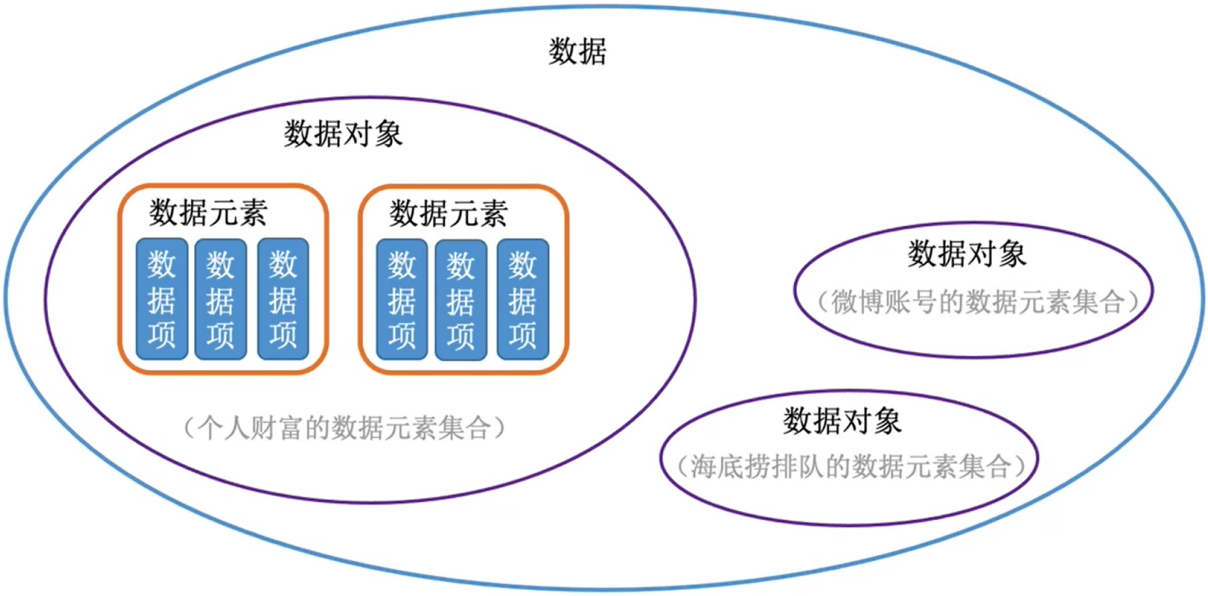
1-绪论

基本概念

- 数据、数据元素、 数据对象、数据项



要点

数据结构的三要素： 数据的逻辑结构、数据的存储结构、数据的运算

- 逻辑结构
  - 线性结构（线性表、栈结构、队列、串、数组等）
  - 非线性结构（集合、树形结构、图状结构等）
- 存储结构
  - 顺序存储、链式存储、索引存储、散列存储（又叫哈希存储）

算法

基本概念： 算法是对特定问题求解步骤的一种描述，他是指令的有限序列，每条指令表示一个或多个操作。
算法的五个重要特性：有穷性、确定性、可行性、输入、输出
算法效率的度量： 时间复杂度 和 空间复杂度

时间复杂度:
`O(1) < O(log~2~n) < O(n) < O(nlog~2~n) < O(n^2^) < O(n^k^) < O(2^n^) < O(n!) < O(n^n^)`
记忆方法： 常 对 幂 指 阶
时间复杂度如何计算:

- 找到基本操作行（如最深层循环）
- 分析执行次数xui问题规模n的关系 x=f(x)
- x的数量级就是算法的时间复杂度

* 若有最坏时间复杂度、最好时间复杂度、平均时间复杂度的情况下，通常考虑平均和最差时间复杂度
加法规则 和 **乘法规则**

- 加法规则： $T(n) = T_1(n) + T_2(n) = O(max(f(n), g(n))$
- 乘法规则： $T(n) = T_1(n) + T_2(n) = O(f(n) \times g(n))$

空间复杂度:
定于： 算法的空间复杂度**S(n)**定义为该算法所耗费的存储空间 S(n) = O(g(n))
内存存放： 1、程序代码内存 2、数据内存（空间复杂度考虑对象）

2-线性表

基本概念

数据结构的三要素： 逻辑结构（数据结构的定义）、数据的运算（基本操作）、存储结构（物理结构）
线性表（linear list）的定义：线性表是具有**相同数据类型**的n个数据元素的**有限数列**，其中n为表长，当n = 0时线性表是一个空表，若用L命名线性表，一般表示为：$L = (a_1, a_2, \ldots\ldots a_n)$

注意事项:

- a_1是唯一的“第一个”数据元素，又叫**表头元素**；a_n是唯一的“最后一个“数据元素，又叫**表尾元素**
- 除表头元素外，每个元素**有且仅有一个直接前驱**；除表尾元素外，每个元素**有且只有一个直接后继。**（直接前驱/前驱 ，直接后继/后继）
- **线性表中元素的位序是从1开始的，而数组元素的下标是从0开始的。**

线性表的特点

- 表中元素的个数有限
- 表中元素具有逻辑上的顺序性，表中元素有其先后次序
- 表中元素都是**数据元素**，每个元素都是单个元素
- 表中元素的数据类型都相同，意味着**每个元素占有相同大小的存储空间**
- 数据元素具有抽象性，此处只考虑逻辑结构，存什么不关心

线性表的基本操作:

- InitList（&L）：初始化表。构建一个空的线性表，分配内存空间。
- Length（L）：求表长，返回线性表L的长度
- LocateElem（L，i）：按照值查找
- GetElem（L，i）按位查找
- ListInsert（&L，i，e）：插入操作，在表L中的第i个位置插入指定元素e
- ListDelete（&L，i，&e）：删除操作，删除表L中第i个位置的元素，并用e返回删除元素的值
- PrintList（L）：按照前后顺序输出表L的所有元素值
- Empty（L）：判断表L是否为空表，是则返回True，否则返回False
- DestroyList（&L）：销毁操作，销毁线性表，并释放内存空间

注意事项：

1. 对数据的操作，无非就是：创、销、增、删、改、查
2. C语言函数定义：`<返回值类型> 函数名 （参数1，参数2）`
3. 实际开发中，可根据实际需求定义其他操作
4. 函数名参数的形式，命名应该具有可读性
5. **传入引用”&“，将参数传回，应当明确什么时候需要传回**，同时，”&“为C++的引用，*C语言中应采用指针* 。

为什么要定义数据操作的剧本操作?
答：1、团队编程，使用方便（封装） 2、将常用函数封装，避免不必要的错误。

顺序表

定义：用顺序存储结构实现的线性表
特点：逻辑顺序与物理顺序相同，都是顺序结构。

- C语言中，`sizeof（ElemType）`可返回贝格数据元素所占存储空间的大小

(***)一维数组可以是**静态分配**的，也可以是**动态分配**的。在**静态分配**时由于数组的大小和空间事先已经固定，(内存大小不可更改)。
而在**动态分配**时，存储数组的空间是在程序执行过程中通过动态分配语句分配的，一旦数据空间占满，就需要另外开辟一块更大的**连续存储空间**，从而达到扩充存储数组空间的目的，不需要为线性表一次性划分所有空间。
C的初始动态分配语句：
`L.data = (ElemType*) malloc(sizeof(ElemType) * InitSize);` 其中ElemType代表指针指向内存存储的类型
(**malloc**) 和 (**free**) 为C中内存动态申请和释放，使用该函数之前需要调用 `#include <stdio.h>`

- 顺序表最主要的特点是**随机访问**，即通过首地址和元素序号可在时间O(1)内找到指定的元素
- 顺序表的存储密度高，每个结点只存储数据元素
- 顺序表逻辑上相邻的元素，物理上也相邻，所以**插入和删除操作需要移动大量元素**

假设线性表的元素类型为ElemType，则线性表的顺序存储类型描述为：

```
# define InitSize 100
typedef struct{
    int MaxSize, length //数组的最大容量和当前元素个数
}SeqList    //动态分配数组顺序表的类型定义
```

线性表动态存储类型描述为：

```
# define InitSize 10  //初始列表长度为10
typedef struct{ //定义新的类型 包括内存指针、最大容量和当前长度
    int *data;
    int MaxSize;
    int length;
}SeqList;

Void InitList(SeqList &L){
    //用 malloc 函数申请一篇连续的存储空间
    L.data = (int *)malloc(InitSize * sizeof(int));
    L.length = 0;
    L.MaxSize = InitSize;
}

// 增加动态数组的长度 功能封装
Void InceaseSize(SeqList &L, int len){  // len代表需要增加的长度
    int *p = L.data;
    L.data = (int *)malloc((L.MaxSize+len) * sizeof(int));
    for (int i=0; i<L.length; i++) {
        L.data[i] = p[i];
    }
    L.MaxSize = L.MaxSize + len;    //顺序表的最大长度增加 len
    free(p);    //释放原来的内存空间
}

int main(){
    SeqList L;
    InitList(L);    //初始化顺序表
    // ...往顺序表中插入元素或其他操作
    IncreaseSize(L, 5); //列表长度+5
    return 0;
}
```

顺序表上基本操作的实现

1、插入操作

注意事项：

1. 插入操作是将第i个元素以及气候的所有元素依次往后移动一个位置，腾出一个空位置插入新元素e，顺序表长度增加1，并且返回true
2. 若操作不合法，则返回false
3. 判断i的范围时，是L.length+1，因为在表尾可以添加元素
4. **在C语言中，尽管data定义为指针，但在使用 `L.data[i]` 调用表数据时，实际操作为：1、找到指针位置 2、取指针所指向的内存数据大小的内存大小**

时间复杂度：

- 最好情况： 在表尾插入，元素后移不执行，时间复杂度为O(1)
- 最坏情况： 在表头插入，所有元素后移，时间复杂度为O(n)
- 平均情况： 时间复杂度为O(n/2)>>O(n)

```
# define InitSize 10
```

```
typedef struct{
    int *data;
    int MaxSize;
    int length;
}SeqList;

bool InsertList(SeqList &L, i, e){
    // 代码的健壮性 》 1、判断插入位置i的范围是否有效  2、判断存储空间是否有空
    if (i<0 || i>L.length + 1)
        return false;
    if (L.length >= MaxSize)
        return false;
    // 数据插入
    for (int j=L.length; j>=i; j--)
        L.data[j] = L.data[j-1];
    L.data[i-1] = e;
    L.length++;
    return true;
}

int main(){
    SeqList L;
    InitList(L);    //初始化
    InserList(L,i,e);
    return 0;
}
```

