQueue& Q, ElemType x) {
    Q.data[Q.rear++] = x;   //在尾指针位置插入元素x后将尾指针后移（尾指针始终指向待入队元素的位置）
    Q.size++;   //位置队列长度
}

//队头元素出队，并返回
ElemType deQueue(SeQueue& Q) {
    if (!isEmpty(Q)) {      //如果队列不为空
        Q.size--;           //队列长度-1
        return Q.data[Q.front++];   //返回队头元素后将队头指针后移
    }
    return -999;        //如果队列为空返回-999代表空
}

//获取队首元素
ElemType getHead(SeQueue Q) {
    if (isEmpty(Q)) {       //如果队列不为空，直接返回队头元素否则返回-999
        return Q.data[Q.front];
    }
    return -999;
}

//获取队列长度
int getSize(SeQueue Q) {
    //return Q.size;        //直接返回Q.size即队列长度
```

```
    return Q.rear - Q.front;    //队尾指针位置-队头指针位置也为队列长度
}

//打印队列
void printQueue(SeQueue Q) {
    if (isEmpty(Q)) {
        return;
    }
    for (int i = Q.front; i < Q.rear; i++) {    //队头开始遍历
        printf("%d ", Q.data[i]);
    }
    printf("\n");
    printf("Q.front = %d ,Q.rear = %d,Q.size = %d\n", Q.front, Q.rear, Q.size);
}
```

3.2循环队列

```
#define MAXSIZE 8
typedef int ElemType;
typedef struct {
    ElemType data[MAXSIZE];
    int front;  //队头指针
    int rear;   //队尾指针
    int size;   //队列大小
}Queue;
/*
*用size判断队列空队列满可以避免一个空间的浪费，如果不使用size的话，至少要浪费一个空间来区分队满和队空
*/
//初始化队列
void initQueue(Queue& Q) {
    //对数据元素进行初始化，防止出现脏数据
    for (int i = 0; i < MAXSIZE; i++) {
        Q.data[i] = 0;
    }
    Q.front = 0;    //初始化队头指针指向0索引
    Q.rear = 0;     //队尾指针也指向0索引
    Q.size = 0;     //队列大小为0
}

//判断队列是否已满

bool isFull(Queue Q) {
    return (Q.rear + 1) % MAXSIZE == Q.front;
    //return Q.size == MAXSIZE;
}

//判断队列是否为空
bool isEmpty(Queue Q) {
        return Q.front == Q.rear;
    //return Q.size == 0;
}

//元素x入队
void enQueue(Queue& Q, ElemType x) {
```

```
    if (isFull(Q)) {
        return;
    }
    Q.data[Q.rear] = x;
    Q.rear = (Q.rear + 1) % MAXSIZE;
    Q.size++;
}

//队头元素出队，并返回
ElemType deQueue(Queue& Q) {
    if (isEmpty(Q)) {
        return -999;
    }
    ElemType x = Q.data[Q.front];
    Q.front = (Q.front + 1) % MAXSIZE;
    Q.size--;
    return x;
}

//获取队首元素
ElemType getHead(Queue Q) {
    return Q.data[Q.front];
}

//获取队列长度
int getSize(Queue Q) {
    return (Q.rear - Q.front + MAXSIZE) % MAXSIZE;
    //return Q.size;
}

//打印队列元素
void printQueue(Queue Q) {
    if (isEmpty(Q)) {
        return;
    }
    int len = getSize(Q);
    int i = Q.front;
    while (len > 0) {
        printf("%d ", Q.data[i]);
        i++;
        if (i >= MAXSIZE) {
            i = 0;
        }
        len--;
    }
    printf("\n");
}
```

3.3链队列

```
typedef int ElemType;
//结点结构
typedef struct LNode {
    ElemType data;
    LNode* next;
```

```
}LNode;

//队列结构
typedef struct {
    LNode* front;   //队列的队头指针
    LNode* rear;    //队列的队尾指针
    int size;       //队列大小
}LinkedQueue;

//初始化队列
void initQueue(LinkedQueue& Q) {
    Q.front = NULL;
    Q.rear = NULL;
    Q.size = 0;
}

//判断队列是否为空
bool isEmpty(LinkedQueue Q) {
    if (Q.front == NULL && Q.rear == NULL) {
        return true;
    }
    else {
        return false;
    }
}

//元素x入队
void enQueue(LinkedQueue& Q, ElemType x) {
    LNode* node = (LNode *)malloc(sizeof(LNode));
    node->data = x;
    node->next = NULL;
    //如果是入队的第一个元素
    if (Q.front == NULL && Q.rear == NULL) {
        Q.front = node;
        Q.rear = node;
    }
    else {
        Q.rear->next = node;    //最后一个结点的next指向新结点元素
        Q.rear = node;          //rear指向新结点
    }
    Q.size++;               //队列长度+1
}

//队头元素出队，并返回
ElemType deQueue(LinkedQueue& Q) {
    if (isEmpty(Q)) {
        return -999;
    }
    LNode* removed = Q.front;   //记录队头（即被删除元素结点以便释放空间）
    ElemType x = removed->data; //接收队头元素以便返回
    Q.front = Q.front->next;    //队头后移
    //如果已经出队直到没元素了，别忘了也让Q.rear = null
    if (Q.front == NULL) {
         Q.rear = Q.front;
    }
```

```
    free(removed);  //释放删除结点空间
    Q.size--;       //队列长度-1
    return x;
}

//获取队首元素
ElemType getHead(LinkedQueue Q) {
    if (isEmpty(Q)) {
        return -999;
    }
    return Q.front->data;
}

//获取队列长度
int getSize(LinkedQueue Q) {
    return Q.size;
}

//打印队列元素
void printQueue(LinkedQueue Q) {
    LNode* p = Q.front;
    while (p) {
        printf("%d ", p->data);
        p = p->next;
    }
    printf("\n");
}
```

3.4双端队列（带头双链表实现）

