实习作业-轻量级员工管理系统的实现

信息管理学院 信息管理与信息系统 戴一诺 211820117

1 作业思路

1.1 目标

实现一款轻量级销售员工管理系统，包含基本人事管理功能，如：录入个人信息、读取员工信息、更改员工信息、人事解雇等。此外，为实现对于员工业务表现的考评，增加业绩排序功能。

1.2 开发工具

Lightly

1.3 运行平台

任何支持 C 语言环境的平台

1.4 I/O 形式

通过附件提供和展示

1.5 主要数据结构

1. 链表：实现员工条目管理
2. 哈希查找：实现快速查找员工信息
3. 快速排序：员工业绩评价管理

2 代码说明

2.1 结构体说明

该代码中，结构体主要用于存储员工信息和节点信息，通过结构体的嵌套和结构体指针的使用，实现了员工信息的存储和管理。

Position 是一个枚举类型，用于表示员工的职位。**Position** 枚举类型包含了不同的职位，例如 **SALESMAN**（销售员）、**MANAGER**（经理）和 **SALESMANAGER**（销售经理）。

Staff 是一个基础的员工信息结构体，包含了所有员工共有的基本属性，例如编号、姓名、年龄等。这个结构体是其他具体员工信息结构体的父结构体，用于提供一个通用的基础。

```
struct staff {
    char name[20];
    int age;
    float salary;
};
```

Salesman 是继承自 **Staff** 的结构体，用于表示销售员的详细信息。它包含了销售员特有的属性，例如销售额。

```
typedef struct {
    Staff staff;
    double sales;
} Salesman;
```

Manager 是继承自 **Staff** 的结构体，用于表示经理的详细信息。它可以包含经理特有的属性，例如团队规模等，但在此情况下，该结构体可能并没有被完整地实现。

```
typedef struct {
    Staff staff;
} Manager;
```

SalesManager 是继承自 **Salesman** 和 **Manager** 的结构体，用于表示销售经理的详细信息。它包含了销售经理特有的属性，例如团队销售额等。

```
typedef struct {
    Staff staff;
    double sales;
} SalesManager;
```

因此，可以看出这些结构体之间存在继承关系，**Salesman** 继承自 **Staff**，**Manager** 继承自 **Staff**，而 **SalesManager** 则同时继承自 **Salesman** 和 **Manager**。

结构体指针 **node**：该结构体指针用于定义一个链表节点，包括一个 staff 结构体和一个指向下一个节点的指针，定义如下：

```
struct node {
    struct staff info;
    struct node *next;
};
```

在主函数中，通过一个链表头指针来操作整个链表，定义如下：

```
struct node *head = NULL;
```

定义了管理系统结构体（ManagementSystem），包括链表头指针和员工数量等信息，定义如下：

```
typedef struct {
    Node* head;
    int staffNum;
} ManagementSystem;
```

2.2 函数说明

已定义一系列函数，用于实现各种操作功能，如初始化管理系统、显示员工信息、统计员工数量、按员工编号或姓名查找员工、录入员工信息、标记待解雇员工、修改员工信息、解雇员工、重组文件和员工销售额排序等。

具体来说包含了以下函数：

1. void initManagementSystem(ManagementSystem* system);：

- 用于初始化管理系统，将系统的头节点和员工数量初始化为默认值

2. void displayAllStaff(Node* head, int numStaff);：

- 显示所有员工的信息
- 参数 head 是链表的头节点，numStaff 是员工数量

3. void displayEmployeeCount(ManagementSystem* system);：

- 显示不同类型员工的数量统计
- 参数 system 是管理系统的指针

4. Node* findStaffByNo(ManagementSystem* system, const char* no);：

- 根据员工编号查找员工节点

- 参数 system 是管理系统的指针，no 是要查找的员工编号

5. void hire(ManagementSystem* system, const char* no, const char* name, int age, Position position, double sales);：

- 雇佣新员工并添加到管理系统中
- 参数 system 是管理系统的指针，no 是员工编号，name 是员工姓名，age 是员工年龄，position 是员工职位，sales 是销售额