2、删除操作

注意事项：

- 先将被删除元素赋给引用变量e，并将第i+1及其后所有元素往前移动一个位置，并且返回true

时间复杂度：

- 最好情况：删除表尾元素，时间复杂度为O(1)
- 最坏情况： 删除表头元素，时间复杂度为O(n)
- 平均情况： 时间复杂度为O(n/2) >> O(n)

```
typedef struct{
    int *data;
    int MaxSize;
    int length;
}SeqList;

bool DeleteList(Seqlist &L; i; &e){
    // 操作合法性
    if (i<0 || i>L.length)
        return false;
    // 删除操作
    e = L.data[i-1];
    for (int j=i; j<L.length;j++)
        L.data[j-1] = L.data[j];
    L.length--;
    return true;
}

```

```c
int main(){
    SeqList L;
    IinitList(L);
    // 删除操作
    if (DeleteList(L,i,e))
        printf("已删除第%d个元素，删除元素值为%d\n",i,e);
    else
        printf("位序%d不合法，操作失败\n",i);
    return 0;
}
```

3、按位查找
注意事项：

- **随机访问**特质，故时间复杂度为O(1)
- 动态存储和静态存储均为 `L.data[i]` 不论data是数组或指针，该指令均可行

```c
typedef struct{
    int data[i];
    int MaxSize;
    int length;
}SeqList

int GetElem(SeqList L, int i){
    return L.data[i-1];
}

int main(){
    SeqList L;
    InitList L;
    GetElem(i);
    return 0;
}
```

4、按值查找
注意事项：

- 在顺序表L中查找第一个元素值等于e的元素，并返回其位序
- **若表中存储的数据不是基本数据类型（int char double float 等）数据时，不可用"=="判断元素是否相当，若为其他结构体，需要分别比较结构体的每个元素是否相等**

时间复杂度：

- 最好情况： 查找元素在表头，时间复杂度为O(1)
- 最坏情况： 查找元素在表尾，时间复杂度为O(n)
- 平均情况：时间复杂度为O(n/2) >> O(n)

```c
typedef struct{
    int *data;
    int MaxSize;
    int length;
}SeqList;

int LocateElem(SeqList L, ElemType e){
```

```
    bool Judge=false;
    for (int i=0; i<L.length; i++)
    {
        if (e == L.data[i-1])
            return i+1;
            Judge = true;
    }
    if (!Judge)
        printf("查找元素没有出现在数列中！");
}
```

链表

单链表

- 顺序表： 可以随机存取表中任一元素，但插入和删除操作需要移动大量元素
- 链式表： 链式存储线性表时，不需要使用地址连续的存储单元，即不要求逻辑上相邻的元素在物理位置上也相邻，他们通过“链”建立元素之间的逻辑关系，因此插入和删除 操作不需要移动元素，而只需要修改指针，但也会失去顺序表可随机存取的有点。

定义：
单链表： 指通过一组任意的存储单元来存储线性表中的数据元素。 **对每个链表结点，除存放元素自身的信息之外，还需要存放一个指向其后继的指针。** *其中data为数据域，存放数据元素；next为指针域，存放其后继节点的地址。*
结点描述如下：

```
typedef struct LNode()  // 定义单链表结点类型
{
    ElemType data;  // 数据存放
    struct LNode *next; // 指针存放
}LNode, *LinkList;  // LNode代表结点，LinkList代表指针，但实际LNode、*LinkList两者所指向的都是struct LNode
```

- 用头指针来标识一个单链表，如单链表L，头指针为NULL时表示一个空表，同时，为了操作上的方便，在第一个结点之前附加一个结点，称为头结点。头结点可以不设任何信息，也可记录表长等信息。**头结点的指针域指向线性表的第一个元素结点。**

typedef关键字：
下述两种表达方式都可以，作用效果和上文程序中表达效果完全相同

```
typedef struct LNode *LinkList;

typedef struct LNode, LNode;
```

调用方式：

- 下述两种表达方式作用相同，都是定义L为指向链表的指针，但是前者偏向说明L是结点的指针，后者偏向说明L是链表的指针

```
LNode *L;    //声明指向单链表第一个结点的指针

LinkList L; //效果相同
```

不带头结点的单链表

```
typedef struct LNode
{
    ElemType data;
    struct LNode *next; // 结构指针，next指针指向struct结构
}LNode, *LinkList;

// 初始化空的链表
bool InitLIst(LinkList &L)  // &L取指针地址，直接修改主程序中L
{
    L = NULL;   // 表示空表
    return true;
}
```

带头结点的单链表:

```
typedef struct LNode
{
    ElemType data;
    struct LNode *next; // 结构指针，next指针指向struct结构
}LNode, *LinkList;

// 初始化空的链表
bool InitLIst(LinkList &L)
{
    L =  (LNode *) malloc(sizeof(LNode));   //分配头结点
    if (L == NULL)  //内存不足，分配失败
        return false;
    L.next = NULL;  // 头结点之后暂时没有结点
    return true;
}

void test()
{
    LinkList L; //声明一个指向单链表的指针

    InitList(L);    //初始化一个空表
}
```

对命令行 `L =  (LNode *) malloc(sizeof(LNode))` 解读:

- sizeof(LNode):首先操作符sizeof计算结构体LNode所占的空间

- malloc(sizeof(LNode)):用操作符sizeof计算完空间，再用malloc()函数,在内存中开辟结构体"LNode"那么大的空间，001x(假设)为该空间的地址。
- (LNode *)malloc(sizeof(LNode)):头部文件调用"#include<stdlib.h>"，malloc()函数返回类型为(void *)，由于 L 是指针变量，直接赋值肯定报错，所以要将malloc()函数的返回值，用(LNode *)强制装换为指针类型。
- L = (LNode *)malloc(sizeof(LNode)):将头结点的地址赋值给指针L，所以现在可以通过指针L访问该节点了。
- 使用 (LNode *)强制转换，代表指针L为LNode结构中的指针类型，即next

注意: `->` 和 `.` 运算符不相同，a->b 的含义是（*a).b，即a是指针，但是 `a.b` 中，a是结构

单链表的基本操作实现:
按位序插入（带头结点）

```
typedef struct LNode    // 定义结点
{
    ElemType data;
    struct LNode *next;
}LNoded, *LinkList;

bool ListInsert(LinkList &L, int i, ElemType e)
{
    if (i<1)
        return false;

    LNode *p;   // 指针p指向当前结点
    int j = 0;  // 当前p指向的是第几个结点
    p = L;  // 指针指向当前结点

    while (p != NULL && j < i-1)    // 循环找到第i-1个结点
    {
        p = p->next;
        j++;
    }

    if (p==NULL)    // i值不合法
        return false;

    LNode *s = (LNode *)malloc(sizeof(LNode));
    s->data = e;
    s->next = p->next;  // 下面两句的顺序不可调换，否则会出错
    p->next = s;

    return true;    // 插入成功
}
```

指定结点的后插操作

```
bool InsertNextNode (LNode *p, ElemType e)
{
    if (p == NULL)
        return false;
    LNode *s = (LNode *)malloc(sizeof(LNode));
    if (s==NULL)    //分配内存失败
        return false;
    s->data = e;
    s->next = p->next;
    p->next = s;

    return true;
}
```

指定位序删除

```
bool ListDelete(LInkList &L, int i, Elemtype &e)
{
    if (i<1)
        return false;
    LNode *p;
    int j=0;
    p = L;

    while (p!=NULL && j<i-1)
    {
        p = p->next;
        j++;
    }
    if (p==NULL)    // i的值不合法
        return false;

    LNode *q = p->next; //令q指向被删除结点
    e = q->data;    // 用e返回元素的值
    p->next = q->next;  // 将q结点从链中断开

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

指定结点删除

```
bool DeleteNode (LNode *p)
{
    if (p == NULL)
        return false;

    LNode *q = p->next;
    p->data = p->next->data;
    p->next = q->next;

    free(q);

    return true;
}
```

按位查找

```
LNode *GetElem(LinkList L, int i)   // 返回类型是LNode类型的指针
{
    if (i<0)
        return NULL;

    LNode *p;   // 指针指向当前扫描的结点
    int j=0;    // 当前p指向的是第几个结点
    p = L;  // L指向头结点，头结点是第0个结点

    while (p != NULL && j<i)
    {
        p = p->next;
        j++;
    }
    return p;
}
```

按值查找

```
LNode *LocateElem(LinkList L, ElemType e)
{
    LNode *p = L->next;

    while (p != NULL && p->data != e)
        p = p->next;

    return p;   // 返回该结点指针，否则返回NULL
}
```

头插法（逆向建立单链表）

```
LinkList List_HeadInsert (LinkList &L)  // 逆向建立单链表
{
    LNode *s; int x;
```

```
    L = (LinkList)malloc(sizeof(LNode));    // 创建头结点
    L->next = NULL; // 初始为空链表
    scanf("%d",&x);

    while(x!=9999)
    {
        s=(LNode *)malloc(sizeof(LNode));
        s->data = x;
        s-next = L->next;
        L->next = s;
        scanf("%d",&x);
    }
    return L;
}
```

尾插法（必须增加一个尾指针r，使其始终指向当前链表的尾结点）

```
LinkList List_TailInsert(LinkList &L)
{
    int x;
    L = (LinkList)malloc(sizeof(LNode));
    LNode *s, *r=L; //r为尾指针，s指向当前指针
    scanf("%d",&x);

    while(x!=9999)
    {
        s = (LNOde *)malloc(sizeof(LNode));
        s->data = x;
        r->next = s;
        r = s;
        scanf("%d",&x);
    }
    r->next = NULL;
    return L;
}
```

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，只能丛头开始遍历，访问后继节点时间复杂度为O(1)，访问前继结点时间复杂度为O(n)，依次引入双链表，**双链表结点只有两个指针prior和next，分别指向其前驱结点和后继节点。**

循环列表：

- 循环单链表： `表尾.next = 表头`
- 循环双链表： `表头.prior = 表尾` 和 `表尾.next = 表头`

3-栈和队列

栈（stack）

定义： 栈是只允许在一端进行插入或删除操作的线性表。即栈是一种线性表，但这种线性表只能在某一端进行插入和删除操作。栈的操作特性可以概括为 *后进先出* （Last In First Out，LIFO）。
栈的基本操作：

- InitStack(&S) : 初始化一个空栈S
- StackEmpty(S) : 判断一个栈是否为空，若栈S为空则返回true，否则返回false
- Push(&S, x) : 进栈，若栈S未满，则将x加入使之成为新的栈顶
- Pop(&S, &x) : 出栈，若栈S非空，则弹出栈顶元素，并用x返回
- GetTop(S, &x) : 读栈顶元素，若栈S非空，用x返回
- DestroyStack(&S) : 销毁栈，并释放栈S占用的存储空间

顺序栈的初始化

