C

一种基于C语言的helloWorld

```
#include<stdio.h>
int main(){
    printf("hello world");
    return 0;
}
```

- 整数类型%d：

1. int，整数，4
2. short，占据空间更小的整数，2
3. long，长整数，8
4. unsigned int，正整数，4
5. unsigned short，占据空间较小的正整数，2
6. unsigned long，长正整数，8

- 小数类型%f：

1. float，小数，4
2. double，更长的小数，8
3. long double，非常长的小数，16

- ASCII码%c：

  char，-127~127，1

- 自动类型转换

  字节数较小的数据可以向字节数较大的进行转换:
  char, short -> int -> unsigned -> long -> double <- float

- 强制转换：（类型）表达式 (int)(10+' a'+i*f-d/e)

输入输出

第一行输入 #include<stdio.h> 来引入如下函数

1. 字符输入：getchar，字符输出"putchar
2. 格式输入：scanf，格式输出：printf
3. 字符串输入：gets，字符串输出：puts

常见字符输出类型：%d，%f，%c，%s

顺序结构，选择结构

优先级：（*，/），（%），（+，-）

注意！ 在表达式中，++和--如果在后面，则会在表达式计算完成后再计算++或--；反之则会优先计算

```
int a = 4, b = 5;
int s;
s = a++ + b;    //此时s=9，a=5，b=5
s = ++a + b;    //此时s=11，b=6，b=5
```

- 与&&, 或||, 非!
- 条件运算

```
// 如果表达式1==True，x=表达式2，否则x=表达式3
x = (表达式1) ? (表达式2) : (表达式3)
```

数组

```
//下面语句中，a[0]=1，a[1]=3，a[2]=5，a[3]=0
int a[4] = {1, 3, 5};

//下面语句相当于 int a[3] = {1, 3, 5}
int a[] = {1, 3, 5};
```

函数

参数

- 函数名括号内的变量成为“形式参数”（简称“形参”）
- 在其它函数内调用函数，则括号内变量称为“实际参数”（“实参”）
- 函数间的数据传递分为值传递和地址传递：
  - 值传递：实参单向传递值给形参（它们的类型必须相同）
  - 地址传递：数据存储地址作为参数双向传递给形参（形参和实参必须暂用相同的存储单元，并且必须是地址常量或变量）
  - 如果实参和形参都是数组，则是地址传递

返回类型

- 在定义函数时需要定义它对应返回值的类型
- 如果是void，则不能出现return，并且在该函数内的功能不会影响函数外的参数

递归

- 函数直接或间接调用自己，称为递归
- 递归三要素：
  1. 初始条件

2. 转移条件
3. 终止条件

```
/*一个递归的例子*/
#include<stdio.h>

void inversion(int n){  //初始条件
    int t;
    if(n > 10){ //终止条件
        t = n%10;
        printf("%d", t);
        inversion(n/10);    //转移条件
    } else {
    printf("%d", n);
    }
}

int main(){
    int a;
    scanf("%d", &a);
    inversion(a);
    printf("\n");
    return 0;
}
```

预编译

- 对命令进行预处理，将其结果和源程序一起进行编译处理，得到目标代码（OBJ文件）。

  1. 宏命令（Macro）
  2. 文件包含命令（include）
  3. 条件编译命令

  这些命令均以#开头

  ```
  //不带参数的宏
  #define pi 3.14
  //终止宏定义
  #undef pi

  //文件包含
  #include "（文件名）"
  ```

指针

内存是按照地址来访问的，地址指是一个个编号，也就是指针（相当于内存被分成一个个小格子）

- 变量地址：系统分配给变量的内存单元的起始地址

- 利用变量名，直接存取变量值，成为“直接访问”
- 定义一个待存放地址的指针：【 数据类型 *变量名; 】

  ```
  int a=1, b=2, c, *pc;
  pc = &c;     //将c的地址赋给pc，此时pc所存放的就是c的地址
  *pc = a+b;     //此时c = a+b = 3
  ```

  由上可知：

  *变量名 表示指针变量，它等于其指向地址所存储的数值
  &变量名 表示该变量的地址，通常为16进制整数，输出时用%p

  对于上式，*pc = c = 3，pc = &c = c的地址（指针的变量只能保存地址量）

- 指针必须绑定数据类型
- 对于 *&变量名 ，其含义为 先取得变量名的地址，再进行*运算。*&a和a等价
- 仅进行地址交换时，不会影响数值；仅进行数值交换时，不会影响地址

指针和数组

将数组起始地址赋给指针，通过该指针可以访问数组中的全部元素。

C语言中数组名代表的时数组的首地址，
因此（int *pa, int a[10]） pa = a 和 pa = &a[0] 是等价的。

指针和函数

一个用指针调用函数的例子

```
#include<stdio.h>

int max(int b[]){}  //选出数组中的最大值

int main(){
    int i, m, a[10], max(int *);
    int (*pf)() //定义指向函数的指针
    for(i=0; i<10; i++)
        scanf("%d", &a[i]);
    pf = max;   //指针初始化，将函数名赋值给指针
    m = (*pf)(a);   //通过指针来调用函数
    printf("max = %d\n", m);
    return 0;
}
```

一个用指针创建函数，并将其调用的例子

```
#include<stdio.h>

float *search(float(*pointer)[4], int n){
    float *pt;
    pt = *(pointer + n);
    return (pt);
}

int main(){

    float score[][4] = {{60, 70, 80, 90}, {56, 89, 67, 88}, {34, 78, 90, 66}};
    float *p;
    int i, m;

    printf("输入学生序号：");
    scanf("%d", &m);
    printf("学生%d的分数如下：\n", m);
    float *search(float(*point)[4], int n);
    p = search(score, m);

    for(i=0; i<4; i++)
        printf("%f\t", *(p+i));

}
```

结构体

结构体是一种构造数据的类型，它把不同类型的数据组合成一个整体。
结构体类型定义只描述结构的组织形式，不分配内存。

```
/*这是一个学生信息结构体的例子*/
struct std{
    int stdNum;
    char stdName[30];
    char stdSex;
    int scrEng;
    int scrMath;
    int scrPhy;
}

struct std std01={1,"Tom","m",88,89,90}, std02;    //声明变量，此时才开始分配内存
/*
声明变量也可以在定义结构体时，
在结构体末尾的【}】后面跟上变量名以创建变量
例如
struct std{
    ...
}std01,std02;
*/
```

```c
std02.stdNum=2;
strcpy(std02.stdName, "Alice");     //strcpy使用时需要【#include<string.h>】
strcpy(std02.stdSex, "f");
std02.scrEng=91;
std02.scrMath=92;
std02.scrPhy=93;

struct std *pstd = &std01;
//其中一种输出方式
printf("学号%d，姓名%c，性别%c，英语成绩%d，数学成绩%d，物理成绩%d", (*pstd).stdNum,
(*pstd).name, (*pstd).stdSex, (*pstd).scrEng, (*pstd).scrMath, (*pstd).scrPhy);
//另一种输出方式
printf("学号%d，姓名%c，性别%c，英语成绩%d，数学成绩%d，物理成绩%d", pstd->stdNum,
pstd->name, pstd->stdSex, pstd->scrEng, pstd->scrMath, pstd->scrPhy);
```

结构体指针

指向结构体**变量**的指针，结构体比那辆的起始地址就是该结构体变量的指针。
当把结构体变量的地址存放在一个指针变量中，该指针变量就指向结构体变量。

单链表

动态进行存储分配的一种结构。

每个节点包含两个部分： 1. 用户需要用到实际数据 2. 下一个节点的地址

可以使用结构体创建链表

```c
struct Std{
    int num;
    struct Std *next;    //next指向结构体变量
};
```

自定义类型声明 typedef

typedef 原类型名 新类型名;

它可以配合struct来自定义一个单链表类型

```c
typedef struct Lnode{
    int data;    //保存节点的数值
    struct Lnode *next;      //指针域
}Lnode, *LinkList; //节点的类型，Lnode是变量，LinkList是指针
/*
在此时
Lnode *p;
则相当于
LinkList p;
*/
```

```c
//给一个节点赋值
Lnode *p;
p = (Lnode*)malloc(sizeof(Lnode));  //分配内存，并将内存首地址赋给p
p -> data=20;
p -> next=NULL;
```

枚举类型

enum 枚举名{枚举元素列表};
枚举元素就是枚举常量

```c
enum Weekday{sum, mon, tue, wed, thu, fri, sat};

enum Weekday workday, weekend;  //声明两个枚举变量，也可以像struct跟在【}】后面直接声明
```

数据结构导论

408不要求一定要用类实现，使用c会简洁一些

数据结构研究的是数值计算问题中的数据组织与操作的问题

逻辑结构

- 集合
- 线性结构，一对一
- 树形结构，一对多
- 图状结构，多对多

注意**数据包含数据元素，数据元素包含数据项**
e.g. 一个学生信息数据表，每个学生是数据元素，学生的具体信息（学号、成绩等）是数据项。

逻辑上把数据结构分为线性结构和非线性结构

线性结构包含：

```
线性表
栈
队列
串
```

非线性结构包含：

```
树
二叉树
图（有向图，无向图，稀疏图，稠密图，带权图）
广义表
多维数组
```

数据的四种基本**存储**结构分为

```
顺序（顺序表，循环队列，顺序栈，邻接矩阵）
索引
链式（单链表，双链表，循环链表，静态链表，链队列，链栈，二叉链树，三叉链树，线索二叉树，邻接表）
散列（哈希表）
```

数据结构中，逻辑结构与所使用的计算机无关

连续存储设计时，存储单元的地址一定连续
对于链式结构，则不一定

算法

特性：有穷性，确定性，可行性，输入，输出

好算法的标准：正确性，可读性，健壮性，通用性，效率与存储需求（与问题规模有关）

时间复杂度

在循环中
若变量在每一次循环都++/--，O(n)
若每次都×k，O(logkn)

- O(1)：常量阶，

```
++x;
s=0;
```

- O(logn)：对数阶，

```
//一种对数阶的例子
while(i<n){
    i *= 2
}
//另一种对数阶的例子
for(k=1;k<=n;k *= 2){
    ...
}
```

- O(n)：线性阶，

```
for(i=1;i<=n;i++){
    ...
}
```

- O(nlogn)：
- O(n^k)：（O(2^n) < O(n!) < O(n^n)），次方阶，

```
    for(i=1;i<=n;i++){
        for(j=1;j<=n;j++){
            ...
        }
```

```
    }
    //当套用多层循环时，若内外层循环变量无关，则其频度可以为O(n)*O(n)
```

关于递归的时间复杂度

1. n ---- n=n-1 ----> 1, O(n)
2. n ---- n=n/k ----> 1, O(logkn)
3. f(n) = f(n-1) + f(n-2), O(2^n)

关于多重循环

1. 若内外层无关：外×内
2. 若内外层有关：需要做递归，无法直接做乘法，

空间复杂度

算法运行时使用的存储空间大小
空间复杂度的度量：程序代码，执行数据，辅助空间/临时变量（此项即为空间复杂度）

e.g.

```
for(i=1;i<=n;i++){
    for(j...){
        x++;
        s+=x;
    }
}
/*
该代码中临时变量为i，j，x，s，空间复杂度为O(1)，常量阶
*/
```

============分割线=================

线性表

链表 Operations见下图。其中T是基本数据类型，即ei的数据类型，我们无需定义； traverse(visit(T))中的visit(T)是回调函数， 必须由traverse函数的调用者提供，它访问（处理）每一个基本数据元素ei。

栈 push(), pop()

· 单向链表实现的队列，其入列操作发生在链表表尾，出列操作发生在链表表头， 需设置两个指针变量，一个指向链表表头，一个指向链表表尾。 · 单向循环链表实现的队列，其入列操作仍发生在链表表尾，出列操作仍发生在链表表头，

但是只需设置一个指向链表表尾的指针变量即可。

队列 put(), get()

