2026 考研数据结构代码题

基础 63 必背版

来源：b 站：我头发还多还能学

微信：Rain-Splash 或 petrichoryin

强化 75 题、408 最终预测 15 题、上机运行脚本以及所有题目对应的视频讲解需要付费，有需要的同学添加微信即可。

本 pdf 加阴影的题目是重点题，务必可以理解并默写出来！

基础篇目录

顺序表

1、在顺序表 L 的第 k 个位置插入元素 x【增】
2、删除顺序表 L 的第 k 个元素并返回其值【删一】
3、将顺序表中的元素逆置【改：经典逆置】
4、查找顺序表中第一个值为 x 的元素的位置【查】
5、顺序表递增有序，插入元素 x，仍递增有序【查+增】
6、用顺序表最后一个元素覆盖整个顺序表中最小元素（若有多个则选取第一个），并返回该最小元素。若最后一个就是最小元素时，则不改变顺序表的状态。仅返回这个最小元素即可。【查+改】
7、删除顺序表中第一个值为 x 的元素【查+删一】
8、删除顺序表中所有值为 x 的元素（拓展：从顺序表中删除给定值在 s 到 t 之间（包含 s 和 t）的所有元素）【查+删多】

链表：

1、分别采用头插法和尾插法建立一个带头结点的单链表（思考：如何创建一个不带头结点的单链表，强化篇讲解）【头插和尾插】
2、一个带头结点递增有序的单链表 L，申请一个值为 x 的结点空间，将其插入 L 后，单链表仍保持递增有序【查+增（增是创建新结点后再插）】
将带头结点的单链表就地逆置，保证空间复杂度为 O(1)【经典头插】
3、删除带头结点单链表中第一个值为 x 的结点【查+删一】
4、删除带头结点单链表中所有值为 x 的结点（拓展：若删除给值在 s 到 t 之间（不包含 s 和 t）的所有结点呢？）【查+删多】
5、试编写算法将带头结点的单链表就地逆置，即不需要借助辅助空间，保证空间复杂度为 O(1)【头插法逆置，十分重要！！！】
6、试编写在带头结点的单链表 L 中删除最小值点的算法【查+删一】
7、递增有序地输出单链表中的各结点的数值，并释放结点空间【查+删多】
8、将一个带头节点单链表值最小的结点移动到链表的最前面【查+插】
9、设有一个由正整数组成的无序单链表，实现以下功能：【查+插+删一】
  1、找出最小值结点(非最后一个且唯一)
  2、若该数值是奇数，将其于后继结点交换（注意不是数值交换）
  3、若该数值是偶数，则将其后继结点删除
10、分别用头插法和尾插法创建一个带头节点的双向链表【对比第 1 题】
11、将带头节点双向链表中值最小的结点移动到链表最前端【对比第 8 题】
12、设有一个带头结点的循环单链表，其结点值为正整数，设计算法反复找出链表内最小值并不断输出，并将结点从链表中删除，直到链表为空，再删除表头结点【对比第 7 题】

栈和队列：

1、栈的基本操作【初始化、入、出、判空、取栈顶】
2、判断单链表中全部 n 个字符是否回文【栈的经典应用】
3、判断一个表达式中圆括号是否配对（拓展：若还有花括号多种类型的括号呢？）【栈的经典应用】
4、假设一个序列为 HSSHHHS，运用栈的知识，编写算法将 S 全部提到 H 之前，即为 SSSHHHH
5、两个栈 s1,s2 采用顺序存储，共享一个存储区[0,...,maxsize-1]。采用栈顶相向，迎面增长的存储方式，设计 s1,s2 入栈和出栈的操作。【共享栈】
6、队列的基本操作【初始化、入、出、判空】
7、设以带头节点的循环单链表表示队列，只设有队尾指针。请写出入队、出队的算法，复杂度要求均为 O(1)。【链表模拟队列】
8、利用两个栈 s1 和 s2 来模拟一个队列。如何利用栈的运算实现该队列的三个运算：入队、出队和判断队列为空。【栈模拟队列】
9、判断字符串是否回文【注意与链式存储判断回文的区别】

10、判断子串 s2 是否匹配母串 s1，若匹配，输出匹配到的第一个字符所在索引。否则输出-1。【顺序存储字符串暴力匹配】

11、有两个链表 A 和 B，判断 B 是否为 A 的连续子序列【链式存储匹配】

树（基础篇都是很简单的题目，一套递归公式全部秒杀）

1、使用先序中序后序递归遍历二叉树

2、计算二叉树中所有结点个数

3、计算二叉树中所有叶子结点的个数

拓展：计算二叉树中所有双分支的结点个数

4、求二叉树中值为 x 的层号

5、计算二叉树的最大深度（高度）

6、找出二叉树中值最大的结点

7、查找二叉树中 data 域等于 key 的结点是否存在，若存在，将 q 指向它，否则 q 为空

8、输出先序遍历第 k 个结点的值

9、把二叉树所有结点左右子树交换

10、判断二叉树是不是正则二叉树（即每个结点的度均为 0 或 2）

11、先序非递归遍历二叉树【牢记模板，学会套用，中、后同理】

12、中序非递归遍历二叉树

13、后序非递归遍历二叉树

14、层次遍历二叉树

图

1、分别使用邻接表和邻接矩阵创建一个图

2、邻接表实现图的广度优先遍历（BFS）

3、邻接矩阵实现图的广度优先遍历（BFS）

4、设计算法，求无向连通图距顶点 v 最远的一个结点（即路径长度最大）

5、邻接表实现图的深度优先遍历（DFS）

6、邻接矩阵实现图的深度优先遍历（DFS）

7、有向图采用邻接表存储，判断顶点 Vi 和顶点 Vj 之间是否存在路径

拓展：有向图采用邻接矩阵存储呢？（经典有无路径问题，DFS/BFS）

8、在有向图中，如果顶点 r 到图中所有顶点都存在路径，则称 r 为图的根结点。编写代码输出有向图中所有根结点。（第 7 题的应用）

9、求无向图的连通分量个数（DFS/BFS）

查找

1、在有序表中二分查找值为 key 的元素【二分查找，递归和非递归解法】

2、判断给定二叉树是否是二叉(搜索)排序树【树递归的应用】

3、寻找二叉排序树中最大值和最小值结点【二叉排序树的应用】

4、求出值为 key 的结点在二叉排序树的层次【二叉排序树的应用】

排序

1、直接插入排序

2、折半插入排序

3、选择排序

4、冒泡排序

5、快速排序

顺序表默认结构体:

```
#define maxsize 50
typedef struct{
    int data[maxsize];
    int length;
} Sqlist;
```

1、在顺序表 L 的第 k 个位置插入元素 x（增）

```
bool insert_L(Sqlist *L, int k, int x){
    if (k < 1 || k > L->length+1){
        printf("插入失败");
        return false;
    }
    for (int i = L->length-1; i >= k-1; i--)
        L->data[i+1] = L->data[i];
    L->data[k-1] = x;
    L->length++;
    return true;
} //时空复杂度分别为 O(N)和 O(1)
```

核心思想：

1、参数合法性检查：检查插入位置 k 是否在有效范围。若越界，输出错误并返回 false

2、元素后移：从最后一个元素开始，到第 k 个位置，依次将元素后移一位，为新元素腾出位置

3、插入元素并更新长度：将 x 放入 k-1 下标处，并将表长加 1

2、删除顺序表 L 的第 k 个元素并返回其值（删）

```
int delete_L(Sqlist *L, int k){
    if (k < 1 || k > L->length){//注意和增的区别
        printf("删除失败");
        return -100;
    }
    int res = L->data[k-1];
    for (int i = k; i < L->length; i++)
        L->data[i-1] = L->data[i];
    L->length--;
    return res;
} //时空复杂度分别为 O(N)和 O(1)
```

核心思想：

1、检查删除位置合法性：如果删除位置 k 不在［1, L->length］范围内，打印“删除失败”并结束

2、保存删除的元素：将待删除的元素存入 res 变量，供后续返回

3、元素前移：从删除位置的下一个元素开始，逐个将元素向前移动，覆盖已删除的元素。

4、更新长度并返回元素值：删除元素后，表长减 1，并返回删除元素 res

3、将顺序表中的元素逆置（改）

```
void Reverse_L(Sqlist *L){
    int i = 0, j = L->length -1;
    for ( ; i < j; i++, j--){
        int temp = L->data[i];
        L->data[i] = L->data[j];
        L->data[j] = temp;
    }
} //时空复杂度分别为 O(N)和 O(1)
```