```
# define MaxSize 50
typedef struct {
    ElemType data[MaxSize];
    int top;
}SqStack;

void InitStack(SqStack &S)
{
    S.top = -1;
}
```

判断栈为空

```
bool StackEmpty(SqStack S){
    if (top == -1)
        return true;
    else
        return false;
}
```

进栈操作

```
bool Push(SqStack &S, ElemType x)
{
    if (top == MaxSize-1)   // 栈满，无法进栈
        return false;
    S.data[++S.top] = x;
    return true;
}
```

出栈操作

```
bool Pop(SqStack &S, ElemType &x)
{
    if (top == -1)  // 空栈
        return false;
    x = S.data[S.top--];
    return true;
}
```

读取栈顶元素

```
bool GetTop(SqStack S, ElmeType &x)
{
    if (top == -1)  // 空战
        return false;
    x = S.data[S.top];
    return true;
}
```

共享栈： 利用栈底位置不变的特性，可以让两个顺序栈共享一个一维数组的空间。当两个栈顶指针相邻（top1-top0=1）时，判断栈满。

栈的链式存储

- 链表的头插法和头结点删除 == 链栈的实现，所有操作和链表完全相同，仅限定头插和头删即可

```
typedef struct Linknode
{
    ElemType data;
    struct Linknode *next;
}*LiStack;
```

队列

定义： 队列，简称队，也是一种操作受限的线性表，只允许在表的一端进行插入，而在表的另一端研究删除。向队列中插入元素称为 *入队* 或 *进队* ，删除 元素称为 *出队* 或 *离队* 。其操作特性是**先进先出**（First In First Out）。
对头（Front）：允许删除的一端，又称队首；队尾（rear）：允许插入的一端；空列表：不含任何元素的空表。

队列的基本操作:

- InitQueue(&Q) : 初始化队列，构建一个空队列Q
- QueueEmpty(Q) : 判队列空，若空则返回true，否则返回false
- EnQueue(&Q,x)： 入队，若队列Q未满，将x加入，使之称为新的队尾
- DeQueue(&Q, x)： 出队，若队列Q非空，删除队头元素，并将值返回给x
- GetHead(Q, &x) : 读取对头元素，若队头非空，则将队头元素赋值给x

队列顺序存储

```
// 队列的创建
# define MaxSize 50 // 定义队列中元素的最大个数
typedef struct{
    ElemType data[MaxSize];
    int formt, rear;    // 对头指针和队尾指针
}SqQueue;

// 队列初始化操作
void InitQueue(SqQueue &Q)
{
   Q.front = Q.rear = 0;
}
```

循环队列

- 不能用 `Q.rear = MaxSize` 判断队列是否已满，以为出队的时候，front的值会改变，会出现假溢出的现象，故引入循环队列概念，即队列首尾相连。

```
Q.fornt = Q.rear = 0;   //初始状态
Q.front = (Q.rear+1)%MaxSize;   //队首指针加一
Q.rear = (Q.rea+1)%MaxSize;     //队尾指针加一
(Q.rear-Q.fornt+MaxSize)%MaxSize;   //队列长度
```

判断循环队列队满的情况：

1. 牺牲一个单元来区分队空和队满，入队是少用一个队列单元， `Q.fornt == (Q.rear+1)%MaxSize;`
2. 队列类型中加入元素个数的数据元素，当 `Q.Size==0` 和 `Q.size==MaxSize` 的时候分别判断队空和队满
3. 队列类型中接入tag数据成员，最近一次操作为删除则 `Q.tag=0` ;最近一次操作为插入则 `Q.tag=1`

循环队列的基本操作
判断队列是否为空

```
bool isEmpty(SqQueue Q)
{
    if (Q.rear == Q.front)
        return true;
    else
        return false;
}
```

入队操作

```
bool EnQueue(SqQueue &Q, ElmeType e)
{
    if ((Q.rear+1)%MaxSize==Q.front)    //队满
        return false;
    Q.data[Q.rear] = e;
    Q.rear = (Q.rear+1)%MaxSize;
    return true;
}
```

出队操作

```
bool DeQueue(SqQueue &Q, ElmeType &e)
{
    if (Q.front == Q.rear)  //队满
        return false;
    e = Q.data[Q.front];
    Q.front = (Q.front+1)%MaxSize;
    return true;
}
```

队列的链式存储结构的操作
链队列定义操作

```
typedef struct LinkNode{    // 定义链队列的结点
    ElemType data;
    struct LinkNode *next;
}LinkNode;

typedef struct {    // 定义链队列
    LinkNode *front, *rear;
}LinkQueue;
```

链队列初始化操作

```
void InitQueue(LinkQueue &Q){
    Q.front = Q.rear =(LinkNode *)(malloc(sizeof(LinkNode)));   //建立头结点
    Q.front->next = NULL;   //初始为空
}
```

链队列判断为空

```
bool IsEmpty(LinkQueue Q)
{
    if (Q.front == Q.rear)
        return true;
    else
        return false;
}
```

入队

```
bool InQueue(LinkQueue &Q, ElemType e)
{
    LinkNode *s = (LinkNode *)malloc(sizeof(LinkNode));
    s->data = e;
    s->next = NULL;
    Q.rear->next = s;
    Q.rear = s;
    return true;
}
```

出队

```
bool DeQueue(LinkQueue &Q, ElemType &e)
{
    if (Q.front == Q.rear)  // 空队列
        return false;
    LinkNode *p = Q.front->next;    // 指向当前需要处理的结点
    e = p->next;
    Q.front->next = p->next;
    if (Q.rear = p) // 原队列只有一个结点时候特殊处理，因为删除后，Q.rear指向NULL，为空队列，此时则需要进行此操作
        Q.rear = Q.front;
    free(p);
    return true;
}
```

双端队列：双端队列是指允许两端哦都看也i进行入队和出队操作的队列，将队列的两端分别称为前端和后端。

栈的引用、队列的引用：

```
// 太懒惰，LeetCode见
```

4-树与二叉树

树的基本概念

定义： 树是n(n>=0)个**结点**的有限集。当n=0时，称空树。特性：有且只有一个根节点， *树的根节点没有前驱，除根结点外的所有结点有且只有一个前驱 ， 树中所有的结点都可以有零个或多个后继。*

区分：根节点、分支结点、叶子结点

区分：祖先结点、子孙结点、双亲结点（父结点）、孩子结点、兄弟结点、堂兄弟结点

结点的度：树中一个 *结点的孩子个数称为该结点的度 ， 树中结点的最大度称为树的度* 。

- 结点的深度：从根结点开始自顶向下 逐层累加
- **结点的高度**：从叶结点开始自底向上逐层累加
- 树的高度：树中结点的最大层数

有序树和无序树：树中结点的各子树从左到右是有次序的，不能互换，称该树为有序树，否则为无序树。

路径和路径长度：树中两个结点之间的路径是由这两个结点之间所经过的结点序列构成的，而路径长度是路径上所经过的边的个数。**树中的路径是自上向下的，同一双亲的两个孩子之间不存在路径。**

森林： 森林是m棵互不相交的树的集合，只要把树的根节点删去就变成森林；反之，只要给m棵独立的树加上一个结点，并把这m棵树作为结点的字数，则森林就变成了树。

二叉树

二叉树 是一种特殊的树形结构，其特点是每个结点至多只有两颗子树（即二叉树中不存在度大于2的结点），并且二叉树的子树也有左右之分，其次序不能任意颠倒。**二叉树也同样是以递归的形式定义**
二叉树与度为2的有序树的区别：

- 度为2的树至少有3个结点，而二叉树可以为空
- 度为2的有序树的孩子的左右次序是相对于另一个孩子而言，若某个结点只有一个孩子，则这个孩子就无需区分其足有次序，而二叉树无论其孩子数是否为2，均需确定其左右次序，即二叉树的结点次序不是相对而言的，而是确定的。

几个特殊的二叉树：

- 满二叉树： 一颗高度为h，且含有2^h-1个结点的二叉树称为满二叉树，即树中的每层都含有最多的结点。
- 完全二叉树： 与高度为h的满二叉树编号一一对应的二叉树
- 二叉排序树：左子树上所有结点的关键字均小于根结点的关键字；右子树上的所有结点的关键字均大于根节点的关键字，且左子树和右子树又是一颗二叉排序树
- 平衡二叉树： 树上任意一个结点的左子树和右子树的深度只差不超过1（平衡二叉树的搜索效率更高）

二叉树的性质：

- $n_0 = n_2 + 1$

- 第i层至多有2^i-1个结点
- 高度为h的二叉树至多有2^n个结点

完全二叉树的性质：

- 具有n个结点的完全二叉树的高度h为【log(n+1)】$_{向下取整}$ + 1
- 可由结点个数推出度为0、1、2的节点个数。n = n_2+n_1+n_0, n = n_1 + 2n_2 + 1 两个公式计算得到

二叉树的存储结构：

顺序存储结构：

- 仅仅适合存放完全二叉树
  - i 的左孩子：2i
  - i 的右孩子：2i + 1
  - i 的父节点：[i / 2]$_{向下取整}$
  - i 所在的层次： [$\log_2$(n+1)]$_{向下取整}$ + 1

```
# define Maxsize 100

struct TreeNode{
    ElemType Data;  // 存储当前结点的关键字
    bool IsEmpty;   // 存储当前结点是否为空结点的布尔判断
}

TreeNode t[Maxsize];
```

链式存储：

```
typedef ElemType{   // 定义结点关键字类型
    int value;
}

typedef struct BiTNode{
    ElemType data;
    struct BiTNode *lchild, *rchild;
}BiTNode, *BiTree;

// 定义一颗空树
BiTNode root = NULL;

// 插入根结点
root = BiTNode (BiTree) malloc(sizeof(BiTNode));
root->data = {1};
root->lchild = NULL;
root->;child = NULL;

// 插入新结点
p = BiTNode (BiTNode) malloc(sizeof(BiTNode));
p->data = 2;
p->lchild = NULL;
p->rchild = NULL;
root->lchild = p;   //作为根结点的左孩子结点
```

若常常调用父节点，则可使用三叉链表：

