软件专硕模拟卷（二）答案

《数据结构》（50 分）

一、简答题（共20分）

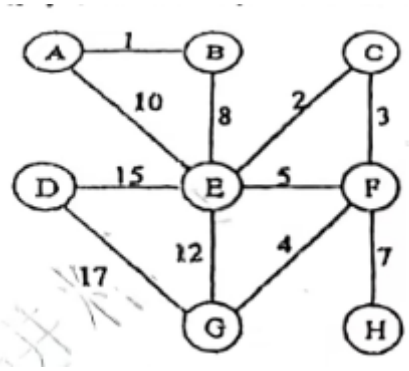
1.给出中缀表达式 `{(a-b)/c-d*[(e+f)-g]+h}/i` 的后缀表达式（5分）

答案： `ab-c/ef+g-*d-h+i/`

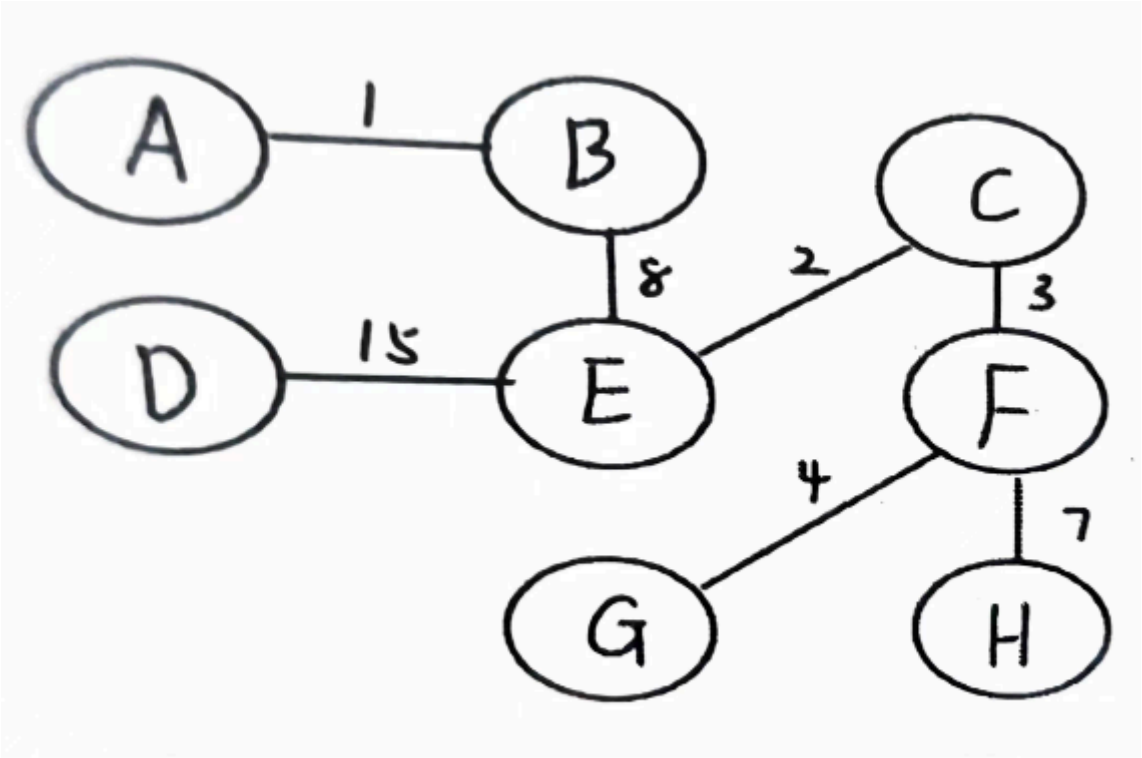
2. 一组记录的关键字为 `{58，81，15， 69， 32， 47， 85， 26， 70}` 给出快速排序（分划交换）排序的过程（5分）

答案：

3. 请画出下图的最小支撑树（5分）



答案:



4. 已知散列表的地址空间为`A[0...10]`，散列函数为`H(K)=k mod 11`，采用线性探测法处理冲突。将下列数据`{24， 15， 38， 46， 79， 82， 52， 39， 85， 143， 231}`依次插入到散列表当中。请写出散列表的结果，并计算在等概率下，查找成功的平均探查次数。（5分）

答案:

下标	0	1	2	3	4	5	6	7	8	9	10
A[0...10]	143	231	24	46	15	36	79	82	52	39	85
探查次数	1	2	1	2	1	1	5	3	1	4	3

$ASL = (1+2+1+2+1+1+5+3+1+4+3)/11 = 24/11$

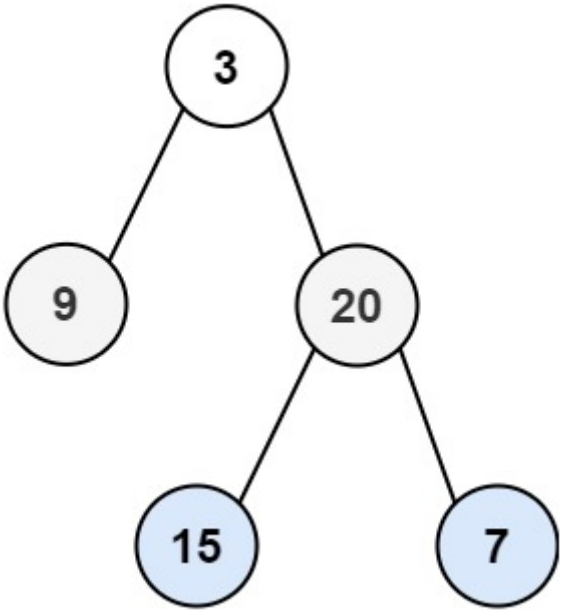
二、算法题（共30分）

答题要求：

- **（1）算法书写可采用`C`，`C++`，`Java`，`ADL`等语言，使用何种语言书写要注明。**
- **（2）在算法开始出必须用自然语言书写注释，说明算法的基本思路，以及使用了那些数据结构。**
- **（3）算法的关键步骤要写注释说明其目的。**

1. 二叉树交替层次遍历算法(即先从左往右，再从右往左进行下一层遍历，以此类推，层与层之间交替进行)（10分）

例如：



该二叉树输出结果为：3,20,9,15,7

算法思路：

> **初始化**：
>
> - 用两个栈 `stack1` 和 `stack2` 分别存储当前层和下一层的节点。
> - 将根节点压入 `stack1`。
>
> **交替遍历**：

- 当

  `stack1`

  或

  `stack2`

  不为空时，循环执行以下操作：
  - 如果当前处理 `stack1`，从左到右遍历当前层，节点的子节点从左到右压入 `stack2`。
  - 如果当前处理 `stack2`，从右到左遍历当前层，节点的子节点从右到左压入 `stack1`。
- 输出当前层的节点值。

结束条件：

- 当 `stack1` 和 `stack2` 都为空时，遍历结束。

答案：

```
#define MAXSIZE 100 // 栈的最大容量

void zigzagTraversal(BiTree root) {
    if (root == NULL) return;

    BiTree stack1[MAXSIZE]; // 当前层栈
    BiTree stack2[MAXSIZE]; // 下一层栈
    int top1 = -1, top2 = -1; // 栈顶指针

    stack1[++top1] = root; // 根节点入栈 stack1
    int leftToRight = 1;   // 方向标志：1 表示从左到右，0 表示从右到左

    while (top1 != -1 || top2 != -1) {
        if (leftToRight) {
            // 从左到右处理 stack1
            while (top1 != -1) {
                BiTree node = stack1[top1--]; // 出栈
                printf("%d ", node->data);

                // 左子树先入栈，右子树后入栈（确保下一层从右到左）
                if (node->lchild) stack2[++top2] = node->lchild;
                if (node->rchild) stack2[++top2] = node->rchild;
            }
        } else {
            // 从右到左处理 stack2
            while (top2 != -1) {
                BiTree node = stack2[top2--]; // 出栈
                printf("%d ", node->data);

                // 右子树先入栈，左子树后入栈（确保下一层从左到右）
                if (node->rchild) stack1[++top1] = node->rchild;
                if (node->lchild) stack1[++top1] = node->lchild;
            }
        }
        leftToRight = !leftToRight; // 切换方向
```

```
    }
}
```