顺序存储的完全二叉树，其空间利用率最高。

线性表的实现

顺序存储

```
//静态分配
int L[MAX_NUM];
//动态分配
int *L = malloc(MAX_NUM * sizeof(int));
```

链式存储

```
typedef struct node *link;
struct node{
    int item;
    link next;
};
link head;
```

栈、队列、数组

栈、队列，基本概念

栈 push(), pop()

队列 put(), get()

```
/*顺序存储*/
//栈
int S[MAX_NUM];
int top;     //栈顶下标
//队列
int Q[MAX_NUM];
int front;   //队头下标
int rear;    //队尾下标

/*链式存储*/
//栈
typedef struct node *link;
struct node{int item, link next;};
link top;    //顶部指针
//队列
typedef struct node *link;
struct node {int item, link next;};
link front; //队头指针
link read;  //队尾指针
```

应用

1. 栈的应用：中断机制，传参，临时变量，表达式求值转换，PostScript， etc；
2. 离散事件仿真，迷宫求解，网络服务， etc；

特殊矩阵压缩

上三角、下三角，etc

树、二叉树

二叉树典操作：前序、中序、后续便利

树典操作：前根、后根遍历

```
/*完全二叉树采用顺序结构，一般二叉树采用链式结构*/
//顺序存储
int T[MAX_NUM];
int root = 0;
//链式存储
typedef struct node *link;
struct node{
    int item;
    link left_child;
    link right_child;
};
link root;


/*遍历*/

//前序便利
void pre_order(link t, void visit(link)){   //visit传递了一个地址
    if (t == NULL) return;
    visit(t);   //访问根
    //递归访问自己的左右节点
    pre_order(t -> left_child, visit);
    pre_order(t -> right_child, visit);
}

//后序遍历
void post_order(link t, void visit(link)){
    if (t == NULL) return;
    //从下往上，先访问左右节点，最后访问根
    post_order(t -> left_child, visit);
    post_order(t -> right_child, visit);
    visit(t);
}

//复制二叉树
link copy(link t){
    if (t == NULL) return NULL;
```

```c
    link s = malloc(sizeof *t); //构造一个新节点
    s -> item = t -> item;  //拷贝数据域
    //先复制左边，再复制右边，最后返回数据域
    s -> left_child = copy(t -> left_child);
    s -> right_child = copy(t -> right_child);
    return s;
}

//销毁二叉树
void destroy(link t){
    if (t == NULL) return;
    destroy(t -> left_child);
    destory(t -> right_child);
    free(t);    //释放根
}

//线索二叉树的基本构造
typedef struct node *link;
struct node{
    int item;
    bool left;
    bool right;
    link left_child;
    link right_child;
}
```

树的存储结构

1. 孩子表示法
2. 双亲表示法（*）
3. 长子-兄弟表示法（*）

树的应用

1. 二叉排序树 BST

2. 平衡二叉树 AVL，

3. （最优二叉树）Huffman树和Huffman编码

   根据字符再通讯信道中出现频率不同，给以不同的编码长度

   应用领域：文件压缩

线性表（很经常考）

基本定义

由n个同类型的数据元素（结点）组成的有限序列。n为线性表长度

线性结构是最常用、最简单的数据结构，线性表是典型的线性结构。
基本特点：线性表中的的元素都是有序且有限的（一对一）。

有首尾元素，有前驱、有后继。

顺序表

顺序表的定义

按照逻辑顺序一次存放在一组地址连续的存储单元。（逻辑顺序和存储顺义一致）

数组具有**随机存取**的特性（存取时间与物理位置无关）：
LOC(ai) = LOC(a1) + (i-1)*l

查找方式：

1. 按位置查找
2. 按值查找

在高级语言（例如C）数组具有随机存储的特性，可以借助数组描述顺序表。

```
/*背下下面的例子*/

#define MAXSIZE 100
typedef struct sqList{  //定义线性表结构体（此处是一个匿名结构体）
    int data[MAXSIZE];  //线性表存储元素的数组
    int length; //记录线性表长度
}sqL, *sqLP;  //线性表名称

/*
三大特性：
1. 空间数组是data
2. 最大长度是MAXSIZE
3. 当前元素个数是length
*/
```

顺序表操作（初始化、增删改查、etc.）

- 初始化

```
bool Init_SqL(sqL *L){
    L->data = (int *)malloc(MAXSIZE * sizeof(int));     //定义了一个100*4B的空间，
首地址为L->data
    if(! L->data) return false;     //分配失败（内存中没有这么大的连续空间）
    else{
        L->length = 0;
        return true;
    }
}
```

- 插入

插入位置开始到后买你的元素均需要往后移动：

1. 移动
2. 放置元素

```
int ListInser(sqL *L, int i, int e){

    /*边界检查*/
    //线性表已满
    if(L->length == MAXSIZE) return 0;
    //当i比第一位置小，后壁最后一位置的后一位置还要大时
    if(i < 1 || i > L->length+1) return 0;

    /*移动*/
    //若插入位置不在表尾
    if(i <= L->length){
        //将要插入位置后的元素向后移一位，从后往前移动
        int k;
        for(k = L->length - 1; k >= i-1; k--)
            L->data[k+1] = L->data[k];
    }

    /*将新元素插入*/
    L->data[i-1] = e;
    L->length++;

    return 1;
}

/* 综上，平均移动次数为E=n/2 */
```

- 删除

删除元素开始后的元素均需要往前移动：

1. 删除元素
2. 移动

```
int ListDelete(sqL *L, int i, ElemType *e){

    /*边界检查*/
    //线性表为空
    if(L->length == 0) return 0;
    //删除位置不正确
    if(i < 1 || i > L->length) return 0;

    /*删除并移动*/
    *e = L->data[i-1];
    //若删除位置不是最后位置
    if(i < L->length){
    //将删除位置后的元素向前移一位，从前往后依次移动
        int k;
        for(k = i; k < L->length; k++)
            L->data[k-1] = L->data[k];
    }

    /*长度-1*/
    L->lenth--;

    return 1;
}

/* 平均移动次数 E=(n-1)/2, O(n) */
```

按值查找并删除：

1. 线性查找值为x的第一个元素，记录位置
2. 从该位置从前往后移动
3. 长度-1

```
void LocateDeleteSqList(sqL *L, int x){
    int i=0;
    while(i < L->length){
        //查找值为x的第一个结点
        if(L->data[i] != x) i++;
        else{
            //找到后移动
            int k;
            for(k = i+1; k < L->length; k++)
                L->data[k-1] = L->data[k];
            L->length--;
            break;
        }
    }
}

/*
时间主要好事在比较和移动的操作
```

```
平均比较次数E=(n+1)/2
平均删除移动次数E=(n-1)/2
合计操作平均时间复杂度为E=n，即为O(n)
*/
```

顺序表总结：**空间连续，随即访问，查找容易，删除难**

单链表

链式存储：用任意存储单元存储线性表中的数据元素
除了存储每个结点的数值，还要存储直接后继结点的地址，称为指针或链：
data：存储数值；next：存储地址

为了操作方便，第一个结点之前设置一个头结点（不存储任何信息），head指向第一个结点
· 头指针：指向头结点
· 头结点：不存放数据，虽然不是必须，但一般都得有
· 第一个数据结点：第一个存放数据的结点

有头结点，单链表判空条件：p->next == null

```
typedef struct Lnode{
    int data;    //数据域
    struct Lnode *next; //指针域
}Lnode, *LinkList; //结点类型，前者用于定义数据结点，后者用于定义头结点
```

结点通过动态分配和释放来实现，需要时分配，不需要时释放

实现时分别使用c提供的标准函数：

1. malloc()，分配
2. realloc()，重新分配
3. sizeof()，大小
4. free(), 释放

动态分配：
//函数malloc分配了一个类型为LNode的结点变量空间，并将其是地址放在指针变量p中
p = (LNode*)malloc(sizeof(LNode));

动态释放 free(p); 系统回收由指针变量p所指向的内存区，p必须是最近一次调用malloc函数时返回的数值

单链表操作（链式结构）【背】

赋值

```
LNode *p;
p = (LNode*)malloc(sizeof(LNode));
p->data = 20;
p->next = null;
```

```
/*
  p
| 20 | NULL |
*/
```

常见指针操作

1. q=p; // 操作前：p=>a；操作后：p=>a, q=>a。
2. q=p->next; // 操作前：p=>a, a、b；操作后：p=>a, q=>b, a、b。
   - 此时p与q的关系：p是q的直接前去，q是p的直接后继
3. p=p->next; // 操作前：p=>a, p->next=>b；操作后：p=>b。
   - 工作指针：用来挨个指向单链表中的每一个结点，以实现对单链表结点的操作
4. 插入一个结点（后插法）; // 操作前：p=>a, a、b；操作后：a、c、b。
   1. 找前驱
   2. 防断链（先右后左）
      - 对于待插入的结点c(指针q)，插入位置为指针p后面：
        q->next = p->next; //也就是让q指向原先p的下一个结点
        p->next = q;
5. 删除一个结点; // 操作前：p=>a，a、b、c；操作后：p=>a，a、c。
   - 一个会导致**内存泄漏**的操作：p->next = p->next->next; //此时原p->next成为野结点
   - 正规操作：
     q = p->next;
     p-next = q;
     free(q);

创建单链表

- 头插法建表

创建表时，每次插入的结点都作为第一个结点
创建过程： 1. 创建头 2. 起循环： 1. 创建 2. 插入

```
LNode *create_LinkList(void){
    int data;
    LNode *head, *p;
    head = (LNode *)malloc(sizeof(LNode));
    head->next = NULL;  //创建头
    while(1){
        scanf("%d", &data);
        if(data == NULL) break;
        p = (LNode *)malloc(sizeof(LNode));
```

```
        p->data = data; //数据域赋值
        /*钩链，新创建的结点总作为第一个结点。下方操作遵循先右后左*/
        p->next = head->next;
        head->next = p;
    }
    return(head);   //链表表头作为返回值
}

/*
头插法的一个致命问题：创建的链表是逆序的（简称头逆）
*/
```

- 尾插法建表

创建表时，每次插入的结点作为链表的表尾
需要引入尾指针，尾指针指向尾结点（尾结点是一个数据结点）

```
LNode *creat_LinkList(void){
    int data;
    LNode *head, *p, *q;
    head = p = (LNode *)malloc(sizeof(LNode));
    p->next = NULL; //创建头、尾结点
    while(1){
        scanf("%d", &data);
        if(data == NULL) break;
        q = (LNode *)malloc(sizeof(LNode));
        q->data = data; //数据域赋值
        /*钩链，新建的结点总是作为最后一个结点*/
        q->next = p->next;
        p->next = q;
        p = q;
    }
    return(head);   //链表表头作为返回值
}

/*
尾插法创建的表即为顺序
*/
```

单链表查找

- 按序号查找，取出链表中第i个元素（从头开始遍历表）

通过引入工作指针来一个个访问

```
int GetElem_L(LNode *L, int i){
    /*
    L为带头结点的单链表头指针
    当第i个元素存在，其赋值为e并返回OK，否则返回ERROR
```

```
    */
    LNode *p;
    p = L->next;
    int j = 1;
    /*工作指针不为空，则链表不为空*/
    while(p && j < i){
        p = p->next;
        ++j;
    }
    if(!p || i) return 0;
    int e = p->data;
    return e;
}
```

- 按值查找

```
LNode *Locate_Node(LNode *L, int key){
    LNode *p = L->next;
    while(p!=NULL && p->data != key) p = p->next;
    if(p->data == key) return p;
    else{
        printf("索要查找的结点不存在");
        return(NULL);
    }
}
```

插入

插入位置为i，需要找到第i-1个元素，以i-1元素为直接前去，做插入

```
int FindElem_L(LNode L, int i){
    LNode *p = L;
    int j = 0;
    while(p && j < i-1){
        //寻找位置
        p = p->next;
        ++j;
    }
    if(!p || j > i-1) return 0;
    /*执行插入操作*/
    //创建节点
    LNode *s;
    s = (LinkedList)malloc(sizeof(LNode));
    s->data = e;
    //插入
    s->next = p->next;
    p->next = s;
    return 1;
}
```