```
typedef struct BiTNode{
    ElemType Data;
    struct BiTNode *lchild, *rchild;
    struct BiTNode *parent; //定义指针指向父节点
}BiTNode, *BiTree;
```

二叉树的前、中、后序遍历

```
// 前序遍历
void ProOrder(BiTree T){
    if (T != NULL)
    {
        visit(T);   // visit为自定义函数
        PreOrder(T->lchild);
        PreOrder(T->rchild);
    }
}


// 中序遍历
void InOrder(BiTree T){
    if (T != NULL)
    {
        InOrder(T->lchild);
        visit(T);
        InOrder(T->rchild);
    }
}


// 后序遍历
void PostOrder(BiTree T){
    if (T != NULL)
    {
        PostOrder(T->lchild);
        PostOrder(T->rchild);
        visit(T);
    }
}


// 不适用递归算法的中序遍历 ************自行理解*******************
void InOrder2(BiTree T){
    InitStack(S);   // 初始化栈S
    BiTree p = T;   // p是遍历指针
    while (p || IsEmpty(S)){
        if (p){
            Push(S,p);  //当前结点入栈
            p = p->lchild;  //当左孩子不为空，一路向左
        }
        else{
            Pop(S,p);   //栈顶元素出栈
```

```
            visit(p);   //访问出栈元素
            p = p->rchild;  //向右子树走
        }
    }
}
```

二叉树遍历的运用：求树的深度

```
// 计算树的深度
int TreeDepth(BiTree T){
    if (T == NULL)
    {
        return 0;
    }
    else
    {
        int r = TreeDepth(T->rchild);
        int l = TreeDepth(T->lchild);
        // 最大深度为  Max[TreeDepth(lchild), TreeDepth(rchild)] + 1
        return (r>l)  ?  r+1 :  l+1;
    }
}
```

二叉树的层次遍历

```
typedef struct BiTNode{ // 定义二叉树的结点（链式存储）
    char data;
    struct BiTNode *lchild, *rchild;
}BiTNode, *BiTree;


typedef struct LinkNode{    // 链式队列结点
    BiTNode *data;
    struct LinkNode *next;
}LinkNode;


void LevelOrder(BiTree T){
    InitQueue(Q);   // 初始化辅助队列Q
    BiTree p;
    EnQueue(p); // 将根节点入队
    while (!IsEmpy(Q))  // 若当前队不为空
    {
        DeQueue(Q,p);
        visit(p);
        if (p->lchild != NULL)  // 左孩子结点入队
            EnQueue(Q, p->lchild);
        if (p->rchild != NULL)  // 右孩子结点入队
            EnQueue(Q, p->rchild);
    }
}
```

仅给你一棵二叉树的前/中/后/层序遍历的一种，不能唯一确定一颗二叉树，需要前/后/层序遍历 + 中序遍历

线索二叉树

传统的二叉树中，仅能体现一种父子关系，不能直接得到结点在遍历中的前驱和后继。传统二叉树中，每次仅能从根节点开始遍历，不能从任一结点出发。**从而，引入线索二叉树正是为了加快查找结点前驱和后继的速度。**

线索二叉树的存储结构描述如下：

```
typedef struct ThreadNode{
    ElemType data;
    struct ThreadNode *lchild, *rchild; // 左、右孩子指针
    int ltag, rtag; // 左、右线索标志
}ThreadNode, *ThreadTree;
```

在传统中序遍历方法中，找到前驱结点和后继结点的方式为：

```
// 定义全局变量
BiTNode *p; // 指向目标结点
BiTNode *pre;   // 指向当前遍历的结点的前驱结点
BiTNode *final; //  存储最终结果指针

void InOrder(BiTree T){
    if (T != NULL)
    {
        InOrder(T->lchild);
        visit(T);
        InOrder(T->rchild);
    }
}

void visit(BiTree q){
    if (q == p) // 找到目标结点
        final = pre;
    else
        pre = q;    // 没有找到目标节点，前驱指针更新为现在的位置
}
```

中序线索化的代码实现：(难点)

```
typedef struct ThreadNode{          // 定义线索二叉树
    ElemType data;
    struct ThreadNode *lchild, *rchild; // 左、右孩子指针
    int ltag, rtag; // 左、右线索标志
}ThreadNode, *ThreadTree;

ThreadNode *pre = NULL; // 定义指针指向前驱结点，全局变量


```

```
void visit(ThreadNode *q){  // 使用该方法，在最后一个结点处，
    if (q->lchild == NULL)      // 若当前结点左子树为空，则建立线索
    {
        q->lchild = pre;
        q->ltag = 1; // 1代表有线索
    }
    if (pre != NULL && pre->rchild == NULL)     // 在遍历完成所有结点后，最后一个右结点需要手动的q->rchild = NULL，将它右结点置空
    {
        pre->rchild = q;
        pre->rtag = 1;
    }
    pre = q;
}


// 中序遍历二叉树，一边遍历一边线索化
void InThread(ThreadTree T){
    if (T != NULL)
    {
        InThread(T->lchild);
        visit(T);   // 判断并线索化当前结点
        InThread(T->rchild);
    }
}
```

先序线索化的代码实现:

```
typedef struct ThreadNode{          // 定义线索二叉树
    ElemType data;
    struct ThreadNode *lchild, *rchild; // 左、右孩子指针
    int ltag, rtag; // 左、右线索标志
}ThreadNode, *ThreadTree;

ThreadNode *pre = NULL;

void visit(ThreadNode *q){
    if (q->lchild == NULL)
    {
        q->lchild = pre;
        q->ltag = 1;
    }
    if (pre != NULL && q->rchild == NULL)
    {
        pre->rchild = q;
        q->rtag = 1;
    }
    pre = q;
}


void PreThread(ThreadTree T){
    visit(T);
    if (T->ltag == 0)   // lchild不是线索结点,若没有if语句，则会陷入死循环
        PreThread(T->lchild);
```

```
    PreThread(T->rchild);
}


void CreatPreThread(ThreadTree T){
    pre = NULL;
    if (T != NULL)
    {
        PreThread(T);
        if (pre->rchild == NULL)    // 定义最后一个结点的线索位
            pre->rtag = 1;
    }
}
```

后续线索化的代码实现：

```
typedef struct ThreadNode{          // 定义线索二叉树
    ElemType data;
    struct ThreadNode *lchild, *rchild; // 左、右孩子指针
    int ltag, rtag; // 左、右线索标志
}ThreadNode, *ThreadTree;


ThreadNode *pre = NULL;


void visit(ThreadNode *q)
{
    if (q->lchild == NULL)
    {
        q->lchild = pre;
        q->ltag = 1;
    }
    if (pre != NULL && q->rchild ==NULL)
    {
        pre->rchild = q;
        pre->rtag = 1;
    }
    pre = q;
}


void PostThread(ThreadTree T){
    if (T != NULL)
    {
        PostThread(T-lchild);
        PostThread(T-rchild);
        visit(T);
    }
}


void CreatPostThreadTree(ThreadTree T){
    pre = NULL;
    if (T != NULL)
```

```
    {
        PostThread(T);
        if (pre->rchild == NULL)        // 处理遍历的最后一个结点
            pre->rtag = 1;
    }
}
```

基于中序线索二叉树的遍历
优缺点分析： 在中序线索二叉树、先序线索二叉树、后序线索二叉树三者中，仅可用中序线索二叉树实现任一结点的前向遍历和后向遍历

代码实现：中序遍历（左 > 根 > 右）
寻找中序线索二叉树中指定结点*p的中序后继next

- 若 `p->rtag == 1, next = p->rchild`
- 若 `p->rtag == 0` 执行如下代码

```
// 找到以p为根的子树中，第一个被中序遍历的结点
ThreadNode *FirtsNode(ThreadNode *p){
    while (p->ltag == 0)    p = p->lchild;
}

// 在中序二叉树中找到结点p的后继结点
ThreadNode *NextNode(ThreadNode *p){
    // 右子树中最左下的结点
    if (p->rtag == 0) return FirstNode(p->rchild);  // 若rtag为0，则p的后继结点是右子树的最左下结点
    else return p->rchild;  // 若rtag==1直接返回后继线索
}

// 对中序线索二叉树进行中序遍历
void InOrder(ThreadNode *T){
    for (ThreadNode *p = FirstNode(T); p != NULL; p = NextNode(p)) // 从第一个结点开始，到最后一个结点结束
        visit(p);
}
```

中序线索二叉树找中序前驱结点

```
// 找到以p为根的子树中，最后一个被中序遍历的结点
ThreadNode *LastNode(ThreaNode *p){
    // 循环找到最右下结点（不一定是叶子结点）
    while (p->rtag == 0) p = p->rchild;
    return p;
}

// 在中序线索二叉树找到结点p的前驱结点
ThreadNode *PreNode(ThreadNode *p){
    // 左子树中最右下结点
    if (p->ltag == 0) return LastNode(p->lchild);
    else return p->lchild;
}
```

```
14
15 // 对中序线索二叉树进行逆向中序遍历
16 void RevInOrder(ThreadNode *T){
17     for (ThreadNode *p = LastNode(T); p != NULL; p=PreNode(p))
18         visit(p);
19 }
```

树、森林

树的存储结构： 1、双亲表示法； 2、孩子表示法；3、孩子兄弟表示法
1、双亲表示法： 每个结点中保存指向parent的“指针”

```
1  # define MAX_TREE_SEZI  100
2  typedef struct {     // 树的结点定义
3      ElemType Data;   // 数据元素
4      int parent;      // “指针”指向父结点，双亲在数组中的下标位置，根结点为0
5  }PTNode;
6
7
8  typedef struct {     // 树的类型定义
9      PTNode nodes[MAX_TREE_SIZE];     // 双亲表示
10     int n;       // 结点数(当前树中结点数量)
11 }PTree;
```

2、孩子表示法： 将每个结点的孩子结点都用单链表链接起来，根结点存储第一个孩子地址，每个孩子又存储其兄弟结点的地址

```
1  struct CTNode{  //定义每一个孩子结点
2      int child;  // 孩子结点在数组中的位置
3      struct CTNode *next;     // 指向下一个孩子
4  }CTNode;
5
6  typedef struct {     // 定义根结点
7      ElemType Data;
8      struct CTNode *FirstChild;  // 指向第一个孩子
9  }CTBox;
10
11 typedef struct {
12     CTBox nodes[MAX_TREE_SIZE];
13     int n, r; // 存储结点数量和根节点的数组下标（位置）
14 };
```

3、 孩子兄弟表示法： 又称二叉树表示法，每个结点包括三部分内容，结点值、指向结点的第一个孩子结点的指针 和指向结点下一个兄弟结点的指针