核心思想：

1、双指针初始化：两个指针 i 和 j 分别指向顺序表起始和末尾

2、交换前后元素：通过循环，交换 i 和 j 指向的元素，每次 i 向后移，j 向前移。直到 i 和 j 相遇或交错，交换完成

4、查找顺序表中第一个值为 x 的元素的位置（查）

```
int find_L(Sqlist L, int x){
    for (int i = 0; i < L.length; i++){
        if (L.data[i] == x){
            printf("查找成功");
            return i+1;
        }
    }
    printf("查找失败");
    return 0;
} //时空复杂度分别为 O(N)和 O(1)
```

核心思想：

1、顺序遍历：从第一个元素开始，逐个检查，直到找到值为 x 的元素或遍历结束

2、找到即返回：如果当前元素等于 x，打印"查找成功"并返回该元素的位置（注意位置是 i+1，而不是下标）

3、找不到返回 0：如果遍历完整个顺序表仍未找到，则打印"查找失败"，并返回 0

5、顺序表递增有序，插入元素 x，仍递增有序

```
int find_L(Sqlist L, int x){
    for (int i = 0; i < L.length; i++){
        if (x < L.data[i])
            return i;
    }
    return L.length;
}
void insert_L(Sqlist *L, int x){
    int pos = find_L(*L, x);
    for (int i = L->length - 1; i >= pos; i--)
        L->data[i + 1] = L->data[i];
    L->data[pos] = x;
    L->length++; //别遗漏
} //时空复杂度分别为 O(N)和 O(1)
```

核心思想：

1、查找插入位置：通过遍历顺序表，找到第一个比 x 大的元素位置，确定插入点，使插入后仍保持递增有序

2、元素后移腾位：从表尾开始，到插入位置，逐个元素向后移动一位，为新元素空出位置

3、插入元素并更新长度：在找到的位置插入元素 x，顺序表长度加 1

这是一个增和查的结合题目。因为是有序的，建议大家学完二分查找之后，再做一下这个题目。

6、用顺序表最后一个元素覆盖整个顺序表中最小元素（若有多个则选取第一个），并返回该最小元素。若最后一个就是最小元素时，则不改变顺序表的状态。仅返回这个最小元素即可。

```
int Del_min(Sqlist *L){
    int pos = 0;
    int min = L->data[0];
    for (int i = 0; i < L->length; i++)
        if (min > L->data[i]){
            min = L->data[i];
            pos = i;
        }
    if (pos == L->length-1)
        return min;
    L->data[pos] = L->data[L->length-1];
    L->length--; //可以不写，这题表达有歧义
    return min;
} //时空复杂度分别为 O(N)和 O(1)
```

核心思想：

1、寻找最小元素：遍历顺序表，记录第一个最小值及其出现的位置

2、判断是否为最后一个元素：如果最小元素已经在最后一个位置，直接返回最小值，不需要改动顺序表

3、覆盖最小元素并更新长度：否则，用最后一个元素覆盖最小元素所在位置（表尾元素补到最小值处），并将顺序表长度减 1（可选操作）

拓展：若有多个最小值选取最后一个呢？

7、删除顺序表中第一个值为 x 的元素

```
bool del_x(Sqlist *L, int x){
    for (int i = 0; i < L->length; i++){
        if (L->data[i] == x){
            for (int j = i+1; j < L->length; j++)
                L->data[j-1] = L->data[j];
            L->length--;
            return true;
        }
    }//这个花括号可省略，for 内就是一个整体
    return false;
} //时空复杂度分别为 O(N)和 O(1)
```

核心思想：

1、查找目标元素：从顺序表头开始遍历，找到第一个等于给定值 x 的位置

2、元素前移：找到后，从下一个元素开始，依次向前覆盖，完成删除操作

3、更新表长并返回：顺序表长度减 1，表示元素减少；删除成功返回 true，否则返回 false

8、删除顺序表中所有值为 x 的元素

```
bool del(Sqlist *L, int x){
    int k = 0;
    for (int i = 0; i < L->length; i++)
        if (L->data[i] != x){
            L->data[k] = L->data[i];
            k++;
        }
    if (L->length == k)
        return false;
    else{
        L->length = k;
        return true;
    }
}//时空复杂度分别为 O(N)和 O(1)
```

核心思想：

1、双指针遍历：使用两个指针：i 遍历原表，k 指向当前保留元素的位置

2、筛选保留元素：遇到不等于 x 的元素，就拷贝到 k 位置，同时 k++，实现覆盖式保留

3、更新表长并返回：遍历完成后，顺序表的新长度为 k。如果没有元素被删除，返回 false；否则返回 true

拓展：若删除给定值在 s 到 t 之间的所有元素？

单链表默认结构体：

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

1、分别采用头插法和尾插法建立一个带头结点的单链表

```
LinkList create(){
    LinkList L = (LinkList)malloc(sizeof(LNode));
    L->next = NULL;
    LinkList r = L;
    int x;
    scanf("%d", &x);
    while (x != 9999){
        LinkList s = (LinkList)malloc(sizeof(LNode));
        s->data = x;
        s->next = NULL;
        r->next = s;
        r = s;
        scanf("%d", &x);
    }
    return L;
}//尾插法   时空复杂度分别为 O(N)和 O(N)
```

核心思想：

1、初始化头结点

2、读入数据，直到遇到结束标志

3、逐个尾插新结点

4、返回头指针

```c
LinkList create(){
    LinkList L = (LinkList)malloc(sizeof(LNode));
    L->next = NULL;
    int x;
    scanf("%d", &x);
    while (x != 9999){
        LinkList s = (LinkList)malloc(sizeof(LNode));
        s->data = x;
        s->next = L->next;
        L->next = s;
        scanf("%d", &x);
    }
    return L;
}//头插法    时空复杂度分别为 O(N)和 O(N)
```

核心思想：

1、初始化头结点：创建一个头结点 L，初始 next 为 NULL，作为链表入口

2、读入数据，直到遇到结束标志：使用 scanf 读入元素，输入 9999 结束

3、逐个头插新结点：每次读入一个值，动态分配新节点 s，将 s->next 指向当前头节点后的第一个节点（即 L->next），然后把 L->next 指向 s，实现头插

4、返回头指针：构建完成后，返回头结点 L（注意，真正的第一个数据节点是 L->next）

2、一个带头结点递增有序的单链表 L，申请一个值为 x 的结点空间，将其插入 L 后，单链表仍保持递增有序

```c
void InsertNode(LinkList L, int x) {
    LinkList p = (LinkList)malloc(sizeof(LNode));
    p->data = x;
    p->next = NULL;
    LinkList r = L;
    while (r->next != NULL){
        if (r->next->data > x)
            break;
        r= r->next;
    }
    p->next = r->next;
    r->next = p;
} //时空复杂度分别为 O(N)和 O(1)
```

核心思想：

1、创建新节点：先动态分配一个新节点 p，存入数据 x，并初始化 p->next = NULL

2、寻找插入位置：指针 r 从头结点开始遍历链表。当 r 的下一个结点值大于 x 时停止，准备在 r 后插入新节点

3、完成插入：将 p->next 指向 r->next，然后让 r->next 指向 p，插入完成，保证链表仍然有序

3、删除单链表中第一个值为 x 的结点

```c
bool del(LinkList L, int x){
    LinkList r=L;
    while (r->next != NULL){
        if (r->next->data == x){
            LinkList p = r->next;
            r->next = p->next;
            free(p);
            return true;
        }
        r = r->next;
    }
    return false;
} //时空复杂度分别为 O(N)和 O(1)
```

核心思想：

1、准备辅助指针，查找目标删除位置：用指针 r 从头结点开始遍历链表。当 r 的下一个结点值等于目标值 x 时。停止遍历

2、删除节点并释放空间：将 r->next 指向被删除节点的后继节点（跳过目标节点），然后释放目标节点的内存

3、返回结果：如果成功删除，返回 true；如果遍历到结尾还没找到，说明删除失败，返回 false

4、删除单链表中所有值为 x 的结点

```
bool Del(LinkList L, int x){
   LinkList r = L;
   int flag = 0;
   while (r->next != NULL)
      if (r->next->data == x){
         flag = 1;
         LinkList p = r->next;
         r->next = p->next;
         free(p);
      }
      else
         r = r->next;
   if (flag == 1)
      return true;
   else
      return false;
} //时空复杂度分别为 O(N)和 O(1)
```