删除

- 按位置删除

找前驱，防断链。删除位置为i，找到i-1个元素，然后做删除

```
int DeleteElem_L(LNode L, int i){
    LNode *p = L;
    LNode *q;
    int j = 0;
    while(p->next && j < i-1){
        //寻找位置
        p = p->next;
        ++j;
    }
    if(!(p->next) || j > i-1) return 0; //删除位置不合理
    /*执行插入操作*/
    q = p->next;
    p->next = q->next;
    free(q);
    return 1;
}
```

- 按值删除

找前驱，防断链，引入结对指针：让p是q的直接前驱，让q是p的直接后继

```
void Delete_LinkList(LNode *L, int key){
    LNode *p = L, *q = L->next;
    while(q != NULL && q->data != key){
        p = q;
        q = q->next;
    }
    if(q->data == key){
        p->next = q->next;
        free(q);
    } else {
        printf("所要删除的节点不存在\n");
    }
}
```

- 变形1：把所有值为key的结点都删除

对每个结点进行检查，值为key则删除，然后继续检查下一个结点，知道所有节点都被检查到。

```
void Delete_LinkList_Node(LNode *L, int key){
    LNode *p = L, *q = L->next;
```

```
    while(q != NULL){
        if(q->data == key){
            //在此处对匹配的结点删除
            p->next = q->next;
            free(q);
            q = p->next;
        } else {
            //若不匹配，则结对指针后移
            p = q;
            q = q->next;
        }
    }
}
```

- 变形2：把所有相等的元素去掉

```
void Delete_Node_value(LNode *L){
    LNode *p = L->next, *q, *ptr;
    /*在该算法里，p为基准（key），q和ptr作为结对指针*/
    while(p != NULL){
        q = p, ptr=p->next;
        while(ptr != NULL){
            if(ptr->data == p->data){
                q->next = ptr->next;
                free(ptr);
                ptr = q->next;
            } else {
                q = ptr;
                ptr = ptr->next;
            }
        }
        p = p->next;
    }
}
```

单链表合并

现有两个有序单链表La、Lb，合并为Lc为表头的有序链表

- 双指针，不回溯
- 尾插法：
    - 谁小，谁尾插
    - 谁小，谁后移
    - 相等即删除

```
LNode *erge_linkList(LNode *La, LNode *Lb){
    LNode *Lc, *pa, *pb, *pc, *ptr;
    Lc = La;
```

```
    pc = La;
    pa = La->next;
    pb = Lb->next;
    while(pa != NULL && pb != NULL){
        /*谁小谁尾插，谁小谁后移*/
        if(pa->data < pb->data){
            pc->next = pa;
            pc = pa;
            pa = pa->next;
        } else if(pa->data > pb->data){
            pc->next = pb;
            pc = pb;
            pb=pb->next;
        } else if(pa->data == pb->data){
            //相等即删除
            pc->next = pa;
            pc = pa;
            pa = pa->next;
            ptr = pd;
            pb = pb->next;
            free(ptr);
        }
    }
    if(pa != NULL) pc->next = pa;
    else pc->next=pd;   //将剩余结点连接上
    free(Lb);
    return(Lc);
}
```

循环链表

尾指针指向头结点

判断是否是空链表：head->next == head;
判断是否表尾结点：p->next == head;

双向链表

每个节点中设立两个指针域，prior指向前驱，next指向后继
(若在此基础上首位连接也可构成双向循环链表)

prior | data | next

(p->prior)->next == p == (p->next)->prior

插入操作

- 后插

1. 找前驱
2. 防断（先右后左）
    1. p->next->prior = s;

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

- 前插

1. 找后继
2. 防断链（先左后右）
    1. p->prior->next = s;
    2. s->prior = p->prior;
    3. p->prior = s;
    4. s->next = p;

删除操作

断链，释放节点

p->prior->next = p->next;
p->next->prior = p->prior;
free(p);

- 后删

删除其直接后继

1. 找前驱
2. 防断链

q = p->next;
p->next = q->next;
p = q->next->prior;
free(q);

- 前删

q = p->prior;
p->prior = p->prior;
q->prior->next = p;
free(q);

静态链表（顺序结构和链式结构的组合体）

用数组来实现链表

1. 使用结构体数组，内含指针域cur和数据域data
2. 一个数组分量表示一个结点，用cur表示结点在数组中相对位置

增删改查只需修改指针即可

链式结构和顺序结构的比较

比较内容	链式结构（重点）	顺序结构
实现形式	结构体	数组
存储空间	可以连续，可以离散	连续
存储效率	低	高
插入元素	无需移动元素	需要移动元素
删除元素	无需移动元素	需要移动元素
查找	顺序查找	顺序和随机存储
扩展	按需扩展	扩展困难

补充细节内容

该部分内容是做例题时的错题知识点

- 线性链表访问第i个元素的时间复杂度为O(n)
- 顺序表访问节点是随机访问，时间复杂度为O(1)；增加、删除结点需要移动大量元素，时间复杂度为O(n)
- 一个时间复杂度为O(1)的顺序表逆置算法

```
/*
该算法利用收尾元素下标和为length-1的特性进行处理
数组逆置策略
*/
void reverse(SqList &L){
    ElemType x;
    for(int i=0; i<L.length/2; i++){
        x = L.data[i];
        L.data[i] = L.data[L.length-i-1];
        L.data[L.length-i-1]=x;
    }
}

/*
如果做的是部分逆序
例如A[to, from]
则是（to+i, from-i）的区间
*/
for(int i=0; i<(to-from+1)/2; i++){
    sawp(); //交换操作
}
```

- 一个高效的奇数提前、偶数放后的算法

```
/*
从左往右找偶数，从右往左找奇数
两边都找到后做交换
直到二者相遇
*/
void oddPreEven(SqList &L){
    int i=0; j=L.length-1, k;
    ElemType temp;
    while(i <= j){
        while(L.data[i]%2 == 1) i++;    //指向偶数
        while(L.data[j]%2 == 0) j--;    //指向奇数
        if(i<j){
            //交换
            temp = L.data[i];
            L.data[i] = L.data[j];
            L.data[j] = temp;
        }
    }
}
```

- 去除单链表中的重复元素

```
void deleteNodeValue(LNode *L){
    LNode *p = L->next, *q, *qtr;
    while(p != NULL){
        *q = p, *ptr = p->next;
        while(ptr != NULL){
            if(ptr->data == p->data){
                q->next = ptr->next;
                free(ptr);
                ptr = q->next;
            } else {
                q = ptr;
                ptr = ptr->next;
            }
        }
        p = p->next;
    }
}
```

- 将一个单链表逆序，可以考虑头插法逆序的特点，来将表倒置
- 时间复杂度为O(1)的顺序表删除指定元素的算法

```
void delLnode(SqList &L, int x){
    int k=0, i=0;
    while(i<L.length){
        if(L.data[i] == x) k++; //如果是要删除的元素，继续遍历
        //如果不是删除元素，则将其放在已经处理好的在最后一个不为x的位置
```

```
        else L.data[i-k] = L.data[i];
        i++;
    }
    L.length -= k;  //删除k个元素，则长度-k
}
```

栈

概念

先进后出，限制在表的一端进行插入删除

设栈S=(a1, a2, ..., an), a1为栈底元素，an为栈顶元素。退栈的第一个元素为栈顶元素。
元素在进栈（压栈）过程中可以随时出栈（弹栈），所以出栈顺序不一定。

出栈排列组合个数：1/(n+1)*Cn_2n*
*e.g. 对于4个元素的栈：1/(5)((8765)/(4*321)) = 14

顺序栈（分为动态和静态）

动态顺序栈用一维数组存储，栈的大小可以增加，实现复杂

静态顺序栈不能增大存储空间，实现简单

顺序栈用的比较多，因为顺序栈在增删时无需移动元素，避开了顺序表插入元素时需要移动大量元素的缺点，同时顺序表的随机存储特性能提高效率。

动态顺序栈

bottom表示栈底指针，固定不变；top表示栈顶指针，随着进栈、退栈变化

空栈：top和bottom都指向第一个位置；
元素a进栈：bottom指向a，top指向a的下一个位置；
出栈：top指向栈顶元素，然后栈顶元素取出。

动态顺序栈存储的临界条件

设开辟n个空间存储栈元素：

1. 空栈：bottom == top
2. 满栈：|bottom - top| >= n
3. 元素个数：|bottom - top| 个
4. 对于指针移动操作
    - 若入栈时先移动指针再入栈，则在出栈时先出栈再移动指针
5. top的指向位置：
    - 若一开始指向合法位置：指向栈顶元素下一位置【不做特殊说明时，此项为默认】
    - 不合法：指向栈顶元素

具体情况：

1. 初始化时，top = bottom = -1
    1. bottom == -1
    2. 空栈：bottom == t == -1
    3. 入栈：t++; push(e)
    4. top指向栈顶元素

        5. 出栈：pop(e); t--
        6. 满栈：top == n-1
        7. 个数：(top - bottom) 个
    2. 初始化时：top = bottom = 0
        1. bottom == 0
        2. 空栈：bottom == top
        3. 入栈：push(e); top++
        4. top指向栈顶元素下一位
        5. 出栈：top--; pop(e)
        6. 满栈：(top-bottom) >= n
        7. 个数：(top-bottom) 个
    3. 初始化时：top = bottom = n
       从上往下入栈
        1. bottom == n
        2. 空栈：bottom == top == n
        3. 入栈：t--; push(e)
        4. top指向栈顶元素
        5. 出栈：pop(e); t++
        6. 满栈：(bottom - top) >= n
        7. 个数：(bottom - top) 个
    4. 初始化时：top = bottom = n-1
       从上往下入栈
        1. bottom == n-1
        2. 空栈：bottom == top == n-1
        3. 入栈：push(e); t--
        4. top指向栈顶元素的下一空位
        5. 出栈：t++; pop(e)
        6. 满栈：(bottom - top) >= n
        7. 个数：(bottom - top) 个

动态的定义和初始化，进栈和出栈

```
#define STACK_SIZE 100      //初始大小
#define STACKINCREMENT 10   //存储空间分配增量

/*定义*/
typedef struct sqstack{
    int *bottom;
    int *top;
    int stacksize;  //当前已分配空间，以元素为单位
}SqStack;

/*初始化*/
int Init_Stack(void){
    SqStack S;
    S.bottom = (int *)malloc(STACK_SIZE *sizeof(int));
    if(! S.bottom) return 0;
    S.top = S.bottom;
```

```
    S.stacksize = STACK_SIZE;
    return 1;
}

/*进栈*/
int push(SqStack S, int e){
    if(S.top->S.bottom > S.stacksize){  //如果满栈，追加存储空间
        S.bottom = (int *)realloc((S.STACKINCREMENT + STACK_SIZE)*sizeof(int));
        if(! S.bottom) return 0;
        S.top = S.bottom + S.stacksize;
        S.stacksize += STACKINCREMENT;
    }
    *S.top = e;
    S.top++;
    return 1;
}   //如果在追加存储空间时，没有那么大的连续存储空间，容易报错

/*出栈*/
int pop(SqStack S, int *e){
    if(S.top == S.bottom) return 0;     //判空
    S.top--;
    *e = *S.top;
    return 1;
}
```

静态顺序栈

静态的定义和初始化，进栈和出栈

```
#define MAX_STACK_SIZE 100      //栈向量大小

/*定义*/
typedef struct sqstack{
    int stack_array[MAX_STACK_SIZE];
    int topl
}SqStack;

/*初始化*/
SqStack Iint_Stack(void){
    Sqstack S;
    S.top = ;
    return(S);
}

/*进栈*/
int push(SqStack S, int e){
    if(S.top == MAX_STACK_SIZE) return 0;   //判满
    S.Stack_array[S.top] = e;
    S.top++;
    return 1;
}
```

```
/*出栈*/
int pop(SqStack S, int *e){
    if(S.top == 0) return 0;    //判空
    S.top--;
    *e=S.stack_array[S.top];
    return *e;
}
```