```c
typedef struct CSNode{  // 定义每个结点
    ELemType Data;
    struct CSNode *FirstChild, *NextSibling;    // 第一个孩子 和 右兄弟 指针
}CSNode, *CSTree;
```

树和森林的遍历

树	森林	二叉树
先根遍历	先序遍历	先序遍历
后根遍历	中序遍历	中序遍历

广度优先遍历： 层次遍历
深度优先遍历： 先序遍历、中序遍历、后续遍历

树的遍历包括： 先根遍历、后根遍历、层次遍历

```c
// 树的先根遍历
void PostOrder(TreeNode *R){
    if (R != NULL){
        visit(R);
        while (R还有下一个子树T)    // 根据不同的存储结构有不同的判断方法
            PostOrder(T);
    }
}

// 树的后根遍历
void PostOrder(TreeNode *R){
    if (R != NULL){
        while (R还有下一个子树T)
            PostOrder(T);
        visit(R);
    }
}
```

树的层次遍历：需要借助辅助队列完成层序遍历(与二叉树的层次遍历相似)

森林的遍历包括： 先序遍历森林、中序遍历森林

- 当森林转换成二叉树时，第一颗子树森林转换成左子树。剩余子树转换成右子树。且森林的先序遍历和中序遍历对应二叉树的先序和中序遍历

树与二叉树的应用

哈夫曼树
在许多应用中，树中的结点常常被赋予一个含有特殊意义的数值，该数值称为该结点的**权**。
从树的根到任意结点的路径长度、经过的边数与该结点上权值的乘积，称为该结点的**带权路径长度**。
树中所有叶子结点的带权路径长度之和称为**该树的带权路径长度**，记为WPL。

在含有n个带权叶子结点的二叉树中，其中带权路径长度（WPL）最小的二叉树称为**哈夫曼树**，也称**最优二叉树**。

哈夫曼树的构造：

1. 将这n个节点分别作为n颗只含一个结点的二叉树，构成森林F
2. 构造一个新结点，从F中选取两颗根结点权值最小的树作为新结点的左、右子树，并且将新结点的权值置为左、右子树上根节点的权值之和
3. 在F中删除刚选出的两棵树，同时将新得到的数加入F中
4. 重复步骤2和3，直到F中只剩下一棵树为止

哈夫曼编码：
即**可变长度编码**，同时，若没有一个编码是另一个编码的前缀，则称这样的编码为前缀编码，在使用过程中不会产生歧义

并查集：（Disjoint set）
并查集是一种简单的集合表示，通常支持3种操作

- Initial(S) : 初始化
- Union(S, Root1, Root2)： 把集合S中每个元素都初始化为只有一个单元素的子集合
- Find(S, x)：查找集合S中单元素x所在的子集合，并返回该子集合的根节点

将集合转换为树结构：

并查集最佳的实现方式是**双亲指针数组**作为存储结构时，因为这样更方便的找到parent结点，从而找到根结点
代码实现：

```
# define SZIE 100
int UFSets[SIZE];       // 集合元素数组（双亲指针数组）

// 初始化操作
void InitSet(int S[]){
    for (int i=0; i<SIZE; i++)  // 每个元素自成元素集合
        S[i] = -1;
}

// Find操作
void Find(int S[], int x){  // x为元素数组下标
    while(S[x] >= 0)     x = S[x];    // 寻找元素x的根
    x = S[x];
    return x;
}

// Union操作
void Union(int S[], int Root1, int Root2){
    if (Root1==Root2) return;       // 若当前元素属于同一个集合则不操作
    S[Root2] = Root1;   // 将根Root2连接到Root1下面
```

```
}
```

并查集的优化：

```
// 优化的核心原则是广度优先

// Union优化操作 使用根的绝对值表示当前树的速度
void Union(int S[], int Root1. int Root2){
    if (Root1 == Root2) return;
    if (S[Root2]>S[Root1]){     // Root2的结点数更少
        S[Root1] += S[Root2];   // 累加结点数量
        S[Root2] = S[Root1];    // 小树合并到大树
    }
    else{   // Root1的结点数更少
        S[Root2] += S[Root1];   // 累加结点数量
        S[Root1] = Root2;   // 小树合并到大树
    }
}


// Find优化操作  压缩路径
int Find(int S[], int x){
    int root = x;
    while (S[root] >= 0)    root = S[root];     // 循环找到根结点
    while (x != root){  // 压缩路径操作
        int t = S[x];   // t指向x的父节点
        S[x] = root;    // x直接挂到根结点下
        x = t;          // 处理父结点
    }
}
```

5-图

图的基本概念

定义以及概念

- **有向图**和**无向图**
- **无向图**中，顶点v的度是指依附于顶点v的边的条数，记为 `TD(v)` ，在有向图中，**入度**为以顶点v为重点的有向边的数目，记为 `ID(v)` ，对应的有**出度** `OD(v)` ，且有向图的**度**为入度加出度 `TD(v)=ID(v)+OD(v)`
- 每条边都可以表上某种具有某种含义的数值，该数值称为该边的**权值**。这种边上带有权值的图称为**带权图**，也称为**网**。
- **生成树**是包含图中全部顶点的一个极小连通子图。若图中顶点数为n，则它们的生成树含有n-1条边。**生成森林**即，在非连通图中，连通分量的生成树构成的。
- 在路径序列中，从顶点u到顶点v的最短路径若存在，则此路径的长度称为从u到v的距离。若两顶点之间不存在路径，则记该距离为无穷∞。

图的存储 （邻接矩阵、邻接表、十字链表、邻接多重表）
邻接矩阵
图的邻接矩阵存储结构定义：

```
# define MaxVertexNum 100   // 顶点数目的最大值
typedef char VertexType;    // 顶点的数据结构
typedef int EdgeType;       // 带权图中边上权值的数据类型
typedef struct {
    VertexType Vex[MaxVertexNum];   // 顶点表
    EdgeType Edge[MaxVertexNum][MaxVertexNum];  // 邻接矩阵，边表
    int vexnum, arcnum;          // 图的当前顶点数和弧数
}MGraph;
```

图的邻接矩阵的特点：

- 无向图的邻接矩阵一定是一个对称矩阵（并且唯一），一次，在实际存储中，只需要存储上三角矩阵或下三角矩阵
- 对于无向图，邻接矩阵的度就是一行中的非0数字的个数
- 对于有向图，出度为一行中非0数字的个数，入度为一列中非0数字的个数，两个相加为有向图的度
- 设图G的邻接矩阵为A，A^n的元素A[i][j]等于由顶点i到顶点j的长度为n的路径数目
- 连接矩阵存储方法更适合稠密图，不适合稀疏图

邻接表法
图的邻接表法结合了顺序存储和链式存储两种存储方法，大大减少了不必要的浪费
其中邻接表中存在两种结点，分别为：**顶点结点**和**边表结点**
顶点结点 包括顶点域（data）和指向第一条邻接边的指针域构成（firstarc）
边表结点 由邻接点域（adjvex）和指向下一条邻接边的指针域（nextarc）构成q

十字链表
十字链表是有向图的一种链式存储结构。包括弧结点和顶点结点
弧结点 tailvex 和 headvex 两个域分别指示弧尾和弧头这两个顶点的编号；hlink域指向弧头相同的下一个弧结点；tlink域指向弧尾相同的下一个弧结点； info域存放该弧的相关信息，如权值
顶点结点 data域存放该顶点的数据信息； firstin域指向以该顶点为弧头的第一个弧结点；firstout域指向以该顶点为弧尾的第一个结点

邻接多重表
邻接多重表是无向图的一种链式存储结构
顶点结点 data域存放顶点信息； firstedge域指向第一条依附于该顶点的边
弧结点 ivex和jvex两个域指示该边依附于的两个顶点的编号；ilink域指向下一条依附于顶点ivex的边；jlink域指向下一条依附于顶点jvex的边，info域存放顶点信息

图的基本操作

- `Adjacent(G, x, y)` ： 判断图G是否存在边<x,y>或者(x,y)
- `neighbors(G, x)` ： 列出G中与结点x邻接的边
- `InsertVertex(G, x)` ：在图G中插入顶点x
- `DeleteVertex(G, x)` ： 在图G中删除顶点x
- `AddEdge(G, x, y)` ： 若无向边(x, y)或者有向边<x, y>不存在，则向图G中添加该边
- `RemoveEdge(G, x, y)` ： 若无向边(x, y)或者有向边<x, y>存在，则从图G中删除该边

- `FirstNeighbor(G, x)`：求图G中顶点x的第一个邻接点，若有则返回顶点号；若x没有邻接点或者图G中不存在x，则返回-1 （常用）
- `NextNeighbor(G, x, y)`：假设图G中顶点y是顶点x的一个邻接点，返回除y外顶点x的下一个邻接点的顶点号，若y是x的最后一个邻接点，则返回-1 （常用）
- `Get_Edge_Value(G, x, y)`：获取图G中边<x,y>或(x, y)对应的权值
- `Set_Edge_Value(G, x, y, v)`：设置图G中边<x,y>或(x,y)对应的权值为v

图的遍历

图的遍历主要分为两种算法：**广度优先遍历（BFS）**和**深度优先遍历（DFS）**

广度优先遍历（Breadth First Search）
可由广度遍历过程中，得到一颗广度优先生成树，不同的极大联通图的广度生成树之间，构成广度生成森林。

BFS核心思想：首先从访问起始顶点v开始，由v出发，依次访问v邻接点，然后再从这些邻接点出发，重复上述步骤，直至所有顶点均被访问。若此时图中尚有顶点未被访问，则选取一个未被访问的顶点，重复上述步骤，直至所有顶点被访问。

```
bool visited[Max_Vertex_Num];   // 访问标记数组

void BFSTraVerse(Graph G){  // 对图G进行BFS
    for (i=0; i<G.VexNum; i++)
        visited[i] = False; // 初始化标记数组
    InitQueue(Q);
    for (i=0; i<G.VexNum; i++)
        if (!visited[i])
            BFS(G,i);
}


void BFS(Graph G, int v){   //BFS算法
    visit(V);
    visited[v] = True;
    EnQueue(Q, v);  // 顶点v入队
    while (!isEmpty(Q))
    {
        DeQueue(Q,v);
        for (w=FirstNeighbor(G,v); w=0; w=NextNeighbor(G,v,w))  // 检测v的所有邻接点
            if(!visited[w]){
                visit(w);
                visited[w] = True;
                EnQueue(Q,w);   // 顶点w入队列
            }// if
    }// while
}
```

深度优先遍历（Depth First Search）

DFS核心思想： 首先访问图中某一起始顶点v，然后从v出发，访问与v邻接且未被访问的任意一个顶点w1，再访问与w1邻接且未被访问的下一个顶点，重复上述步骤，直至不能继续向下访问时，回到最近被访问的顶点，继续上述操作，直至图中所有顶点均被访问为止。