```
//用双链表(带头结点)实现双端队列
typedef int ElemType;

typedef struct DNode {
    ElemType data;      //数据域
    DNode* prior;       //指向前驱结点的指针域
    DNode* next;        //指向后继结点的指针域
}DNode;

typedef struct {
    DNode* front;   //队头指针
    DNode* rear;    //队尾指针
    int size;       //队列长度
}DeQueue;

//初始化双端队列
void initDeQueue(DeQueue& Q) {
    DNode* head = (DNode*)malloc(sizeof(DNode));    //申请头结点
    //初始化头结点
    head->next = NULL;
    head->prior = NULL;
    Q.front = head; //队列的头指针和尾指针都指向头结点
    Q.rear = head;
    Q.size = 0;     //队列大小初始化为0
}
```

```
//判断队列是否为空
bool isEmpty(DeQueue Q) {
    return Q.size == 0;
}

//向队头添加元素e
void pushHead(DeQueue& Q, ElemType e) {
    DNode* node = (DNode*)malloc(sizeof(DNode));    //申请结点空间
    node->data = e; //为新元素结点赋值
    node->next = NULL;  //新结点结点前序后继初始化指向空
    node->prior = NULL;
    if (Q.front == Q.rear) {    //如果此时队列还无元素，插入第一个元素时
        node->prior = Q.front;      //新结点的前序是头结点
        Q.front->next = node;       //头结点的后继是当前新结点
        Q.rear = node;              //队尾来到新结点的位置
    }
    //如果入队的元素不是第一个
    else {
        node->next = Q.front->next; //当前结点的后继是头结点的后继
        Q.front->next->prior = node;    //头结点的后继的前序是当前节点
        Q.front->next = node;       //头结点的后继是当前结点
        node->prior = Q.front;      //当前结点的前序是头结点
    }
    Q.size++;           //队列长度+1
}

//从队头移除元素并返回
ElemType popHead(DeQueue& Q) {
    if (isEmpty(Q)) {       //如果队列为空返回-999代表空
        return -999;
    }
    DNode* removed = Q.front->next;     //记录被删除结点
    ElemType x = removed->data;         //记录被删除结点的元素值
    if (removed->next == NULL) {        //如果队列只有一个元素了
        Q.front->next = NULL;           //直接让头结点的next指向NULL
        Q.rear = Q.front;               //队列的尾指针指向头结点表示队列已经为空了
    }
    else {
        Q.front->next = removed->next;  //头结点的后继指向被删除结点的后继
        removed->next->prior = Q.front; //被删除结点的后继的前序指向头结点
    }
    free(removed);  //释放空间队列长度-1
    Q.size--;
    return x;
}

//向队尾添加元素e
void pushTail(DeQueue& Q, ElemType e) {
    DNode* node = (DNode*)malloc(sizeof(DNode));    //申请新结点空间
    node->data = e;     //给新结点赋值
    node->next = NULL;  //新结点的next先指向空
    Q.rear->next = node;    //队尾指针的next指向新结点
    node->prior = Q.rear;   //新结点的prior指向尾指针结点
    Q.rear = node;          //尾指针来到新结点位置
```

```
    Q.size++;   //队列长+1
}

//从队尾删除元素并返回
ElemType popTail(DeQueue& Q) {
    if (isEmpty(Q)) {
        return -999;
    }
    DNode* removed = Q.rear;    //记录被删除结点
    ElemType x = removed->data; //记录被删除结点的元素值
    Q.rear->prior->next = NULL; //队尾指针结点的前序的后继指向空，即倒数第二个结点的next
指向空
    Q.rear = Q.rear->prior; //队尾指针来到倒数第二个结点处
    free(removed);      //释放最后一个结点空间
    Q.size--;
    return x;
}

//获取队头元素
ElemType peekHead(DeQueue Q) {
    if (isEmpty(Q)) {
        return -999;
    }
    return Q.front->next->data; //第一个元素是头结点的next
}

//获取队尾元素
ElemType peekTail(DeQueue Q) {
    if (isEmpty(Q)) {
        return -999;
    }
    return Q.rear->data;
}

//从前往后打印队列元素
void printDeQueuefromHead(DeQueue Q) {
    if (isEmpty(Q)) {
        return;
    }
    DNode* p = Q.front->next;
    while (p) {
        printf("%d ", p->data);
        p = p->next;
    }
    printf("\n");
}

//从后往前打印队列元素
void printDeQueuefromTail(DeQueue Q) {
    if (isEmpty(Q)) {
        return;
    }
    DNode* p = Q.rear;
    while (p!=Q.front) {
        printf("%d ", p->data);
```

```
        p = p->prior;
    }
    printf("\n");
}

//获取队列长度
int getSize(DeQueue Q) {
    return Q.size;
}
```

3.5栈和队列的相关题目

01.设单链表的表头指针为L,结点结构由data和next两个域构成，其中data域为字符型。
试设计算法判断该链表的全部n个字符是否中心对称。例如xyx、xyyx都是中心对称。

```
bool isSymmetry(LinkedList L) {
    int len = length(L) / 2;    //计算表长的一半，将一半元素入栈
    LNode* p = L;
    SqStack S;
    initStack(S);
    while (len > 0) {   //将一半元素入栈
        push(S, p->data);
        p = p->next;
        len--;
    }
    if (length(L) % 2 == 1) {   //如果表长为奇数，跳过最中间的一个
        p = p->next;
    }
    while (p) {
        if (pop(S) != p->data) {    //从右半边的第一个开始和栈中元素依次比较
            return false;       //如果有一个不满足栈顶元素和当前元素值相等返回false
        }
        p = p->next;
    }
    return true;
}
```

02.假设以I和O分别表示入栈和出栈操作。栈的初态和终态均为空，入栈和出栈的操作序列可表示为仅由I和O组成的序列，可以操作的序列称为合法序列，否则称为非法序列。

1)下面所示的序列中哪些是合法的?

A. IOIIOIOO

B. IOOIOIIO.

C. IIIOIOIO

D. IIIOOIO0

2)通过对1) 的分析，写出一个算法，判定所给的操作序列是否合法。若合法，返回

true,否则返回false (假定被判定的操作序列已存入一维数组中)。
***/**