对顶栈

【408还没出现（截至2022），北京大学考过】

若内存不足，可以考虑把两个栈共享一片空间

两个栈共享同一片空间;
把两个栈的栈底设在数组的两端;
|top1 - top2| == 1 表示栈满

链栈

存储结构为链式存储的栈，一种运算首先的单链表

比起顺序栈，它不会出现满栈的情况

定义和初始化，入栈和出栈

```
/*定义*/
typedef struct Stack_Node{
    int data;
    struct Stack_Node *next;
}STack_Node;

/*初始化*/
Stack_Node *Init_Link_Stack(void){
    Stack_Node *top;
    top = (Stack_Node *)malloc(sizeof(Stack_Node));
    top->next = NULL;
    return(top);
}

/*入栈*/
int push(Stack_Node *top, int e){
    Stack_Node *p;
    p = (Stack_Node *)malloc(sizeof(Stack_Node));
    if(!p) return 0;    //空间申请失败，结点未创建
    p->data = e;
    p->next = top->next;
    top->next = p;  //钩链
    return 1;
}
```

```
/*出栈*/
int pop(Stack_Node *top, int *e){
    Stack_Node *p;
    int e;
    if(top->next == NULL) return 0; //栈空，返回错误
     p = top->next;
     *e = p->data;  //取出栈顶元素
     top->next = p->next;   //修改栈顶指针
     free(p);
     return 1;
}
```

应用

考试中可以直接使用，无需定义：

top(), bottom(), push(), pop(), initStack()

数学运算中的括号匹配

读到左括号，pop()，读到右括号，push与读到的左括号匹配
匹配成功，继续读入；反之返回FALSE

```
int Match_Brackets(){
    char ch, x;
    scanf("%c", &ch);
    while(asc(ch) != 13){   //程序到回车结束
        if(ch == '(' || ch == '[') push(S, ch);
        else if(ch == ']'){
            x = pop(S);
            if(x != '[') {
                printf("括号不匹配");
                return 0;
            } else if(ch == ')'){
                x = pop(S);
                if(x != '('){
                    printf("括号不匹配");
                    return 0;
                }
            }
            if(S>top != 0){
                printf("括号数量不匹配");
                return 0;
            } else return 1;
        }
    }
}
```

进制转换（辗转相除法）

e.g. 十进制转八进制：
(D/8)%8，得到的余数进栈，
随后(D/8/8)%8，得到的余数进栈，
...
依此类推，直到D==0，将栈中元素逐个出栈，即可得到八进制数

```
void conversion(int D, int d){
    //将十进制D转为d进制数
    SqStack S;
    int k, *e;
    S = Init_Stack();
    while(D>0){
        k = D%d;
        push(S, k);
        D = D/d;
    }
    while(S.top != 0){
        pop(S, e);
        printf("%d", *e);
    }
}
```

将递归调用用栈实现

递归的最后一次调用会先进行处理（top），贴合于栈的后进先出的特性，可以借助栈来转换为非递归算法

表达式求值

e.g. 对于b-(a+5)*3 //这是一种中缀表达式*
后缀表达式：ba5+3- //使用最多 前缀表达式：-b*+a53
注意：数据顺序不限，变化的是运算符号的位置和顺序

在使用栈进行表达式求值时，设立两个栈：数栈、符栈。当符栈push进高优先级的运算符号时，数栈进行相应运算。

中缀转后缀

中缀转后缀同样需要两个栈，根据运算符的优先级，将符栈的元素pop并push进数栈

队列

一种先进先出的运算受限的线性表，只在表的一端插入，另一端删除

队首（front）：只允许删除
队尾（rear）：只允许插入

基本操作：

- Create();　//创建一个空队列
- EmptyQue();　//判空
- InsertQue(x);　//向队尾插入元素
- DeleteQue(x);　//删除队首元素

顺序队列

利用一维数组（连续存储单元）存储队列，和线性表一样由着动、静态之分

静态顺序队列

```
#define MAX_QUEUE_SIZE 100

/*初始化*/
typedef struct queue{
    int queueArray[MAX_QUEUE_SIZE];
    int front;  //始终指向头元素
    int rear;   //始终指向队尾元素的下一位
}sqQueue;
```

入队：
　先放元素
　rear++
出队：
　先出元素
　front++

此操作容易假溢出，解决方案：

- 引入length：
    - 入队：length++
    - 出队：length--
    - 此时，length==0为空；length==n为满。入队和出队共享资源，需要互斥访问，需要相关的进程管理机制，对结构进行控制
- 引入一个flag：
    - 当队列满的时候flag=1
    - 反之flag=0

注意：**以上两种方法容易造成更复杂的管理方式**

- 将队列看成一个首尾相连的队列，形成**循环队列**

循环队列【重点】

循环队列舍弃一个空间用于判断队空、队满

```
if(i+1 == MAX_QUEUE_SIZE) i = 0;
else i++;

//初始化
rear = front = 0

//循环队列为空
front == rear

//循环队列满
front == (rear+1) % MAX_QUEUE_SIZE

//入队
rear = (rear+1) % MAX_QUEUE_SIZE

//出队
rear = (rear+1) % MAX_QUEUE_SIZE

//计算元素个数
len = (rear - front + MAX_QUEUE_SIZE) % MAX_QUEUE_SIZE

/*
如果非空状态下，front指向队首元素的前一个空位，rear指向队尾
入队、出队需要分别让rear、fornt先++，然后再执行操作
同时初始化时，需要rear = front = n-1
*/

/*
如果非空状态下，front指向队首，rear指向队尾，需要引入length字段
判空：front == rear；length = 0
判满：front == rear；length = n
入队时先rear++，再入队
出队时先出队，再front++
*/
```

循环队列的初始化，入队、出队、计算长度

```
/*循环队列初始化*/
sqQueue initCirQueue(void){
    sqQueue Q;
    Q.front = Q;
    Q.rear = 0;
```

```
    return(Q);
}

/*入队*/
int insertCirQueue(sqQuene Q, int e){
    if((Q.rear+1) % MAX_QUEUE_SIZE == Q.front)  return 0;   //判满
    Q.queueArray[Q.rear] = e;   //插入e
    Q.rear = (Q.rear+1) % MAX_QUEUE_SIZE    //入队指针移动
    return 1;
}

/*出队*/
int deleteCirQueue(sqQueue Q, int *x){
    if(Q.front == Q.rear) return 0;     //判空
    *x = Q.queueArray[Q.front];     //取出队首元素
    Q.front = (Q.front+1) % MAX_QUEUE_SIZE; //移动首指针
    return 1;
}
```

链式队列

限制表头删除、表尾插入的单链表

头出：头指针始终指向头结点
尾插：尾指针始终指向尾节点

定义

```
/*定义数据指针节点*/
typedef struct QNode{
    int data;
    struct Qnode *next;
}qNode;

/*定义首尾指针结点*/
typedef struct LinkQueue{
    qNode *fornt, *rear;
}linkQueue;
```

链队运算以及指针变化

- 若带头结点：
    1. 插入：尾改，头不改
    2. 删除：
        1. **当删除最后一个节点时，头尾均要改**
        2. 其它情况头改，尾不改
- 若不带头结点：
    1. 插入：
        1. **插入第一个结点，头尾均要改**

        2. 插入其他结点时，尾改，头不改
    2. 删除：
        1. 当删除最后一个结点，头尾均要改
        2. 其它情况头改，尾不改

链队列的初始化，入队、出队

```
/*初始化*/
linkQueue *initLinkQueue(void){
    linkQueue *q;
    qNode *p;
    p = (qNode *)malloc(sizeof(qNode)); //开辟头结点
    p->next = NULL;
    q = (linkQueue *)malloc(sizeof(linkQueue)); //开辟联队指针结点
    q.front = q.rear = p
    return(Q);
}

/*入队*/
int insertCirQueue(linkQueue *q, int e){
    p = (qNode *)malloc(sizeof(qNode));
    if(!p) return 0;  //结点申请失败
    p->data = e;
    p->next = null;
    q.rear->next = p;
    q.rear = p; //在队尾插入新节点
    return 1;
}

/*出队*/
int deleteLinkQueue(linkQueue *q, int *x){
    qNode *p;
    if(q.front == q.rear) return 0;    //判空
    p = q.front->next;  //取队首结点
    *x = p->data;
    q.front->next = p->next;    //修改队首指针
    if(p == q.rear) q.rear = q.front;   //队列只有一个结点时，防止丢失队尾指针
    free(p);
    return 1;
}
```

双端队列（2010、2021）

栈和队列的组合体，分为两种类型：

- 输出受限：两端均可插入，限制一端删除
- 输入受限：两端均可删除，限制一端插入

队列的应用

- 排队业务、打印机服务、挂号系统、etc.
- 树的层次遍历
- 图的广度优先遍历

补充

- 用单链表的队列的几种形式：
    1. 带有头指针和尾指针，最简便
    2. 带尾指针的单循环链表
    3. 双向链表：
        1. 带尾指针的双向循环列表
        2. 带头指针的双向循环列表
- 对于第一个进入队列的存储位置在A[0]的循环队列中（数组A[0..n-1]用于存储），初始时front=0，rear=n-1
- 不带头结点的链队列在出队操作时，修改尾指针的情况发生在出队后队列为空的时候

树和二叉树

树是一种非线性结构，特点为：分支关系，一对多，层次结构。

二叉树是度为2的树。

基本概念

1. 结点node：一个数据元素和若干指向子树的分支。
2. 结点的度、树的度degree：结点度是结点所拥有的子树数，树的度是树中结点度的最大值。
3. 叶子节点left、非叶子节点（非终端节点、分支节点）：叶子结点是度为零的结点，非叶子节点反之。
4. 孩子节点（子节点）、双亲结点、兄弟节点：孩子节点是某一结点的子树根，双亲结点为该结点，兄弟结点是来自同一个根的子树根。
5. 层次、堂兄弟结点：从根开始，根为第一层，其孩子为第二层；堂兄弟节点是双亲结点在同一层的所有结点。
6. 层次路径、祖先ancestor、子孙descent：层次节点是从根到某一结点的层次路径，祖先是该路径上所有除自己的结点，子孙结点是以某一点为根的子树中任意节点。
7. 深度depth：树中结点的最大层次值，又为数的高度。
8. 有序树、无序树：有序树是每一个节点的子树有一定次序，否则即为无序树。
9. 森林forest：若干棵互不相交的树集合（若删除一棵树的根节点，子树就构成森林）。

性质

1. 节点个数等于所有节点的度之和+1。
2. 定义m度树的叶子节点数量为n0，节点总数为N，度为1的节点数量为n1...度为m的节点个数为nm:
   - n0 = n2 + 2×n3 +...+ (m-1)×nm + 1
   - N = n0 + n1 + n2 +...+ nm
     - = ∑D + 1
     - = 0×n0 + 1×n1 + 2×n2 +...+ m×nm + 1
3. 在非空m度树，第i层至多由m^(i-1)个结点（i>=1）。
4. 高度为h的m度树，最多有(m^h-1)/(m-1)个结点：
   - 最多情况下结合性质3，将构成首项为1，公比为m的等比数列，当要求解前h项时：Sh = a1(1-m^n) / (1-m) = (m^n-1) / (m-1)
5. 具有n个结点的m度树，最小高度 = ┍logm(n×(m-1)+1)┐：
   1. 当高度一定，每层节点个数越多，其对应的总结点数就越多；反之越少。
   2. 当总节点个数一定，每层节点个数越多，其高度越小；反之越高。

