软件专硕模拟卷（一）答案

《数据结构》（50 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. 已知线性表（`a1`，`a2`，`a3`，...，`an`）存放在一维数组A中。试设计一个在时间和空间两方面都尽可能高效的算法，将所有奇数号元素移到所有偶数号元素前，并且不得改变奇数号（或偶数号）元素之间的相对顺序。（15分）

算法思路：

> 为了保持元素的相对顺序，同时避免使用过多的额外空间，可以采用以下方法：
>
> 1. 使用单个辅助数组：
>   - 遍历原数组，先将奇数号（即索引为偶数的元素）依次放入辅助数组的前半部分。
>   - 再次遍历原数组，将偶数号（即索引为奇数的元素）放入辅助数组的剩余部分。
> 2. 写回原数组：
>   - 将辅助数组中的内容覆盖回原数组，完成重排。
>
> 该方法只需要一个额外的辅助数组，空间复杂度为 O(n)，时间复杂度为 O(n) 。

答案：

```
void rearrangeArray(int A[], int n) {
    // 创建辅助数组，用于存储重新排列的结果
    int temp[n];
    int oddIndex = 0;  // 奇数号元素的写入位置
    int evenIndex = 0; // 偶数号元素的写入位置起点

    // 第一遍遍历，将奇数号元素存入辅助数组
    for (int i = 0; i < n; i += 2) {
        temp[oddIndex++] = A[i];
    }

    // 第二遍遍历，将偶数号元素存入辅助数组
    evenIndex = oddIndex; // 偶数号元素从奇数号结束的位置开始写入
    for (int i = 1; i < n; i += 2) {
        temp[evenIndex++] = A[i];
    }

    // 将辅助数组内容写回原数组
    for (int i = 0; i < n; i++) {
        A[i] = temp[i];
```

```
    }
}
```

2. 把二叉查找树转换为双向循环链表（递增），要求不创建新的节点，只能由原来的节点转化。（15分）

二叉树结构体定义为：

```
typedef struct BTNode {
    int data; // 数据域
    struct BTNode *lchild, *rchild; // 左右子树指针
} BTNode, *BiTree;
```

算法思路：

> 将二叉查找树转换为双向循环链表的问题要求使用原树的节点，且链表中的节点需按照递增顺序排列。可以通过**中序遍历**实现这一目标，因为中序遍历的结果就是二叉查找树的递增序列。
>
> #### 步骤：
>
> 1. 递归中序遍历:
>   - 通过中序遍历，按照左子树 -> 根节点 -> 右子树的顺序访问节点。
> 2. 双向链表链接:
>   - 在遍历过程中，维护一个指针 `prev` 指向当前链表的最后一个节点。
>   - 将当前节点的 `lchild` 指针设置为 `prev`（即前驱），`prev` 的 `rchild` 指针设置为当前节点（即后继）。
> 3. 形成循环:
>   - 在遍历完成后，链表的首节点和尾节点相连，形成循环链表。

答案：

```
// 递归函数：将二叉查找树转换为双向循环链表
void convertToDoublyLinkedList(BiTree root, BiTree head, BiTree prev) {
    if (root == NULL) return;

    convertToDoublyLinkedList(root->lchild, head, prev);// 递归处理左子树

    // 处理当前节点
    if (prev == NULL) {
        // 如果前驱为空，说明当前节点是链表的第一个节点
        head = root;
    } else {
        // 前驱的右指针指向当前节点，当前节点的左指针指向前驱
        prev->rchild = root;
        root->lchild = prev;
    }
    // 更新前驱为当前节点
    prev = root;
```

```
    // 递归处理右子树
    convertToDoublyLinkedList(root->rchild, head, prev);
}

// 主函数：二叉查找树转换为双向循环链表
BiTree BSTToCircularDoublyLinkedList(BiTree root) {
    if (root == NULL) return NULL;

    BiTree head = NULL; // 链表的头节点
    BiTree prev = NULL; // 当前链表的最后一个节点

    // 调用递归函数进行中序遍历并建立双向链表
    convertToDoublyLinkedList(root, head, prev);

    // 链表首尾相连，形成循环
    head->lchild = prev;
    prev->rchild = head;

    return head;
}
```