```
bool isLegal(SqList L) {
    SqStack S;
    initStack(S);
    for (int i = 0; i < L.length; i++) {
        if (L.data[i] == 0) {   //用0表示O，1表示I。如果要出栈先看是不是空栈
            if (isEmpty(S)) {   //是空栈返回false
                return false;
            }
            else {      //不是空栈直接把栈顶元素出站
                pop(S);
            }
        }
        else {      //入栈元素（随便入个元素模拟即可）
            push(S, -1);
        }
    }
    return isEmpty(S);  //最后如果栈为空证明合法。否则不合法
}
```

03.若希望循环队列中的元素都能得到利用，则需设置一个标志域tag,并以tag的值为0
或1来区分队头指针front和队尾指针rear相同时的队列状态是"空"还是"满"。
试编写与此结构相应的入队和出队算法。

```
typedef int ElemType;
typedef struct {
    ElemType data[MAXSIZE];
    int front;  //队头指针
    int rear;   //队尾指针
    int tag;    //用来标记队空还是队满 0 表示队空，1表示堆满
}Queue;

//初始化队列
void initQueue2(Queue& Q) {
    //对数据元素进行初始化，防止出现脏数据
    for (int i = 0; i < MAXSIZE; i++) {
        Q.data[i] = 0;
    }
    Q.front = 0;    //初始化队头指针指向0索引
    Q.rear = 0;     //队尾指针也指向0索引
    Q.tag = 0;      //初始化时tag=0表示队列为空
}

//判断队列是否已满

bool isFull2(Queue Q) {
    return Q.tag == 1 && Q.front == Q.rear;
}

//判断队列是否为空
```

```
bool isEmpty2(Queue Q) {
    return Q.tag == 0 && Q.front == Q.rear;
}

//元素x入队
void enQueue2(Queue& Q, ElemType x) {
    if (isFull2(Q)) {
        printf("队列已满\n");
        return;
    }
    Q.data[Q.rear] = x;
    Q.rear = (Q.rear + 1) % MAXSIZE;
    //入队元素之后可能导致队列变满，如果满注意修改tag
    if (Q.rear == Q.front) {
        Q.tag = 1;
    }
}

//队头元素出队，并返回
ElemType deQueue2(Queue& Q) {
    if (isEmpty2(Q)) {
        printf("队列已空\n");
        return -999;
    }
    ElemType x = Q.data[Q.front];
    Q.front = (Q.front + 1) % MAXSIZE;
    //出队后可能导致队列变空
    if (Q.front == Q.rear) {
        Q.tag = 0;
    }
    return x;
}
```

04.Q是一个队列，S是一个空栈，实现将队列中的元素逆置的算法。

```
//借助栈
void inverQueue(Queue& Q) {
    SqStack S;
    initStack(S);
    while (!isEmpty(Q)) {   //如果队列不为空
        ElemType x = deQueue(Q);    //依次出队并将出队元素进栈
        push(S, x);
    }
    while (!isEmpty(S)) {       //如果栈不为空
        enQueue(Q, pop(S));     //出栈并将出栈元素进队
    }
}

//利用递归
void inverQueue2(Queue& Q) {

    if (isEmpty(Q)) {       //终止条件
        return;
    }
    ElemType x = deQueue(Q);    //元素出队
```

```
    inverQueue2(Q);                 //递归后续队列元素
    enQueue(Q, x);                  //从前往后入队
}
```

第四章 树

4.1 二叉树的顺序存储

```
#define MAXSIZE 16
typedef int ElemType;
typedef struct {
    ElemType data[MAXSIZE];
    int size;
}Tree;

//初始化二叉树
void initTree(Tree& T) {
    for (int i = 0; i < MAXSIZE; i++) {
        T.data[i] = 0;  //假设0表示空节点
    }
    T.size = 0;
}
//创建二叉树
void createTree(Tree& T) {
    ElemType x;
    scanf("%d", &x);
    int i = 0;
    while (x != 999) {
        T.data[i++] = x;
        T.size++;
        scanf("%d", &x);
    }
}
//打印二叉树
void printTree(Tree T) {
    for (int i = 0; i < T.size; i++) {
        printf("%d ", T.data[i]);
    }
    printf("\n");
}
```

01.已知一棵二叉树按顺序存储结构进行存储，设计一个算法，求编号分别为i和j的两个
结点的最近的公共祖先结点的值。

```
ElemType findCommonAncestors(Tree T, int i, int j) {
    if (T.data[i] != 0 && T.data[j] != 0) { //两个结点都存在在进行后续操作
        //父节点的下标为 x/2
        /*
                1               对应数组[1,2,3,4,5,6,7]
```

```
                /   \            对应下标[0,1,2,3,4,5,6]
               2     3
              / \   / \          注意：如果是下标从0开始，
             4   5 6   7          i的左孩子下标是 i*2 +1   右孩子下标是i*2+2
                                  i的父节点是(i-1)/2
                                 如果下标从1开始
                                  i的左孩子是i*2,右孩纸是i*2+1
                                  i的父节点是i/2
        */
        while (i != j) {
            if (i > j) {
                i /= 2;
            }
            else {
                j /= 2;
            }
        }
        return T.data[i];
    }
    return 0;   //0代表为空
}
```

4.2二叉树的链式存储

二叉树的结构