```
bool visited[Max_VerTex_NUM];   // 访问标记数组

void DFSTraverse(Graph G){  // 对图G进行深度优先遍历
    for (v=0; v<G.vexnum; v++)
        visited[v] = False; // 初始化标记数组
    for (v=0; v<G.vexnum; v++)
        if (!visited[v])
            DFS(G, v);
}

void DFS(Gragh G, int v){   // 从顶点v出发，执行DFS算法
    visit(v);
    visited[v] = True;
    for (w=FirstNeighbor(G, v); w>=0; w=NextNeighbor(G,v,w))
        if (!visited[w])
            DFS(G,w);   // 递归调用实现DFS算法
}
```

图的应用

区别 ： **最小生成树** 和 **最短路径问题** 的区别：

- 最小生成树能够保证整个拓扑图的所有路径之和最小，但是不能保证任意两点之间是最短路径
- 最短路径是从一点出发，到达目的地的路径最小

最小生成树：Prim算法 和 Kruskal算法
比较：

- Prim算法仅与顶点数vexnum有关，与其他无关，因此，该算法适合稠密图
- Kruskal算法仅与边数有关，与其他无关，因此，该算法适合稀疏图

Prim算法： 初始时从图中任取一顶点，加入树T中，此时树中只含有一个顶点，之后选择一个与当前T中顶点集合距离最近的顶点，并将该顶点和相应的边加入T，每次操作后T中的顶点和边数加1。以此类推，直至图中所有顶点都加入T中，得到的T就是最小生成树。此时T中必然有n-1条边。
Kruskal算法： 初始时为只有n个顶点而无边的非连通图T，每个顶点自成一个连通分量，然后按照边的权值由小到大的顺序，不断选取当前未被选取过且权值最小的边，若该边依附的顶点落在T中不同的连通分量上，则将此边加入T，否则舍弃此边而选择下一条权值最小的边。依此类推，直至T中所有顶点都在一个连通分量上。

最短路径
常用的最短路径算法有三种： 1、BFS算法； 2、Dijkstra算法；3、Floyd算法。
其中BFS和Dijkstra算法可以求解单源最短路径问题，而Floyd算法可以求解各顶点之间的最短路径问题。
BFS算法： 仅适用于非权图的单源最短路径问题，该算法基于广度优先算法（广度优先算法 > 深度最小 > 路径最短），等价于广度优先生成树的最短路径问题
通常适用如下数组来存储距离和上一个路径来源

代码实现：

```c
void BFS_MIN_Distance(Graph G, int u){
    // d[i]表示从u到i结点的最短路径
    // path[i]表示若要形成最短路径达到该结点，上一个走过的结点序号
    for (i=0; i<G.vexnum; i++){
        d[i] = INF; // 初始化路径长度为无穷
        path[i] = -1;   // 最短路径从哪个顶点过来
    }
    visited[u] = True;
    d[u] = 0;
    EnQueue(Q,u);

    while (!isEmpty(Q))
    {
        DeQueue(Q,u);
        for (w=FirstNeighbor(G,u); w<G.vexnum; w=NextNeighbor(G,u,w))
            if (!visited[w]){
                visited[w] = True;  // 设为已访问标记
                d[w] = d[u] + 1;    // 路径长度加1
                path[w] = u;    // 最短路径是从u到w
                EnQueue(Q, w);  // 顶点w入队
            }
    }
}
```

Dijkstra算法 和 Floyd算法

`Dijkstra` 算法：(不适用于带负权值的边！)

算法过程：

1. 需要初始化三个数组，第一个final数组存储是否该结点完成运算，第二个dist数组存储距离上一结点的距离，path存储上一结点的数组下表
2. 判断当前结点相邻的结点是否有有更短路径，有则更新数组
3. 判断当前数组中final的值为false且距离最短的点，将它的final值设为true
4. 重复2、3两步骤，直至final中所有值均变成true

代码实现：

```c
// 邻接矩阵
typedef struct _graph
{
    char vexs[MAX];       // 顶点集合
    int vexnum;           // 顶点数
    int edgnum;           // 边数
    int matrix[MAX][MAX]; // 邻接矩阵
}Graph, *PGraph;

// 边的结构体
typedef struct _EdgeData
{
    char start; // 边的起点
    char end;   // 边的终点
    int weight; // 边的权重
}EData;
```

```
17
18
19  /*
20   * Dijkstra最短路径。
21   * 即，统计图(G)中"顶点vs"到其它各个顶点的最短路径。
22   *
23   * 参数说明：
24   *       G -- 图
25   *      vs -- 起始顶点(start vertex)。即计算"顶点vs"到其它顶点的最短路径。
26   *    prev -- 前驱顶点数组。即，prev[i]的值是"顶点vs"到"顶点i"的最短路径所经历的全
     部顶点中，位于"顶点i"之前的那个顶点。
27   *    dist -- 长度数组。即，dist[i]是"顶点vs"到"顶点i"的最短路径的长度。
28   */
29  void dijkstra(Graph G, int vs, int prev[], int dist[])
30  {
31      int i,j,k;
32      int min;
33      int tmp;
34      int flag[MAX];      // flag[i]=1表示"顶点vs"到"顶点i"的最短路径已成功获取。
35
36      // 初始化
37      for (i = 0; i < G.vexnum; i++)
38      {
39          flag[i] = 0;              // 顶点i的最短路径还没获取到，  相当于上述final
     数组
40          prev[i] = 0;              // 顶点i的前驱顶点为0，相当于上述path数组
41          dist[i] = G.matrix[vs][i];// 顶点i的最短路径为"顶点vs"到"顶点i"的权。
42      }
43
44      // 对"顶点vs"自身进行初始化
45      flag[vs] = 1;
46      dist[vs] = 0;
47
48      // 遍历G.vexnum-1次；每次找出一个顶点的最短路径。
49      for (i = 1; i < G.vexnum; i++)
50      {
51          // 寻找当前最小的路径；
52          // 即，在未获取最短路径的顶点中，找到离vs最近的顶点(k)。
53          min = INF;
54          for (j = 0; j < G.vexnum; j++)
55          {
56              if (flag[j]==0 && dist[j]<min)
57              {
58                  min = dist[j];
59                  k = j;
60              }
61          }
62          // 标记"顶点k"为已经获取到最短路径
63          flag[k] = 1;
64
65          // 修正当前最短路径和前驱顶点
66          // 即，当已经"顶点k的最短路径"之后，更新"未获取最短路径的顶点的最短路径和前驱顶
     点"。
67          for (j = 0; j < G.vexnum; j++)
68          {
69              tmp = (G.matrix[k][j]==INF ? INF : (min + G.matrix[k][j])); //
     防止溢出
```

```
            if (flag[j] == 0 && (tmp  < dist[j]) )
            {
                dist[j] = tmp;
                prev[j] = k;
            }
        }
    }

    // 打印dijkstra最短路径的结果
    printf("dijkstra(%c): \n", G.vexs[vs]);
    for (i = 0; i < G.vexnum; i++)
        printf("  shortest(%c, %c)=%d\n", G.vexs[vs], G.vexs[i], dist[i]);
}
```

Floyd算法 （基于动态规划思想）（可以解决带负权边的图，但是不能解决带有负权回路的图）

使用动态规划思想，将问题的求解分为多个阶段： 对于n个顶点的图G，求任意一对顶点V_i -> V_j之间的最短路径可分为如下几个阶段：

1. 初始： 不允许在其他顶点中转，最短路径是?
2. 若允许在V_0中转，最短路径是?
3. 若允许在V_0、V_1中转，最短路径是?
4. 若允许在V_0、V_1、V_2、……V_{n-1}中转，最短路径是?

使用两个矩阵存储最短路径和路径方向
代码实现：

```
// 在执行下面代码前，需要准备工作，即初始化矩阵A和path数组
for (int k=0; k<n; k++){    // 考虑意vk作为中转点
    for (int i=0; i<n; i++){    // 遍历整个矩阵，i为行号，j为列号
        for (int j=0; j<n; j++){
            if (A[i][j] > A[i][k]+A[k][j]){     // 以vk为中转点的路径更短
                A[i][j] = A[i][j]+A[k][j];      // 更新最短路径
                path[i][j] = k;                 // 更新中转点数组下标
            }
        }
    }
}
```

总结和对比：

6-查找

基本概念

查找： 在数据结合中寻找满足某种条件的数据元素的过程称为查找
查找表：用于查找的数据集合称为查找表，通常为同一类型的数据元素。 对于一个查找表，常见的操作有： 查找、插入和删除数据。若一查找表只需要进行查找操作，则称为静态查找表
关键字： 数据元素中唯一标识该元素的某个数据项的值，使用基于关键字的查找，查找的结果应该是唯一的，如学号
平均查找长度（ASL）： 在查找过程中进行关键字的比较次数的平均值

顺序、折半、分块查找

顺序查找
顺序查找又叫线性查找，它对顺序表和链表都是适用的。对于顺序表，可以通过递增数组下标来扫描每个元素；对于链表，可以通过next依次扫描每个元素
代码实现：

```c
// 方法1,直观思维实现
typedef struct {
    ELemType *elem;  // 指向元素存储空间的地址
    int TableLen;   // 当前表的长度
}SSTable;


/*
* ST 需要查找的数组或者链表
* key 需要查找的关键字
*/
int Search_Seq(SSTable ST, ElemType key){
    for (int i=0; i<ST.TableLen && ST.elem!=key; i++)
        return i==ST.TableLen ? -1 : i;
}


// 方法2，加入哨兵倒序查找
typedef struct {
    ElemType *elem;
    int TableLen;
}SSTable;

int Search_Seq(SSTable ST, ElemType key){
    ST.elem[0]=key;     // 哨兵，且该数组的0号位置不存放数据
    for (int i=ST.TableLen; ST.elem[i]!=key; i--)   // 从后往前查找
        return i;   // 若表中不存在关键词为key的元素，将查找到i为0时退出for循环
}
```

折半查找
折半查找又叫**二分查找**，他仅使用于有序的**顺序表**，不适合链表结构