3. 有向加权图，设计一个算法判断该图中是否存在起点为v，长度为`len`的路径，并说明时间复杂度和空间复杂度。（20分）

算法思路：

1. **数据结构**
   - `ArcNode`：用于表示边的邻接节点，包含目标节点 `adjvex` 和边的权重 `weight`。
   - `VNode`：每个顶点的信息，包括数据 `data` 和第一条边的指针 `firstarc`。
   - `ALGraph`：图的整体结构，包含顶点表 `vertices`、顶点数 `vexnum` 和边数 `arcnum`。
2. **DFS 函数**
   - 输入参数：
     - `G`：图的邻接表表示。
     - `v`：当前访问的顶点。
     - `len`：目标路径长度。
     - `currentLen`：当前路径长度。
     - `visited`：标记已访问的顶点，防止重复访问形成环。
   - 核心逻辑：
     - 如果当前路径长度等于 `len`，直接返回 `true`。
     - 如果当前路径长度超过 `len`，剪枝返回 `false`。
     - 否则，遍历所有邻接节点，递归探索可能的路径。

- **回溯**：递归返回时，将 `visited[v]` 重置为 `false`，使得其他路径可以再次访问该顶点。

3. **主函数 `hasPathOfLength`**
   - 调用 DFS，从起点 `v` 开始进行深度优先搜索。
   - 返回是否存在满足条件的路径。

答案：

```
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>

// 图的邻接表定义（严格按照王道定义）
typedef struct ArcNode {
    int adjvex;                 // 该弧指向的顶点
    int weight;                 // 边的权值
    struct ArcNode *nextarc;    // 指向下一条弧的指针
} ArcNode;

typedef struct VNode {
    int data;                   // 顶点数据
    ArcNode *firstarc;          // 指向第一条依附于该顶点的弧
} VNode, AdjList[100];

typedef struct {
    AdjList vertices;           // 邻接表
    int vexnum, arcnum;         // 图的顶点数和弧数
} ALGraph;

// 深度优先搜索函数
bool DFS(ALGraph *G, int v, int len, int currentLen, bool *visited) {
    // 如果当前路径长度正好为 len，则找到路径
    if (currentLen == len) {
        return true;
    }

    // 如果当前路径长度超过 len，剪枝
    if (currentLen > len) {
        return false;
    }

    // 标记当前顶点已访问
    visited[v] = true;

    // 遍历当前顶点的所有邻接节点
    ArcNode *p = G->vertices[v].firstarc;
    while (p != NULL) {
        int w = p->adjvex;
        if (!visited[w]) {  // 避免重复访问
            // 递归探索以 w 为起点的路径
            if (DFS(G, w, len, currentLen + p->weight, visited)) {
                return true;  // 找到路径则立即返回
            }
        }
```

```
            p = p->nextarc;  // 考虑下一个邻接节点
        }

        // 回溯，取消当前顶点的访问状态
        visited[v] = false;

        return false;  // 未找到路径
}

// 判断是否存在起点为 v，长度为 len 的路径
bool hasPathOfLength(ALGraph *G, int v, int len) {
        // 辅助数组，记录顶点是否已访问
        bool visited[100] = {false};

        // 从起点 v 开始深度优先搜索
        return DFS(G, v, len, 0, visited);
}
```

《高级语言程序设计》（100分）

1.求sinx近似值(25分)

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots$$编写程序，求$sinx$的近似值，要求误差小于10^{-8}

算法思路：

1. 根据$sin(x)$的展开式：$sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$.
2. 每一项可以通过上一项递推得到：$\text{term}_{n+1} = -\text{term}_n \times \frac{x^2}{(2n+2)(2n+3)}$
3. 初始项设置为x，逐项累加，直到当前项的绝对值小于10^{-8}
4. 使用递推减少重复计算，优化性能。

