大连海事大学实验报告

设计题目 : 实验一 线性表

专业 : 信息管理与信息系统

班级 : 二班

学号 : 2220192279

姓名 : 王心昊

指导教师 : 李晔

航运经济与管理学院 二○二一年 三 月

一、设计需求描述

编制一个能演示执行集合的并、交和差运算的程序

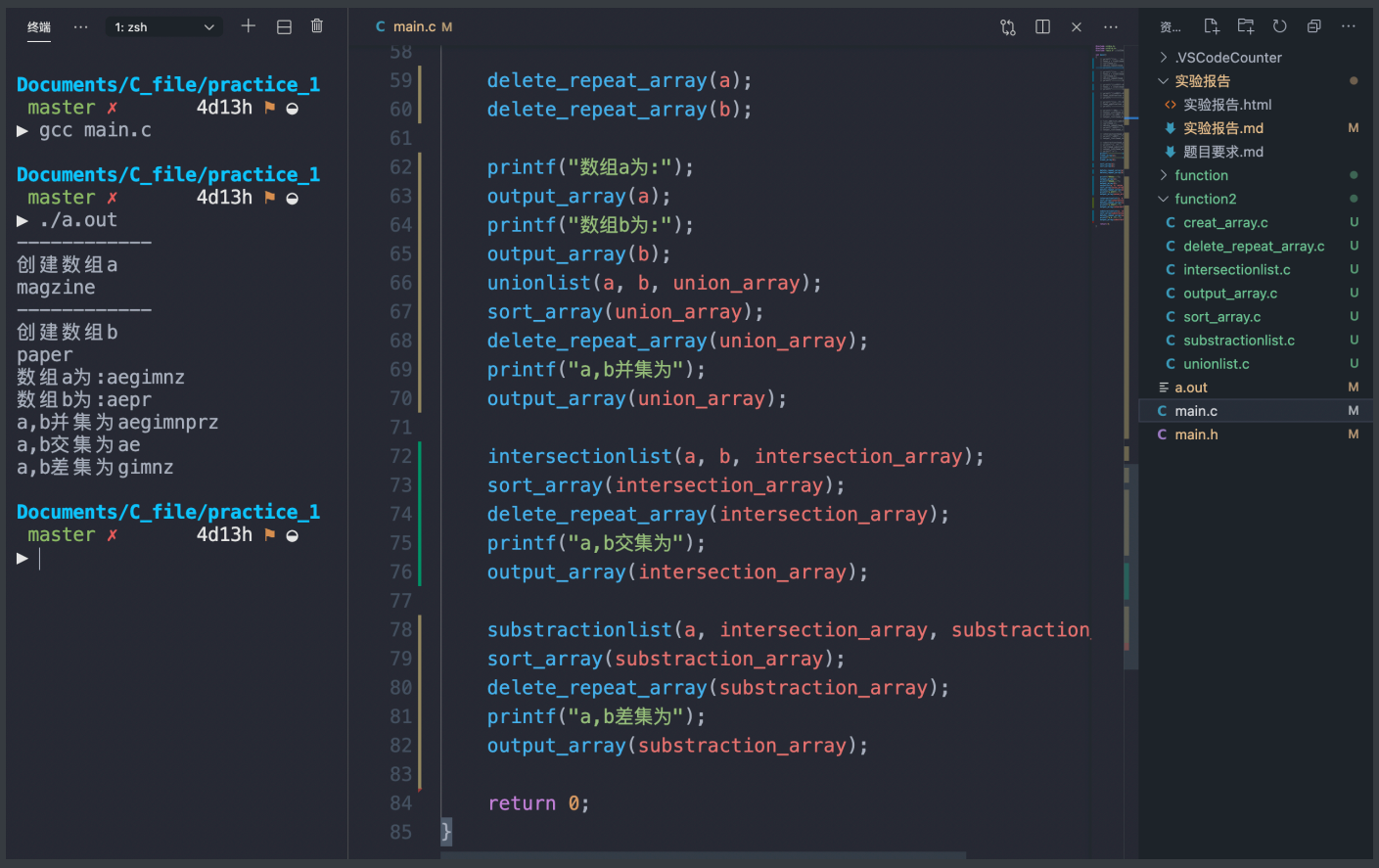
二、程序设计指导思想

利用单链表的基本特点,利用指针操作实现对集合的交集、补集、并集运算

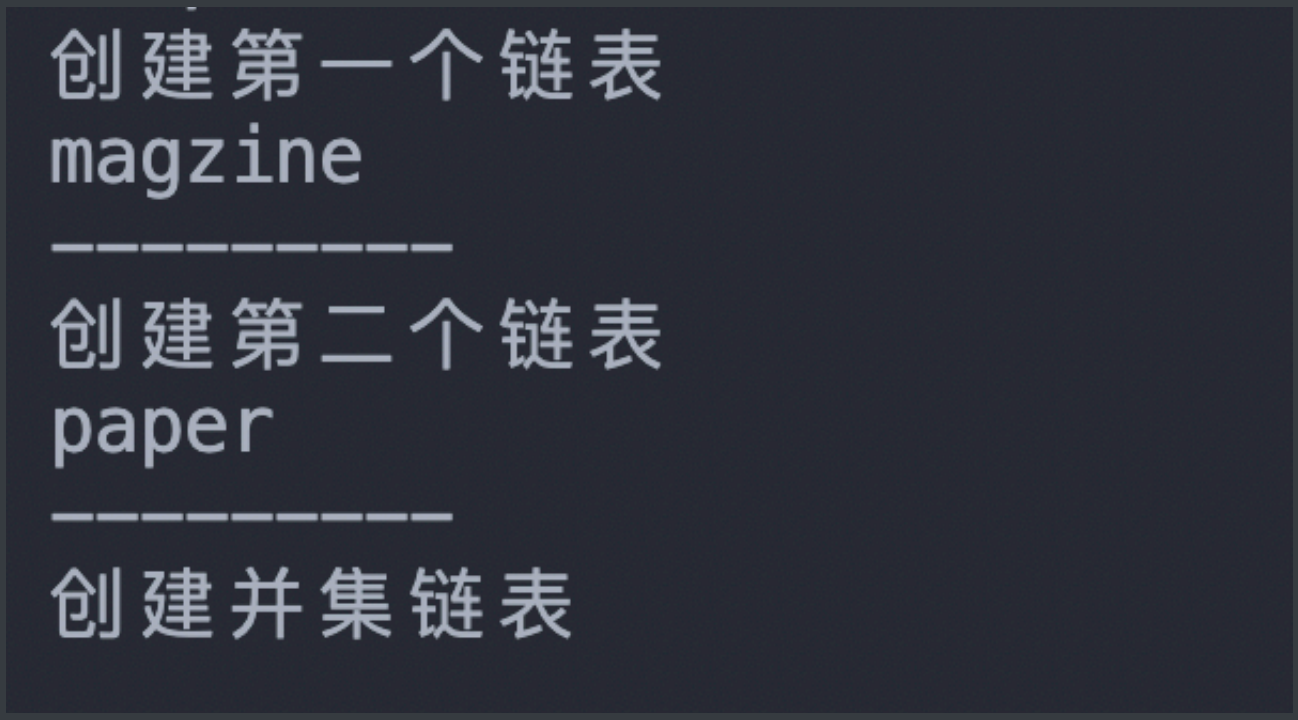
三、程序算法设计

- 程序中的主要数据结构:单链表
- 程序算法的总体设计:程序分为三个部分 分别为 main.c function main.h,其中在main.h中定义了全局使用的变量以及结构体等
  Function文件夹中有函数文件,所有文件中包含一个行为函数由main.c集中调用
- 程序框图及必要的说明
  程序运行过程:程序开始,录入两个待操作链表,之后分别进行取交集 并集 差集运算

四、设计过程(界面)



运行结果



```
----------
创建交集链表

----------
创建差集链表

----------
链表a为:aegimnz
链表b为:aepr
并集为:aegimnprz
交集为:ae
差集为:gimnz
--------------
创建数组a
magzine
--------------
创建数组b
paper
数组a为:aegimnz
数组b为:aepr
a,b并集为aegimnprz
a,b交集为ae
```

a,b差集为gimnz

五、设计总结

- 在程序设计过程中练习了链表的操作,并体会到了面向过程变成的本质,锻炼了逻辑思维
- 存在问题及改进
  在后续编程中需要改进算法记录方法,需要有一个直观并且完整的记录
- 其它需说明的情况（如果有请说明）

六、程序代码

main.c文件

```
#include <stdio.h>
#include <stdlib.h>
#include "main.h" //定义全局变量 头指针 创建链表过程中所需指针

int main()
{
    printf("创建第一个链表\n");
    head_a = creat(head_a); // 创建第一个链表Í
    sort(head_a);
    delete_repeat(head_a);
    printf("---------\n");

    printf("创建第二个链表\n");
    head_b = creat(head_b); //创建第二个链表
    sort(head_b);
    delete_repeat(head_b);
    printf("---------\n");

    printf("创建并集链表\n");
```

```
    head_c = creat(head_c);
    printf("---------\n");

    printf("创建交集链表\n");
    head_intersection = creat(head_intersection);
    printf("---------\n");

    printf("创建差集链表\n");
    head_substraction = creat(head_substraction);
    printf("---------\n");

    printf("链表a为:");
    output_list(head_a);
    printf("\n链表b为:");
    output_list(head_b);

    list_add(list_add(head_c,head_a),head_b);//并集
    sort(head_c);
    delete_repeat(head_c);
    printf("\n并集为:");
    output_list(head_c);


    intersection(head_a, head_b, head_intersection); //交集
    printf("\n交集为:");
    output_list(head_intersection);

    substraction(head_a, head_intersection, head_substraction);
    printf("\n差集为:");
    sort(head_substraction);
    output_list(head_substraction);
    printf("\n");
    return 0;
}
```