6. void markForTermination(ManagementSystem* system);：

- 标记员工待解雇状态
- 参数 system 是管理系统的指针

7. void modifyStaff(ManagementSystem* system);：

- 修改员工信息，包括销售人员的销售额
- 参数 system 是管理系统的指针

8. void terminateStaff(ManagementSystem* system);：

- 解雇员工
- 参数 system 是管理系统的指针

9. void recombineFile(ManagementSystem* system);：

- 重新组合文件，将当前管理系统中的员工信息保存到文件中
- 参数 system 是管理系统的指针

10. void selectFunction(ManagementSystem* system);：

- 提供菜单选择功能，根据用户输入选择相应的操作
- 参数 system 是管理系统的指针

11. void quicksort(Staff** staffList, int left, int right);：

- 快速排序算法，用于对员工列表按照销售额进行排序
- 参数 staffList 是员工列表的指针数组，left 和 right 是排序范围的左右边界

12. int partition(Staff** staffList, int left, int right);：

- 快速排序中的划分函数，用于确定基准值的位置
- 参数 staffList 是员工列表的指针数组，left 和 right 是划分范围的左右边界

13. void swap(Staff** a, Staff** b);:

- 交换两个指针的值
- 参数 a 和 b 为要交换的指针

2.3 选择功能函数

调用选择功能函数（selectFunction）来执行用户选择的操作：**selectFunction** 函数提供了一个交互式的菜单界面，用户可以通过选择不同的操作来管理员工信息，包括显示、录入、修改、解雇等操作，并可以将员工信息保存到文件或按销售额排序。

定义一个名为 **selectFunction** 的函数，它接受一个 **ManagementSystem** 类型的指针作为参数。该函数通过一个循环，让用户选择不同的操作来管理员工信息。

首先显示一个菜单，列出了可用的操作选项，包括显示员工信息、显示员工总人数及各类员工人数、按员工号或姓名索引信息、录入员工信息、标记待解雇员工、修改员工信息、解雇员工、重组文件和员工销售额排序等。

然后，根据用户的选择，使用 **switch** 语句执行相应的操作。每个操作对应一个具体的函数调用，例如调用 **displayAllStaff** 函数显示所有员工信息，调用 **displayEmployeeCount** 函数显示员工总人数及各类员工人数。

循环会一直执行，直到用户选择退出操作（输入 0），此时函数会打印一条退出提示并返回。

2.4 快速排序算法

```
void quicksort(Staff** staffList, int left, int right) {
    if (left < right) {
        int pivot = partition((Staff**)staffList, left, right);
        quicksort(staffList, left, pivot - 1);
        quicksort(staffList, pivot + 1, right);
    }
}


void swap(Staff** a, Staff** b) {
    Staff* temp = *a;
    *a = *b;
```

```
    *b = temp;
}

int partition(Staff** staffList, int left, int right) {
    double pivot = ((Salesman*)staffList[right])->sales;
    int i = left - 1;

    for (int j = left; j <= right - 1; j++) {
        if (((Salesman*)staffList[j])->sales >= pivot) {
            i++;
            swap(&staffList[i], &staffList[j]);
        }
    }

    swap(&staffList[i + 1], &staffList[right]);
    return (i + 1);
}
```

利用快速排序实现了对 **staffList** 数组进行排序，排序的依据是员工的销售额。

首先，定义了两个辅助函数：**swap** 和 **partition**。

swap 函数用于交换两个指针类型的变量的值，这里用于交换 **Staff*类型的指针。在排序过程中，**swap** 函数用于交换 **staffList** 数组中的元素位置。

partition 函数用于选择一个基准 pivot 元素，并将数组划分为两部分，使得左边的元素都小于等于 pivot，右边的元素都大于 pivot。在这里，pivot 被选为 **staffList[right]**（数组的最后一个元素），即最右边的元素。然后，通过遍历数组中的元素，将大于等于 pivot 的元素交换到左边，小于 pivot 的元素交换到右边。最后，将 pivot 元素放置到正确的位置，返回其索引。