```c
int Binary_Search (SSTable L, ElemType key){
    int low=0, high=L.TableLen-1,mid;
    while (low<high){       // low>high时，查找失败
        mid = (low+high)/2; // 取中间位置
        if (L.elem[mid]==key)
            return mid; // 查找成功则返回当前所在位置
        else if (L.elem[mid]>key)
            high = mid - 1; // 从前半部分继续查找
        else
            low = mid + 1;  // 从后半部分继续查找
    }
    return -1;  // 查找失败，返回-1
}
```

分块查找
分块查找右脚索引顺序查找。分块查找的优先是，块内可以无序，但是块间是有序的。
查找过程为：1、在索引表中确定待查记录所在的块，可以顺序查找或折半查找；2、在块内顺序查找

树型查找

二叉排序树（BST，Binary Search Tree）： 构造BST的目的不是为了排序，而是为了提高查找、插入和删除关键字的速度
定义： BST又叫二叉查找树或者是一颗空树。**可用二叉树的中序遍历，即左根右得到一个递增的有序序列。

BST查找的代码实现：

```c
BSTNOde *BST_Search(BiTree T, ElemType key){
    while (T!=NULL && key!=T->data){
        if (key<T->data) T=T->lchild;   // 小于，则在左子树上查找
        else T=T->rchild;   // 大于，则在右子树查找
    }
    return T;
}
```

BST插入的代码实现：

```c
// 递归实现BST的插入操作 （可以考虑一下非递归实现，这样空间复杂度更低）
int BST_Insert(BiTree &T, KeyType k){
    if (T==NULL){   // 原树为空，新插入的记录作为根节点，递归的基本操作
        T=(BiTree)malloc(sizeof(BSTNode));
        T->data = k;
        T->lchild = T->rchild = NULL;
        return 1;
    }
    else if (k==T->data)    // 树中存在相同关键字的结点，则插入失败
        return 0;
    else if (k<T->data) // 插入到T的左子树中
        return BST_Insert(T-lchild);
    else if (k>T->data) // 插入到T的右子树中
        return BST_Insert(T->rhild);
```

```
}
```

BST的删除：

1. 若被删除结点z是叶子结点，直接删除
2. 若被删除结点z只有一颗左子树和右子树，则让z的子树成为z父结点的子树，代替z的位置
3. 若结点z有左右两颗子树，则令z的直接后继（或者直接前驱）替代z，然后从BST树中删去这个结点，这样就变成了第一或第二种情况

平衡二叉树
为了避免树的高度增长过快，降低BST的性能，规定在插入和删除结点时，需要保证任一结点的左右子树高度差的绝对值不超过1，这样的二叉树叫做平衡二叉树（Balanced Binary Tree），又叫做AVL树。

平衡二叉树的定义：（引入平衡因子的概念，左右子树的平衡因子绝对值应当小于1）

```
// 平衡二叉树的结点

typedef struct AVLNode{
    int key;    // 数据域
    int balance;    // 平衡因子
    struct AVLNode  *lhcild, *rchild;   // 左、右孩子指针，这里struct定义与递归定义不同，这里指的是指向AVL结点的指针，而不是定义结构体
}AVLNode, *AVLTree;
```

平衡二叉树的插入：
插入新结点后，可能会导致不平衡。若导致了不平衡的发生，找到**最小不平衡子树**，后根据RR、LL、RL、LR类型进行处理

1. LL平衡旋转（右旋转），由于在A的左孩子（L）的左子树（L）上插入了新结点，导致A失去平衡，则需要将A的左孩子B向右上旋转代替A成为根结点
2. RR平衡旋转（左旋转），由于在A的右孩子（R）的右子树（R）上插入新结点，导致A失去平衡，则需要将A的右孩子向左旋转代替A成为根结点
3. LR平衡旋转（先左后右双旋转），由于在A的左孩子（L）的右子树（R）上插入新结点，导致A失去平衡，则需先将A结点的左孩子B的右子树的根结点C向左上旋转提升到B结点的位置，然后把C向右上旋转提升到A结点的位置
4. RL平衡旋转（先右后左双旋转），由于A的右孩子（R）的的左子树（L）上插入新结点，导致A失去平衡，则需先将A结点的右孩子B的左子树C向右上旋转提升到B结点的位置，然后把C结点向左上旋转提升到A结点的位置

代码实现：

```
// 右旋的实现，其中f是爹，p是左孩子，gf为g的爹。实现f向右下旋转，p向右上旋转
f->lchild = p->rchild;
p->rchild = f;
gf->lchild/rchild = p;

// 左旋的实现。实现了f向左下旋转，p向左上旋转
f-rchild = p->lchild;
p->lchild = f;
gf->lchild/rchild = p;
```

平衡二叉树的删除：

1. 删除结点（方法同“二叉排序树”）
2. 一路向上找到最小不平衡子树，找不到就结束该操作
3. 找最小不平衡子树下，“个头”最高的儿子和孙子
4. 根据孙子的位置，调整平衡（LL/RR/RL/LR）
5. 如果不平衡向上传到，继续2，否则结束该操作

红黑树
工程中，由于RBT更加优异，故通常采用红黑树，而非AVL

```
// 难度有点大，后续遇到再解决

struct RBNode{  // 红黑树的结点定义
    int key;    // 关键字的值
    RBNode* parent;
    RBnode* lchild;
    RBNode* rchild;
    int color;  // 结点颜色定义
}
```

B树和B+树

B树
定义：　所谓m阶 B 树是所有结点的平衡因子均等于 0 的m路平衡查找树，且 B 树必须满足如下特性：

- 树中每个结点至多有m棵子树，即至多含有m-1个关键字
- 若根结点不是叶结点，则至少有两棵子树
- 除根结点以外的所有非叶结点至少有 $[m/2]_{向上取整}$ 棵子树，即至少含有 $[m/2]_{向上取整}$ - 1 个关键字
- 所有结点必须绝对平衡，即没棵子树的高度必须相同
- 叶子节点（败者结点）均在最下一层

B树的插入
定位 > 插入 ，插入的位置一定是最底层中的某个非叶子结点
例：一次插入操作的示意图

B树的删除：
被删除的关键字k不在终端结点（最底层的非叶子结点）中时，可用k的前驱（或后继）k'来代替k，然后就变成删除终端结点的问题
删除终端结点：

1. 直接删除关键字。若被删除关键字所在结点的关键字个数满足最低子树要求，则可以直接删去关键字
2. 兄弟够借。若被删除的关键字所在结点删除前的关键字小于最低子树要求，但是与其相邻的左、右兄弟结点的关键字够借用，则调整兄弟结点和父结点的关系，以达到新的平衡
3. 兄弟不够借，若被删除的关键字所在节点删除前的关键字小于最低子树要求，且与其相邻的左、右兄弟结点的关键字也不够用，则需要合并与其相邻的兄弟节点和父节点，以达到新的平衡

总结：

B+树

要求：

1. 每个分支结点最多有m棵树（孩子结点）
2. 非根结点至少有两棵子树，其他每个分支结点至少有两棵子树，其他每个分支结点至少有 $[m/2]_{向上取整}$棵子树
3. **结点的子树个数与关键字个数相等**
4. **所有叶子节点包含全部关键字以及指向相应记录的指针**，且**叶子节点将关键字按大小顺序排序**，并且相邻叶子结点之间按照顺序相互链接起来
5. **所有分支结点中仅包含它各个子结点中关键字的最大值以及指向其子结点的指针**

在B+树中查找时，无论查找成功与否，每次查找都是一条从根结点到叶子结点的路径

散列表（Hash 表）

Hash函数： 一个把查找表中的关键字映射成该关键字对应的地址函数，记为 `Hash(key) = Addr`

Hash表： 根据关键字而直接进行访问的数据结构。 *也就是说，**Hash表建立了关键字和存储地址之间的一种映射关系*** 。

冲突： 散列函数中可能会把两个或两个以上的不同关键字映射到同一个地址，这种情况就叫做冲突，这些发生冲突的不同关键字之间叫做**同义词。**冲突是不可避免的，但是应当尽量减少冲突。

理想情况下，对Hash表进行查找的时间复杂度是O(1)，但是空间复杂度很高，即**用空间换时间**，将具体的数值与存储的地址相关联起来。

构建Hash函数的方法

1. 直接定值法： `H(key) = key` 或者 `H(key) = a*key + b` ，这种方法非常适合关键字连续的情况，且不会产生冲突
2. 除留余数法： `H(key) = key % p` ，假定一个hash表的长度为m，取一个不大于m但非常接近或者等于m的质数p，用上述公式即可
3. 数字分析法： 取r个数码中，尽量均匀随机分布的数值，比如电话号码138xxxx0767，前三位大多都一样，但是后4位随机分布，则可以按照后四位建立hash表
4. 平方取中法（需要的时候查资料）

处理冲突的方法

1. 开放定址法（需要的时候查资料）
2. 拉链法（chaining）

7-排序

基本概念

排序算法分为：1． **内部排序**；2． **外部排序。**这是根据排序过程中，数据元素是否完全在内存中来划分的。
排序算法需要考虑的是：

1. 时间复杂度
2. 空间复杂度
3. 算法稳定性： 关键字相同的元素在排序之后的相对位置不变，反之则是不稳定的排序算法

若当前是**外部排序**，即数据存储在磁盘中，则还需要考虑磁盘读写的次数，因为机械硬盘是慢速设备，会影响程序运行时间。

插入排序

直接插入排序：每次将一个待排序的记录按照其关键字的大小，插入前面已排好的子序列中，直至全部记录插入完成。该算法**适用于顺序表和链表**
代码实现：

```
/*
 * @brief 直接插入排序
 * @param A[] 需要排序的数组
 * @param n 数组长度
 * @return 无
*/
void InsertSort(int A[], int n){
    int i,j,temp;
    for (i=1; i<n; i++) // 逐个扫描需要排序的数组元素
        if (A[i]<A[i-1]){   // 若A[i]关键字小于前驱
            temp = A[i];    // temp暂存A[i]
            for (j=i-1; j>=0 && A[j]>temp; j--) // 检查所有前面已经排序好的元素
                A[j+1] = A[j];  // 所有大于temp的元素都向后挪位置
            A[j+1] = temp;  // 当前排序关键字复制到插入位置
        }
}


