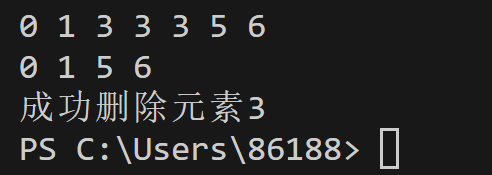
作业一结果

对于顺序储存结构来说：

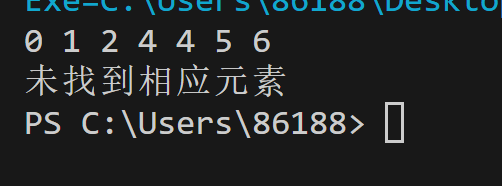
① 删除给定元素的算法。

1.1删除指定元素

数据样例： 数组为[0,1,3,3,3,5,6] 目标删除数据为3 ；

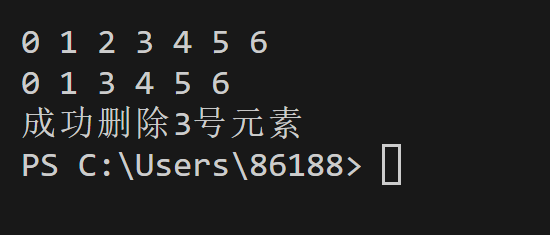


数据样例：数组[0,1,2,4,4,5,6] 目标删除数据为 3；

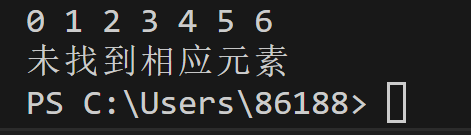


1.2删除指定位置元素

数据样例：数组[0,1,2,3,4,5,6] 目标删除位置 3；

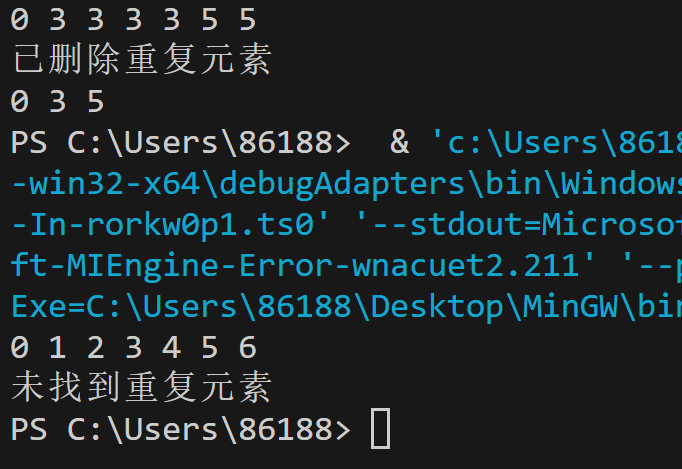


数据样例：数组[0,1,2,3,4,5,6] 目标删除位置8；

  
② 对于已排好序的线性表，删除所有重复元素的算法。

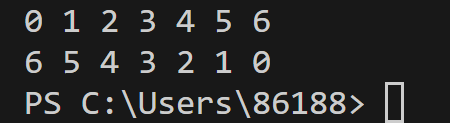
数据样例1：数组[0,3,3,3,3,5,5]

数据样例2：数组[0,1,2,3,4,5,6]



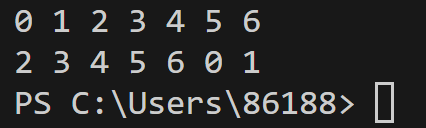
③ 线性表“逆置”算法。

数据样例：数组[0,1,2,3,4,5,6]