```
typedef int ElemType;
typedef struct BiTNode {
    ElemType data;      //数据域
    BiTNode* lchild;    //左孩子指针
    BiTNode* rchild;    //右孩子指针
}BiTNode,*BiTree;
```

创建二叉树

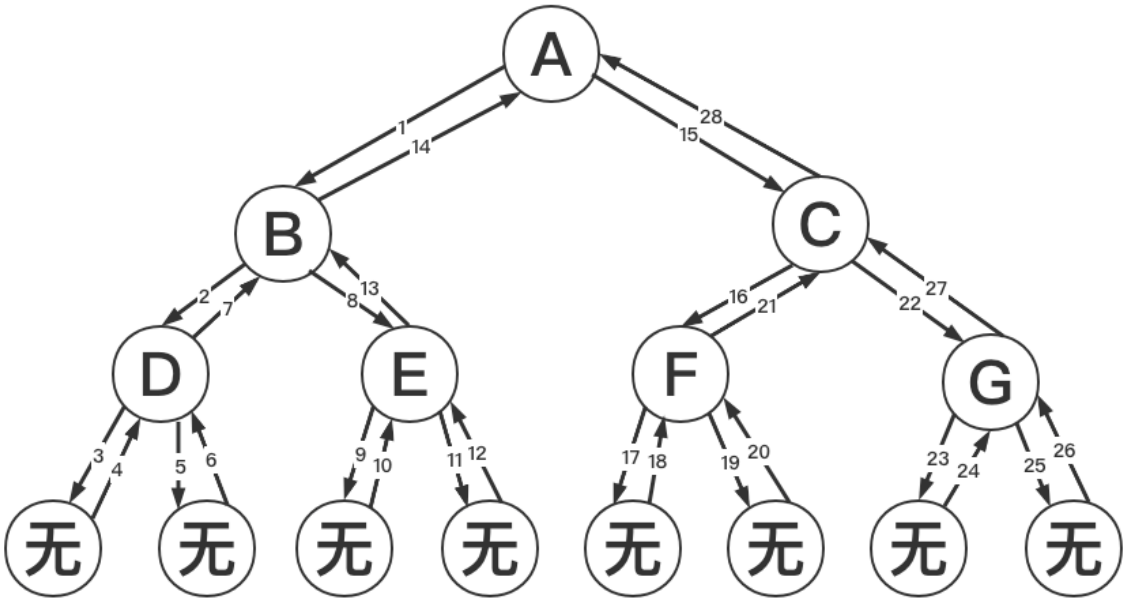
```
//前序创建二叉树
void createTree(BiTree& T) {
    ElemType x;
    scanf("%d", &x);
    if (x == 0) {       //用0代表空结点
        return;
    }
    else {
        T = (BiTNode *)malloc(sizeof(BiTNode)); //创建结点结构
        T->data = x;
        T->lchild = NULL;
        T->rchild = NULL;
        createTree(T->lchild);  //递归创建左子树
        createTree(T->rchild);  //递归创建右子树
    }
}
```

二叉树的前中后序遍历（递归）

```
//前序遍历递归
void preOrder(BiTree T) {
    if (T == NULL) {
        return;
    }
    printf("%d ", T->data);
    preOrder(T->lchild);
    preOrder(T->rchild);
}

//中序遍历递归
void inOrder(BiTree T) {
    if (T == NULL) {
        return;
    }
    inOrder(T->lchild);
    printf("%d ", T->data);
    inOrder(T->rchild);
}

//后序遍历递归
void postOrder(BiTree T) {
    if (T == NULL) {
        return;
    }
    postOrder(T->lchild);
    postOrder(T->rchild);
    printf("%d ", T->data);
}
```

大家可以发现，前中后序遍历时只有打印位置不同，其他代码完全一样，为什么在不同的位置打印会得到3种正确的遍历结果呢？为此，我们先了解一下递归序。

```
void f(BiTree T) {
    if (T == NULL) {
        return;
    }
    /*
        递归序：每个函数都会有三次机会来到。
        第一次来打印先序，第二次来打印中序，第三次来打印后序
    */
    //前序
    f(T->lchild);
    //中序
    f(T->rchild);
    //后序
}
```



如果在三个位置都进行打印的话，输出结果是：A B D D D B E E E B A C F F F C G G G C A

先序遍历：第一次经过时打印，先序序列：A B D E C F G

中序遍历：第二次经过时打印，中序序列：D B E A F C G

后序遍历：第三次经过时打印，后序序列：D E B F G C A

前序遍历二叉树（非递归）

```
//前序遍历非递归
void preOrderNoRecursion(BiTree T) {
    SqStack S;
    initStack(S);
    BiTNode* p = T;     //p作为树的遍历指针
    while (p || !isEmpty(S)) {  //如果p不为空或者栈中还有元素
        if (p) {
            printf("%d ", p->data); //打印当前结点的值
            push(S, p);             //当前元素
            p = p->lchild;          //先一直走左子树
        }
        else {
        //如果此时遍历到了空结点，证明这条路一路走到头了，弹出后再去遍历右子树
            BiTNode* top = pop(S);
            p = top->rchild;
        }
    }
}
```

中序遍历二叉树（非递归）

```
//中序遍历非递归
void inOrderNoRecursion(BiTree T) {
    SqStack S;
    initStack(S);
    BiTNode* p = T;
```

```
    while (p || !isEmpty(S)) {  //如果p不为空或者栈中还有元素
        if (p) {
            push(S, p);         //还是一路向左走
            p = p->lchild;
        }
        else {
            BiTNode* top = pop(S);  //左边到头了，弹出栈顶元素并打印
            printf("%d ", top->data);
            p = top->rchild;    //接着去走弹出元素的右子树的路
        }
    }
}
```

后序遍历二叉树（非递归）

```
//后序遍历非递归
void postOrderNoRecursion(BiTree T) {
    SqStack S;
    initStack(S);
    BiTNode* p = T;
    BiTNode* last = NULL;       //用来记录最近一次出栈的元素
    while (p || !isEmpty(S)) {
        if (p) {
            push(S, p);
            p = p->lchild;
        }
        else{
            BiTNode* top = peek(S);     //获取栈顶元素
            if (top->rchild == NULL||top->rchild == last) {
                last = pop(S);  //出栈并更新最近出栈的元素
                printf("%d ", top->data);
            }
            else {
                p = top->rchild;
            }
        }
    }
}
```

层次遍历二叉树

```
//层次遍历
void levelOrder(BiTree T) {
    Queue Q;
    initQueue(Q);
    BiTNode* p = T;
    if (p) {
        enQueue(Q, p);      //先将根节点元素进队
        while (!isEmpty(Q)) {
            BiTNode* head = deQueue(Q); //出队
            printf("%d "T, head->data); //打印出队元素
            if (head->lchild) {         //如果出队元素有左孩子先将左孩子入队
```

