数据结构算法题

- 几轮复习
  - 基础
    - 全覆盖
    - 无遗漏
  - 强化
    - 抓大放小
    - 花更少的精力，建立起更强的底气
  - 冲刺
    - 抓总体，查缺补漏
  - 忍法核心要义
    抓总体、抓重点、抓总体、抓重点、抓总体
- 给分规则
  - 第一小问，语言障碍，表达不清，怎么办？ —— 只要第二小问代码逻辑清晰，第一小问也给满分
  - 第三小问，分析时间、空间复杂度
    - 要和你的算法相匹配，否则不得分
    - 根据你算法的整体复杂度来决定给你多少分——看整体，不看局部。 —— 如：采用快排预处理，导致时间复杂度为O(nlogn)，则接下来的查找过程，可以用顺序查找O(n)，没必要挣扎着写折半查找O(logn)
  - 在408中，代码健壮性重要吗？是否需要考虑边界情况？
    - 不太重要
    - 工作中很重要!
- 树相关算法
  - 基于顺序存储的二叉树树
    - 遍历
      - 层序遍历 —— 顺序遍历数组即可
      - 先、中、后序遍历 —— Key：改造链式存储的思路；封装
    - 建立 —— 给你一颗链式存储的二叉树，建立与之对应的顺序存储二叉树
    - 堆排序（排序+树的结合考点）
      - 不负责任的咸鱼语言
        - 考应用题的概率较大（画图）
        - 代码，考的概率不大，但依然建议整一下
      - 练习
        - 给你一个乱序数组[1....n]，让你建堆?
        - 建好堆之后，堆排序代码?
    - 封装
      - 找左孩子
      - 找右孩子
      - 找父节点
  - 基于链式存储的二叉树
    - 遍历
      - 先序
      - 中序 —— 练习：给你一个表达式树，计算该表达式的值。只有+、-、*、/，假设给你的表达式树是没有错误的。
      - 后序
      - 层次遍历
        - 练习：给你一棵链式存储的二叉树，让你建立与之对应的顺序存储的二叉树?
        - 如果判断链式存储的二叉树是否是完全二叉树?
      - 常用的处理 —— 递归遍历的时候，如何判断当前遍历的结点，处于第几层?
    - 建立 —— 给你一个顺序存储的二叉树，建立与之对应的链式存储的二叉树?
    - BST（查找+树）
      - 建立 —— 给你一个关键字插入序列（用长度为n的数组给出），让你建立一棵BST
      - 查找
        - 递归实现
        - 非递归实现
      - 插入
        - 递归实现
        - 非递归实现
      - 删除
        - ？？？？？有点难？？？，给你某结点的指针，让你删除这个结点？？
        - 考察的可能性存在，但不大。有能力的同学练练，吃力的同学？可躺平
  - 同
    封装了之后，先、中、后序遍历
    findLeft、findRight 两个操作，是粘合剂
  - 逆向操作
  - 核心
- 图的算法
  - 存储结构
    - 邻接表 —— 数据结构定义，类似与?
      - 基数排序
      - 树的存储，孩子表示法 —— 思考：BFS、DFS之于孩子表示法存储的树，相当于？？？？
      - 散列查找表，拉链法实现
    - 邻接矩阵
  - 算法
    - BFS
    - DFS
    - 拓扑排序的实现? —— 是否类似? —— 判断有没有环?
      - 上课时候给过基于DFS的逆拓扑排序实现
      - 拓扑排序的实现呢？怎么搞? —— stack或原地逆置
      - 两种方法：数组标记法？DFS?
    - 有内在联系，一起搞
- 线性表
  - 常用套路
    - 设计一个时间上尽可能高效的算法
      - ？？？？ 空间复杂度，随意，高高高。辅助数组
      - 空间换时间的思想
    - 给你关键字的取值范围
      - ？？？？，辅助数组
      - 无序 变 有序，是否更好处理?
    - 本身有序，常查找思想
      - 折半查找的代码，给老子熟悉
  - 练习
    - 单链表
      - 原地逆置算法
        - int func(？, ？) —— 定义一个单链表，并完成原地逆置
      - 单链表双指针处理
        - int func(???) —— 用一次遍历，找到链表中间结点 (n/2 那个点)
        - int func(???) —— 用一次遍历，找到链表的倒数第k个元素
    - 顺序表
      - 原地逆置算法
        - int func(a[] , n) —— a[n]，完成原地逆置
      - 查找
        - (二)顺序查找法
        - 算法考的较少 —— (三)分块查找法
        - int zheBanSearch (int a[], n, int key) —— (四)折半查找法
        - (六)散列(Hash)表
          - 用拉链法解决冲突的散列表可以关注一下
          - 练习
            - 散列表长度为n，用拉链法解决冲突
            - 定义这个散列表
            - 写代码实现关键字 插入、删除、查找 的基本操作
          - 树的存储之孩子表示法、图的存储之邻接表、排序之鸡排
        - 主串中找子串
        - (七)字符串模式匹配
        - 朴素模式次优解解决
      - 排序
        - 可用于顺序+单链表。需要练习熟悉代码
        - 只能用于顺序表，不能用于单链表
          - 1.直接插入排序
          - 2.折半插入排序
          - (二)插入排序
        - (三)起泡排序(bubble sort)
        - (四)简单选择排序
        - 不太可能考算法 —— (五)希尔排序(shell sort)
        - 重要啊重要啊重要，救命排序 —— (六)快速排序
        - 周三讲树的时候再说 —— ？？？ —— (七)堆排序
        - 重要啊重要啊重要，多个有序序列的合并 —— (八)二路归并排序(mergesort)
        - (九)基数排序
          - 关键字的取值范围是0~999，乱序存在单链表中，元素有好几十亿个，用鸡排实现排序
          - 练习
            - 如何去除个十百 三个位?
            - 如何分配? 如何回收?
            - 参考散列表 —— 如何设计数据结构?
        - 如果考，只可能是归并 —— (十)外部排序