核心思想：

1、设置辅助指针和标记：使用指针 r 从头结点开遍历链表，同时定义 flag 标记是否曾删除节点

2、遍历链表，逐个检查：判断 r 的下一个结点值是否等于 x。若相等，删除其下一个结点，不移动 r，以防连续值被跳过；否则，r 后移继续遍历

3、返回删除结果：遍历完成后，根据 flag 判断是否成功删除至少一个节点，返回 true 或 false

5、试编写算法将带头结点的单链表就地逆置，即不需要借助辅助空间，保证空间复杂度为 O(1)

```
void reverse(LinkList L){
   LinkList p = L->next;
   L->next = NULL;
   LinkList r;
   while (p != NULL){
      r = p->next;
      p->next = L->next;
      L->next = p;
      p = r;
   }
} //时空复杂度分别为 O(N)和 O(1)
```

核心思想（头插法）：

1、初始化指针：用指针 p 遍历原链表的每个结点，同时将头结点的 next 置为 NULL，表示新链表为空

2、逐个反转结点指向：在循环中，先暂存 p->next 到 r，然后将 p->next 指向当前新链表的头部（L->next）。再将 L->next 指向当前的 p，相当于把 p 头插到了新链表的最前面（自己动手画图更直观）

3、继续遍历：用 p = r 进入下一个未处理的结点，继续上述过程直到原链表遍历完毕

6、试编写在带头结点的单链表 L 中删除最小值点的高效算法（已知最小值唯一）

```
void del_min(LinkList L){
   LinkList p = L;
   LinkList pos = L;
   while (p->next != NULL){
      if (p->next->data <pos->next->data)
         pos = p;
      p = p->next;
   }
   LinkList u= pos->next;
   pos->next = u->next;
   free(u);
} //时空复杂度分别为 O(N)和 O(1)
```

核心思想：

1、初始化指针：使用指针 p 遍历链表，用 pos 来记录当前已知最小值结点的前驱位置，初始都指向头结点 L

2、查找最小值节点：遍历整个链表，若发现某个结点的值小于 pos->next->data，则更新 pos 为该结点的前驱，确保最终 pos->next 指向最小结点

3、删除最小值节点：删除记录下的最小值结点：令 pos->next 指向最小值结点的下一个结点，然后释放最小值结点的空间，完成删除操作

7、给定一个单链表，按递增有序地输出单链表中各结点的数值，并释放结点所占空间（不断寻找最小值）

```
void del_min(LinkList L){
   while (L->next != NULL){
      LinkList p = L;
      LinkList pos = L;
      while (p->next != NULL){
         if (p->next->data< pos->next->data)
            pos = p;
         p = p->next;
      }
      printf("%d", pos->next->data);
      LinkList u = pos->next;
      pos->next = u->next;
      free(u);
   }
   free(L);
} //时空复杂度分别为 O(N²)和 O(1)
```

核心思想：

1、循环找最小值节点：每次从链表头开始，用两个指针 p 和 pos 找到当前链表中值最小的结点，其中，pos 用来指向最小值结点的前驱

2、输出最小值并删除节点：找到后，输出其值，然后将其从链表中断开，并释放对应内存

3、重复直到链表为空：重复上述过程，直到链表中所有元素都被删除，最后，记得释放头结点 L 所占内存

8、将一个带头节点单链表中值最小的结点移动到整个链表的最前面。

```
bool move(LinkList L){
   LinkList p = L;
   LinkList pos = L;
   while (p->next != NULL){
      if (p->next->data < pos->next->data)
         pos = p;
      p = p->next;
   }
   if (pos != L){
      LinkList r = pos->next;
      pos->next = r->next;
      r->next = L->next;
      L->next = r;
      return true;
   }
   else
      return false;
} //时空复杂度分别为 O(N)和 O(1)
```

核心思想：

1、查找最小值的前驱结点：遍历整个链表，用指针 pos 记录当前最小值结点的前驱位置，即 pos->next 是最小值结点

2、若最小值结点不在首节点位置：如果最小值结点不是第一个节点，将其从当前位置删除，并插入到链表首部

3、返回处理结果：如果移动操作执行了，返回 true；否则，不做处理，返回 false

9、设有一个由正整数组成的无序单链表，编写程序之下以下功能：

1、找出最小值结点(非最后一个且唯一)

2、若该数值是奇数，将其于后继结点的结点交换（注意不是数值交换）

3、若该数值是偶数，则将其后继结点删除

```
bool func(LinkList L){
   LinkList p = L, pos=L;
   while (p->next != NULL){
      if (p->next->data < pos->next->data)
         pos = p;
      p = p->next;
   }
   if (pos->next->next == NULL)
      return false;
   LinkList u = pos->next;
   LinkList r = u->next;
   u->next = r->next;
   if (pos->next->data % 2 == 0)
      free(r);
   else{
      r->next = pos->next;
      pos->next = r;
   }
   return true;
} //时空复杂度分别为 O(N)和 O(1)
```

核心思想：

1、查找最小值结点的前驱

2、排除最后一个结点的情况

3、判断最小值奇偶，分别处理

双向链表默认结构体

```c
typedef struct DNode {
        int data;
        struct DNode* next;
        struct DNode* prev;
} DNode, *DinkList;
```

10、分别采用头插法和尾插法创建一个带头节点的双向链表

```c
DinkList create() {
    DinkList L = (DinkList)malloc(sizeof(DNode));
    L->next = NULL;
    L->prev = NULL;
    DinkList r = L;
    int x;
    scanf("%d", &x);
    while (x != 9999) {
        DinkList s =(DinkList)malloc(sizeof(DNode));
        s->data = x;
        s->next = NULL;
        r->next = s;
        s->prev = r;
        r = s;
        scanf("%d", &x);
    }
    return L;
} //尾插法  时空复杂度分别为 O(N)和 O(N)
```

核心思想：

1、初始化链表

2、尾部添加新节点

3、更新尾指针

```c
DinkList create(){
    DinkList L = (DinkList)malloc(sizeof(DNode));
    L->next = NULL;
    L->prev = NULL;
    int x;
    scanf("%d", &x);
    while(x != 9999){
        DinkList s= (DinkList)malloc(sizeof(DNode));
        s->data = x;
        s->next = L->next;
        if (L->next != NULL)
            L->next->prev = s;
        L->next = s;
        s->prev = L;
        scanf("%d", &x);
    }
    return L;
}//头插法      时空复杂度分别为 O(N)和 O(N)
```

核心思想：

1、初始化链表：创建头结点 L，初始化 next 和 prev 为 NULL

2、头部插入新节点：每次读取数据，动态申请新节点 s，将其插入到 L 后面（即链表头部）

3、维护双向关系：若 L 后已有节点，要更新其 prev 指向新节点；新节点 prev 指向头结点 L

11、将一个带头节点双向链表中值最小的结点移动到整个链表的最前面。

```c
bool move(DinkList L){
    DinkList p = L, pos = L;
    while (p->next != NULL){
        if (p->next->data < pos->next->data)
            pos = p;
        p = p->next;
    }
    if (pos != L){
        DinkList r = pos->next;
        pos->next = r->next;
        //加个 if 语句，防止最后一个结点值最小
        if (r->next != NULL)
            r->next->prev = pos;
        r->next = L->next;
        L->next->prev = r;
        L->next = r;
        r->prev = L;
        return true;
    }
    else
        return false;
} //时空复杂度分别为 O(N)和 O(1)
```

核心思想：

1、查找最小值结点的前驱 pos

2、摘除并断链处理：若最小值结点不是第一个数据结点，则从当前位置摘除；若是尾结点，需额外处理其后继指针

3、将最小值结点插入到头结点之后

12、设有一个带头结点的循环单链表，其结点值为正整数，设计算法反复找出链表内最小值并不断输出，并将结点从链表中删除，直到链表为空，再删除表头结点

```
void del_min(LinkList L){
   while (L->next != L){
      LinkList p = L;
      LinkList pos = L;
      while (p->next != L){
        if (p->next->data < pos->next->data)
             pos = p;
        p = p->next;
      }
      printf("%d", pos->next->data);
      LinkList u = pos->next;
      pos->next = u->next;
      free(u);
   }
   free(L);
}
```

时空复杂度分别为 $O(N^2)$和 O(1)

核心思想：

1、循环找最小：每轮遍历循环链表，找到最小值结点的前驱 pos，为删除操作做准备

2、输出并删除：输出最小值，断开其连接，释放对应结点，循环继续直至链表仅剩头结点

3、释放头结点：所有数据结点删除完毕后，最后释放头结点所占空间，完成整体销毁

其实循环单链表的题目都可以看作是单链表题目的拓展，例如遍历、逆置、删除、合并等等，区别主要体现在循环终止条件

栈的结构体：

```
#define maxsize 50
typedef struct{
   char data[maxsize];
   int top;
} stack;
```