存储结构

双亲表示法（顺序存储）

用顺序存储来保存树的结点，同时每个节点附加一个指示器（整数域）来表示双亲结点的位置（下标值）。

利用了双亲唯一的性质，可以快速找到任意父节点，但子节点需要遍历整个数组。

```c
#define MAX_SIZE 100

typedef struct PTNode{
    int data;
    int parent;     //标记双亲结点
}PTNode;

typedef struct{
    PTNode nodes[MAX_SIZE];
    int root;    //根节点位置
    int num;     //节点数
}PTree;
```

孩子链表示法（链式结构）

每个节点有多个指针域指向对应子树的根节点。有定长节点结构，不定长结构，孩子兄弟表示法。

1. 定长节点结构

结构简单统一，指针域浪费明显。在一颗n个结点的树，度为k的树中必有n(k-1)+1个空指针域。

2. 不定长结构

树中每个结点的指针域数量不同，以此作为该节点的度。没有多余的指针域，但操作不便。

3. 孩子兄弟表示法（最常用）

用二叉链表作为存储结构（故也叫**二叉树表示法**），用两个指针域分别指向第一个子节点和下一个兄弟节点。

```c
typedef struct Csnode{
    ElemType data;
    struct Csnode *firstchild, *nextsibing;
}CSNode;
```

存储

顺序存储

自上而下、自左而右的完全二叉树，完全按照其编号来存储。

对于非完全二叉树，将用空指针填充，使其“成为完全二叉树”，容易造成空间浪费问题，性能较差。

```c
#define MAX_SIZE 100

int SQBTree[MAX_SIZE];
```

链式存储

每个节点包含三个域：数据域，左子节点指针域，右子节点指针域。

```
typedef struct btNode{
    int data;
    struct btNode *LChild, *RChild;
}BTNode;
```

对于三叉链表

额外增加一个指向父节点的指针域

```
typedef struct btNode_3{
    ElemType data;
    struct btNode_3 *LChild, *RChild, *parent;
}BTNode_3;
```

遍历

对二叉树的各个节点进行一次访问，按照先左后右的原则。

所有遍历里，叶子节点的顺序一定不变。

以此分为三种遍历情况：DLR、LDR、LRD，外加一个逐层遍历的层次遍历。

- DLR：先序遍历（根左右）

```
//递归先序遍历
void PreOrderTraversal(BTNode *root){
    if(root != NULL){
        printf(root->data);
        PreOrderTraversal(root->LChild);
        PreOrderTraversal(root->RChild);
    }
}
```

- LDR：中序遍历（左根右）

```
//递归中序遍历
void inOrderTraversal(BTNode *root){
    if(root != NULL){
        inOrderTraversal(root->LChild);
        printf(root->data);
        inOrderTraversal(root->RChild);
```

```
    }
}
```

- LRD：后序遍历（左右根）

```
//递归后序遍历
void postOrderTraversal(BTNode *root){
    if(root != NULL){
        postOrderTraversal(root->LChild);
        postOrderTraversal(root->RChild);
        printf(root->data);
    }
}
```

- 层次遍历（逐层遍历）
  从根出发，逐层遍历

```
#define MAX_NODE 50
void levelOrder(BTNode *T){
    BTNode *Queue[MAX_NODE], *p = T;
    int front = 0, rear = 0;
    if(p != NULL){
        Queue[rear++] = p;
        while(front < rear){
            p = Queue[++front];
            printf(p->data);
            if(p->LChild != NULL) Queue[rear++] = p->LChild;
            if(p->RChild != NULL) Queue[rear++] = p->RChild;
        }
    }
}
```

二叉树的构造

中序确定左右，先序or后续确定根。如下遍历结果的组合可以唯一确定一棵二叉树：

- 中序遍历 + 层次遍历
- 中序遍历 + 后序遍历
- 中序遍历 + 先序遍历

线索二叉树

一颗二叉树有n个结点，有n-1条边（即指针的连线），有2n个指针域，有n+1个空闲指针域——那么可以利用它们来存放**遍历后**的直接前驱和直接后继的信息：

- 若结点没有左孩子，则LChild指向直接前驱
- 若结点没有右孩子，则RChild指向直接后继

但是对于后序遍历（后序二叉线索树）找直接后继节点依然很困难

树与树之间的转换

遵循先先后中：树的先序对应转化后二叉树的先序；树的后续对应转化后二叉树的中序。

将树转化为二叉树

对于非二叉树，可以将其转换为一颗唯一二叉树。具体方法如下：

1. 逐层遍历，从左往右在兄弟节点之间虚线连接；
2. 随后除了最左的第一个子节点，去除父节点与其它子节点的连线；随后顺时针旋转45°，原有实线左旋；
3. 最后所有虚线改为实线并右斜。

如此转换后，根节点没有右子树；左子树中沿右链往下的右子节点均为原来树中的兄弟结点。

将二叉树转回树

右子节点与父节点连上虚线，去除右子节点的连线，随后即可还原树。

森林转为二叉树【典型考题】

1. 将森林里的每棵树均转为二叉树。
2. 随后从最后一棵二叉树开始，每棵二叉树作为前一棵二叉树的根节点的右子树——
3. ——这样一来，第一棵树的根节点就成为转换后的二叉树根节点。

哈夫曼树Huffman（最优二叉树）

源于哈夫曼编码：一种不等长编码，更为常用的编码更短，不常用的可以用较长的编码——这样使得整段字节的编码量通常不会太长。

- 节点路径：一个节点到另一个结点之间的分支构成该二者间的路径
- 路径长度：节点路径上的分支数目
- 权（值）：各种开销、代价、频度、etc
- 结点的带权路径长度：某节点到根节点间的 路径长度 × 权
- 树的路径长度：树根到每一个结点的路径长度之和
- 树的带权路径长度（WPL）：所有叶子节点的**带权路径长度**之和
- Huffman树：在具有n个叶子节点的二叉树中，WPL值最小的数（让权重较大的结点放置于路径长度较小的位置）

Huffman树构造【高考频】

1. 根据n个权值W构成n棵二叉树集合F（每棵二叉树只有权值为Wi的结点，无左右子树）；
2. 在F中取两棵权值最小的树，作为左右子树，来构成新二叉树，其根节点的权值为左右子树权值之和；
3. 删除这两棵树，并将组合成的新树加入F；
4. 重复2和3，直到剩下一棵树。
5. P.S. 为了规范，权值较小的作为左子树。权值一样则让低的在左，高的在右。

如此操作后，每个字符都是叶子节点，字符不可能出现在路径上——所以每个字符的Huffman编码不可能是另一个字符编码的前缀。

哈夫曼树结论

- 只有0度和2度的结点
- WPL值最小
- 由于左右子树可以交换（规范上权值较小的在左边），哈夫曼树不唯一，但是WPL唯一
- 虽然本质上不属于二叉树，但考试中认为其是二叉树
- 上层结点权值不小于下层节点
- 哈夫曼编码只讨论叶子的编码

_

- n个叶子结点的哈夫曼树共有2n-1个结点
- 判断Huffman编码的步骤：
  1. 先找前缀，若存在编码可以作为其它编码的前缀，则判其不是Huffman
  2. 再按照左0右1的规律还原哈夫曼树，若出现度为1（n1）的结点，则不是Huffman树
- 平均编码长度 = WPL / ∑Wi

并查集【2022新加】

通常用树来表示。

并查集存储在一组不相交集合的动态集合S={S1, S2,...,Sk}，每个集合包含一个或多个元素，并选出某个元素作为代表，不关注其具体包含了何种元素——而关注于可以快速找到指定元素所在的集合，以及合并两个元素所在集合。

并查集的操作

1. makeSet(s); // 建立一个新的并查集，其中包含s个单元素集合
2. unionSet(x, y); // 把元素x和元素y所在集合合并，x和y所在集合不能相交（相交则不进行合并操作）。分为按高度合并（按秩合并）、按节点数量合并（数量较少的树的根其父节点指向节点数较多的树根）
3. find(x); // 找到元素x所在集合的代表，时间复杂度为树的高度。此操作亦可用于判断两个元素是否再同一个集合中，只需要比较集合代表即可

图

一个图对应一个偶对，G = (V, E)。
V（vertex）是顶点的非空集合；E（edge）是无需集V&V的一个子集，记为E(G)，元素为弧（边Arc）：

$G = (V, E)$
$V = \{v \mid v \in \text{data object}\}$
$E = \{\langle v, w\rangle \mid V, W \in V \wedge p(v, w)\}$
P(v, w)表示从顶点v到顶点w有一条直通通路

图分为有向图Digraph和无向图Undigraph（E = {(v, w) | ... }）。

设图中顶点数为n，边的数目为e，则
对于无向图：$e \in [0, n(n-1)/2]$；
对于有向图：$e \in [0, n(n-1)]$。

408要求中都是简单图，不含平行边和自环。

图的种类

1. 完全无向图
   具有 n(n-1)/2 条边的无向图，即不同顶点间均有一条无向边。

2. 完全有向图
   具有n(n-1)条边的有向图，即不同顶点间均有一条弧。

3. 稀疏图和稠密图
   有很少的边或弧的图（e<nlogn）称为稀疏图，反之为稠密图。

4. 子图
   设有图 G=(V, E) 和 G'=(V', E')，V'∈V且E'∈E，则G'是G的子图。

5. 包含子图
   子图包含原图的全部顶点和部分边，称其为包含子图（V'=V, E'∈E）。

图的基本概念

设图G=(v, w)，若为有向图则为G=<v, w>

1. 如果两点互为邻接点，则边(v, w)依附于v和w。对于有向图，若两点相互有弧，则弧<v, w>与v和w相关联。

2. 度：某一顶点为起点的有向边的数量为顶点的出度OD；反之以其为终点的则为入度ID。该顶点的出度和入度之和称为度TD=OD+ID。

3. 路径长度：路径上边的数量。

4. 简单路径：一条路径中没有重复同一个顶点。

5. 回路（环）：一条路径中第一个顶点和最后一个顶点相同。

6. 简单回路（简单环）： 回路中除了头尾外没有重复顶点。
7. 连通图：图中 任意vi,vj∈V 都是联通的。反之则是非连通图
8. 连通分量：极大的连通子图。
9. 强连通图：任意两个点相互为起点、终点出发，都有有向路径。反之则为非强连通图，其极大的强连通子图称为G的强连通分量。
10. 生成树：一个连通无向图的一个极小连通子图，它有原图的全部顶点以及刚好构成一棵树的边数量。对于生成树：
    1. n个顶点的生成树有且仅有n-1条边；
    2. 如果一个图有n个顶点和n-1条边，则是非连通图；
    3. 多于n-1条边，则一定有环；
    4. 有n-1条边的不一定是生成树。
11. 网：边带有权值的图。

补充

- 对于求n条边至少要多少个顶点为连通的问题，设有x个顶点：
    - 先设置两个极端的情况：
        - 0条边
        - 构成完全图 （ n(n-1)/2 ）
    - 那么此时可以考虑减少一个顶点，即可得到 (x-1)(x-2)/2 ≥ n。
    - 对于n个顶点求至少要多少条连通边亦是如此。

存储结构

邻接矩阵，邻接链表

邻接矩阵（数组）表示法

用一维数组`vexs[n]`存放顶点，用二维数组`A[n][n]`存放顶点之间的关系。

邻接矩阵是对称方阵，并且是唯一的；对于顶点vi，其度数是第i行的非0元素的个数；无向图的边数是上三角（或下三角）矩阵中的非零元素个数。

特性：

1. 给定图，邻接矩阵唯一；
2. 所占空间开销`O(N^2)`，N为顶点个数，与边的个数无关；
3. 适用于稠密图，不适用于稀疏图；
4. 计算度需要用遍历或列来实现。