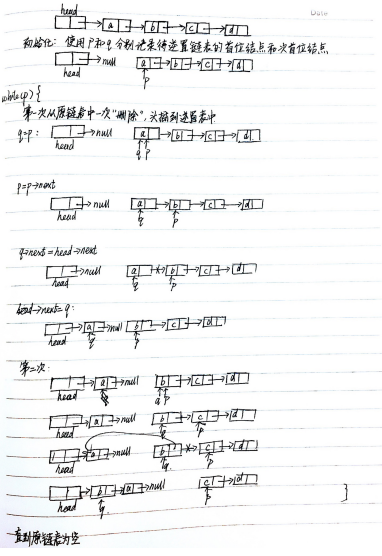
单链表定义：

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

线性表

1.单链表

（1）定义一个单链表，原地逆置



```
LNode* converse(LinkList *head)
{
    LinkList *p,*q;
    p=head->next;
    head->next=NULL;
    while(p)
    {
        q=p;        //向后挪动一个位置
        p=p->next;
        q->next=head->next;
        head->next=q;
    }
}
```

头插操作

//头插

（2）用一次遍历，用双指针找到链表的中间节点（n/2的那个点）

双指针 两步走

```
LNode * findMidNode(LinkList head){
    LNode * p=head; //p指针，每次往后走一步
    LNode * q=head; //q指针，每次往后走两步，当q指向最后一个结点时，p刚好指向中间那个结点
    while(q->next!=NULL){
        q = q->next; //q往后走一步
        if (q->next!=NULL)
        q = q->next; //q再往后走一步
        p=p->next; //p往后走一步
    }
        if(p==head)
        printf("一个空链表，让我找中间结点？你一定是在逗我~");
        else
        printf("中间结点的值为：%d\n", p->data);
        return p; //返回p所指向的结点
    }
```

（3）用一次遍历，用双指针找到链表的倒数第k个元素

```
LNode * findDaoShuKNode(LinkList head, int k){
    LNode * p=head; //p指针，等q指针走到第k个结点时，p指针再出发
    LNode * q=head; //q指针，每次往前走一步
    int count=0;
    bool chong = false; //p指针要不要冲?
    while(q->next!=NULL){
        q = q->next; //q往后走一步
        count++; //计数+1
        if (count==k)
            chong = true; //q走到第k个结点时，p指针开冲
        if(chong)
            p = p->next;
    }
    if(p==head)
        printf("这个链表长度小于%d，并不存在倒数第k个结点\n", k);
    else
        printf("倒数第%d个结点的值为：%d\n", k, p->data);
    return p; //返回p所指向的结点
}
```

p指针开始走的条件

2.顺序表

（1）给一个数组A[n]，完成原地逆置

```
void Reverse(int a[],int from,int to){
    int i,temp;
    for(i=0;i<(to-from)/2;i++)
    {
        temp=a[from+i];
        a[from+i]=a[to-i];
        a[to-i]=temp;
    }
}
```

Sweep(a[from+i], a[to-i]); 封装

（2）折半查找

```
int Binary_Search(SeqList L,ElemType key){
    int low=0,high=L.TableLen-1,mid;
    while(low<=high){
        mid=(low+high)/2;           //取中间位置
        if(L.elem[mid]==key)
            return  mid;            //查找成功则返回所在位置
        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
}
```