接下来，定义了 **quicksort** 函数来实现快速排序算法。该函数接受 **staffList** 数组的指针、左边界索引 **left** 和右边界索引 **right** 作为参数。

在 **quicksort** 函数内部，首先检查左边界是否小于右边界，如果满足条件，则进行排序操作。

首先调用 **partition** 函数，将数组划分为两部分，并返回 pivot。然后，递归地调用 **quicksort** 函数，对左半部分（左边界到 `pivot - 1`）进行排序。接着，递归地调用

quicksort 函数，对右半部分（`pivot + 1` 到右边界）进行排序。这样，通过递归的方式，不断地划分和排序数组的左右部分，直到最终完成整个数组的排序。

时间复杂度分析：

根据给出的代码实现，这段快速排序的时间复杂度也是 O(n*logn)：快速排序的时间复杂度主要由划分和递归决定。

划分操作的时间复杂度是 O(n)，其中 n 是待排序数组的长度。

递归操作的时间复杂度是 O(logn)，因为每次递归调用都将数组划分为两部分，每部分的长度约为原数组的一半。

3 算法改进

3.1 排序算法

首先考虑到在轻量级员工管理系统下，可以选择使用计数排序 **Counting Sort** 来实现更低的时间复杂度。

计数排序适用于待排序元素的取值范围比较小且已知的情况。在这个例子中，我们可以假设销售额是非负整数，并且我们已知最大的销售额。假设最大销售额为 maxSales。计数排序的基本思想是创建一个长度为 maxSales + 1 的计数数组，用于记录每个销售额出现的次数。然后，根据计数数组中的统计信息，将员工按照销售额的顺序重新排列。以下是使用计数排序实现员工按销售额排序的示例代码：

```
void countingSort(Staff** staffList, int size, int maxSales) {
    int* count = (int*)calloc(maxSales + 1, sizeof(int));

    // 统计每个销售额的出现次数
    for (int i = 0; i < size; i++) {
        int sales = ((Salesman*)staffList[i])->sales;
        count[sales]++;
    }

    // 计算每个销售额的累积次数
    for (int i = 1; i <= maxSales; i++) {
        count[i] += count[i - 1];
    }
```

```
    // 创建临时数组用于存储排序结果
    Staff** sorted = (Staff**)malloc(size * sizeof(Staff*));

    // 根据销售额和累积次数，将员工排序到临时数组中
    for (int i = size - 1; i >= 0; i--) {
        int sales = ((Salesman*)staffList[i])->sales;
        int index = count[sales] - 1;
        sorted[index] = staffList[i];
        count[sales]--;
    }

    // 将排序结果拷贝回原数组
    for (int i = 0; i < size; i++) {
        staffList[i] = sorted[i];
    }

    // 释放临时数组和计数数组的内存
    free(sorted);
    free(count);
}
```

使用计数排序的时间复杂度是 O(n + k)，其中 n 是待排序数组的长度，k 是销售额的取值范围。

但是在本系统中，销售额的最大值 maxSales 可能不是一个较小的常数（在 10^6 级别以上），那么计数排序不合适，仍然是快速排序更合适，因而保持原设置。

3.2 查找算法

要降低查找算法的时间复杂度，可以考虑使用更高效的数据结构，例如哈希表（Hash Table）。通过使用哈希表，可以将查找的时间复杂度降低到 O(1)。

当以编号查找员工时，哈希表比顺序查找更快。哈希表具有平均时间复杂度为 O(1)的查找操作，而顺序查找的时间复杂度为 O(n)，其中 n 为员工数量。

哈希表利用哈希函数将键（员工编号）映射到存储位置，因此可以直接通过计算哈希函数得到目标员工的存储位置，并在该位置进行查找。无论员工数量如何增加，哈希表的查找速度几乎保持不变，因为它的平均查找时间不依赖于数据规模。

相比之下，顺序查找需要遍历整个员工列表，逐个比较员工编号，直到找到匹配的员工或遍历完所有员工。随着员工数量的增加，顺序查找的时间复杂度呈线性增长，查找速度会随之变慢。