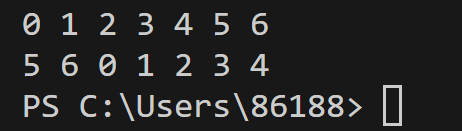


④ 线性表循环左移/右移 k 位的算法。

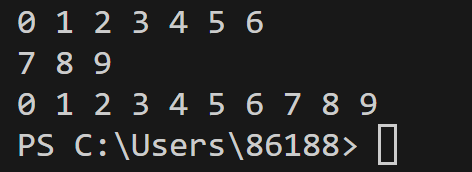
数据样例：数组[0,1,2,3,4,5,6] 左移2位（-2）



数据样例：数组[0,1,2,3,4,5,6] 右移9位（9）



⑤ 合并两个已排好序的线性表的算法。

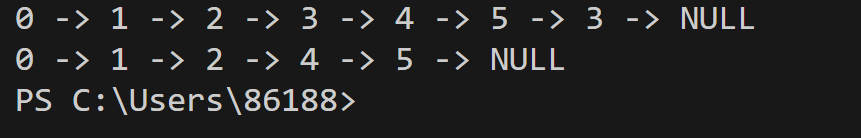


对于链式储存结构来说：

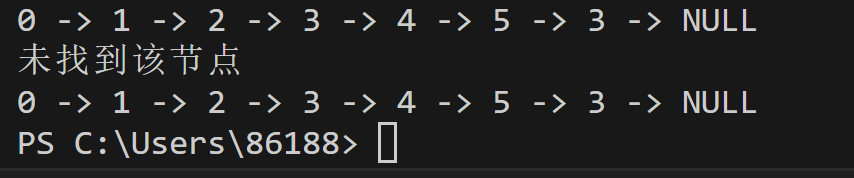
① 删除给定元素的算法。

1.1删除指定元素

数据样例： 链表为0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 3 -> NULL 目标删除数据为3 ；

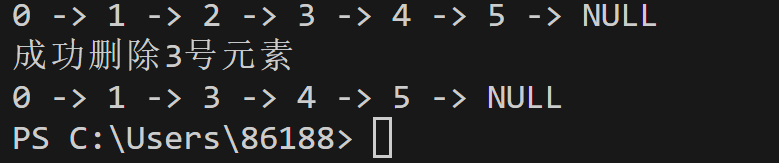


数据样例： 链表为0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 3 -> NULL 目标删除数据为8 ；

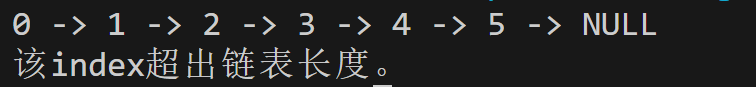


1.2删除指定位置元素

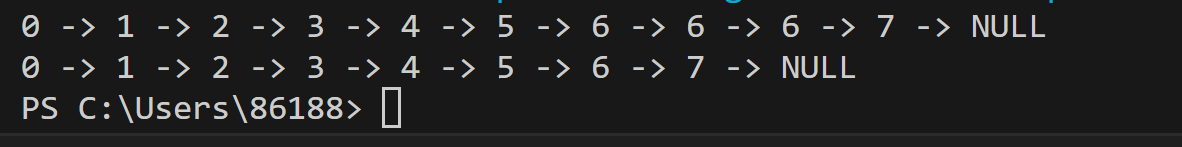
数据样例：链表为0 -> 1 -> 2 -> 3 -> 4 -> 5 -> NULL 目标删除位置 3；



数据样例：链表为0 -> 1 -> 2 -> 3 -> 4 -> 5 -> NULL 目标删除位置8；

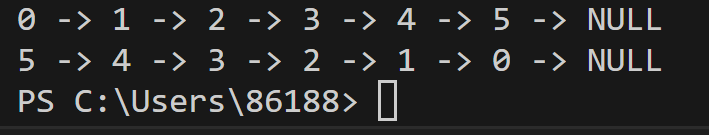
  
② 对于已排好序的线性表，删除所有重复元素的算法。

数据样例：链表为0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 6 -> 6 -> 7 -> NULL