（3）散列表长度为n，用拉链法解决冲突（定义散列表并实现插入、删除、查找操作）

1）定义散列表

```
//用于存储关键字的结点
```

```
typedef struct HaNode {
    int key;
    struct HaNode * next;
} HaNode;
#define N 7 //散列表的长度
typedef struct {
    HaNode * h; //链头指针
} HashTable[N];
/**
* 初始化散列表
*/
void InitHashTable (HashTable t){
    for (int i=0; i<N; i++)
        t[i].h = NULL;
}
```

2）插入元素key

```
int InsertElem (HashTable t, int key){
//申请新结点
    HaNode * p = (HaNode *)malloc(sizeof(HaNode));
    p->key = key;
    //头插法插入关键字key对应的拉链
    int index = key%N;
    p->next = t[index].h;
    t[index].h = p;
    return 1; //插入成功
}
```

3）删除值为key的元素

```
int DeleteElem (HashTable t, int key){
    int index = key%N;
    int flag = 0; //flag=0表示删除失败，flag=1表示删除成功
    HaNode * pPre = NULL; //pPre在遍历过程中指向p的前驱结点
    HaNode * p = t[index].h; //p指针从第一个结点开始遍历
    while (p!=NULL) {
        //找到链表中值为key的结点并删除
        if (p->key==key){
            if (pPre==NULL){ //pPre==NULL说明p结点是当前这个链表的第一个元素
                t[index].h = p->next; //头指针指向p的下一个结点
                HaNode * s = p;
                p = p->next;
                free(s);
            } else {
```

```
                pPre->next = p->next;
                HaNode * s = p;
                p = p->next;
                free(s);
            }
            flag = 1;
        } else {
            //当前结点 p 的值不等于 key，继续检查下一个结点
            pPre = p;
            1p = p->next;
        }
    }
    return flag; //遍历整个链表，没找到关键字为key的结点，返回0表示删除失败
}
```

4）查找值为key的元素

```
HaNode * GetElem (HashTable t, int key){
    int index = key%N;
    HaNode * p = t[index].h; //p指针从第一个结点开始遍历
    while (p!=NULL) {
        //找到链表中值为key的结点并删除
        if (p->key==key){
            return p; //返回第一个关键字为key的结点
        }
        p = p->next;
    }
    return NULL; //遍历整个链表，没找到关键字为key的结点，返回NULL表示查找失败
}
```

（4）简单插入排序

```
void InsertSort(ElemType A[],int n){
    int i,j;
    for(i=2;i<=n;i++)                    //依次将 A[2]～A[n]插入前面已排序序列
        if(A[i]<A[i-1]){                 //若 A[i]关键码小于其前驱，将 A[i]插入有序表
            A[0]=A[i];                   //复制为哨兵，A[0]不存放元素
            for(j=i-1;A[0]<A[j];--j)//从后往前查找待插入位置
                A[j+1]=A[j];             //向后挪位
            A[j+1]=A[0];                 //复制到插入位置
        }
}
```

（5）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
        while(low<=high){           //折半查找(默认递增有序)
            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];            //统一后移元素，空出插入位置
        A[high+1]=A[0];             //插入操作
    }
}
```

(6) 冒泡排序

```
void BubbleSort(ElemType A[],int n){
    for(i=0;i<n-1;i++){
        flag=false;                 //表示本趟冒泡是否发生交换的标志
        for(j=n-1;j>i;j--)          //一趟冒泡过程
            if(A[j-1]>A[j]){        //若为逆序
                swap(A[j-1],A[j]);  //交换
                flag=true;
            }
        if(flag==false)
            return;                 //本趟遍历后没有发生交换，说明表已经有序
    }
}
```

(7) 快速排序

```
void QuickSort(ElemType A[],int low,int high){
    if(low<high){                   //递归跳出的条件
    //Partition()就是划分操作，将表A[low…high]划分为满足上述条件的两个子表
        int pivotpos=Partition(A,low,high);  //划分
        QuickSort(A,low,pivotpos-1);  //依次对两个子表进行递归排序
        QuickSort(A,pivotpos+1,high);
    }
}
int Partition(ElemType A[],int low,int high){ //一趟划分
    ElemType pivot=A[low];          //将当前表中第一个元素设为枢轴，对表进行划分
    while(low<high){                //循环跳出条件
        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;                     //返回存放枢轴的最终位置
}
```

（8）二路归并排序