答案：

```
#include <stdio.h>

// 函数计算 sin(x)
double computeSin(double x) {
        double term = x;          // 当前项的值
```

```
    double sum = term;        // 累计的结果
    double threshold = 1e-8; // 误差阈值
    int n = 1;                // 当前项的阶数（1 表示 x^1）

    // 逐项计算直到误差小于阈值
    while (1) {
        term *= -x * x / ((2 * n) * (2 * n + 1)); // 计算下一项
        if (term > -threshold && term < threshold) {
            break; // 如果当前项绝对值小于误差阈值，停止计算
        }
        sum += term; // 累加到结果
        n++;         // 更新阶数
    }
    return sum;
}

int main() {
    double x;
    printf("请输入角度（弧度制）x: ");
    scanf("%lf", &x);

    double result = computeSin(x);
    printf("sin(%.6f) 的近似值为: %.8f\n", x, result);

    return 0;
}
```

2. 编写递归函数，实现从小到大有序的整数数组中进行二分检索，找到数据则返回所在的下标，没找到数据就会返回-1，注：数组下标从0开始。（25分）

算法思路

> 1. 输入参数:
>   - 有序整数数组 `arr`。
>   - 待查找值 `target`。
>   - 当前搜索范围的起始下标 `low` 和结束下标 `high`。
> 2. 递归基准:
>   - 如果 `low > high`，表示范围内无数据，返回 `-1`。
> 3. 核心逻辑:
>   - 计算中间下标 `mid = (low + high) / 2`。
>   - 如果 `arr[mid] == target`，返回 `mid`。
>   - 如果 `arr[mid] < target`，递归查找右半部分。
>   - 如果 `arr[mid] > target`，递归查找左半部分。

实现代码

```
// 递归实现二分检索
int binarySearchRecursive(int arr[], int low, int high, int target) {
    if (low > high) {
        return -1; // 基准情况：范围无效，未找到
    }

    int mid = low + (high - low) / 2; // 防止溢出的写法
    if (arr[mid] == target) {
        return mid; // 找到目标值，返回下标
    } else if (arr[mid] < target) {
        return binarySearchRecursive(arr, mid + 1, high, target); // 查找右半部分
    } else {
        return binarySearchRecursive(arr, low, mid - 1, target); // 查找左半部分
    }
}
```

3. 学生成绩信息包含学号、姓名和成绩三项，定义存储上述学生成绩信息的单向链表的结点类型，并编写函数，由键盘输入n个学生的成绩信息，创建一个用于管理学生成绩信息的单向链表 A，并在创建过程中随时保证单向链表的结点顺序满足成绩从低到高。（25分）

算法思路

> 1. 链表结点定义:
>   - 每个结点包含学生的 `学号`、`姓名` 和 `成绩`，以及指向下一个结点的指针。
> 2. 链表操作:
>   - 创建链表头指针 `head`，初始化为空。
>   - 通过输入 `n` 个学生信息，逐个插入结点到链表中。
>   - 每次插入时，按照成绩从低到高找到正确位置，插入结点。
> 3. 插入操作:
>   - 如果链表为空或新结点的成绩小于链表第一个结点的成绩，插入到链表头。
>   - 遍历链表，找到第一个成绩大于新结点的位置，将新结点插入到前面。

答案:

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

// 定义链表结点类型
typedef struct StudentNode {
    int id;                    // 学号
    char name[50];             // 姓名
    float grade;               // 成绩
    struct StudentNode *next; // 指向下一个结点
```

```c
} StudentNode, *SList;          // 定义 SList 类型

// 创建链表，输入 n 个学生信息
SList createStudentList(int n) {
    SList head = NULL; // 链表头指针
    for (int i = 0; i < n; i++) {
        int id;
        char name[50];
        float grade;
        printf("请输入第 %d 个学生的学号、姓名和成绩（用空格分隔）：\n", i + 1);
        scanf("%d %s %f", &id, name, &grade);

        // 创建新结点
        SList newNode = (SList)malloc(sizeof(StudentNode));
        newNode->id = id;
        strcpy(newNode->name, name);
        newNode->grade = grade;
        newNode->next = NULL;