因此，当要以编号查找员工时，使用哈希表能够提供更快的查找速度，尤其是在员工数量较大的情况下。

下面是使用哈希表优化后的代码：

```
#define HASH_TABLE_SIZE 40009


typedef struct {
    char no[20];
    char name[50];
    int age;
} Staff;


typedef struct Node {
    Staff* staff;
    int isEmployed;
    struct Node* next;
} Node;


typedef struct {
    Node* head;
    int staffNum;
} ManagementSystem;


typedef struct {
    char key[20];
    Node* value;
} HashEntry;


typedef struct {
    HashEntry* table;
    int size;
} HashTable;
```

```
// 创建哈希表
HashTable* createHashTable(int size) {
    HashTable* hashTable = (HashTable*)malloc(sizeof(HashTable));
    hashTable->table = (HashEntry*)malloc(size * sizeof(HashEntry));
    hashTable->size = size;

    // 初始化哈希表
    for (int i = 0; i < size; i++) {
        strcpy(hashTable->table[i].key, "");
        hashTable->table[i].value = NULL;
    }

    return hashTable;
}

// 哈希函数
int hashFunction(const char* key, int size) {
    unsigned long hash = 5381;
    int c;

    while ((c = *key++)) {
        hash = ((hash << 5) + hash) + c; // djb2 哈希算法
    }

    return hash % size;
}

// 向哈希表插入键值对
void insertIntoHashTable(HashTable* hashTable, const char* key, Node* value) {
    int index = hashFunction(key, hashTable->size);

    while (strcmp(hashTable->table[index].key, "") != 0) {
        index = (index + 1) % hashTable->size;
```

```
    }

    strcpy(hashTable->table[index].key, key);
    hashTable->table[index].value = value;
}

// 从哈希表中查找键对应的值
Node* findInHashTable(HashTable* hashTable, const char* key) {
    int index = hashFunction(key, hashTable->size);

    while (strcmp(hashTable->table[index].key, "") != 0) {
        if (strcmp(hashTable->table[index].key, key) == 0) {
            return hashTable->table[index].value;
        }
        index = (index + 1) % hashTable->size;
    }

    return NULL;
}

// 销毁哈希表
void destroyHashTable(HashTable* hashTable) {
    free(hashTable->table);
    free(hashTable);
}
```

创建一个哈希表，将员工信息存储在哈希表中，然后通过员工编号快速查找对应的员工信息。这种方式相比顺序查找可以显著提高查找效率。

- **HashTable** 结构体定义了哈希表的基本属性，包括一个指向 HashEntry 数组的指针 table 和哈希表的大小 size。
- **HashEntry** 结构体定义了哈希表中的每个条目，包括一个键 key 和一个值 value，其中键是员工的编号，值是指向对应员工信息的节点 Node 的指针。
- **createHashTable** 函数用于创建哈希表。它动态分配内存并初始化哈希表中的所有条目，将键和值都初始化为空。

- **hashFunction** 函数是一个哈希函数，它将给定的键（员工编号）映射到哈希表的索引位置。这里使用的是 djb2 哈希算法，通过对每个字符进行位运算和累加来计算哈希值。
- **insertIntoHashTable** 函数用于向哈希表插入键值对。它通过计算键的哈希值找到合适的索引位置，并处理哈希冲突（如果发生）来寻找空槽位，然后将键和值存储在该位置。
- **findInHashTable** 函数用于从哈希表中查找给定键的值。它通过计算键的哈希值找到对应的索引位置，并在该位置及后续位置进行线性探测，直到找到匹配的键或遇到空槽位。
- **destroyHashTable** 函数用于释放动态分配的内存。

使用哈希表优化后，查找的时间复杂度降低到 O(1)，但需要额外的空间来存储哈希表，以空间换时间。在该系统的实际应用中，会根据数据规模和性能需求综合考虑是否使用哈希表优化。

4 代码运行效果展示

```
开始运行...

请选择操作：
1. 显示所有员工信息
2. 显示员工总人数及各类员工人数
3. 按员工号或者姓名索引信息
4. 录入员工信息
5. 标记待解雇员工
6. 修改员工信息
7. 解雇员工
8. 重组文件
9. 员工销售额排序(升序)
0. 退出
请输入操作编号：
```