main.h文件

```
struct link//定义结构体
{
    char data;
    struct link *next;
};
#define LEN sizeof(struct link)


struct link *head_a;//定义第一个链表头指针
struct link *head_b;//定义第二个链表头指针
struct link *head_c;//定义第合并链表头指针
struct link *head_intersection;//定义第交集链表头指针
struct link *head_substraction;//定义第差集链表头指针

struct link *test;



#include "./function/creat.c"
#include "./function/output_list.c"
#include "./function/sort.c"
#include "./function/delete_repeat.c"
#include "./function/list_add.c"
#include "./function/intersection.c"
#include "./function/substraction.c"
```

```
struct link *creat(struct link *head) //创建链表函数
{
    struct link *p_present, *p_open;
    p_present = p_open = (struct link *)malloc(LEN); //分配内存空间
    if (p_present == NULL)                           //判断是否成功创建
    {
```

```
            printf("分配内存失败\n");
            printf("回车退出\n");
            getchar();
            getchar();
            exit(0);
        }
        head = p_present;
        while ((p_open->data=getchar()) != '\n') //输入为回车结束创建
        {
            p_present = p_open;                  //将操作对象移动至下一个节点
            p_open = (struct link *)malloc(LEN); //分配新的内存
            p_present->next = p_open;            //指向下一个节点
            if (p_open == NULL)                  //判断是否分配失败
            {
                printf("分配内存失败\n");
                return 0;
            }
        }
        p_present->next = NULL; //尾节点null结束创建
        struct link *tmp;       //创建空头节点
        tmp = (struct link *)malloc(LEN);
        tmp->next = head;
        return tmp;
    }
```

```
    int delete_repeat(struct link *pt)
    {
        // 前期准备
        struct link *current, *to_connect;
        current = pt->next;

        //循环结束标志
        while (current->next != NULL)
        {
            if (current->data == current->next->data) //删除部分
```

```
        {
            to_connect = current->next->next;
            free(current->next);
            current->next = to_connect;
        }
        else
        {
            current = current->next; //指针下移
        }
    }
    return 0;
}
```

```
void intersection(struct link *list_a, struct link *list_b, struct link
*list_intersection)//对已排序且删除重复元素的链表进行合并
{
    struct link *list_a_node, *list_b_node, *list_intersection_node;
    list_a_node = list_a->next;
    list_b_node = list_b->next;
    list_intersection_node = list_intersection->next;
    while (list_a_node != NULL)
    {
        while (list_b_node != NULL)
        {

            if (list_a_node->data == list_b_node->data)
            {
                list_intersection_node->data = list_b_node->data;
                list_intersection_node->next = (struct link
*)malloc(LEN);
                list_intersection_node = list_intersection_node->next;
                list_intersection_node->next = NULL;
                break;
            }
                list_b_node = list_b_node->next;
```

```
        }
        list_a_node = list_a_node -> next;
        list_b_node = list_b->next;
    }
}
```

```
struct link *list_add(struct link *end_of_receiver, struct link *adder)
{
    struct link *receiver_last_node;
    receiver_last_node = end_of_receiver;
    // while (receiver_last_node->next != NULL)
    // {
    //     receiver_last_node = receiver_last_node->next;
    // } //已经移动到了receiver最后面

    struct link *open;
    struct link *new_node_to_add;
    new_node_to_add = adder->next;
    while (new_node_to_add != NULL)
    {
        open = (struct link *)malloc(LEN);
        receiver_last_node->next = open;
        receiver_last_node->next->data = new_node_to_add->data;
        receiver_last_node = receiver_last_node->next; //move forward
        new_node_to_add = new_node_to_add->next;       //move forward
    }
    receiver_last_node->next = NULL;
    return receiver_last_node;
}
```

```c
void output_list(struct link *head) //输出链表函数
{
    struct link *pt;
    pt = head->next;
    while (pt != NULL)
    {

        printf("%c", pt->data);
        pt = pt->next;
    }
}
```

```c
void sort(struct link *head)
{
    struct link *current, *fetch, *move, *pre, *save_fetch_next; //变量定义

    //前期的准备 双指针初始位置
    pre = head;             //空头节点定义为“前指针pre”
    current = head->next;  //第一个有数据节点定义为“current指针”
    fetch = current->next; //取出待排序链表第一个节点

    current->next = NULL; //切断链表完整性 构造已排序链表和未排序链表
    while (fetch != NULL) //●待排序节点为空无节点可排序
    {
        move = fetch; //将“取元素指针fetch”传递给已排好序链表的“排序指针move”
        save_fetch_next = fetch->next;
        move->next = NULL;      //从未排序链表中分离出来
        while (current != NULL) //已排序链表遍历到最后一个之前,排序的操作是重复的
        {
            if (move->data <= current->data) //当找到适合的位置的时候开始插入
            {
                pre->next = move;     //前指针指向待排序节点
                move->next = current; //待排序节点指向后一个节点    连接起来
```

```
                //双指针复位
                pre = head;
                current = head->next;
                break; //排序完毕跳出循环
            }
            else //没有找到合适的位置,将双指针后移
            {
                if (current->next != NULL) //双指针可以后移时才后移,
                {
                    pre = pre->next;
                    current = current->next;
                }
                else //不可后移时说明已经到了结尾
                {
                    current->next = move;
                    move->next = NULL;
                    //双指针复位
                    pre = head;
                    current = head->next;
                    break; //插入结束
                }
            }
        }

        //节点已经已经插入完毕取新元素
        fetch = save_fetch_next; //十分简单的一个句子
    }
    //函数结束


}
```

```c
void substraction(struct link *head_a, struct link *head_intersection,
struct link *substraction)
{
    struct link *head_a_node;
    struct link *head_intersection_node;
    struct link *head_substraction_node;
    head_a_node = head_a->next;
    head_intersection_node = head_intersection->next;
    head_substraction_node = substraction->next;
    while (head_a_node != NULL)
    {
        while (head_intersection_node != NULL)
        {
            if (head_a_node->data == head_intersection_node->data)
            {
                break;
            }
            head_intersection_node = head_intersection_node->next;
        }

        if (head_intersection_node == NULL) //全部遍历结束
        {
            head_substraction_node->data = head_a_node->data;
            head_substraction_node->next = (struct link *)malloc(LEN);
            head_substraction_node = head_substraction_node->next;
            head_substraction_node->next = NULL;
        }
        head_a_node = head_a_node->next;//链表a跳到下一个节点
        head_intersection_node = head_intersection->next;//交集链表复位
    }

}
```