队列的结构体

```
typedef struct{
   char data[maxsize];
   int front, rear;
} queue;
```

1、栈的基本操作

```
void init(stack* s){
      s->top = -1;
}//初始化

bool IsEmpty(stack s){
   return s.top == -1;
}//判断栈空，这里可以不用指针传递

bool push(stack* s, int x){
   if (s->top == maxsize - 1)
      return false;
   s->data[++s->top] = x;
   return true;
}//入栈

bool pop(stack* s, int* x){
   if (s->top == -1)
      return false;
   *x = s->data[s->top--];
   return true;
}//出栈

bool GetTop(stack s,int *x){
   if (s.top == -1)
      return false;
   *x = s.data[s.top];
   return true;
}//获取栈顶元素
```

2、判断单链表中全部 n 个字符是否回文

```
bool func(LinkList L){
   stack s;
   s.top = -1;
   LinkList p = L->next;
   while (p != NULL){
      s.data[++s.top] = p->data;
      p = p->next;
   }
   p = L->next;
   while (p != NULL){
      if (s.data[s.top--] != p->data)
         return false;
      p = p->next;
   }
   return true;
} //时空复杂度分别为 O(N)和 O(N)
```

核心思想：

1、入栈保存：遍历链表，将所有结点数据依次压入栈中，保留原始顺序的反转副本

2、逐一对比：再次从链表头开始遍历，依次与栈顶元素比较，若有不等则说明不是回文

3、判断结果：若全部匹配成功，说明链表构成回文，返回 true；否则返回 false

3、判断一个表达式中圆括号是否配对

```
bool fun(char A[]){
   stack s;
   s.top = -1;
   for (int i = 0; A[i] != '\0'; i++)
      if (A[i] == '(')
         s.data[++s.top] = '(';
      if (A[i] == ')')
         if(s.top == -1)
            return false;
         else
            s.top--;
   }
   if (s.top == -1)
      return true;
   else
      return false;
} //时空复杂度分别为 O(N)和 O(N)
```

拓展：若还有花括号多种类型的括号呢？

核心思想：

1、左括号入栈：遇到（ 就压栈，用于记录未匹配的左括号

2、右括号匹配：遇到 ）时弹出栈顶，如果栈为空说明缺左括号，直接返回 false

3、最终判断：遍历结束后栈应为空，若非空说明有未配对的左括号，返回 false；否则返回 true

4、假设一个序列为 HSSHHHS，运用栈的知识，编写算法将 S 全部提到 H 之前，即为 SSSHHHH

```
void fun(char A[]){
   int k = 0;
   stack s;
   s.top = -1;
   for (int i = 0; A[i] != '\0'; i++){
      if (A[i] == 'S')
         A[k++] = A[i];
      else
         s.data[++s.top] = A[i];
   }
   while (s.top != -1)
      A[k++] = s.data[s.top--];
} //时空复杂度分别为 O(N)和 O(N)
```

核心思想：

1、遍历字符串：扫描每个字符，将所有 'S' 依次保存在原数组前面，记录位置 k

2、栈存非 'S'：遇到非 'S' 字符则入栈，实现逆序保存（其实用队列保存也可以，不一定逆序）。

3、填充尾部：栈中字符依次出栈，补到数组后面，形成 'S' 在前，其它字符在后

5、两个栈 s1,s2 都采用顺序存储，并共享一个存储区[0,...,maxsize-1]。采用栈顶相向，迎面增长的存储方式，设计 s1,s2 入栈和出栈的操作。

```
typedef struct {
    int data[maxsize];
    int top1;
    int top2;
} stack;
bool push(stack *s, int i, int x){
    if (s->top2 - s->top1 == 1 || (i != 1 && i != 2))
        return false;
    if (i == 1)   s->data[++s->top1] = x;
    if (i == 2)   s->data[--s->top2] = x;
    return true;
} //时空复杂度分别为 O(1)和 O(1)
bool pop(stack *s, int i, int *x){
    if (i !=1 &&i != 2)
        return false;
    if (i == 1)
        if (s->top1 == -1)
            return false;
        else
            *x= s->data[s->top1--];
    if (i == 2)
        if (s->top2 == maxsize)
            return false;
        else
            *x = s->data[s->top2++];
    return true;
} //时空复杂度分别为 O(1)和 O(1)
```

核心思想：

1、栈结构设计：用一个数组 data[0...maxsize-1] 存放两个栈的数据。top1 从左向右增长（初值为 -1），top2 从右向左增长（初值为 maxsize）。当 top2 - top1 == 1 时，空间已满

2、入栈操作：

若栈空间满或编号非法（不是 1 或 2），返回 false。

若操作栈 1，则执行 data[++top1] = x

若操作栈 2，则执行 data[--top2] = x

3、出栈操作：

若栈 1 空（操作栈 1 的前提）或栈 2 空（操作栈 2 的前提），或者编号非法（不是 1 或 2），返回 false。

若操作栈 1，则执行 *x= s->data[s->top1--]

若操作栈 2，则执行 *x = s->data[s->top2++]

6、队列的基本操作

```
void init(queue* q){
    q->front = q->rear = 0;
}//初始化

bool IsEmpty(queue q){
    if (q.front == q.rear)
        return true;
    else
        return false;
}//判断队空

bool enqueue(queue* q, int x){
    if ((q->rear + 1) % maxsize == q->front)
        return false;
    q->data[q->rear] = x;
    q->rear = (q->rear + 1) % maxsize;
    return true;
}//入队

bool dequeue(queue* q, int* x){
    if (q->front == q->rear)
        return false;
    *x = q->data[q->front];
    q->front = (q->front + 1) % maxsize;
    return true;
}//出队
```

7、设以带头节点的循环单链表表示队列，只设有队尾指针。请写出入队、出队的算法，复杂度要求均为 O(1)。

```
void enqueue(LinkList *tail, int x){
    LNode* p = (LNode*)malloc(sizeof(LNode));
    p->data = x;
    p->next = (*tail)->next;
    (*tail)->next = p;
    (*tail) = p;
} //时空复杂度分别为 O(1)和 O(1)
```

核心思想

1、创建新结点：申请一个新结点 p，存入数据 x

2、新结点插入到尾结点之后：p 结点的 next 指向头结点。让原尾结点的 next 指向新结点 p

3、更新尾指针：令 tail 指向新结点，即新结点成为新的尾结点

```
bool dequeue(LinkList *tail, int *x){
    if ((*tail)->next == (*tail)){
        return false;
    }
    LinkList p = (*tail)->next;
    LinkList r = p->next;
    p->next = r->next;
    *x = r->data;
    free(r);
    if (p->next == p)
        *tail = p
    return true;
} //时空复杂度分别为 O(1)和 O(1)
```

核心思想：

1、**判断队列是否为空**：若 tail->next == tail，说明循环链表中无其他结点，返回 false

2、**删除队首结点**：找到队首结点 r，将其从链中断开，释放空间并将其数据赋值给 x

3、**更新头或尾指针**：若删除后只剩一个结点（即 tail->next == tail），则更新 tail 指向该唯一结点

8、用两个栈 s1 和 s2 来模拟一个队列。如何利用栈的运算实现该队列的两个运算：入队和出队。

```
bool enqueue(stack* s1, stack* s2, int x){
    if (s1->top == maxsize-1 && s2->top != -1)
        return false;
    if (s1->top == maxsize-1 && s2->top == -1)
        while (s1->top != -1)
            s2->data[++s2->top]=s1->data[s1->top--];
    s1->data[++s1->top] = x;
    return true;
} //时空复杂度分别为 O(N)和 O(N)
```

核心思想：

1、判断空间是否溢出：若栈 1 已满，且栈 2 不为空，说明总空间已用完，入队失败，直接返回 false

2、元素转移（仅在必要时）：若栈 1 满但栈 2 为空，则将栈 1 中元素逐个弹出并压入栈 2，为入队腾出空间

3、完成入队：将新元素压入栈 1 顶，入队成功，保持栈 1 负责入队的职责

```
bool dequeue(stack* s1, stack* s2, int* x){
    if (s2->top == -1&&s1->top == -1)
        return false;
    if (s2->top == -1&&s1->top != -1)
        while (s1->top != -1)
            s2->data[++s2->top]=s1->data[s1->top--];
    *x = s2->data[s2->top--];
    return true;
} //时空复杂度分别为 O(N)和 O(N)
```