2. 自由树（即无环连通图）T=(V,E)的直径是所有顶点之间最短路径的最大值，请设计一个时间复杂度尽可能低的算法求T的直径。并分析算法的时间复杂度（20分）

算法思路：

> - 先从任意一个顶点找到该顶点最短路径中最长的一个，这个点为直径的某个端点
> - 然后再从这个顶点出发，再进行一次BFS,求出距离该顶点最短路径最长的一个，该顶点为直径的另一端
> - 返回两个顶点的路径长度，就是自由树的直径

答案：

```
typedef struct ArcNode{     //边节点
    int adjvex;
    struct ArcNode *next;
}ArcNode;

typedef struct VNode{       //顶点节点
    int data;
    struct ArcNode *firstarc;
}VNode;

typedef struct AGraph{      //领接表
    VNode adjlist[maxsize];
    int vexnum, edgenum;
}AGraph;

int MaxLenBFS(AGraph *G, int v, int dist[]){
    int visited[maxsize] = {0};
    int queue[maxsize];
    int front = -1, rear = -1,i,k,temp,max=0;
    ArcNode *p = G->adjlist[v].firstarc;

    for(i = 0; i < G->vexnum; i++){
        dist[i] = -1;
    }

    queue[++rear] = v;
    visited[v] = 1;
    dist[v] = 0;

    while(rear != front){
        k = queue[++front];
        p = G->adjlist[k].firstarc;
        while(p!=NULL){
            temp = p->adjvex;
            if(visited[temp]==0){
                queue[++rear] = temp;
                visited[temp] = 1;
                dist[temp] = dist[k] + 1;
```

```
            }
            p = p->next;
        }
    }

    for(i=0; i < G->vexnum; i++){
        if(dist[i]>dist[max]){
            max = i;
        }
    }
    return max;          //返回端点
}

int Diameter(AGraph *G){
    int dist[maxsize];
    int first = MaxLenBFS(G,0,dist);
    int last = MaxLenBFS(G,first,dist);
    printf("直径为：%d",dist[last]);
    return dist[last];                    //返回直径长度
}
```

《高级语言程序设计》（100分）

1.编写函数计算滑动窗口的最大值。函数对给定的一个数组和滑动窗口的大小，返回所有滑动窗口里数值的最大值。如果输入数组 `{1，3，-1，-3，5，3，6，7}` 及滑动窗口的大小为3，那么一共存在6个滑动窗口，它们的最大值分别为 `{3，3，5，5，6，7}`。(25分)

```
输入: nums = [1,3,-1,-3,5,3,6,7], k = 3
输出: [3,3,5,5,6,7]
解释:
滑动窗口的位置                最大值
---------------               -----
[1  3  -1] -3  5  3  6  7       3
 1 [3  -1  -3] 5  3  6  7       3
 1  3 [-1  -3  5] 3  6  7       5
 1  3  -1 [-3  5  3] 6  7       5
 1  3  -1  -3 [5  3  6] 7       6
 1  3  -1  -3  5 [3  6  7]      7
```