顺序表

```
void creat_array(char *head)
{
    int i;
    char c;
    while ((c = getchar()) != '\n')
    {
        *(head++) = c;
    }
}
```

```
void delete_repeat_array(char *head)
{
    int i;
    for (i = 0; *(head + i) != '\0';)
    {
        if (*(head + i) == *(head + i + 1))
        {
            int j;
            for (j = i; *(head + j) != '\0'; j++)
            {
                *(head + j) = *(head + j + 1);
            }
        }
        else
        {
            i++;
        }
    }
}
```

```
void intersectionlist(char *a, char *b, char *intersection_array)
{
```

```
    int i, j, k = 0;
    for (i = 0; *(a + i) != '\0'; i++)
    {
        for (j = 0; *(b + j) != '\0'; j++)
        {
            if (*(a + i) == *(b + j))
            {
                *(intersection_array + k) = *(a + i);
                k++;
            }
        }
    }
}
```

```
void output_array(char *head)
{
    int i;
    for (i = 0; *(head + i) != '\0'; i++)
    {
        printf("%c", *(head + i));
    }
    putchar('\n');
}
```

```
void sort_array(char *head)
{
    int i, j;
    char tmp;
    for (i = 0; *(head + i + 1) != '\0'; i++)
    {
        for (j = i + 1; *(head + j) != '\0'; j++)
        {
            if (*(head + j) < *(head + i))
            {
                tmp = *(head + i);
```

```
                *(head + i) = *(head + j);
                *(head + j) = tmp;
            }
        }
    }
}
```

```
void substractionlist(char *a, char *intersection_array, char
*substraction_array)
{
    int i, j, k = 0;
    for (i = 0; *(a + i) != '\0'; i++)
    {
        int flag = 1;
        for (j = 0; *(intersection_array + j) != '\0'; j++)
        {
            if (*(a + i) == *(intersection_array + j))
            {
                flag = 0;
                break;
            }
        }
        if (flag != 0)
        {
            *(substraction_array + k) = *(a + i);
            flag = 1;
            k++;
        }
    }
}
```

```c
void unionlist(char *a, char *b, char *union_array)
{
    int i, j;
    for (i = 0; *(a + i) != '\0'; i++)
    {
        *(union_array + i) = *(a + i);
    }
    for (j = 0; *(b + j) != '\0'; j++, i++)
    {
        *(union_array + i) = *(b + j);
    }
}
```