无向图的邻接矩阵

1. 无权无向图：
   对于`A[i][j]`：若`(vi, vj) ∈ E`，二者邻接；否则不邻接。

2. 带权无向图：
   对于`A[i][j]`：若`Wij (vi, vj) ∈ E`，二者邻接，权值为Wij；否则∞，不邻接。

有向图的邻接矩阵

1. 无权有向图：
   对于`A[i][j]`：若`<vi, vj> ∈ E`，从vi到vj有弧；否则没有弧。

2. 带权有向图：
   对于`A[i][j]`：若`Wij (vi, vj) ∈ E`，二者邻接，权值为Wij；否则∞，不邻接。

连接矩阵的定义和基本操作

```
#define INFINITY MAX_VAL     //最大值∞
#define MAX_VEX 30           //最大顶点数

typedef enum {DG, AG, WDG, WAG} GraphKind;  //{有向图，无向图，带权有向图，带权无向图}

typedef struct ArcType{
    int vex1, vex2;          //弧或边所依附的两个顶点
    ArcValType arcVal;       //弧或边的权值
    ArcInfoType arcInfo;     //弧或边的其它信息
}arcType;    //弧或边的定义

typedef sturct{
    GraphKind kind;              //图的类型
    int vexnum, arcnum;          //图当前顶点数和弧数
    int vexs[MAX_VEX];           //顶点向量
    int adj[MAX_VEX][MAX_VEX];   //连接矩阵
}MGraph;     //结构定义

/*创建图*/
AdjGraph *createGraph(MGraph *G){
    printf("请输入图种类标志：");
    scanf("%d", &G->kind);
    G->vexnum = 0;   //初始化顶点数量
    return(G);
}

/*顶点定位
（确定一个顶点在vexs数组中的位置（下标），等同于在顺序存储的线性表中查找一个数据元素）*/
int locateVex(MGraph *G, int *vp){
    int k;
    for(k=0; k < G->vexnum; k++){
        if(G->vexs[k] == *vp) return(k);
    }
    return(-1);      //图中无此顶点
}

/*增加顶点
```

```
（类似在线性表的末尾增加一个元素）*/
int addVertex(MGraph *G, int *vp){
    int k, j;
    k = G->vexnum;
    G->vexs[G->vesnum++] = *vp;
    if(G->kind == DG || G->kind == AG){     //是否为带权的有向图或无向图
        for(j=0; j < G->vexnum; j++){
            G->adj[j][k].arcVal = G->adj[k][j].arcVal = 0;
        }
    } else {
        for(j=0; j < G->vexnum; j++){
            G->adj[j][k].arcVal = G->adj[k][j].arcVal = INFINITY;
        }
    }
    return(k);
}

/*增加一条边或弧*/
int addArc(MGraph *G, arcType *arc){
    int k, j;
    k = locateVex(G, &arc->vex1);
    j = locateVex(G, &arc->vex1);
    if(G->kind == DG || G->kind == WDG){    //是否为有向图或带权有向图
        G->adj[k][j].arcVal = arc->arcVal;
        G->adj[k][j].arcInfo = arc->arcInfo;
    } else {    //对于无向图或带权无向图，需要对称赋值
        G->adj[k][j].arcVal = arc->arcVal;
        G->adj[j][k].arcVal = arc->arcVal;
        G->adj[k][j].arcInfo = arc->arcInfo;
        G->adj[j][k].arcInfo = arc->arcInfo;
    }
    return(1);
}
```

邻接表

分别设各个顶点为头结点，将与顶点相邻接的边构成一个单链表。于是分别需要表结点和顶点结点

表结点：adjvex | info | nextarc 【邻接点域 | 数据域 | 下一邻接表结点】
顶点结点（表头结点）：data | firstarc 【数据域 | 链域】

特点：

1. 表头中每个分量就是一个单链表的头结点，分量个数是图中顶点的数目；
2. 稀疏图的情况下更省空间；
3. 无向图中，Vi的度就是第i个链表的节点数；
4. 有向图中：正邻接表出度直观；逆邻接表入度直观。

邻接表的定义和基本操作

```
#define MAX_VEX 30      //最大顶点数

typedef enum{DG, AG, WDG, WAG}GraphKind;

/*弧或边的结构定义*/
typedef struct ArcType{
    int vex1, vex2;
    infoType info;      //与弧相关的信息，例如权值
}arcType;

/*图的结构定义*/
typedef sturct{
    GraphKind kind;
    int vexnum;
    vexNode adjList[MAX_VEX];
}ALGraph;

/*表结点定义*/
typedef struct LinkNode{
    int adjvex;     //邻接点在头结点数组中的位置（下标）
    infoType info;  //弧或边的相关信息，例如权值
    struct LinkNode *nextarc;   //指向下一个表结点
}linkNode;

/*创建图*/
ALGraph *createGraph(ALGraph *G){
    printf("请输入图类型的标志：");
    scanf("%d", &G->kind);
    G->vexnum = 0;
    return(G);
}

/*图顶点的定位*/
int locateVex(ALGraph *G, int *vp){
    int k;
    for(k=0; k < G->vexnum; k++){
        if(G->adjList[k].data == *vp) return(k);
    }
    return(-1);     //图中没有这个顶点
}

/*增加顶点
直接在顶点结点插入*/
int addVertex(ALGraph *G, vexType *vp){
    int k, j;
    G->adjList[G->vexnum].data = *vp;
    G->adjList[G->vexnum].degree = 0;
    G->adjList[G->vexnum].firstarc = NULL;
    k = ++G->vexnum;
    return(k);
}

/*增加边（弧）*/
```

```
int addArc(ALGraph *G, arcType *arc){
    int k, j;
    linkNode *p, *q;
    k = locateVex(G, &arc->vex1);
    j = locateVex(G, &arc->vex2);
    p = (linkNode *)malloc(sizeof(linkNode));
    p->adjvex = arc->vex1;
    p->info = arc->info;
    p->nextarc = NULL;       //边的起始表结点赋值
    q = (linkNode *)malloc(sizeof(linkNode));
    q->adjvex = arc->vex2;
    q->info = arc->info;
    q->nextarc = NULL;       //边的末尾表结点赋值
    if(G->kind == AG || G->kind == WAG){
        /*无向图，用头插法插入到两个单链表*/
        q->nextarc = G->adjList[k].firstarc;
        G->adjList[k].firstarc = q;
        p->nextarc = G->adjList[j].firstarc;
        G->adjList[j].firstarc = p;
    } else {
        /*有向图，用头插法*/
        q->nextarc = G->adjList[k].firstarc;
        G->adjList[k].firstarc = q;     //简历正邻接链表
    }
    return(1);
}
```

补充内容

- 邻接矩阵表示是唯一的，邻接表表示不唯一

遍历【2009-2023都没考代码】

从某一点出发各访问一次其余点。在遍历过程中借助一个辅助向量`Visited[1...n]`记录被访问过的点。

图的遍历算法分为广度优先搜索算法和深度优先搜索算法。
采用邻接表时，由于邻接表不唯一，遍历结果不唯一，但给定邻接表后遍历结果唯一；邻接矩阵的遍历结果唯一。

不同的遍历顺序会生成不同的树；非连通图则会生成森林。

深度优先遍历DFS

从一点出发沿着一条路径深入，直到无法深入后再转回前一个可继续深入其它路径的点，直到遍历完全部顶点。
深度优先遍历利用栈来存储。图中有e条边时时间复杂度为`O(e)`，总时间复杂度为`O(n+e)`；有n个顶点式，`O(n)`。
通常用于拓扑排序、连通分量等。

广度优先遍历BFS

从一点触发访问所有相邻点，再依次访问这些点的相邻点，直到遍历完。
广度优先遍历用队列来存储。图中有e条边时时间复杂度为`O(e)`，总时间复杂度为`O(n+e)`；有n个顶点式，`O(n)`。
通常用于求最短路径。

图的应用

生成树,

生成树

连通图的生成树是一个极小连通子图（包含全部的点，和刚好足够连通的边）：

有`n`个顶点的生成树仅有`n-1`条边；
若`<n-1`，则是非连通图；
`>n-1`则一定有环；
有`n-1`条边不一定是生成树。

一种典型的应用是城市间修路（点表示城市、边表示路、边的权值表示距离，n个城市间可修n(n-1)/2条路，选出其中n-1条使得总路程最短（权值最小））。其基本思想为：

- 选择n-1条边构成最小生成树；
- 尽可能选权值最小的边，且不构成回路。

最小生成树MST算法

有两种算法：普利姆Prim算法（一种贪心算法，贪吃蛇），克鲁斯卡尔Kruskal算法（Prim的延申，相对更为符合最小生成树思想）

普利姆Prim算法

从v0出发，先找权值最小的边，并且不构成环，找到后则构成一个整体，继续找对于这个整体而言权值最小的边，一直重复直到连通所有的点，即为最小生成树。

时间复杂度为`O(n^2)`，与边数目无关。

克鲁斯卡尔Kruskal算法

将边的权值排序，每次选取最小边（前提不构成回路），直到选取完n-1条边。

数组初始化的时间复杂度为`O(n)`；排序权值采用堆排序或快速排序，时间复杂度为`O(eloge)`。

克鲁斯卡尔算法更适用于稀疏图（较多点的度为1）。

MST唯一性讨论

1. 权重均不相等，MST必定唯一；
2. 存在环，且环上有想等权重，同时相等权重比别的全重大，MST不唯一——但权值之和是唯一的。

最短路径

从一个点到另一个点，所经过的边的权值和最小。有迪杰斯特拉Dijkstra算法（贪心算法，路径按长度递增次序产生最短路径）和弗洛伊德Floyd算法（构建二维数组，记录任意两点的最短路径并逐步更新数组）。

Dijkstra

从一点出发，求出到各个顶点的最短路径和路径长度，按长度递增次序生成个顶点的最短路径，直到求出长度最长的最短路径。

但由于是一种贪心算法，所以只关心局部最优解，且**不适用于负权值和负回路。**

步骤如下，采用做表的形式

1. 指定点后，初始化它到各个点的距离（若无法直接到达，则标记为∞），选取其中最小点作为下一个到的点；
2. 随后将该点加入，再求该整体到剩余点的距离（如果更近了（权值更小），则更新，并将其pre更新为该点），选出最近的点作为下一个点；
3. 将该点加入，重复2；
4. 直到所有的点遍历完成。

算法分析

- 初始化时间复杂度`O(n)`；
- 求最短路径二重循环的时间复杂度`O(n^2)`。

因此，整体的时间复杂度为`O(n^2)`。

但以上只是从一点到其余点的时间复杂度。如果是求各个点到其余点的最短路径，时间复杂度将是`O(n^3)`。

Floyd

构建二维数组，记录最短长度，然后逐步更新数组中的数值，直到求出所有结点之间的最短路径。

适用于**稠密图**，且可以处理负权边（但不允许负回路），时间复杂度`O(n^3)`；
空间复杂度`O(n^2)`。

步骤如下

1. 列出原图的邻接矩阵A，途径点矩阵Path（初始值均为-1），途径点数组S（初始化为空）；
2. 随后选一个点作为途径点，若经过该点可以获得更短的路径，则更新A上的权值；
3. 将该点加入S；
4. 对于更新权值的点，更新Path为当前途径点；
5. 重复2、3、4，直到S内点的数量等于图的数量；
6. 结合A和Path，Path中每一行的最大值，即为对应A上该点到另一点的最短路径。

AOV网与拓扑排序【占题目的40%】

AOV网是顶点表示活动的有向无环图。
AOV网用于工程项目中的工序、工程时间进度的问题，是一种有向无环图，顶点表示活动，有向边表示活动间优先关系。

拓扑排序是AOV的遍历算法，用于确定活动执行顺序。
当一个有向无环图所组成的序列中：当每个顶点仅出现一次，且后继节点绝对没有通往前驱节点的路径，即可使用拓扑排序。