```
    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
        }//if
    }
ElemType *B=(ElemType *)malloc((n+1)*sizeof(ElemType)); //辅助数组B
void Merge(ElemType A[],int low,int mid,int high){
//表A的两段A[low…mid]和A[mid+1…high]各自有序，将它们合并成一个有序表
    for(int 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++];                 //将较小值复制到A中
        else
            A[k]=B[j++];
    }//for
    while(i<=mid)  A[k++]=B[i++];  //若第一个表未检测完，复制
    while(j<=high)  A[k++]=B[j++];  //若第二个表未检测完，复制
}
```

*（9）基数排序（关键字取值范围为0~999）

```
//用于存储关键字的结点
typedef struct RaNode {
    int key;
    struct RaNode * next;
} Node;
//队列结点定义
typedef struct {
    RaNode * front; //队头指针
    RaNode * rear; //队尾指针
} Queue;


//取出关键字 key 的第 d 位（第1位是个位，第2位是十位，第3位是百位
int getRadix(int key, int d){
    if(d==1) return key % 10;
    if(d==2) return (key/10) % 10;
    if(d==3) return (key/100) % 10;
    return -1; //参数d有误
}


//第 k 趟分配
void distribute (RaNode * head, int k, Queue list[]) {
//每次摘下链头元素进行分配
    while (head->next!=NULL){
        RaNode * p = head->next; //p指向链头元素
        head->next = p->next; //摘下链头元素
```

```
            p->next=NULL;
            int r = getRadix(p->key, k); //取出当前结点第k位
            //目前第r个分配队列为空
            if (list[r].front == NULL){
                list[r].front = p; //将结点p插入第r个队列
                list[r].rear = p;
            } else {
            list[r].rear->next = p; //将结点p插入第r个队列的队尾
            list[r].rear = p; //修改队尾指针
            }
        }
}

//第 k 趟回收
void collect(RaNode * head, int k, Queue list[]) {
//依次将 9~0 队列中的元素整体摘下，用头插法插入单链表 head
        for (int i=9; i>=0; i--) {
            if (list[i].front == NULL) continue; //空队列直接跳过
            list[i].rear->next = head->next; //将第i个分配队列整体头插到链表中
            head->next = list[i].front;
            list[i].front = NULL;
            list[i].rear = NULL;
        }
}

//对单链表 head 进行基数排序，其中指针 head 指向头结点
int RadixSort(RaNode * head) {
        Queue list[10]; //用于实现基数排序的10个队列
        for (int i=0; i<10; i++){
            list[i].front = NULL;
            list[i].rear = NULL;
        }
//由于关键字取值范围为 0~999，因此仅需 3 趟分配/回收
for (int r=1; r<=3; r++){
        distribute(head, r, list);
        collect(head, r, list);
        }
}
```

***（10）堆排序**

3.树相关算法

（1）二叉树链式存储结构

```
//二叉排序树结点
typedef struct BSTNode{
        int key; //数据域
        struct BSTNode *lchild,*rchild; //左、右孩子指针
```

```
}BSTNode,*BSTree;


//平衡二叉树结点
typedef struct AVLNode{
    int key; //数据域
    int balance; //平衡因子
    struct AVLNode *lchild,*rchild;
}AVLNode,*AVLTree;
```

（2）二叉树先中后序遍历

```
//访问结点p
void visit(BSTNode * p){
    printf("%d,", p->key);
}

//先序遍历
void PreOrder(BSTree T){
    if(T!=NULL){
        visit(T); //访问根结点
        PreOrder(T->lchild); //递归遍历左子树
        PreOrder(T->rchild); //递归遍历右子树
    }
}
//中序遍历
void InOrder(BSTree T){
    if(T!=NULL){
        InOrder(T->lchild); //递归遍历左子树
        visit(T); //访问根结点
        InOrder(T->rchild); //递归遍历右子树
    }
}
//后序遍历
void PostOrder(BSTree T){
    if(T!=NULL){
        PostOrder(T->lchild); //递归遍历左子树
        PostOrder(T->rchild); //递归遍历右子树
        visit(T); //访问根结点
    }
}
```

（3）求树的深度

```
    int treeDepth(BSTree T){
        if (T == NULL) {
            return 0;
        }
        else {
            int l = treeDepth(T->lchild);
            int r = treeDepth(T->rchild);
            //树的深度=Max(左子树深度，右子树深度)+1
            return l>r ? l+1 : r+1;
```