/*
 * @brief 直接插入排序(带哨兵)
 * @param A[] 需要排序的数组
 * @param n 数组长度
 * @return 无
*/
void InsertSort(int A[], int n){
    int i,j;
    for (i=2; i<n; i++)
        if (A[i]<A[i-1]){   // 若A[i]关键字小于前驱
            A[0] = A[i];    // 将当前需要排序的值，复制为哨兵，A[0]不存放元素
```

```
            for (j=i-1; A[j]>A[0]; j--) // 从后往前查找待插入的位置
                A[j+1] = A[j];  // 向后挪位置
            A[j+1] = A[0];  // 将需要插入的值复制到需要插入的位置
        }
}
```

折半插入排序
该算法是基于插入排序算法的优化，对于直接插入排序，其分为两个步骤，1、定位到需要插入的位置；2、给插入位置腾出空间，而在折半插入排序中，使用折半查找的方法找出元素的待查入位置，然后统一的移动待插入位置之后的所有元素。该算法**仅适用于顺序表**
代码实现：

```
// 折半插入排序
void InsertSort(ElemType A[], int n){
    int i, j, low, high, mid;
    for (i=2; i<n; i++){    // 依次将A[2]~A[n]插入前面的已排序序列
        A[0] = A[i];    // 将A[i]暂存到A[0]
        low=1; high=i-1;    // 设置折半查找的范围，即当前元素前面所有的元素
        while (low<=high){  // 折半查找开始
            mid = (low+high) / 2;   // 取中间点
            if (A[mid])>A[0]) high = mid-1; //查找左半子表
            else low = mid+1;   //查找右半子表
        }
        for (j=i-1; j>=high+1; j--)
            A[j+1] =  A[j];  // 统一后移，空出插入位置
        A[high+1]=A[0]; // 插入操作
    }
}
```

希尔排序 (Shell Sort)
分析：直接插入排序算法的时间复杂度为$O(n^2)$，但若待排序为“正序”时，其时间效率可提升至O(n)，由此可见直接插入排序更适合基本有序的排序表和数据量不大的排序表。希尔排序正是基于这两点分析对直接插入排序进行改进而得来的，又叫缩小增量排序。算法的时间复杂度降低到了$O(n^{1.3})$。
核心：先追求表中元素的部分有序，再逐渐逼近全局有序。首先，将待排序表分割成若干形如 `L[i, i+d, i+2d,..., i+kd]` 的“特殊”子表，即把像个某个“增量”的记录组成一个子表，对各个子表分别进行直接插入排序，当整个表中的元素呈现“基本有序”的时候，再对全体记录进行一次直接插入排序。
代码实现：

```c
// 下述代码不完全符合代码逻辑，可以更改代码结构使之符合的代码逻辑
void ShellSort (ElemType A[], int n){
    // A[0]是暂存单元
    int dk, i, j;
    for (dk=n/2; dk>=1; dk=dk/2)     // 增量变化定义
        for (i=dk+1; i<n; i++)  //**在不同子表中逐个遍历**
            if (A[i] < A[i-dk]){
                A[0] = A[i];    // 暂存在A[0]
                for (j=i-dk; j>0 && A[0]<A[j]; j-=dk)
                    A[j+dk] = A[j]; // 记录向后移动，查找插入的位置
                A[j+dk] = A[0];
            }
}
```

交换排序

冒泡排序
概念：从后往前（后者从前往后）两两比较相邻元素的值，若为逆序（即 `A[i-1]>A[i]` ），则交换它们，知道序列比较完。称这样过程为一趟冒泡排序。结果是将最小的元素交换到待排序列的第一个位置，关键字最小的元素如气泡一样逐渐向上漂浮直至睡眠（或者关键字最大的元素如石头一般下沉直至水底）。下一趟冒泡时，前一趟确定的最小元素不再参与比较，每趟冒泡的结果是把序列中的最小元素（或者最大元素）放在了序列的最终位置。……这样最多进行n-1次冒泡就可以将所有元素排好序。**若一趟中交换不再发生，表示当前数组已经有序，不需要继续冒泡排序**
代码实现：

```c
// Bubble Sort，逆向的冒泡排序，每次都将当前子表中最小元素移动到最前面
void swap(int a, b){
    int temp;
    temp = a;
    a = b;
    b = temp;
}

void BubbleSort(ElemType A[], int n){
    for (int i=0; i<n-1; i++){
        bool Flag = false;  // 表示本次冒泡是否发生交换的标志
        for (int j=n-1; j>i; j--)   // 一趟冒泡过程
            if (A[j-1]>A[j]){
                swap(A[j-1], A[j]); // 交换
                Flag = true;    // 标志位变化，代表本次发生交换
            }
        if (Flag==false)
            return; // 本次遍历后没有发生交换，说明表已经有序，结束循环
    }
}
```

快速排序

算法思想： 在待排序表 `L[1...n]` 中任意选取一个元素pivot作为枢轴（或基准，通常取首元素），通常一趟排序将待排序表划分为独立的两部分 `L[1...k-1]` 和 `L[k+1...n]` ，使得左半边表中所有元素小于pivot，右半边表中所有元素大于等于pivot，则pivot放在了其最终位置L(k)上，这个过程称为一次"划分"。然后分别**递归**地对两个子表重复上述过程，直至每部分内只有一个元素或空为止，即所有元素都放在了其最终位置上。平均时间复杂度为 $O(nlog_2n)$
代码实现：

```
int Partition(ElemType A[], int low, int high){ // 一次划分
    ElemType pivot=A[low];  // 将当前表中第一个元素设为枢轴，对表进行划分。更好的建议是找中间位置的元素建立枢轴，可以避免最坏情况发生
    while (low<high){
        while (low<high && A[high]>=pivot) high--;
        A[low] = A[high];   // 将比枢轴小的元素移动到左端
        while (low<high && A[low]<=pivot) low++;
        A[high] = A[low];   // 将比枢轴大的元素移动到右端
    }
    A[low]=pivot;   // 枢轴元素存放到最终位置
    return low; // 返回房钱存放数轴元素的数组下标
}


void QuickSort(ElemType A[], int low, int high){
    if (low<high){//递归结束的条件
        int PivotPos = Partition(A, low, high); // 划分操作
        QuickSort(A, low, PivotPos-1);  // 对左右两个子表进行递归排序
        QuickSort(A, PivotPos+1, high);
    }
}
```

选择排序

简单选择排序
每一趟在待排序元素中选取关键字最小的元素加入有序子序列，时间复杂度为 `O(n)` ，适用于顺序表和链表
代码实现：

```
void SeletSort(ElemType A[], int n){
    for (int i=0; i<n-1; i++){      // 一共进行n-1躺，不需要对数组最后一个元素进行操作
        int min=i;  // 记录最小元素的位置
        for (int j=i+1; j<n; j++)
            if (A[j]<A[min]) min=j; // 更新最小元素的位置
        if (min != i) swap(A[i], A[min]);   // 交换最小元素到当前子序列的最前端
    }
}
```

堆排序（Heap Sort）
堆的定义如下，n个关键字序列 `L[1...n]` 称为堆，当且仅当该序列满足：

1. `L(i)>=L(2i)` 且 `L(i)>=L(2i+1)` 或

2. `L(i)<=L(2i)` 且 `L(i)<=L(2i+1)` `(1 <= i <=n/2)`

可以将堆视为一颗**完全二叉树**，满足条件1的称为**大根堆（大顶堆）**，大根堆的最大元素存放在根结点，且其任意一个非根结点的值小于或者等于其双亲结点值。
满足条件2的称为**小根堆（小顶堆）**，定义与大根堆刚好相反，根结点是最小元素。
代码实现：

```
// 建立大根堆
void BuildMaxHeap(ElemType A[], int len){
    for (int i=len/2; i>0; i--) // 从i=[n/2]到1，反复调整堆。从堆的最底层根结点开始调整，直至根结点
        HeadAdjust(A, i, len);
}


// 调整大根堆
void HeadAdjust(ElemType A[], int k, int len){
    // 函数HeadAdjust将元素k为根的子树进行调整
    A[0] = A[k];    // 将A[0]暂存子树的根节点
    // i指向根结点的左右孩子的更大的一个，k指向根节点
    for (int i=2*k; i<=len; i*=2){  // 沿key较大的子结点向下筛选，防止交换堆顶元素后破坏堆的结构
        if (i<len && A[i]<A[i+1])   // 寻找左、右孩子中较大的一个
            i++;
        if (A[0]>=A[i]) break;  // 筛选结束
        else{
            A[k] = A[i];    // 将A[i]调整到双亲结点上
            k = i;
        }
        A[k] = A[0];    // 被筛选节点的值放入最终位置
    }
}


// Heap Sort
void HeapSort(ElemType A[], int len){
    BuildMaxHeap(A, len);   // 初始建堆
    for (int i=len; i>1; i--){  // n-1次的交换和建堆的过程
        Swap(A[i], A[1]);   // 输出堆顶元素（和堆底元素交换）
        HeapAdjust(A, 1, i-1);  // 调整，把剩余的i-1个元素整理成堆
    }
}
```

归并排序

归并排序（Merge）
“归并”的含义是将两个或两个以上的有序表合并成一个新的有序表。假定待排序表含有n个记录，则可以将其视为n个有序的子表，每个子表的长度为1，然后两两归并，得到n/2个长度为2或1的有序表，继续两两归并……直至合并称为一个长度为n的有序表为止，这种排序方法称为2路归并排序。时间复杂度为O($nlog_2n$).
代码实现:

```c
// Merge的功能是将前后相邻的两个有序表归并为一个有序表，需要借助辅助数组B
ElemType *B = (ElemType *)malloc((n+1)*sizeof(ElemType));   // 辅助数组B

void Merge(ElemType A[], int low, int mid, int high){
    int i, j, k;
    for (k=low; k<=high; k++)
        B[k] = A[k];    // 将A中所有元素复制到B中
    for (i=low, j=mid+1, k=i; i<=mid && j<=high; k++){
        if (B[i] < B[j])    // 比较B的左右两段中的元素
            A[k] = B[i++];
        else
            A[k] = B[j++];
    }
    while (i<=mid)  A[k++] = B[i++];    // 若第一个表未检测完，复制到队尾
    while (j<=high) A[k++] = B[j++];    // 若第二个表未检测完，复制到队尾
}


// 归并排序
void MergeSort(ElemType A[], int low, int high){
    if (low<high){
        int mid = (low+high) / 2;   // 从中间划分两个子序列
        MergeSort(A,low,mid);   // 对左侧子序列进行递归排序
        MergeSort(A,mid+1,high);    // 对右侧子序列进行递归排序
        Merge(A,low,mid,high);  // 归并
    }
}
```