时间复杂度为`O(n+e)`。

拓扑排序的过程

1. 选择一个没有前驱结点的顶点并输出；
2. 删除该点，以及从该店出发的所有弧；
3. 重复1、2，直到全部顶点输出（或图中不存在无前驱的顶点）。

一种快速排除AOV错误遍历的方式

若选项中出现了违逆弧指向的遍历，则为错误。

AOE网与关键路径

AOE网是边表示活动的有向无环图，通常边的权值是活动时间。

关键路径是从起点到终点的最长路径（完成工程的最长时间，是影响整个工程的关键）。
关键活动是关键路径上的活动。增加关键活动，关键路径必然增长。

事件最早/最晚发生时间

1. 绘制表格，顶横为活动（边），顶列为最早发生时间、最晚发生时间、松弛量（最早、最晚发生事件的差值）。
2. 求最早发生时间，用正推法（从左到右），取权值和最大的那条路径；
3. 求最晚发生时间，用倒推法（从右到左），取权值和最小的哪条路径，并用关键路径的长度-该路径长度；
4. 求松弛量。

时间复杂度总结

`O(n+e)`

邻接表创建的：BFS、DFS、拓朴排序、AOE网，创建表、插入边、删除边。

`O(n^2)`

邻接矩阵创建的：BFS、DFS、拓朴排序、AOE网，创建表、插入边、删除边。

普利姆算法，迪杰斯特拉算法。

`O(eloge)`

克鲁斯卡尔算法。

`O(n^3)`

弗洛伊德算法。

查找

给定数值并在查找表中确定关键字等于给定值的记录或数据元素。

基本概念：

- 查找表：相同类型的数据元素的集合，每个元素由若干数据项构成；
- 关键字（Key，码）：数据元素中数个数据项的数值，可以表示一个数据元素。
- 静态查找staticSearch：只对数据元素进行查询火箭所，静态查找表；
- 动态查找dynamicSearch：查找的同时插入表中不存在的记录，或删除表中已存在的记录，动态查找表。
- 平均比较次数ASL：衡量查找算法效率的高低`ASL = n∑i=1(Pi×Ci)`，n为查找表中的记录个数：
    1. ASL成功 = 比较次数/元素个数；
    2. ASL失败 = 比较次数/不成功的位置个数。

四种查找方式：

1. 顺序表查找：给定值与表中记录相比较；
2. 链表查找：给定值与表中记录逐个比较；
3. 散列表查找：给定值直接访问表记录；
4. 索引表查找：根据索引确定带查找记录所在的块，从块中查找。

顺序查找

从表的一端关键词逐个将记录中的关键字与给定值比较。若扫描完整个表均无相应记录，则查找失败。

算法分析：

查找成功比较次数`n`（元素在第几位，n就等于几），查找失败`n+1`次；

ASL：查找成功`(n+1)/2`，查找失败`3(n+1)/2`。

折半查找（二分查找）

查找表需要是有序的（升序、降序均可），先确定带查找记录在表中的范围（与表中的中值相比较，大于或小于中值均表示查找数值的范围在中值的其中一侧），然后逐步缩小（每次缩小一半），直到记录的存在与否被确定。

前提条件：必须是有序的，且用顺序结构存储。

折半算法思想，用low、high、mid分别表示上界、下界、中间位置指针，初值low=1、high=n：

1. 取中间位置mid：`mid = └(low+high)/2┘`；
2. 比较中间位置的值和查找值：
    1. 相等：查找成功；
    2. 大于：查找值在区间的前半段，修改上界指针：`high = mid-1`，回到1.取mid；
    3. 小于：查找值在区间的后半段，修改下届指针：`low = mid+1`，回到1.取mid；
3. 直到`low > high`越界，查找失败。

```
int binSearch(int []st, int n, int key){
    int low = 0, high = n-1, mid;
    while(low < high){
        mid = (low+high)/2;
        if(st[mid] == key) return mid;
        else if(st[mid] < key) low = mid+1;
        else high = mid-1;
    }
    return -1;  //查找失败
}
```

折半查找的查找表可以构建一颗折半树，根为mid、左子树low、右子树high；
折半树只有最下层是不满的，元素个数为n时树高`h = ⌈log2(n+1)⌉`，时间复杂度`O(h) = O(⌈log2(n+1)⌉)`；
ASL成功 = 比较次数/元素个数，
ASL失败 = 比较失败次数/不成功的位置个数；

若无特殊说明，默认向下⌊⌋取整。

动态查找与BST、AVL树

根据查找结果进行 增、删、改 操作。

二叉排序树BST

左子树的结点小于根节点且不为空，右子树的结点大于根节点且不为空，左右子树均为二叉排序树。

查找效率较高，但容易受树的形态所影响。

BST查找

待查找数值K与根节点比较：相等，查找成功；小于，继续沿左子树查找；大于，继续沿右子树查找。

```
/*一种基于递归的BST查找算法*/
BSTNode *BSTSearch(BSTNode *T, keyType){
  if(t == NULL) return NULL;
  else{
    if(T->key == key) return T;
    else if(key < T->key) return BSTSearch(T->LChild, key);
    else return BSTSearch(T->RChild, key);
  }
}
```

BST插入

插入结点s时若BST为空，则s作为根节点。
否则，与根节点比较：相等，不插入；小于，进入左子树，继续比较；大于，进入右子树，继续比较。

```
/*一种基于递归的插入算法*/
void BSTInsert(BSTNode *T, int key){
  BSTNode *x;
  x = (BSTNode *)malloc(sizeof(BSTNode));
  x->key = key;
  x->LChild = x->RChild = NULL;
  if(T == NULL) T = x;
  else{
    if(T->key == x->key) return NULL; //已有结点，无需插入
    else if(x->key < T->key) BSTInsert(T->LChild, key);
    else BSTInsert(T->RChild, key);
  }
}
```

BST删除

删除结点p，其父节点为f。
如果p是叶子节点，直接删除；如果p只有左子树或右子树，则直接用子树取代p，称为f的子树；
如果p同时又左右子树，有两种方式：

1. 直接用p的中序前驱节点取代p，从p左子树选择最大结点s放在p的位置，然后删除s。
2. 用p的直接中序后继结点代替p，从p右子树选择最小节点s放在p的位置，然后删除s。

BST构造

```
BSTNode *createBST(){
  keyType key;
  BSTNode *T = NULL;
  scanf("%d", &key);
  while(key != 65535){
    insertBST(T, key);
    scanf("%d", &key);
  }
  return T;
}
```

平衡二叉树AVL

形态总体均匀，查找效率最好。左右子树的深度差不超过1，且都是平衡二叉树。

平衡因子：节点左子树的深度减去右子树的深度。所以平衡二叉树的平衡因子只会是-1、0、1，否则不是平衡二叉树。

如果一个树同时满足二叉排序树和平衡二叉树的特点，则称为平衡二叉排序树BBST。
BBST的平均查找长度为`O(log2n)`，平均时间复杂度为`O(log2n)`。

一般二叉排序树不是平衡的，可以通过构造平衡二叉树进行平衡化旋转。

对AVL树进行删除或插入操作，通常会影响从根节点到插入/删除结点路径上的某些节点，使得这些结点的子树可能会变化（**LL，LR，RR，RL**）。

平衡化旋转

层次路径是从根到叶子的一条路径。

新插入的结点会影响叶子的层次路径，沿着插入节点上行到根节点——这样就可以找到失衡节点。从失衡结点往下推三个点，这三个点就是要调整的点。

红黑树RedBlackTree【2022新增】

红黑树在AVL的基础上放宽条件：左右子树高度差不超过两倍，一种近似平衡的结构。

查找

索引

一种尽可能减低磁盘I/O次数的索引组织方式。采用B树（B-树）这一多路平衡查找树（B+树为其变体）。

	B树（B-树）	B+树
	子树个数_；_关键字个数	子树个数_；_关键字个数
根节点	2 ~ m； 1 ~ (m-1)	2 ~ m； 2 ~ m
中间节点	⌈3/2 ⌉~ m；⌈m/2⌉-1 ~ m-1	⌈m/2⌉ ~ m；⌈m/2⌉ ~ m
节点重复	否	是
查找	只有随机查找	随机查找和顺序查找
关键字	各节点包含的含剪子不重复	叶子结点包含全部关键字，非叶子节点中出现的关键字也会出现于叶子节点
存储信息	节点中都包含了关键字对应记录的存储地址	叶子节点包含信息；所有非叶子节点仅作为索引，且每个索引项只含有对应子树最大关键字和指向该子树的指针，不含有该关键字对应记录的存储地址
n个关键字	对应n+1个子树	对应n个子树

B树

树中每个节点的大小为一个磁盘页，结点中所含关键字及其孩子数目取决于页的大小。度为m的B树称为m阶B树，是满足以下性质的m叉树（或空树）：

1. 根节点至少有两棵子树、至多有m棵子树（或者根节点为叶子节点）；
2. 除根结点外，所有非终端节点至少有`⌈m/2⌉`棵子树，至多有`m`棵子树;
3. 所有叶子节点都在同一层。

4. 每个结点包含`(n, A0, K1, A1, k2, ... , Kn, An)`，其中：
    1. `n`是节点中关键字个数，`⌈m/2⌉-1 ≤ n ≤ m-1`，`n+1`为子树棵树——用关键字分割子树；
    2. `Ki`为关键字，`Ki < K(i+1)`；
    3. `Ai`是指向孩子节点的指针，`A(i-1)`所指向的子树中所有节点的关键字都小于`Ki`，`Ai`则均大于Ki。

B树查找

从根节点T开始，在T所指结点的关键字向量`key[1...keynum]`中查找给定值K（折半查找）：

1. 若`key[i] == K (1 ≤ i ≤ keynum)`，查找成功返回结点和关键字位置；
2. 否则，将K与key中各个值比较，以选定查找子树：
    - 若`K < key[1]`：`T = T->prt[0]`；
    - 若`key[i] < K < key[i+1] (i = 1,2,..keynum)`：`T = T->ptr[i]`；
    - 若`K > key[keynum]`：`T = T->ptr[keynum]`；
3. 若均不满足，跳转1，知道T是叶子节点且未找到相等关键字，查找失败。

查找分析：

对于第h层：

- 最多节点数为`m^(h-1)`，最多关键字数为`(m-1)m^(h-1)`；
- 最少节点数为`2*(⌈m/2⌉)^(h-2)`，最少关键字数为`2(⌈m/2⌉-1)(⌈m/2⌉)^(h-2)`。

B树插入

插入时首先在最低层的叶子节点添加一个关键字，然后有可能“分裂”，插入过程如下（**插入看上界，超过要分裂，根分高一层**）：

1. 在B树种查找关键字K，若找到则表明已存在，否则K的查找操作失败与某个叶子节点；
2. 随后将K插入该叶子节点，插入时：
    - 若叶子节点关键字数 < m-1：直接插入；
    - 若叶子节点关键字数 == m-1：结点“分裂”。
3. 根节点分列式，由于没有父节点，则建立一个新根，B树增高一层。

B树删除

对于删除一个关键字K：

1. 找到其所在结点N并删除关键字K。
2. 如果N不是叶子节点，设K是N的第i个关键字，将指针A(i-1)所指子树中的最大关键字K'（或最小关键字）放在(K)的位置；
3. 然后删除K'，而K'在叶子节点上。

删除看下界，若自己够就从自己删除，树不调整；
自己不够，找左兄弟的最大值，或找右兄弟的最小值——兄弟上，父亲下；
若都不够，让自己、左兄弟（或右兄弟）、父亲三方合并，此时父亲也进行上述操作，逐级向上递归。

B+树

只有叶子节点存储信息，非叶子部分均为索引，同时支持顺序查找和随机查找。

散列表Hash

在记录存储地址和它的关键字之间简历一个确定的对应关系，不经比较，一次存取获得查找元素的查找方式。

基本概念：

- **哈希函数**：在关键字与存储地址之间建立关系，从关键字空间到存储地址空间的一种映像，从而得出哈希地址（哈希值）。
- **哈希表**：用哈希函数的映像记录在表中的地址，并将记录置入此地址，构成哈希表。
- **冲突**：不同关键字但生成了相同哈希值的情况。
- **同义词**：相同哈希值的两个不同关键字称为同义词。
- **哈希查找**（散列查找）：利用哈希函数进行查找。
- **散列表设计**：
    1. 空间范围，确定散列函数的值域；
    2. 构造合适的散列函数，使对于所有可能元素的哈希值均在散列表的地址空间范围，且冲突尽可能小；
    3. 设计合适的冲突处理方式。
- **哈希表评估因素**：
    1. 散列函数构造是否简单；
    2. 能否均匀将关键字映射到地址空间（冲突尽可能少）。