```
                enQueue(Q, head->lchild);
            }
            if (head->rchild) {      //如果出队元素有右孩子再将左孩子入队
                enQueue(Q, head->rchild);
            }
        }
    }
}
```

4.3二叉树题目

01.试给出二叉树的自下而上、从右到左的层次遍历算法。

```
void inverLevelOrder(BiTree T) {
    //用将层次遍历结果放到栈中，在依次打印栈中的元素即可
    Queue Q;
    SqStack S;
    initQueue(Q);
    initStack(S);
    BiTNode* p = T;
    if (p) {
        enQueue(Q, p);
        while (!isEmpty(Q)) {
            BiTNode* head = deQueue(Q);//队头出队
            push(S, head);
            if (head->lchild) {
                enQueue(Q, head->lchild);    //有左孩子左孩子入队
            }
            if (head->rchild) {
                enQueue(Q, head->rchild);    //有右孩子右孩子入队
            }
        }
    }
    while (!isEmpty(S)) {    //栈中此时存在的遍历层数遍历的逆序列，依次打印即可
        BiTNode* cur = pop(S);
        printf("%d ", cur->data);
    }
    printf("\n");
}
```

02.假设二叉树采用二叉链表存储结构，设计一个非递归算法求二叉树的高度。

```
//非递归
int getTreeHigh2(BiTree T) {
    if (T == NULL) {     //如果是空树直接返回0
        return 0;
    }
    //在进行树的层次遍历时统计树的高度
    Queue Q;
    initQueue(Q);
    BiTNode* p = T;
    enQueue(Q, p);
```

```
    int count = 0;  //count为最终要返回树的大小
    while (!isEmpty(Q)) {
        int size = Q.size;      //获取当前队列的长度，表示该层有几个结点。通过for循环表示结束了该层遍历，结束之后将层数count+1。
        for (int i = 0; i < size; i++) {
            BiTNode* head = deQueue(Q);
            if (head->lchild) {
                enQueue(Q, head->lchild);
            }
            if (head->rchild) {
                enQueue(Q, head->rchild);
            }
        }
        count++;
    }
    return count;
}
```

```
//递归
int getTreeHigh(BiTree T) {
    if (T == NULL) {    //如果是空树直接返回0
        return 0;
    }
    if (T->lchild == NULL && T->rchild == NULL) {   //如果是叶子结点高度为1
        return 1;
    }
    int lhigh = getTreeHigh(T->lchild);     //递归求左子树的高度
    int rhigh = getTreeHigh(T->rchild);     //递归求右子树的高度
    //总高度等于左右子树的最高度+自己的高度1
    return lhigh >= rhigh ? lhigh + 1 : rhigh + 1;
}
```

03.求树的最小深度

```
/*
    给定一个二叉树，找出其最小深度。
        最小深度是从根节点到最近叶子节点的最短路径上的节点数量。
        说明：叶子节点是指没有子节点的节点。
          例子    3              最小深度为2
                /   \
               9    20
                   /  \
                  15   7

                10             最小深度为3
               /  \
              15   20
              /     \
             25      30
*/
//递归版本
int minDepth(BiTree T) {
    if (T == NULL) {
        return 0;
```

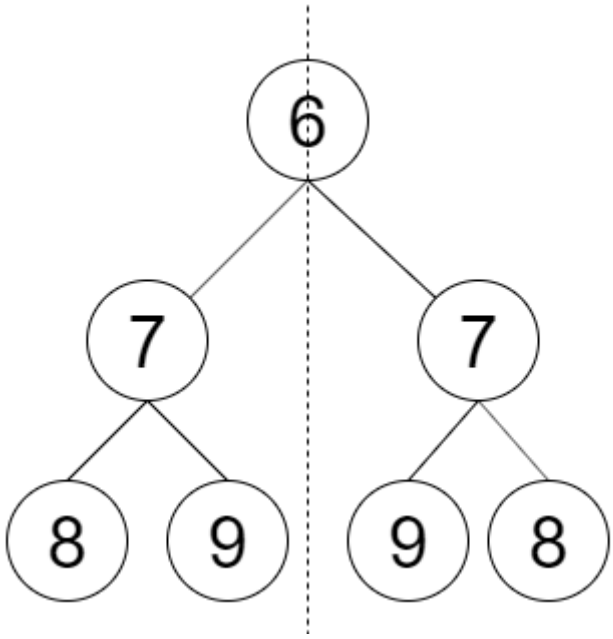
```
    }
    //如果是叶子结点高度为1
    /*if (T->lchild == NULL && T->rchild == NULL) {
        return 1;
    }*/
    int lhigh = getTreeHigh2(T->lchild);
    int rhigh = getTreeHigh2(T->rchild);
    if (lhigh == 0) {
        return rhigh + 1;
    }
    if (rhigh == 0) {
        return lhigh + 1;
    }
    return lhigh <= rhigh ? lhigh + 1 : rhigh + 1;

}
//非递归使用层次遍历实现
int minDepth2(BiTree T) {
    if (T == NULL) {
        return 0;
    }
    Queue Q;
    initQueue(Q);
    BiTNode* p = T;
    enQueue(Q, p);
    int count = 0;  //用来记录层数（高度）
    while (!isEmpty(Q)) {
        int size = Q.size;
        count++;
        for (int i = 0; i < size; i++) {
            BiTNode* head = deQueue(Q);
            if (head->lchild == NULL && head->rchild == NULL) {
                return count;
            }
            if (head->lchild) {
                enQueue(Q, head->lchild);
            }
            if (head->rchild) {
                enQueue(Q, head->rchild);
            }
        }
    }
    return count;
}
```

04.判断两棵树是否相同

```
/*
    04.给你两棵二叉树的根节点 T1 和 T2 ，编写一个函数来检验这两棵树是否相同。
        如果两个树在结构上相同，并且节点具有相同的值，则认为它们是相同的。
             1               1
           /   \           /   \
          2     3         2     3
```