```
        }
    }
```

（4）在树T中寻找结点P的父节点

```
BSTNode * findFather(BSTree T, BSTNode * p) {
//检查T是否是p的父节点
    if (T==NULL)
        return NULL;
    if (T->lchild==p || T->rchild==p)
        return T;
    //在左子树找p的父节点
    BSTNode * l = findFather(T->lchild, p);
    if (l != NULL)
        return l;
    //在右子树找p的父节点
    BSTNode * r = findFather(T->rchild, p);
    if (r != NULL)
        return r;
    //左右子树中都没找到父节点。或者，根节点也没有父节点
    return NULL;
}
```

（5）在二叉排序树中寻找值为key的节点

非递归算法：

```
BSTNode *BST_Search(BSTree T,int key){
    while(T!=NULL&&key!=T->key){ //若树空或等于根结点值，则结束循环
        if(key<T->key) T=T->lchild; //小于，则在左子树上查找
        else T=T->rchild; //大于，则在右子树上查找
    }
    return T;
}
```

递归算法：

```
BSTNode *BSTSearch(BSTree T,int key){
    if (T==NULL)
        return NULL; //查找失败
    if (key==T->key)
        return T; //查找成功
    else if (key < T->key)
        return BSTSearch(T->lchild, key); //在左子树中找
    else
        return BSTSearch(T->rchild, key); //在右子树中找
}
```

（6）在二叉排序树中插入关键字为k的新结点

```
int BST_Insert(BSTree &T, int k){
    if(T==NULL){ //原树为空，新插入的结点为根结点
        T=(BSTree)malloc(sizeof(BSTNode));
        T->key=k;
        T->lchild=T->rchild=NULL;
```

```
        return 1; //返回1，插入成功
    }
    else if(k==T->key) //树中存在相同关键字的结点，插入失败
        return 0;
    else if(k<T->key) //插入到T的左子树
        return BST_Insert(T->lchild,k);
    else //插入到T的右子树
        return BST_Insert(T->rchild,k);
}
```

（7）层序遍历二叉树

```
void LevelOrder(BSTree T){
    LinkQueue Q;
    InitQueue(Q); //初始化辅助队列
    BSTree p;
    EnQueue(Q,T); //将根结点入队
    while(!IsEmpty(Q)){ //队列不空则循环
        DeQueue(Q, p); //队头结点出队
        visit(p); //访问队头元素
        if(p->lchild!=NULL)
            EnQueue(Q,p->lchild); //左孩子入队
        if(p->rchild!=NULL)
            EnQueue(Q,p->rchild); //右孩子入队
    }
}
```

4、图相关算法

（1）图邻接表、邻接矩阵存储结构

邻接表：

```
#define MaxVertexNum 100          //图中顶点数目的最大值
typedef struct ArcNode{           //边表结点
    int adjvex;                   //该弧所指向的顶点的位置
    struct ArcNode *next;         //指向下一条弧的指针
    //InfoType info;              //网的边权值
}ArcNode;
typedef struct VNode{             //顶点表结点
    VertexType data;              //顶点信息
    ArcNode *first;               //指向第一条依附该顶点的弧的指针
}VNode,AdjList[MaxVertexNum];
typedef struct{
    AdjList vertices;             //邻接表
    int vexnum,arcnum;            //图的顶点数和弧数
} ALGraph;                        //ALGraph 是以邻接表存储的图类型
```

邻接矩阵：

```
#define MaxVertexNum 100                    //顶点数目的最大值
typedef char VertexType;                    //顶点的数据类型
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;
```

（2）DFS,BFS算法

```
//广度优先遍历
void BFS(Graph G,int v){    //从顶点v出发，广度优先遍历图G
    visit(v);                   //访问初始顶点v
    visited[v]=TRUE;            //对v做已访问标记
    Enqueue(Q,v);               //顶点v入队列Q
    while(!isEmpty(Q)){
        DeQueue(Q,v);           //顶点v出队列
        for(w=FirstNeighbor(G,v);w>=0;w=NextNeighbor(G,v,w))
            //检测v所有邻接点
            if(!visited[w]){    //w为v的尚未访问的邻接顶点
                visit(w);       //访问顶点w
                visited[w]=TRUE;//对w做已访问标记
                EnQueue(Q,w);   //顶点w入队列
            }//if
    }//while
```

```
void DFS(Graph G,int v){          //从顶点v出发，深度优先遍历图G
    visit(v);                     //访问顶点v
    visited[v]=TRUE;              //设已访问标记
    for(w=FirstNeighbor(G,v);w>=0;w=NextNeighor(G,v,w))
        if(!visited[w]){          //w为u的尚未访问的邻接顶点
            DFS(G,w);
        }   //if
}
```