思路：

> 需要求出的是滑动窗口的最大值，如果当前的滑动窗口中有两个下标 i 和 j，其中 i 在 j 的左侧（i<j），并且 i 对应的元素不大于 j 对应的元素（nums[i]≤nums[j]），那么会发生什么呢？
>
> 当滑动窗口向右移动时，只要 i 还在窗口中，那么 j 一定也还在窗口中，这是 i 在 j 的左侧所保证的。因此，由于 nums[j] 的存在，nums[i] 一定不会是滑动窗口中的最大值了，我们可以将 nums[i] 永久地移除。
>
> 因此我们可以使用一个队列存储所有还没有被移除的下标。在队列中，这些下标按照从小到大的顺序被存储，并且它们在数组 nums 中对应的值是严格单调递减的。因为如果队列中有两个相邻的下标，它们对应的值相等或者递增，那么令前者为 i，后者为 j，就对应了上面所说的情况，即 nums[i] 会被移除，这就产生了矛盾。
>
> 当滑动窗口向右移动时，我们需要把一个新的元素放入队列中。为了保持队列的性质，我们会不断地将新的元素与队尾的元素相比较，如果前者大于等于后者，那么队尾的元素就可以被永久地移除，我们将其弹出队列。我们需要不断地进行此项操作，直到队列为空或者新的元素小于队尾的元素。
>
> 由于队列中下标对应的元素是严格单调递减的，因此此时队首下标对应的元素就是滑动窗口中的最大值。但与方法一中相同的是，此时的最大值可能在滑动窗口左边界的左侧，并且随着窗口向右移动，它永远不可能出现在滑动窗口中了。因此我们还需要不断从队首弹出元素，直到队首元素在窗口中为止。
>
> 为了可以同时弹出队首和队尾的元素，我们需要使用双端队列。满足这种单调性的双端队列一般称作「单调队列」。

答案：

```c
int* maxSlidingWindow(int* nums, int numsSize, int k, int* returnSize) {
    int q[numsSize];
    int left = 0, right = 0;
    for (int i = 0; i < k; ++i) {
        while (left < right && nums[i] >= nums[q[right - 1]]) {
            right--;
        }
        q[right++] = i;
    }
    *returnSize = 0;
    int* ans = malloc(sizeof(int) * (numsSize - k + 1));
    ans[(*returnSize)++] = nums[q[left]];
    for (int i = k; i < numsSize; ++i) {
        while (left < right && nums[i] >= nums[q[right - 1]]) {
            right--;
        }
        q[right++] = i;
        while (q[left] <= i - k) {
            left++;
        }
        ans[(*returnSize)++] = nums[q[left]];
    }
    return ans;
}
```

2.编写程序实现：若一个数字的各个数位阶乘之和等于它本身，则称该数字为一个阶乘数，如（145=1！+4！+5！），输出除1,2之外，小于2000000的所有阶乘数。（25分）

思路：

> **阶乘计算（`factorial`）**：计算数字 0-90-90-9 的阶乘，并将其存储在数组 `factorials[10]` 中，减少重复计算。
>
> **判断阶乘数（`is_factorion`）**：将输入数字的每一位分离，查找其阶乘值，并累加。如果累加和等于原数字，则该数字是阶乘数。
>
> **主程序**：
>
> - 预计算 0 到 9 的阶乘。
> - 从 10 开始检查每个数字是否是阶乘数。
> - 输出符合条件的阶乘数。

答案：

```c
#include <stdio.h>

// 计算阶乘的函数
int factorial(int n) {
    int result = 1;
    for (int i = 2; i <= n; i++) {
        result *= i;
    }
    return result;
}

// 判断一个数字是否是阶乘数
int is_factorion(int num, int factorials[]) {
    int sum = 0;
    int temp = num;

    while (temp > 0) {
        sum += factorials[temp % 10]; // 加上当前数位的阶乘
        temp /= 10;
    }

    return sum == num;
}
int main() {
    // 预计算 0-9 的阶乘
    int factorials[10];
    for (int i = 0; i <= 9; i++) {
        factorials[i] = factorial(i);
    }

    // 查找小于 2,000,000 的阶乘数
    printf("小于 2000000 的阶乘数为：\n");
    for (int i = 10; i < 2000000; i++) { // 排除 1 和 2
        if (is_factorion(i, factorials)) {
```

```
            printf("%d\n", i);
        }
    }

    return 0;
}
```