```
            20                20
          /    \            /    \
                30          30
 */
bool isSameTree(BiTree T1, BiTree T2) {
    if (T1 == NULL && T2 == NULL) {
        return true;
    }
    if (T1 == NULL && T2 != NULL || T1 != NULL && T2 == NULL) {
        return false;
    }
    if (T1->data != T2->data) {
        return false;
    }
    return isSameTree(T1->lchild, T2->lchild) && isSameTree(T1->rchild, T2-
>rchild);
}
```

05.判断二叉树是不是对称的




```
//05.判断二叉树是不是对称的
bool checkSymmetricTree(BiTree T) {
    if (T == NULL) {    //认为空树是对称的
        return true;
    }
    return check(T->lchild, T->rchild);
}
bool check(BiTree left, BiTree right) {
    if (left == NULL && right == NULL) {    //如果左右孩子都是空返回true
        return true;
    }
    //如果左右孩子一个为空一个不为空返回false
```

```
    if (left == NULL && right != NULL || left != NULL && right == NULL) {
        return false;
    }
    //如果左右孩子的值不等返回false
    if (left->data != right->data) {
        return false;
    }
    return check(left->lchild, right->rchild) && check(left->rchild, right-
>lchild);
}
```

4.4先序线索二叉树

```
typedef int ElemType;
typedef struct ThreadNode{
    ElemType data;  //数据域
    ThreadNode* lchild;
    ThreadNode* rchild; //左右孩子指针
    int ltag, rtag; //前驱后继线索
}ThreadNode,*ThreadBinaryTree;
//初始化二叉树
void initTree(ThreadBinaryTree& T) {
    T = NULL;
}
//创建二叉树
void createTree(ThreadBinaryTree& T) {
    ElemType x;
    scanf("%d", &x);
    if (x == 0) {
        return;
    }
    else {
        T = (ThreadNode*)malloc(sizeof(ThreadNode));
        T->data = x;
        T->lchild = NULL;
        T->rchild = NULL;
        T->ltag = 0;
        T->rtag = 0;
        createTree(T->lchild);
        createTree(T->rchild);
    }
}

//普通二叉树的先序遍历
void printPreTree(ThreadBinaryTree T) {
    if (T != NULL) {
        printf("%d ", T->data);
        printPreTree(T->lchild);
        printPreTree(T->rchild);
    }
}
//先序遍历给二叉树线索化
void preThread(ThreadBinaryTree& p, ThreadBinaryTree& pre){
```

```
    if (p != NULL) {
        if (p->lchild == NULL) {
            p->lchild = pre;
            p->ltag = 1;
        }
        if (pre != NULL && pre->rchild == NULL) {
            pre->rchild = p;
            pre->rtag = 1;
        }
        pre = p;
        if (p->ltag == 0) {
            preThread(p->lchild, pre);
        }
        if (p->rtag == 0) {
            preThread(p->rchild, pre);
        }
    }
}
//创建先序搜索二叉树
void createPreThreadTree(ThreadBinaryTree& T) {
    ThreadNode* pre = NULL;
    if (T != NULL) {
        preThread(T, pre);
        pre->rchild = NULL;
        pre->rtag = 1;
    }
}

//线索二叉树的先序遍历(递归)
void printPreThreadTree(ThreadBinaryTree T) {
    if (T != NULL) {
        printf("%d ", T->data);
        if (T->ltag == 0) {
            printPreThreadTree(T->lchild);
        }
        else {
            printPreThreadTree(T->rchild);
        }
    }
}
//线索二叉树的先序遍历（非递归）
void printPreThreadTree2(ThreadBinaryTree T) {
    ThreadNode* p = T;
    while (p != NULL) {
        printf("%d ", p->data);
        if (p->ltag == 0) {
            p = p->lchild;
        }
        else {
            p = p->rchild;
        }
    }
}
```

题目1：写出在中序线索二叉树里查找指定结点在后序的前驱结点的算法。

```
ThreadNode* inpostPre(ThreadBinaryTree T, ThreadBinaryTree p) {
    ThreadNode* q = NULL;   //最终要找的p在后序中前驱结点
    //p结点有右孩子的时候
    if (p->rtag == 0) {
        q = p->rchild;
    }
    //p结点无右孩子但是有左孩子时
    else if (p->ltag == 0) {
        q = p->lchild;
    }
    //同时没有左右孩子
    else {
        while (p->ltag == 1 && p->lchild != NULL) {
            p = p->lchild;
        }
        if (p->ltag == 0) {
            q = p->lchild;
        }
    }
    return q;
}
```

第五章 图

5.1图的邻接矩阵存储

```
//无向图的邻接矩阵存储
#define MAXSIZE 16      //图的最大顶点个数
typedef int VertexType; //顶点类型
typedef int EdgeType;   //边类型

typedef struct {
    VertexType Vex[MAXSIZE];        //图的顶点集
    EdgeType Edge[MAXSIZE][MAXSIZE];    //图的边集
    int verNum;     //实际顶点的数量
    int edgeNum;    //实际边的数量
}Graph;

//图的初始化
void initGraph(Graph& G) {
    //给顶点集设置初始值
    for (int i = 0; i < MAXSIZE; i++) {
        G.Vex[i] = 0;
    }
    //给边集设置初始值
    for (int i = 0; i < MAXSIZE; i++) {
        for (int j = 0; j < MAXSIZE; j++) {
            G.Edge[i][j] = 0;
        }
    }
    //初始化顶点和边的个数都为0
    G.verNum = 0;
```

```
    G.edgeNum = 0;
}


//添加顶点
void addVertex(Graph& G, VertexType v) {
    if (G.verNum >= MAXSIZE) {
        printf("顶点集已满\n");
        return;
    }
    G.Vex[G.verNum++] = v;
}
//查找顶点v在图G的顶点集中的下标
int verIsExist(Graph G, VertexType v) {
    for (int i = 0; i < G.verNum; i++) {
        if (G.Vex[i] == v) {
            return i;
        }
    }
    return -1;
}