核心思想：

1、判断是否为空队列：如果两个栈都为空，说明队列为空，出队失败，返回 false

2、若 s2 不为空：直接从 s2 弹出栈顶元素，即可完成出队操作，满足先进先出

3、若 s2 为空而 s1 不为空：将 s1 中的所有元素依次弹出并压入 s2，使得原本先进的元素位于 s2 的栈顶，再从 s2 弹出栈顶元素

9、判断字符串是否回文

```
bool is_true(char str[]) {
    int N = strlen(str);//c 语言内置，计算长度
    for (int i = 0; i < N/ 2; i++)
        if (str[i] != str[N-i-1])
            return false;
    return true;
} //时空复杂度分别为 O(N)和 O(1)
```

核心思想：

1、双指针比较：从字符串的两端开始（i 从左，N-i-1 从右），逐个字符进行比较

2、不相等立即返回：一旦发现任意一对字符不相等，即说明不是回文串，直接返回 false

3、全部匹配即为回文：若所有对应字符都相等，则说明字符串为回文，返回 true

10、判断子串 s2 是否匹配母串 s1，若匹配，输出匹配到的第一个字符所在索引。否则输出-1。

```
int find(char s1[], char s2[]) {
    int pos = 0;
    int p1 = pos, p2 = 0;
    while (s1[p1] != '\0' && s2[p2] != '\0') {
        if (s1[p1] == s2[p2]){
            p1++;
            p2++;
        }
        else{
            p2=0;
            pos++;
            p1=pos;
        }
    }
    if (s2[p2] == '\0')
        return pos;
    else
        return -1;
}//时空复杂度分别为 O(M*N)和 O(1)
```

核心思想：

1、指针控制：p1 指向主串 S1 待匹配位置，p2 指向模式串 S2 待匹配位置。当字符匹配时，两指针都后移；否则，主串起点位置 pos 加 1，p1 重设为新起点 pos，p2 归 0

2、判断匹配成功：如果 s2[p2] == '\0'，说明全部字符都成功匹配，返回起始位置 pos；否则返回 -1 表示未找到

11、有两个链表 A 和 B，判断 B 是否是 A 的连续子序列

```
bool find(LinkList A, LinkList B){
    LinkList pos = A->next;
    LinkList p1 = pos, p2=B->next;
    while (p1 != NULL && p2 != NULL){
        if (p1->data == p2->data){
            p1 = p1->next;
            p2 = p2->next;
        }
        else{
            pos = pos->next;
            p1 = pos;
            p2 = B->next;
        }
    }
    if (p2 == NULL)
        return true;
    else
        return false;
}
```

核心思想：

1、指针控制：p1 指向主串 A 待匹配位置，p2 指向模式串 B 待匹配位置。当字符匹配时，两指针都后移；否则，主串起点位置 pos 往后移动，p1 重设为新起点 pos，p2 回到 B 的起始位置

2、判断匹配成功：如果 p2 最后指向空，返回 true，表示链表 B 是链表 A 的一个子链表；如果匹配失败，返回 false，表示没有找到匹配

二叉树的结构体（链式存储）（二叉链表存储）

```
typedef struct BTNode{
    char data;
    struct BTNode *lchild, *rchild;
} BTNode, *BiTree;
```

1、使用先序中序后序递归遍历二叉树

```
void pre_print(BiTree T){
    if (T!=NULL){
        printf("%c", T->data);
        pre_print(T->lchild);
        pre_print(T->rchild);
    }
}
void in_print(BiTree T){
    if (T!=NULL){
        in_print(T->lchild);
        printf("%c", T->data);
        in_print(T->rchild);
    }
}
void post_print(BiTree T){
    if (T!=NULL){
        post_print(T->lchild);
        post_print(T->rchild);
        printf("%c" ,T->data);
    }
} //时空复杂度分别为 O(N)和 O(N)
```

二叉树递归遍历这部分的算法思想不再描述，大家看视频去理解，一通百通~

2、计算二叉树中所有结点个数

计算型：

```
int count(BiTree p){
    if (p == NULL)
        return 0;
    else{
        int n1 = count(p->lchild);
        int n2 = count(p->rchild);
        return n1 + n2 + 1;
    }
}
```

操作型：

```
void count(BiTree p, int* n){
    if (p != NULL){
        ++(*n);
        count(p->lchild, n);
        count(p->rchild, n);
    }
} //时空复杂度分别为 O(N)和 O(N)
```

3、计算二叉树中所有叶子结点的个数

计算型：

```
int count(BiTree p){
    if (p == NULL)
        return 0;
    if (!p->lchild && !p->rchild)
        return 1;
    else{
        int n1 = count(p->lchild);
        int n2 = count(p->rchild);
        return n1 + n2;
    }
}
```

操作型：

```
void count(BiTree p, int* n){
    if (p != NULL){
        if (!p->lchild && !p->rchild)
            ++(*n);
        count(p->lchild, n);
        count(p->rchild, n);
    }
} //时空复杂度分别为 O(N)和 O(N)
```

拓展：如何计算二叉树中所有双分支的结点个数

计算型：

```
int count(BiTree p){
    int n1, n2;
    if (p == NULL)
        return 0;
    int n1 = count(p->lchild);
    int n2 = count(p->rchild);
    if (p->lchild && p->rchild)
        return n1 + n2 + 1;
    else
        return n1 + n2;
}
```

操作型：

```
void count(BiTree p, int* n){
    if (p != NULL){
        if (p->lchild && p->rchild)
            ++(*n);
        count(p->lchild, n);
        count(p->rchild, n);
    }
} //时空复杂度分别为 O(N)和 O(N)
```

//如果是单分支结点个数呢？

4、求二叉树中值为 x 的层号

计算型：

```
int find(BiTree p, int x) {
    if (p == NULL)
        return 0;
    if (p->data == x)
        return 1;
    int L1 = find(p->lchild, x);
    if (L1 != 0)
        return L1 + 1;
    int L2 = find(p->rchild, x);
    if (L2 != 0)
        return L2 + 1;
    return 0;
}
```

操作型：

```
void find(BiTree p, int x, int *L){
    if (p != NULL){
        ++(*L);
        if (p->data == x)
            printf("%d", *L);
        find(p->lchild, x, L);
        find(p->rchild, x, L);
        --(*L);
    }
} //时空复杂度分别为 O(N)和 O(N)
```

5、计算二叉树的最大深度（高度）

计算型：

```
int func(BiTree p){
    if (p == NULL)
        return 0;
    else{
        int L1 = func(p->lchild);
        int L2 = func(p->rchild);
        return (L1 > L2 ? L1 : L2) + 1;
    }
}
```

操作型：

```
void func(BiTree p, int *L, int *max_L){
    if (p != NULL){
        ++(*L);
        if (*L >= *max_L)
            *max_L = *L;
        func(p->lchild, L, max_L);
        func(p->rchild, L, max_L);
        --(*L);
    }
} //时空复杂度分别为 O(N)和 O(N)
```

6、找出二叉树中值最大的结点

```
void func(BiTree p, BiTree *m){
    if (p !=NULL){
        if ((*m)==NULL || p->data>(*m)->data)
            *m=p;
        func(p->lchild, m);
        func(p->rchild, m);
    }
}//时空复杂度分别为 O(N)和 O(N)
```

7、查找二叉树中 data 域等于 key 的结点是否存在，若存在，将 q 指向它，否则 q 为空

```
void func(BiTree p, BiTree *q, int key){
    if (p != NULL){
        if (p->data == key)
            *q = p;
        func(p->lchild, q, key);
        func(p->rchild, q, key);
    }
} //时空复杂度分别为 O(N)和 O(N)
```

8、输出先序遍历第 k 个结点的值

```
void func(BiTree p, int k, int* n){
    if (p != NULL){
        ++(*n);
        if (*n == k)
            printf("%c", p->data);
        func(p->lchild, k, n);
        func(p->rchild, k, n);
    }
} //时空复杂度分别为 O(N)和 O(N)
```

9、把二叉树所有结点左右子树交换

```
void swap(BiTree p){
    if (p != NULL){
        BiTree temp = p->lchild;
        p->lchild = p->rchild;
        p->rchild = temp;
        swap(p->lchild);
        swap(p->rchild);
    }
} //时空复杂度分别为 O(N)和 O(N)
```