        // 按成绩插入新结点
        if (head == NULL || head->grade >= newNode->grade) {
            // 插入到链表头
            newNode->next = head;
            head = newNode;
        } else {
            // 找到插入位置
            SList current = head;
            while (current->next != NULL && current->next->grade < newNode-
>grade) {
                current = current->next;
            }
            // 插入到当前位置之后
            newNode->next = current->next;
            current->next = newNode;
        }
    }
    return head;
}

// 主函数
int main() {
    int n;
    printf("请输入学生数量：");
    scanf("%d", &n);

    SList studentList = createStudentList(n); // 创建链表

    return 0;
}
```

4. 编写函数，从文件`classB.txt`中读取另一个班级的学生成绩信息创建链表B(文件`classB.txt`中的信息按照成绩从低到高的顺序存储)，将单向链表B与上题中的单向链表A归并为一个按学生成绩从低到高排序的单向链表。（25分）

思路介绍：

> `createListFromFile`:
>
> - 打开指定的文件 `classB.txt`。
> - 按行读取学生信息并创建链表 B。
> - 如果文件打开失败，返回 `NULL`。
>
> `mergeLists`:
>
> - 合并两个按成绩从低到高排序的单向链表。
> - 使用一个新的链表头指针，将 A 和 B 中较小的结点依次加入新链表。
> - 将剩余的结点直接拼接到合并链表的末尾。
>
> **主函数**:
>
> - 用户输入班级 A 的学生数量，调用 `createStudentList` 创建链表 A。
> - 调用 `createListFromFile` 从文件 `classB.txt` 创建链表 B。
> - 调用 `mergeLists` 将 A 和 B 合并为新的升序链表。
> - 打印结果并释放内存。

答案：

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

// 定义链表结点类型
typedef struct StudentNode {
    int id;                    // 学号
    char name[50];             // 姓名
    float grade;               // 成绩
    struct StudentNode *next; // 指向下一个结点
} StudentNode, *SList;

// 从文件读取学生信息，创建链表B
SList createListFromFile( char *filename) {
    FILE *file = fopen(filename, "r");
    if (!file) {
        printf("无法打开文件 %s\n", filename);
        return NULL;
    }

    SList head = NULL, tail = NULL;
    while (!feof(file)) {
        int id;
        char name[50];
```

```
        float grade;

        // 从文件中读取一行学生信息
        if (fscanf(file, "%d %s %f", &id, name, &grade) != -1) {
            // 创建新结点
            SList newNode = (SList)malloc(sizeof(StudentNode));
            newNode->id = id;
            strcpy(newNode->name, name);
            newNode->grade = grade;
            newNode->next = NULL;

            // 将结点加入链表
            if (head == NULL) {
                head = newNode;
                tail = newNode;
            } else {
                tail->next = newNode;
                tail = newNode;
            }
        }
    }

    fclose(file);
    return head;
}

// 合并两个按成绩升序排列的链表A和B
SList mergeLists(SList A, SList B) {
    SList mergedHead = NULL, mergedTail = NULL;

    while (A != NULL && B != NULL) {
        SList smallerNode = (A->grade <= B->grade) ? A : B;
        SList newNode = smallerNode;
        smallerNode = smallerNode->next;

        if (mergedHead == NULL) {
            mergedHead = newNode;
            mergedTail = newNode;
        } else {
            mergedTail->next = newNode;
            mergedTail = newNode;
        }
    }

    // 将剩余结点加入合并链表
    SList remaining = (A != NULL) ? A : B;
    if (mergedTail) {
        mergedTail->next = remaining;
    } else {
        mergedHead = remaining;
    }

    return mergedHead;
}

// 主函数
```

```c
int main() {
    int n;
    printf("请输入班级A学生数量：");
    scanf("%d", &n);

    SList listA = createStudentList(n);  // 创建链表A
    SList listB = createListFromFile("classB.txt"); // 从文件读取链表B

    // 合并链表A和B
    SList mergedList = mergeLists(listA, listB);



    return 0;
}
```