3. {}、[]、()又称大括号、中括号和小括号；括号匹配除了必要的同类型括号左右成对儿且不交叉的规定；还增加如下规则：大括号内能包含大、中、小括号，中括号内只能包含中、小括号，小括号内只能包含小括号。请编写程序判断从键盘输入的以#结束的字符串（可能包含空格、回车、换行和制表等符号，字符串长度不限），其中的{}、[]、()是否按照上述规则匹配。如果匹配成功，则输出提示信息MATCHED和匹配的括号对的数目；如果匹配不成功，则输出提示信息ERR，以及在出现第一个错误前、已经匹配的括号对的数目（提示信息和括号对数目的中间，以一个西文空格间隔）。匹配括号对的数目不超过int型可表示范围，且括号嵌套层数不超过100层。（25分）

思路：

> - 括号匹配的扩展
> - 当出现{前一个必须是{
> - 当出现[，前面如果是(，也不符合规则
> - 其他的就是正常的左括号入栈，右括号出栈进行匹配

答案：

```
#include<stdio.h>
char c, s[100100]; // 定义字符变量c和字符数组s，用于存储括号序列
int top=0, ans; // 定义栈顶指针top和标记变量ans

int main() {
    int count = 0; // 定义计数器变量count，用于统计匹配的括号对数
    ans = 1; // 初始化ans为1，表示序列默认为有效
    while (1) { // 无限循环，直到遇到break语句
        scanf("%c", &c); // 读取一个字符
        if (c == '#') break; // 如果读取到#，结束循环
        if (c == '(' || c == '[' || c == '{') { // 如果是左括号
            s[++top] = c; // 将左括号压入栈中
            // 检查特定规则
            if (top != 1) {
                if (c == '{') { // 如果是{，前一个必须是{
                    if (s[top - 1] != '{') {
                        ans = 0; // 如果不满足规则，标记为无效
                        break; // 结束循环
                    }
```

```
                }
                if(c == '[') { // 如果是[，前一个不能是(
                    if (s[top - 1] == '(') {
                        ans = 0; // 如果不满足规则，标记为无效
                        break; // 结束循环
                    }
                }
            }
        }
        // 以下是右括号的匹配逻辑
        if (c == ')') { // 如果是右括号)
            if (top && s[top] == '(') { // 如果栈顶是左括号(
                top--; // 弹出栈顶元素
                count++; // 匹配对数加1
            } else {
                ans = 0; // 如果不匹配，标记为无效
                break; // 结束循环
            }
        }
        // 类似逻辑应用于其他右括号
        if (c == ']') {
            if (top && s[top] == '[') {
                top--;
                count++;
            } else {
                ans = 0;
                break;
            }
        }
        if (c == '}') {
            if (top && s[top] == '{') {
                top--;
                count++;
            } else {
                ans = 0;
                break;
            }
        }
    }
    // 根据ans的值输出结果
    if (ans==1) {
        printf("MATCHED"); // 如果序列有效，输出MATCHED
        printf(" %d", count); // 输出匹配的括号对数
    }
    else {
        printf("ERR"); // 如果序列无效，输出ERR
        printf(" %d", count); // 输出匹配的括号对数
    }
}
```

4. 构造一个表示教师的结构体（包含三个字段，姓名，性别，年龄），编写函数，读入M个教师的信息，存入一个结构体数组中，如下所示：（25分）

张三 男（0） 45

李四 男（0） 24

...

赵九 女（1） 32

```c
#include <stdio.h>
#include <string.h>

// 定义教师结构体
typedef struct {
    char name[50]; // 教师姓名
    int gender;    // 性别：0 表示男，1 表示女
    int age;       // 教师年龄
} Teacher;

// 函数：读取教师信息
void readTeachers(Teacher teachers[], int count) {
    for (int i = 0; i < count; i++) {
        printf("请输入第 %d 位教师的信息 (姓名 性别[0男/1女] 年龄):\n", i + 1);
        scanf("%s %d %d", teachers[i].name, &teachers[i].gender,
&teachers[i].age);
    }
}

int main() {
    int M; // 教师数量
    printf("请输入教师的数量 M: ");
    scanf("%d", &M);

    if (M <= 0) {
        printf("教师数量必须为正数。\n");
        return 1;
    }

    Teacher teachers[M]; // 定义结构体数组
    readTeachers(teachers, M); // 读入教师信息

    return 0;
}
```