需要讲义对应讲解视频，可加下方微信获取

10、判断二叉树是不是正则二叉树（即每个结点的度均为 0 或 2）

计算型：

```
int func(BiTree p) {
    if (p == NULL)
        return 1;
    if ((p->lchild == NULL && p->rchild != NULL)
        ||(p->lchild != NULL && p->rchild == NULL))
        return 0;
    int L1 = func(p->lchild);
    int L2 = func(p->rchild);
    return L1&&L2;
}
```

操作型：

```
void func(BiTree p, int *flag){
    if (p!=NULL){
        if ((p->lchild==NULL && p->rchild!=NULL)
            ||(p->lchild!=NULL && p->rchild==NULL))
            *flag = 0;
        func(p->lchild, flag);
        func(p->rchild, flag);
    }
} //时空复杂度分别为 O(N)和 O(N)
```

11、先序非递归遍历二叉树

```
void Nonpre(BiTree bt){
    BiTree S[maxsize];
    int top = -1;
    while (bt || top != -1)
        if (bt != NULL){
            printf("%d",bt->data);
            S[++top] = bt;
            bt = bt->lchild;
        }
        else{
            bt = S[top--];
            bt = bt->rchild;
        }
} //时空复杂度分别为 O(N)和 O(N)
```

核心思想：

1、栈模拟递归：递归依赖系统栈保存上下文，非递归手动创建数组 S 作栈，用 top 指示栈顶。top 初始化为-1 表示栈空，bt 设为根节点

2、循环遍历逻辑：

bt 不为空，访问节点、入栈，更新 bt 为左子节点，遵循先根后左的先序规则

bt 为空，出栈，更新 bt 为右子节点，开始遍历右子树

3、终止条件：bt 为空且栈空时结束，意味着树遍历完成

12、中序非递归遍历二叉树

```
void Nonin(BiTree bt){
    BiTree S[maxsize];
    int top = -1;
    while (bt || top != -1)
        if (bt){
            S[++top] = bt;
            bt = bt->lchild;
        }
        else{
            bt = S[top--];
            printf("%d", bt->data);
            bt = bt->rchild;
        }
} //时空复杂度分别为 O(N)和 O(N)
```

核心思想：

1、栈模拟递归：递归依赖系统栈保存调用上下文，非递归实现手动用数组 S 作栈，top 指示栈顶。初始时 top 设为-1 表示栈空，bt 为二叉树根节点，开启遍历

2、循环遍历逻辑：

bt 不为空：按“左-根-右”顺序，将 bt 入栈，更新 bt 为左子节点，深入左子树

bt 为空：左子树遍历完，出栈节点并访问，bt 指向其右子节点，遍历右子树

3、终止条件：bt 为空且栈空时结束，意味着树遍历完成

13、后序非递归遍历二叉树

```
void Nonpost(BiTree bt){
    BiTree S[maxsize], nowp, tag = NULL;
    int top = -1;
    while (bt || top != -1)
        if (bt){
            S[++top] = bt;
            bt = bt->lchild;
        }
        else{
            nowp = S[top];
            if (nowp->rchild && nowp->rchild != tag)
                bt = nowp->rchild;
            else{
                nowp = S[top--];//或写 top--也可
                printf("%d", nowp->data);
                tag = nowp;
            }
      }
} //时空复杂度分别为 O(N)和 O(N)
```

核心思想：

1、栈模拟递归：需额外定 nowp 临时保存节点，tag 指针初始 NULL，用于标记已访问的右子树

2、循环遍历逻辑：bt 不为空：将 bt 入栈，bt 指向左子节点，持续深入左子树；bt 为空：取栈顶节点到 nowp。若 nowp 右子树存在且未被访问（即不等于 tag），则 bt 指向右子树；若右子树不存在或已访问，弹出栈顶节点 nowp 并访问，更新 tag 为 nowp

3、终止条件：同先序和中序

14、层次遍历二叉树

```
void level(BiTree bt){
    BiTree que[maxsize];
    int front = 0, rear = 0;
    if (bt != NULL){
        que[++rear] = bt;
        while (front != rear){
            bt = que[++front];
            printf("%d", bt->data);
            if (bt->lchild != NULL)
                que[++rear] = bt->lchild;
            if (bt->rchild != NULL)
                que[++rear] = bt->rchild;
        }
    }
} //时空复杂度分别为 O(N)和 O(N)
```

核心思想：

1、初始化与队列模拟：为实现二叉树的层序遍历，手动用数组 que 模拟队列，front 和 rear 分别表示队首和队尾，初始都为 0。若根节点 bt 不为空，则将其入队

2、循环遍历逻辑：

先出队：将队首元素出队到 bt，并访问该节点（打印数据）

再入队：若 bt 的左子节点不为空，将其左子节点入队；若右子节点不为空，将其右子节点入队。

3、终止条件：当 front 等于 rear，即队列空时，层序遍历结束。

//这里入队和出队的时候，先++再赋值或者先赋值再++都是可以的，rear 和 front 保持一致即可

图的结构体

邻接表存储：

```
typedef struct ANode{
    int adjvex; //边所指向结点的位置
    struct ANode *nextarc;
} ANode, *Node; //边结点结构体

typedef struct{
    int data;
    ANode *firstarc;
} Vnode; //顶点结构体

typedef struct{
    int numver, numedg;
    Vnode adjlist[maxsize];
} Graph;
```

邻接矩阵存储：

```
typedef struct{
    int numver, numedg;
    int verticle[maxsize];
    int Edge[maxsize][maxsize];
} mGraph;
```

1、分别使用邻接表和邻接矩阵创建一个图

```
void create(Graph *G){
    for (int i = 0; i < G->numver; i++)
        G->adjlist[i].firstarc = NULL;
    for (int i = 0; i < G->numedg; i++){
        int v1, v2;
        scanf("%d%d", &v1, &v2);
        Node p = (Node)malloc(sizeof(ANode));
        p->adjvex = v2;
        p->nextarc = G->adjlist[v1].firstarc;
        G->adjlist[v1].firstarc = p;
        Node q = (Node)malloc(sizeof(ANode));
        q->adjvex = v1;
        q->nextarc = G->adjlist[v2].firstarc;
        G->adjlist[v2].firstarc = q;
    }
}//邻接表创建无向图（去除下划线为有向图）

void create(MGraph *G){
    for (int i = 0; i < G->numver; i++)
        for (int j = 0; j < G->numver; j++)
            G->Edge[i][j] = 0;
    for (int i = 0; i < G->numedg; i++){
        int v1, v2;
        scanf("%d%d",&v1,&v2);
        G->Edge[v1][v2] = 1;
        G->Edge[v2][v1] = 1;
    }
}//邻接矩阵创建无向图（去除下划线为有向图）
```

2、邻接表实现图的广度优先遍历（BFS）

```
void BFS(Graph G, int v){
   int visit[maxsize] = {0};
   int que[maxsize];
   int front = 0, rear = 0;
   visit[v] = 1;
   que[++rear] = v;
   while (front != rear){
      v = que[++front];
      printf("%d", v);
      Node p = G.adjlist[v].firstarc;
      for( ; p != NULL; p = p->nextarc)
         if (visit[p->adjvex] == 0){
            visit[p->adjvex] = 1;
            que[++rear] = p->adjvex;
         }
   }//p 的初始化最好写到 for 循环里面
} //时空复杂度分别为 O(V+E)和 O(V)
```

核心思想

1、初始化：为实现图的广度优先搜索，定义 visit 整型数组，初始化为 0，用于标记顶点是否被访问；同时创建数组 que 模拟队列。将起始顶点 v 标记为已访问，并加入队列

2、循环遍历逻辑：从队列取出队首顶点 v 并访问（打印顶点编号）。遍历顶点 v 的邻接表，对未访问的邻接顶点 p->adjvex，先标记为已访问，再将其入队

3、终止条件：当队列为空，表示图中所有可达顶点均已访问，BFS 结束。

3、邻接矩阵实现图的广度优先遍历（BFS）

```
void BFS(MGraph G, int v){
   int visit[maxsize] = {0};
   int que[maxsize];
   int front = 0, rear = 0;
   que[++rear] = v;
   visit[v] = 1; //本句写在入队前或入队后都可
   while (front != rear){
      v = que[++front];
      printf("%d", v);
      for (int i=0; i<G.numver; i++){
         if (G.Edge[v][i] == 1 && visit[i] == 0){
            que[++rear] = i;
            visit[i] = 1;
         }
      }
   }
} //时空复杂度分别为 O(V^2)和 O(V)
```