//添加边集
void addEdge(Graph& G, VertexType v1,VertexType v2,EdgeType weight) {
    //查找v1和v2顶点在图G顶点集中的下标索引
    int index1 = verIsExist(G, v1);
    int index2 = verIsExist(G, v2);
    if (index1 == -1 ||index2 == -1) {
        return;
    }
    //给边集设置权重，因为是无向图所以要进行双向设置
    G.Edge[index1][index2] = weight;
    G.Edge[index2][index1] = weight;    //如果是有向图把这一句去掉就行然后创建时所有的边
都要输入就行
    G.edgeNum++;
}

//打印图
void printGraph(Graph G) {
    printf("图G的顶点集：\n");
    for (int i = 0; i < G.verNum; i++) {
        printf("%d ", G.Vex[i]);
    }
    printf("\n");
    printf("图G的边集：\n\t");
    for (int i = 0; i < G.verNum; i++) {
        printf("v%d\t", G.Vex[i]);
    }
    printf("\n");
    for (int i = 0; i < G.verNum; i++) {
        printf("v%d\t", G.Vex[i]);
        for (int j = 0; j < G.verNum; j++) {
            printf("%d\t", G.Edge[i][j]);
        }
        printf("\n\n");
```

```
    }
    printf("\n");
}

//创建图的顶点集和边集
void createGraph(Graph& G) {
    VertexType v;
    printf("请输入要创建图G的顶点集：\n");
    scanf("%d", &v);
    while (v != 999) {
        addVertex(G, v);
        scanf("%d", &v);
    }
    printf("请输入要创建图G的边集（v1 v2 weight）：\n");
    VertexType v1, v2;
    EdgeType weight;
    scanf("%d%d%d", &v1, &v2, &weight);
    while (weight != 0) {
        addEdge(G,v1, v2, weight);
        scanf("%d%d%d", &v1, &v2, &weight);
    }
}
//针对无向图
//找顶点x的度
int degree(Graph G, VertexType x) {
    int index = verIsExist(G, x);
    if (index == -1) {
        return 0;
    }
    int count = 0;  //用来记录下顶点x的度数
    for (int i = 0; i < G.verNum; i++) {
        if (G.Edge[index][i] != 0) {
            count++;
        }
        /*if (G.Edge[i][index] != 0) {
            count++;
        }*/
    }
    return count;
}

//针对有向图
//找顶点x的入度
int inDegree(Graph G, VertexType x) {
    int index = verIsExist(G, x);
    if (index == -1) {
        return 0;
    }
    int count = 0;  //用来记录下顶点x的度数
    for (int i = 0; i < G.verNum; i++) {
        if (G.Edge[i][index] != 0) {
            count++;
        }
    }
    return count;
```

```
}

//找顶点x的出度
int outDegree(Graph G, VertexType x) {
    int index = verIsExist(G, x);
    if (index == -1) {
        return 0;
    }
    int count = 0;  //用来记录下顶点x的度数
    for (int i = 0; i < G.verNum; i++) {
        /*if (G.Edge[i][index] != 0) {
            count++;
        }*/
        if (G.Edge[index][i] != 0) {
            count++;
        }
    }
    return count;
}
```

5.2图的邻接表存储

```
#define MAXSIZE 16
typedef int VertexType; //顶点类型
//边表结点
typedef struct ArcNode {
    int adjVex;     //理解成顶点的data域
    ArcNode* next;  //指向下一个边表中的结点
}ArcNode;
//顶点表
typedef struct VNode {
    VertexType data;    //顶点信息
    ArcNode* first;     //指向边表
}VNode,adjList[MAXSIZE];

//图的邻接表存储
typedef struct {
    adjList vertices;   //邻接表
    int vexNum;     //顶点数
    int arcNum;     //边数
}Graph;


//初始化图
void initGraph(Graph& G) {
    for (int i = 0; i < MAXSIZE; i++) {
        G.vertices[i].data = 0;
        G.vertices[i].first = NULL;
    }
    G.arcNum = 0;
    G.vexNum = 0;
}
//添加顶点
void addVertex(Graph& G, VertexType vertex) {
    if (G.vexNum >= MAXSIZE) {
```

```
        printf("顶点集已满\n");
        return;
    }
    G.vertices[G.vexNum++].data = vertex;
}
//如果顶点在图中存在范围其在顶点表中的下标，不存在返回-1
int verIsExist(Graph G, VertexType v) {
    for (int i = 0; i < G.vexNum; i++) {
        if (G.vertices[i].data == v) {
            return i;
        }
    }
    return -1;
}

//添加边
void addEdge(Graph& G, VertexType v1, VertexType v2) {
    int index1 = verIsExist(G, v1);
    int index2 = verIsExist(G, v2);
    if (index1 == -1 || index2 == -1) {
        return;
    }
    /*//头插法创建
    //创建v1的边表结点
    ArcNode* node1 = (ArcNode *)malloc(sizeof(ArcNode));
    node1->adjVex = v2;
    node1->next = G.vertices[index1].first;
    G.vertices[index1].first = node1;
    //创建v2的边表结点
    ArcNode* node2 = (ArcNode*)malloc(sizeof(ArcNode));
    node2->adjVex = v1;
    node2->next = G.vertices[index2].first;
    G.vertices[index2].first = node2;
    G.arcNum++;*/

    //尾插法创建
    // 创建v1的边表结点
    ArcNode* node1 = (ArcNode*)malloc(sizeof(ArcNode));
    node1->adjVex = v2;
    node1->next = NULL;
    if (G.vertices[index1].first == NULL) {
        G.vertices[index1].first = node1;
    }
    else {
        ArcNode* tail = G.vertices[index1].first;
        while (tail->next != NULL) {
            tail = tail->next;
        }
        tail->next = node1;
    }
    // 创建v2的边表结点
    ArcNode* node2 = (ArcNode*)malloc(sizeof(ArcNode));
    node2->adjVex = v1;
    node2->next = NULL;
```

```
    if (G.vertices[index2].first == NULL) {
        G.vertices[index2].first = node2;
    }
    else {
        ArcNode* tail = G.vertices[index2].first;
        while (tail->next != NULL) {
            tail = tail->next;
        }
        tail->next = node2;
    }

    G.arcNum++;