————开始运行————

【录入员工信息】

Output | Build Log

```
请输入操作编号：4
----------------------
请输入员工编号：002
请输入员工姓名：yi
请输入员工年龄：35
请选择员工职位（1-经理，2-销售员）：2
请输入员工销售额：4000
------------end-------------

请选择操作：
1．显示所有员工信息
2．显示员工总人数及各类员工人数
3．按员工号或者姓名索引信息
4．录入员工信息
5．标记待解雇员工
6．修改员工信息
7．解雇员工
8．重组文件
9．员工销售额排序(升序)
0．退出
请输入操作编号：4
----------------------
请输入员工编号：003
请输入员工姓名：nuo
请输入员工年龄：44
请选择员工职位（1-经理，2-销售员）：2
请输入员工销售额：9090
------------end-------------

请选择操作：
1．显示所有员工信息
2．显示员工总人数及各类员工人数
3．按员工号或者姓名索引信息
4．录入员工信息


5．标记待解雇员工
6．修改员工信息
7．解雇员工
8．重组文件
9．员工销售额排序(升序)
0．退出
请输入操作编号：4
----------------------
请输入员工编号：004
请输入员工姓名：zhu
请输入员工年龄：41
请选择员工职位（1-经理，2-销售员）：2
请输入员工销售额：80880
------------end-------------
```

【显示员工人数】

```
请选择操作：
1．显示所有员工信息
2．显示员工总人数及各类员工人数
3．按员工号或者姓名索引信息
4．录入员工信息
5．标记待解雇员工
6．修改员工信息
7．解雇员工
8．重组文件
9．员工销售额排序(升序)
0．退出
请输入操作编号：2
----------------------
员工总人数：4
经理人数：1
销售员人数：3
------------end-------------
```

【显示员工信息】

```
1．显示所有员工信息
2．显示员工总人数及各类员工人数
3．按员工号或者姓名索引信息
4．录入员工信息
5．标记待解雇员工
6．修改员工信息
7．解雇员工
8．重组文件
9．员工销售额排序(升序)
0．退出
请输入操作编号：1
----------------------
所有员工信息：
编号：001
姓名：dai
年龄：34
----------------------
编号：002
姓名：yi
年龄：35
----------------------
编号：003
姓名：nuo
年龄：44
----------------------
编号：004
姓名：zhu
年龄：41
----------------------
------------end-------------
```

【按员工号索引信息】

```
请选择操作：
1．显示所有员工信息
2．显示员工总人数及各类员工人数
3．按员工号或者姓名索引信息
4．录入员工信息
5．标记待解雇员工
6．修改员工信息
7．解雇员工
8．重组文件
9．员工销售额排序(升序)
0．退出
请输入操作编号：3
----------------------
请输入要查找的员工编号：003
员工信息：
编号：003
姓名：nuo
年龄：44
------------end-------------
```

【标记待解雇员工】

为了防止误删除员工信息，只有经过标记待解雇后才可以被删除员工数据。

标记后不影响信息修改。

```
请选择操作：
1．显示所有员工信息
2．显示员工总人数及各类员工人数
3．按员工号或者姓名索引信息
4．录入员工信息
5．标记待解雇员工
6．修改员工信息
7．解雇员工
8．重组文件
9．员工销售额排序(升序)
0．退出
请输入操作编号：5
----------------------
请输入要标记解雇的员工编号：003
员工003已被标记为待解雇。
------------end-------------
```

【修改员工销售额】

```
请选择操作：
1．显示所有员工信息
2．显示员工总人数及各类员工人数
3．按员工号或者姓名索引信息
4．录入员工信息
5．标记待解雇员工
6．修改员工信息
7．解雇员工
8．重组文件
9．员工销售额排序(升序)
0．退出
请输入操作编号：6
----------------------
请输入要修改信息的员工编号：003
当前销售额：9090.00
请输入新的销售额：9099
员工销售额修改成功。
------------end-------------
```