哈希函数构造方式

直接定址法：使用一元线性方程生成哈希值，关键字个数和地址个数一样，不会发生冲突，但由于占用空间过高，实际很少使用。

除留余数法：对关键字取余得到哈希值。取余数的大小不大于哈希表长度。是一种简单常用的构造方式。

冲突的处理方式

开放定址法：

冲突发生时，可以（由某种给定的方式，但得确保能够被找到）放置于值域的任何位置。
其公式为`Hi(key) = (H(key)+di) % m, i = 1,2,..,k (k≤m-1)`，
H(key)为哈希函数，m为散列表长度，di为第i此探测时的增量，Hi(key)是经过第i此探测后得到的散列表地址。

对于di的算法：

1. 线性探测法：发生冲突时，从发生冲突的位置一次向后探查。只要表中未满，总会找到位置。但每个冲突记录被散列到冲突最近的空地址，增加了更多冲突，容易“聚集”，ASL增大。查找失败的比较次数将从哈希值出发，一直到空位置为止。

2. 二次探查法：di的增量为`1², -1², 2², -2²,..., ±k² (k≤⌊m/2⌋)`，相较于当前所在位置做平方勘察。采用较大的跨距跳跃到散列表，不容易“聚集”，但不能保证准确使用到所有空间。

再哈希法：

备置多个哈希函数，冲突时使用另一个哈希函数，直到没有冲突发生。不容易“聚集”，但会增加计算时间。

链地址法：

哈希值相同的关键字存储在一个单链表，并用一维数组存放头指针。

哈希查找分析

查找效率基于ASL，关键字和给定值比较次数基于：哈希函数，处理冲突的方式，哈希表的装填因子α = (表中填入记录数)/(哈希表长度)。

ASL成功 = 比较次数 / 元素个数；ASL失败 = 比较次数 / 失败的位置量个数

排序

排序是将文件中的记录进行按关键字有序排序的处理过程，是数据处理种常用的操作。

排序的基本操作：比较关键字大小，存储位置移动。

稳定的排序：记录中两个或以上关键字相等的记录，在排序前和排序后的顺序一致，则该排序方法稳定。

排序类型：内部排序（记录不多，都可以在内存储排序），外部排序（记录过多，需要内、外存进行数据交换）。

排序算法的优劣取决于：执行时间（时间复杂度），所需辅助空间（空间复杂度），算法稳定性。

各个排序算法的性能：

方法	平均时间	最坏所需时间	辅助空间	稳定性
直接插入	O(n^2)	O(n^2)	O(1)	稳定
Shell排序	O(n^1.3)		O(1)	不稳定
直接选择	O(n^2)	O(n^2)	O(1)	不稳定
堆排序	O(nlog2n)	O(nlog2n)	O(1)	不稳定
冒泡排序	O(n^2)	O(n^2)	O(1)	稳定
快速排序	O(nlog2n)	O(n^2)	O(log2n)	不稳定
归并排序	O(nlog2n)	O(nlog2n)	O(n)	稳定
基数排序	O(d(n+r))	O(d(n+r))	O(n+r)	稳定

1. 快、希、选、堆 不稳定；
2. 快归堆 nlogn；
3. 快排 log2n，归并 o(n)；
4. 借助于顺序结构：折半插入、希尔、归并、堆、快排；
5. 每趟排序均有元素排好：交换类、选择类；
6. 排序趟数与数据初始状态有关：冒泡、快排；
7. 比较次数与数据初态无关：简单选择、折半插入、基数排序；
8. 时间复杂度与初态无关：简单选择、基数排序、堆排序、归并排序；

插入类排序

将待插入元素Ri插入已排好的记录中的适当位置。

直接插入排序

将未排好序的无序部分逐个插入进已排好序的部分。

最好情况：待排序记录已经按指定次序排序。元素月有序，直接插入排序越快。

平均时间复杂度是`O(n^2)`。

折半插入排序

在将未排好序的无序部分逐个插入进已排好序的部分时，可以转变为折半插入排序，减少比较次数，使得性能效率更高。
并且折半插入排序只能用顺序存储。

由于仅仅减少关键字比较次数，没有减少移动次数，所以时间复杂度O`(n^2)`。

希尔排序

按照一定步长跨越地选择对应元素，对这些对应元素并在步长内采用直接插入排序，随后减少步长再排序，从而逐渐让整体逐渐趋向于有序。

交换类排序

不断交换反序的偶对，直到不再有反序偶对。

冒泡排序

依次比较相邻元素，通过两两交换，（例如按从大到小排序）将大的元素前置、小的元素后置——通过多趟这样的处理后，得到有序序列。

时间复杂度：`O(n^2)`；
空间复杂度：`O(1)`。

快速排序【☆☆☆☆☆算法，最经常考】

以一个记录为参照R[s]，一趟排序后，R[s]的左半边均会小于（大于）右半边。
下述步骤以递增排序为例来描述：

1. 在记录中任取一个记录作为参照R[s]（一般选择第一个记录），从末尾开始往前比较；
2. 找到第一个小于R[s]的元素R[1]插入到R[s]的位置；
3. 随后从插入位置出发继续比较：
    1. 若是小于R[s]，则继续往后比较；
    2. 若是大于R[s]的元素R[2]，则插入到R[1]的位置，再继续往前比较；
    3. 直到找到小于R[s]的元素，插入到R[2]的位置；
4. 重复2、3的操作，直到让序列分为两个部分——这样一来，左半边的元素会小于右半边；
5. 随后对这两个部分内部再依此方式排序、分割——直到整个序列有序。

```
/*一趟快速排序*/
int quickOnePass(sqList *L, int low, int high){
    int i = low, j = high;
    L->R[0] = L->R[i];  //R[0]作为临时单元和哨兵
    do{
        while(L->R[0].key < L->R[j].key && (j > i))
            j--;
        if(j > i){
            L->R[i] = L->R[j];
```

```
            i++;
        }
        while(L->R[i] = L->R[0].key && (j > 1))
            i++;
        if(j > i){
            L->R[j] = L->R[i];
            j--;
        }
    }while(i != j);     //i==j时退出扫描
    L->R[i] = L->R[0];
    return i;
}

/*递归*/
void quickSort(sqList *L, int low , int high){
    int k;
    if(low < high){
        /*序列分为两个部分后，分别对每个子序排序*/
        k = quickOnePass(L, low, high);
        quickSort(L, low, k-1);
        quickSort(L, k+1, high);
    }
}
```

排序后，每一趟所选的R[s]可以构成一颗二叉树：树高==排序趟数，每一层的元素即为每一趟的基准元素。
若序列在排序前有序，那么会构成一棵单支树。

所以对于快速排序，元素越有序，速度越慢，反之速度越快。
元素**乱序**：时间复杂度为`O(nlogn)`，空间复杂度`O(logn)`；
元素**有序**：时间复杂度为`O(n^2)`，空间复杂度`O(n)`。

选择类排序

选择当前序列中最小值，放置到最后（或第一位），依此类推分成有序和无序两个部分，直到整个记录有序。

简单选择排序

若序列中有n个元素，通过`n-1`次关键字比较，从`n-i+1`个记录中选取关键字最小的记录，然后和第`i`个记录进行交换。

时间复杂度：`O(n^2)`；
空间复杂度：`O(1)`。

```
void simpleSelectionSort(sqList *L){
    int m, n, k;
    for(m=1; m<L->length; m++){
        k = m;
        for(n=m+1; n<=L->length; n++)
            if(LT(L->R[n].key, L->R[k].key)) k = n;
        if(k != m){
            /*交换记录*/
```

```
            L->R[0] = L->R[m];
            L->R[m] = L->R[k];
            L->R[k] = L->R[0];
        }

    }
}
```

堆排序

n个元素的序列，满足
`ki ≤ k2i, ki ≤ k(2i+1)` （小根堆）
或
`ki ≥ k2i, ki ≥ k(2i+1)` （大根堆）
所得到一个以k1为根且从上到下、从左到右编号的完全二叉树，得到的序列则是将二叉树以顺序结构存储，堆的结构和该序列结构一致。

每次输出一个堆顶元素，将从最底层的叶子节点补充，再重新调整。

时间复杂度`O(nlogn)`； 空间复杂度`O(1)`。

堆排序的构思：

1. 堆一组待排序的记录，按堆的定义建立堆；
2. 将堆顶记录和最后一个记录交换位置，则得到前n-1个记录是无序的，最后一个记录是有序的；
3. 堆顶记录交换后，前n-1个记录不再是堆，需要重新组织成一个堆，然后堆顶记录和倒数第二个记录交换位置，将整个序列中次小关键字的记录调整出无序区；
4. 重复上述步骤，直到记录有序。

排序过程：

1. 建立初始堆：
    1. 按照完全二叉树的层次逐个插入；
    2. 按照从下往上、从右往左，先兄弟、后双亲的顺序来比较；
    3. （此处以小根堆为例子）按照如上顺序，将较小值置上；
2. 让堆顶于最后一个元素交换；
3. 重新调整，从上往下调。

归并类排序

将序列中的每个元素都视为单个序列，相邻的序列合成一个序列并比较排序，依此类推，直到最终合成一个有序序列。
遵循——双指针、不回溯、尾插法，谁小谁尾插、谁小谁后移、相等即删除。

时间复杂度：`O(nlog2n)`；
空间复杂度：`O(n)`。

基数类排序

将序列中的单个元素拆成多个关键字（每个元素按位拆成子序列），再将子序列中对应位进行比较，按个位、十位、百位依次在将各个子序排序（低位优先），使得整个序列逐渐有序。

在实际操作中，将会构建`0~9`的桶，先按元素的个位数字分配进对应的桶——桶内按队列（FIFO），桶间从小到大——从而得到一个排序好一趟的序列。随后再将元素的十位、百位等依此类推进行排序，直到整个序列有序。

数组

数组是一个包含{下标值、数据元素}（偶数对）的集合，数据元素类型是一样的。

数组一旦建议，结构中的元素个数和元素间关系就不再变化，也因此采用顺序存储来表示数组。

408一般是按行存储。

数组地址的计算

设二维数组`A = (a(ij))(m*n)`，每个元素占用存储单元`l`个，`LOC[a11]`表示元素`a11`的首地址（即数组首地址）那么有：

第m行中每个元素对应（首）地址：`LOC[a(mj)] = LOC[a11] + (m-1)×n×l + (j-1)×l, j=1,2,...,n`；
同理可得按列优先存储。

特殊矩阵

三角矩阵，对称矩阵，带状矩阵，稀疏矩阵等。

三角矩阵

包含上三角和下三角两种，它们以对角线分开的另外半边元素均为常数c（一般是0）。
因此三角矩阵的重复元素可以都存储在向量`sa[0...n(n+1)/2]`中，sa中的下标值与`(i,j)`之间的关系为：

- if(i≥j)：`i(i-1)/2 + j - 1`；
- if(i<j)：`j(j-1)/2 + i - 1`。

上三角矩阵的元素a(ij)存在一维数组中时，前`i-1`行：`[n+(n-i+2)×(i-1)/2]`
第i行：`j-i`

对称矩阵

对称矩阵满足 `a(ij) = a(ji)`，则其中`n²`个元素可压缩为`n(n+1)/2`个存储空间。
`a(ij)`之前的`i-1`行共有`i(i-1)/2`个元素。

对角矩阵

呈现中心对称的矩阵。

稀疏矩阵

有多个非零元素的矩阵。对其存储形式可以采用
三元组：`(i,j,a(ij))`，前二个存储行列，最后一个存储数值。

但丧失了随机存储的能力。