}
//打印图
void printGraph(Graph G) {
    printf("图G的邻接表存储：\n");
    for (int i = 0; i < G.vexNum; i++) {
        printf("%d: ", G.vertices[i].data);
        ArcNode* cur = G.vertices[i].first;
        while (cur) {
            printf("%d->", cur->adjVex);
            cur = cur->next;
        }
        printf("\n");
    }
}
//创建图的顶点集和边集
void createGraph(Graph& G) {
    VertexType v;
    printf("请输入要创建图G的顶点集：\n");
    scanf("%d", &v);
    while (v != 999) {
        addVertex(G, v);
        scanf("%d", &v);
    }
    printf("请输入要创建图G的边集（v1 v2）：\n");
    VertexType v1, v2;
    scanf("%d%d", &v1, &v2);
    while (v1!=999||v2!=999) {
        addEdge(G, v1, v2);
        scanf("%d%d", &v1, &v2);
    }
}

//判断图G中v1、v2是否有边
bool adjacent(Graph G, VertexType v1, VertexType v2) {
    int index1 = verIsExist(G, v1);
    int index2 = verIsExist(G, v2);
    if (index1 == -1 || index2 == -1) {
        return false;
    }
    ArcNode* cur = G.vertices[index1].first;    //拿到v1顶点在页表中的第一个节点
    while (cur) {
        if (cur->adjVex == v2) {
```

```
            return true;
        }
        cur = cur->next;
    }
    return false;
}

//列出图G中与所有x邻接的顶点
void findAllNeighbors(Graph G, VertexType x, VertexType neighbors[]) {
    int index = verIsExist(G, x);
    if (index == -1) {
        printf("顶点%d不存在\n", x);
        return;
    }
    int count = 0;
    ArcNode* cur = G.vertices[index].first;
    while (cur) {
        neighbors[count++] = cur->adjVex;
        cur = cur->next;
    }
}

//求图G中顶点x的第一个邻接点，若有则返回顶点，没有返回-1
VertexType FirstNeighbor(Graph G, VertexType x) {
    int index = verIsExist(G, x);
    if (index == -1) {
        printf("顶点%d不存在\n",x);
        return -1;
    }
    ArcNode* cur = G.vertices[index].first;
    return cur ? cur->adjVex : -1;
}
//假设图G中顶点y是顶点x的一个邻接点，返回除y之外顶点x的下一个邻接点的顶点，如y是x的最后一个顶点，返回-1
VertexType nextNeighbor(Graph G, VertexType x, VertexType y) {
    int index = verIsExist(G, x);
    if (index == -1) {
        printf("顶点%d不存在\n", x);
        return -1;
    }
    ArcNode* cur = G.vertices[index].first;
    while (cur && cur->adjVex != y) {
        cur = cur->next;
    }
    if (cur && cur->next) {
        return cur->next->adjVex;
    }
    return -1;
}

//找x顶点的入度
int inDegree(Graph G, VertexType x) {
    int count = 0;
    for (int i = 0; i < G.vexNum; i++) {
        ArcNode* cur = G.vertices[i].first;
```

```
        while (cur) {
            if (cur->adjVex == x) {
                count++;
            }
            cur = cur->next;
        }
    }
    return count;
}

//找x顶点的出度
int outDegree(Graph G, VertexType x) {
    int index = verIsExist(G, x);
    if (index == -1) {
        return 0;
    }
    ArcNode* cur = G.vertices[index].first;
    int count = 0;
    while (cur) {
        count++;
        cur = cur->next;
    }
    return count;
}
```

第六章 排序

6.1冒泡排序

```
void swap(int arr[], int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

//外层循环是说明n个元素排好序需要经过n-1轮
    for (int i = n - 1; i > 0; i--) {
        bool flag = true;
        for (int j = 0; j < i; j++) {
            if (arr[j] > arr[j + 1]) {
                flag = false;
                swap(arr, j, j + 1);
            }
        }
        if (flag == true) {
            break;
        }
    }
    /*for (int i = 0; i < n - 1; i++) {
        for (int j = n - 1; j > i; j--) {
            if (arr[j] > arr[j - 1]) {
                swap(arr, j, j - 1);
```

```
            }
        }
    }*/
//递归写法
void bubbleSort2(int arr[], int n) {
    if (n == 1) {
        return;
    }
    for (int i = 0; i < n-1; i++) {
        if (arr[i] > arr[i + 1]) {
            swap(arr, i, i + 1);
        }
    }
    bubbleSort2(arr, --n);
}

void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
```

6.2选择排序

```
//简单选择排序
void selectSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int min = arr[i];
        int minIndex = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < min) {
                min = arr[j];
                minIndex = j;
            }
        }
        if (minIndex != i) {
            swap(arr, i, minIndex);
        }
    }
}
```

6.3直接插入排序

```
//插入排序
void insertSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j > 0 && arr[j] > arr[j - 1]; j--) {
            swap(arr, j, j - 1);
        }
    }
}
```

6.4折半插入排序

```
int insertIndex(int arr[],int target, int n) {
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (arr[mid] <=  target) {
            low = mid + 1;
        }
        else {
            high = mid - 1;
        }
    }
    return low;
}
//折半插入排序
void binaryInsertSort(int arr[], int n) {
    for (int i = 1; i < n; i++) {
        int index = insertIndex(arr, arr[i], i);
        for (int j = i; j > index; j--) {
            swap(arr, j, j - 1);
        }
    }
}
```

6.5希尔排序

```
//希尔排序
void shellSort(int arr[], int n) {
    for (int gap = n / 2; gap >= 1; gap /= 2) {
        for (int i = gap; i < n; i++) {
            for (int j = i; j >= gap && arr[j] < arr[j - gap]; j -= gap) {
                swap(arr, j, j - gap);
            }
        }
    }

}

//希尔排序
void shellSort(int arr[], int n) {
    for (int gap = n / 2; gap >= 1; gap /= 2) {
        for (int i = 0; i < n - gap; i++) {
            for (int j = i + gap; j >= gap && arr[j] < arr[j - gap]; j -= gap) {
                swap(arr, j, j - gap);
            }
        }
    }
}
```

如果代码看不懂，配套讲解（B站：执念讶）持续更新中.....