【解雇员工】

```
请选择操作：
1．显示所有员工信息
2．显示员工总人数及各类员工人数
3．按员工号或者姓名索引信息
4．录入员工信息
5．标记待解雇员工
6．修改员工信息
7．解雇员工
8．重组文件
9．员工销售额排序(升序)
0．退出
请输入操作编号：7
----------------------
请输入要解雇的员工编号：003
该员工已标记为待解雇状态。
------------end-------------

请选择操作：
1．显示所有员工信息
2．显示员工总人数及各类员工人数
3．按员工号或者姓名索引信息
4．录入员工信息
5．标记待解雇员工
6．修改员工信息
7．解雇员工
8．重组文件
9．员工销售额排序(升序)
0．退出
请输入操作编号：2
----------------------
员工总人数：3
经理人数：1
销售员人数：2
------------end-------------
```

```
请选择操作：
1．显示所有员工信息
2．显示员工总人数及各类员工人数
3．按员工号或者姓名索引信息
4．录入员工信息
5．标记待解雇员工
6．修改员工信息
7．解雇员工
8．重组文件
9．员工销售额排序(升序)
0．退出
请输入操作编号：7
----------------------
请输入要解雇的员工编号：  003
未找到该员工。
------------end-------------
```

```
请选择操作：
1．显示所有员工信息
2．显示员工总人数及各类员工人数
3．按员工号或者姓名索引信息
4．录入员工信息
5．标记待解雇员工
6．修改员工信息
7．解雇员工
8．重组文件
9．员工销售额排序(升序)
0．退出
请输入操作编号：1
----------------------
所有员工信息：
编号：001
姓名：dai
年龄：34
----------------------
编号：002
姓名：yi
年龄：35
----------------------
编号：004
姓名：zhu
年龄：41
----------------------
------------end-------------
```

【保存文件】

```
请选择操作：
1. 显示所有员工信息
2. 显示员工总人数及各类员工人数
3. 按员工号或者姓名索引信息
4. 录入员工信息
5. 标记待解雇员工
6. 修改员工信息
7. 解雇员工
8. 重组文件
9. 员工销售额排序(降序)
0. 退出
请输入操作编号：8
----------------------
文件保存成功，程序退出。

运行结束。
```

【销售额业绩排序】

对销售额业绩排序进行从低到高的排序。

```
请选择操作：
1. 显示所有员工信息
2. 显示员工总人数及各类员工人数
3. 按员工号或者姓名索引信息
4. 录入员工信息
5. 标记待解雇员工
6. 修改员工信息
7. 解雇员工
8. 重组文件
9. 员工销售额排序(升序)
0. 退出
请输入操作编号：9
----------------------
员工销售额排序结果：
员工编号：004
员工编号：003
员工编号：002
员工编号：001
------------end-------------
```

【结束运行】

```
请选择操作：
1. 显示所有员工信息
2. 显示员工总人数及各类员工人数
3. 按员工号或者姓名索引信息
4. 录入员工信息
5. 标记待解雇员工
6. 修改员工信息
7. 解雇员工
8. 重组文件
9. 员工销售额排序(升序)
0. 退出
请输入操作编号：0
----------------------
程序已退出。

运行结束。
```

5 结论

该段程序实现了一款轻量级销售员工管理系统，包含基本人事管理功能，如：录入个人信息、读取员工信息、更改员工信息、人事解雇等。而且业绩排序功能可以辅助考评员工业务表现。

在基础的数据结构应用中，利用链表实现了员工数据管理。为了提高员工信息的查找效率，减少大规模员工管理情境下的时间复杂度，使用了哈希表进行改进。同时在排序算法方面进行了改进的尝试：经过分别比较快速排序与计数排序在本系统中的适用性，考虑到系统规模与营业额上限，最终选择快速排序作为排序算法。

但是这份作业也有其局限性，包括对于轻量级员工管理系统的应用场景的实战经验与深层理解不足、使用的是比较传统的链表结构、排序算法的改进有限等。尽管能够胜任轻量级、数据级不高的业务应用场景，但仍然需要进一步的思考与改进。

感谢老师的阅读与指导!