核心思想

1、初始化：同邻接表

2、循环遍历逻辑：从队列中取出队首顶点 v 并进行访问（打印顶点编号）。遍历图中所有顶点，对于与顶点 v 有边相连（即 G.Edge[v][i] == 1）且未被访问过（即 visit[i] == 0）的顶点 i，先将其标记为已访问，再将其入队

3、终止条件：当队列为空，表示图中所有可达顶点均已访问，BFS 结束。

4、设计算法，求无向连通图距顶点 v 最远的一个结点（即路径长度最大）

```
int BFS(Graph G, int v){
   int visit[maxsize] = {0};
   int que[maxsize];
   int front = 0, rear = 0;
   visit[v] = 1;
   que[++rear] = v;
   while (front != rear){
      v = que[++front];
      Node p = G.adjlist[v].firstarc;
      for( ; p != NULL; p = p->nextarc)
         if (visit[p->adjvex] == 0){
            visit[p->adjvex] = 1;
            que[++rear] = p->adjvex;
         }
   }
   return v;
} //时空复杂度分别为 O(V+E)和 O(V)
```

核心思想：

参考第二题，即从 v 顶点开始广度有限遍历，最后遍历到的结点一定是距离 v 最远的一个结点

5、邻接表实现图的深度优先遍历（DFS）

```
void DFS(Graph G, int v, int visit[]){
    printf("%d", v);
    visit[v] = 1;
    Node p = G.adjlist[v].firstarc;
    for( ; p != NULL; p = p->nextarc)
        if (visit[p->adjvex] == 0)
            DFS(G, p->adjvex, visit);
} //时空复杂度分别为 O(V+E)和 O(V)
```

6、邻接矩阵实现图的深度优先遍历（DFS）

```
void DFS(MGraph G, int v,int visit[]){
    printf("%d", v);
    visit[v] = 1;
    for (int i = 0; i < G.numver; i++)
        if (G.Edge[v][i] == 1 && visit[i] == 0)
            DFS(G, i, visit);
} //时空复杂度分别为 O(V²)和 O(V)
```

5、6 两题的核心思想：

1、访问与标记：先访问顶点 v 并打印其值，然后将 visit[v]置为 1，标记该顶点已访问

2、邻接顶点遍历：获取顶点 v 的邻接表，用指针 p 遍历。若邻接顶点未被访问，则递归调用 DFS 继续搜索

3、终止条件：当一个顶点的所有邻接顶点都已访问，或无邻接顶点时，递归逐层返回，直至遍历完所有可达顶点

7、有向图采用邻接表存储，设计算法判断顶点 Vi 和顶点 Vj 之间是否存在路径

```
void f1(Graph G, int i, int j, int visit[], bool* res){
    if (i == j)
        *res = true;
    visit[i] = 1;
    Node p = G.adjlist[i].firstarc;
    for( ; p != NULL; p = p->nextarc)
        if (visit[p->adjvex] == 0))
            f1(G, p->adjvex, j, visit, res);
}//法一：DFS
```

核心思想：

1、目标与初始判断：此函数 f1 旨在利用 DFS 判断图 G 中顶点 i 到顶点 j 是否存在路径。进入函数后，首先检查 i 是否等于 j，若相等，说明已找到目标路径，将 *res 设为 true

2、标记当前顶点：将当前顶点 i 标记为已访问，即把 visit[i]置为 1，防止后续重复访问该顶点。

3、邻接顶点遍历：获取顶点 i 的邻接表，用指针 p 遍历。对于每个邻接顶点，若其未被访问，则递归调用 f1 函数，从该邻接顶点继续进行深度搜索，尝试寻找通向顶点 j 的路径

4、终止条件：当一个顶点的所有邻接顶点都已被访问，递归调用会逐层返回，直至完成对所有可达顶点的搜索。最终*res 的值表明顶点 i 到顶点 j 是否存在路径

```
bool f2(Graph G, int i, int j){
        int visit[maxsize] = {0};
        int que[maxsize];
        int front = 0, rear = 0;
        que[rear++] = i;
        visit[i] = 1;
        while (front != rear){
            i = que[front++];
            if (i == j)
                return true;
            Node p = G.adjlist[i].firstarc;
            for ( ; p != NULL; p = p->nextarc){
                if(visit[p->adjvex] == 0){
                    que[rear++] = p->adjvex;
                    visit[p->adjvex] = 1;
                }
            }
        }
        return false;
}//法二：BFS
```

1、初始化：创建 visit 数组，创建 que 数组模拟队列，将起始顶点 i 入队，同时标记为已访问

2、队列遍历：每次从队列取出队首顶点，若该顶点为目标顶点 j，说明存在路径

3、邻接顶点处理：获取当前顶点的邻接表，遍历邻接顶点。对于未访问的邻接顶点，将其入队并标记为已访问，以便后续搜索

4、结果判定：若循环结束都未找到目标顶点 j，则返回 false，表示不存在顶点 i 到顶点 j 的路径

8、在有向图中，如果顶点 r 到图中所有顶点都存在路径，则称 r 为图的根结点。编写代码输出有向图中所有根结点。

```
void func(Graph G){
    for(int i = 0; i < G.numver; i++){
        int visit[maxsize] = {0};
        int flag = 0;
        DFS(G, i, visit);
        for (int j = 0; j < G.numver; j++){
            if (visit[j] == 0)
                flag = 1;
        }
        if (flag == 0)
            printf("%d", i);
    }
} //时空复杂度分别为 O(V*(V+E))和 O(V)
```

核心思想：

1、遍历顶点：对图 G 中的每个顶点 i 进行遍历，为每个顶点执行可达性检查

2、可达性检查：每次检查前，初始化 visit 数组和 flag 变量。调用 DFS 函数从顶点 i 开始搜索，之后遍历 visit 数组，若有未访问顶点，将 flag 置为 1

3、输出结果：若 flag 为 0，意味着从顶点 i 出发能到达图中所有顶点，打印该顶点编号

9、求无向图的连通分量个数

```
int func(Graph G){
    int visit[maxsize] = {0};
    int count = 0;
    for (int i = 0; i < G.numver; ++i)
        if (visit[i] == 0){
            DFS(G, i, visit);
            count++;
        }
    return count;
} //时空复杂度分别为 O(V*(V+E))和 O(V)
```

核心思想：

1、初始化访问标记数组：使用一个数组 visit[] 来标记每个顶点是否被访问过，初始时全部设为未访问（0），为后续遍历做准备

2、遍历每个顶点，启动 DFS：遍历图中所有顶点 i，如果某个顶点尚未被访问，则从该顶点出发进行一次深度优先搜索（DFS），标记该连通块内所有可达节

3、统计连通分量数量：每当从一个未访问的顶点启动 DFS，就说明发现了一个新的连通分量，使用变量 count 累加连通分量的个数，最终返回

1、在有序表中二分查找值为 key 的元素

```
int binsearch(Sqlist L, int key){
    int low = 0, high = L.length - 1, mid;
    while (low <= high){
        mid = (low + high) / 2;
        if (L.data[mid] == key)
            return mid + 1;
        else if (L.data[mid] < key)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;
} //非递归，时空复杂度分别为 O(logN)和 O(1)
```

核心思想：

1、设定查找范围的边界指针：初始化两个指针 low 和 high 分别指向顺序表的起始位置和末尾位置，表示当前查找的区间范围

2、不断缩小查找区间：在 low <= high 的条件下，循环执行：计算中间位置 mid，判断中间元素与目标值 key 的大小关系，从而选择继续在左半区或右半区查找，并更新 low 或 high

3、返回结果或未找到：如果中间值等于 key，返回其逻辑位置（mid + 1）；如果查找结束仍未匹配到目标值，说明查找失败，返回 -1

```c
int func(Sqlist L, int key, int low, int high){
   if (low > high)
      return -1;
   int mid = (low + high) / 2;
   if (L.data[mid] == key)
      return mid + 1;
   else if (L.data[mid] < key)
      return func(L, key, mid + 1, high);
   else
      return func(L, key, low, mid - 1);
} //递归，时空复杂度分别为 O(logN)和 O(logN)
```

由于每次递归调用消耗的栈空间与递归深度成正比，因此空间复杂度也是 O(logN)

核心思想：

1、递归终止条件：若 low 大于 high，意味着查找范围为空，函数返回-1 表示未找到目标值 key

2、中间元素判断：若 L.data[mid] 等于 key，则找到目标，返回 mid + 1 作为目标位置

3、递归查找：若 L.data[mid]小于 key，目标在右半部分，递归调用函数在 mid + 1 到 high 范围查找；否则，目标在左半部分，递归调用函数在 low 到 mid - 1 范围查找

2、判断给定二叉树是否是二叉(搜索)排序树

```c
void inprint(BiTree T, int res[], int *index) {
   if (T != NULL) {
      inprint(T->lchild, res, index);
      res[(*index)++] = T->data;
      inprint(T->rchild, res, index);
   }
}
bool isture(int res[], int index){
   for(int i = 0; i < index-1; i++)
      if(res[i] >= res[i+1])
        return false;
   return true;
} //法一，时空复杂度分别为 O(N)和 O(N)
```

核心思想：

1、中序遍历二叉树：inprint 函数对二叉树 T 进行中序遍历，将中序遍历结果顺序存入数组。

2、检查数组有序性：isture 函数用于检查 res 数组是否严格升序。遍历数组，若发现相邻元素 res[i]大于等于 res[i+1]，则返回 false，表明数组无序；若遍历完都未发现此类情况，返回 true

3、功能用途：结合两个函数，若 isture 对 inprint 存储结果的数组返回 true，则该二叉树是二叉搜索树

```c
bool func(BiTree T,int low,int high){
      if(T == NULL)
            return true;
      if(T->data <= low || T->data >= high)
            return false;
      bool left = func(T->lchild, low, T->data);
      bool right = func(T->rchild, T->data, high);
      return left && right;
} //法二，时空复杂度分别为 O(N)和 O(N)
```

核心思想：

1、空树判断：函数 func 用于判断二叉树 T 是否为二叉搜索树（BST）。若树为空，直接返回 true，因为空树可视为二叉搜索树

2、节点值范围检查：对于非空节点，检查其值是否在 low 和 high 规定的范围内。若节点值小于等于 low 或者大于等于 high，说明不满足二叉搜索树性质，返回 false

3、递归判断子树：对当前节点的左子树，递归调用 func 函数。更新范围为 low 到当前节点值；对右子树，递归调用时更新范围为当前节点值到 high。最终返回左右子树判断结果的逻辑与，只有左右子树都是二叉搜索树，整棵树才是二叉搜索树

3、寻找二叉排序树中最大值和最小值结点

```
BiTree Min(BiTree bt){
    while (bt->lchild)
        bt = bt->lchild;
    return bt;
}
BiTree Max(BiTree bt){
    while (bt->rchild)
        bt = bt->rchild;
    return bt;
} //时空复杂度分别为 O(logN)和 O(1)
```

核心思想：

1、查找最小节点：Min 函数用于在二叉搜索树中查找最小节点。由于左子树的节点值小于根节点值，所以不断沿着左子树向下遍历，直到找到没有左子节点的节点，此节点即为最小节点

2、查找最大节点：Max 函数用于在二叉搜索树中查找最大节点。由于右子树的节点值大于根节点值，因此持续沿着右子树向下遍历，直至找到没有右子节点的节点，该节点就是最大节点

4、求出值为 key 的结点在二叉排序树的层次

```
int level(BiTree bt, int key){
    int n = 1;
    while (bt != NULL)
        if (bt->data == key)
            return n;
        else if (bt->data < key){
            bt = bt->rchild;
            n++;
        }
        else{
            bt = bt->lchild;
            n++;
        }
    return -1;
} //时空复杂度分别为 O(logN)和 O(1)
```

核心思想：

1、初始化：函数开始时，将层次计数 n 设为 1，表明从根节点开始查找值为 key 的节点

2、查找流程：借助 while 循环遍历二叉搜索树。若当前节点值等于 key，返回当前层次 n；若小于 key，转向右子树并将 n 加 1；若大于 key，转向左子树并将 n 加 1

3、结果判定：若遍历完树仍未找到 key，返回 -1，否则返回节点所在层次

1、直接插入排序

```
void InsertSort(int A[], int n){
    int j;
    for (int i = 2; i <= n; i++){
        A[0] = A[i];
        for (j = i - 1; A[0] < A[j]; j--)
            A[j+1] = A[j];
        A[j+1] = A[0];
    }
} //时空复杂度分别为 O(N^2)和 O(1)
```

核心思想：

1、逐步插入元素：从第二个元素开始，依次将当前元素与已排序部分的元素进行比较，寻找合适的位置插入

2、元素向右移动：在内层循环中，当当前元素比前一个元素小（即需要插入），就将较大的元素向右移动，腾出空位

3、插入当前元素：将当前元素插入到找到的合适位置，保证前面部分的数组始终保持有序，继续处理下一个元素，直到整个数组有序

2、折半插入排序

```
void BinInsert(int A[], int n){
   int j, low, high, mid;
   for (int i = 2; i <= n; i++){
      A[0] = A[i];
      low = 1;
      high = i-1;
      while (low <= high){
         mid = (low + high) / 2;
         if (A[mid] > A[0])
            high = mid - 1;
         else
            low = mid + 1;
      }
      for (j = i-1; j >= low; j--)
         A[j+1] = A[j];
      A[low] = A[0];
   }
}//时空复杂度分别为 O(N²)和 O(1)
```

核心思想：

1、通过二分法查找插入位置：在每一轮外循环中，使用二分查找在已排序部分找到合适的插入位置 low，降低插入时的查找时间复杂度

2、将元素插入正确位置：将当前元素 A[i] 临时存储在 A[0]，然后通过内层循环将已排序部分的元素向后移动，为新元素腾出空间

3、插入新元素：将当前元素插入到合适的位置，保持数组的有序性，并继续处理下一个元素，直至数组完全排序

3、选择排序

```
void SelectSort(int A[], int n){
   for (int i = 0; i < n; i++){
      int pos = i;
      for (int j = i + 1; j < n; j++)
         if (A[pos] > A[j])
            pos = j;
      int temp = A[i];
      A[i] = A[pos];
      A[pos] = temp;
   }
}//时空复杂度分别为 O(N²)和 O(1)
```

核心思想：

1、找到最小元素并交换：在每一轮排序中，从当前未排序部分中找到最小元素，并将它与当前元素交换。这样，每一轮之后，当前元素就排好了一个位置

2、逐步缩小未排序部分：外层循环控制排序的轮数，每次外层循环后，未排序部分逐渐缩小，确保每次选择最小的元素并放到已排序部分的末尾

4、冒泡排序

```
void swap(int *x, int *y){
      int temp = *x;    *x = *y;    *y = temp;
}
void BubbleSort(int A[], int n){
   for (int i = n-1; i > 0; i--){
      int flag = 0;
      for (int j = 1; j <= i; j++)
         if (A[j-1] > A[j]){
            swap (&A[j-1], &A[j]);
            flag = 1;
         }
      if (flag == 0)
         return;
   }
}//时空复杂度分别为 O(N²)和 O(1)
```

核心思想：

1、比较并交换相邻元素：对数组中的相邻元素进行逐一比较，如果前一个元素比后一个元素大，则交换它们的位置。通过不断的交换，较大的元素会逐渐被移动到数组的末尾

2、逐步缩小排序范围：每次外层循环结束后，当前最大元素已经排好位置，因此内层循环的排序范围逐步缩小，减少无效的比较，提升效率

3、提前结束排序的优化：引入标志 flag，用于检测当前轮次是否发生了交换。如果没有交换发生，说明数组已经是有序的，可以提前终止排序，避免不必要的操作，从而提高算法效率

5、快速排序

```
int Partition(int A[], int low, int high){
    int pivot = A[low];
    while (low < high){
        while (low < high && A[high] >= pivot)
            high--;
        A[low] = A[high];
        while (low < high && A[low] < pivot)
            low++;
        A[high] = A[low];
    }
    A[low] = pivot;
    return low;
}

void QuickSort(int A[], int low, int high){
    if (low < high){
        int pos = Partition(A, low, high);
        QuickSort(A, low, pos - 1);
        QuickSort(A, pos + 1, high);
    }
} //时空复杂度分别为 O(NlogN)和 O(logN)
```

核心思想：

1、选择枢轴元素：在 Partition 函数中，选择数组的第一个元素作为枢轴（pivot）。然后通过两个指针（low 和 high）将数组分割成两部分：左边部分所有元素小于枢轴，右边部分所有元素大于等于枢轴

2、分割操作：通过 low 和 high 两个指针，从两端向中间扫描，直到找到一个不满足条件的元素，交换这两个元素的位置。继续扫描，直到指针相遇，完成一次分区操作

3、递归排序：在 QuickSort 函数中，通过递归调用分别对枢轴左侧和右侧的子数组进行排序，不断细分，最终实现整个数组的排序