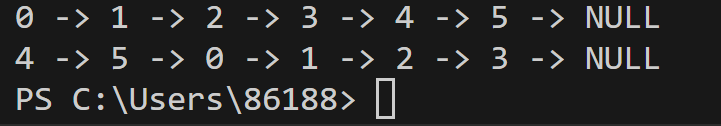
③ 线性表“逆置”算法。

数据样例：链表为0 -> 1 -> 2 -> 3 -> 4 -> 5 -> NULL

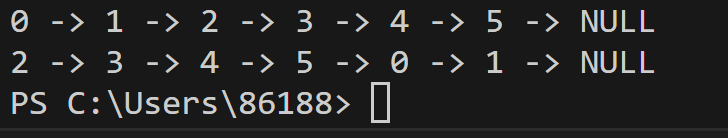


④ 线性表循环左移/右移 k 位的算法。

数据样例：链表为0 -> 1 -> 2 -> 3 -> 4 -> 5 -> NULL 右移2位（-2）

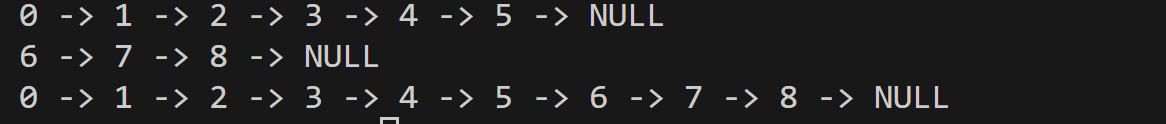


数据样例：链表为0 -> 1 -> 2 -> 3 -> 4 -> 5 -> NULL 左移8位（9）



⑤ 合并两个已排好序的线性表的算法。

数据样例：链表1为0 -> 1 -> 2 -> 3 -> 4 -> 5 -> NULL 链表2为 6 -> 7 -> 8 -> NULL



顺序储存代码：

#include <stdio.h>

#include <string.h>

#define Mix\_size 100

#define OK 1

#define ERROR 0

typedef int ElemType;

typedef int Status;

typedef struct {

    ElemType data[Mix\_size];

    int length;

} Slist;

// 打印数组

void Print\_Data(Slist s) {

    for (int i = 0; i < s.length; i++) {

        printf("%d ", s.data[i]);

    }

    printf("\n");

}

// 1.1删除给定元素

Status Delete\_Slist(Slist \*s, int Data) {

    int i = 0;

    int flag = 1;

    // 检查线性表是否为空

    if (s->length == 0) return ERROR;

    for (i = 0; i < s->length; i++) {

        if (s->data[i] == Data) {

            flag = 0;

            for (int j = i; j < s->length; j++) {

                s->data[j] = s->data[j + 1];

            }

            s->length--;

            i--;  // 回退一步 这样可以删除数组内所有的Data

        }

    }

    if (flag == 1) return ERROR;

    return OK;

}

// 1.2删除指定位置的元素

Status Delete\_Slist\_index(Slist \*s, int index) {

    int i = 0;

    int flag = 1;

    // 检查线性表是否为空

    if (s->length == 0) return ERROR;

    for (i = 0; i < s->length; i++) {

        if (i == index - 1) {  // 这次是输入index来删数据

            flag = 0;

            for (int j = i; j < s->length; j++) {

                s->data[j] = s->data[j + 1];

            }

            s->length--;

            break;

        }

    }

    if (flag == 1) return ERROR;

    return OK;

}

// 2.对于已经排好的线性表，删除所有重复元素

Status Delete\_Slist\_repeat(Slist \*s) {

    int i = 0;

    int j = 0;

    int flag = 1;

    // 检查线性表是否为空

    if (s->length == 0) return ERROR;

    // 查找重复元素

    for (i = 0; i < s->length; i++) {

        j = i + 1;

        while (s->data[i] == s->data[j]) {

            flag = 0;

            for (int k = j; k < s->length; k++) {

                s->data[k] = s->data[k + 1];

            }

            s->length--;

        }

    }

    if (flag == 1) {

        printf("未找到重复元素\n");

        return ERROR;

    } else {

        printf("已删除重复元素\n");

    }

    return OK;

}

// 3.逆置

void Inversion\_Slist(Slist \*s) {

    int i = 0;

    int l = s->length;

    int temp;  // 用于交换的临时变量

    for (i = 0; i < l / 2; i++) {

        temp = s->data[i];

        s->data[i] = s->data[l - i - 1];

        s->data[l - i - 1] = temp;

    }

}

// 4.左移右移

void R\_L\_Move\_Slist(Slist \*s, int k) {

    // k>0,右移；k<0,左移

    int l = s->length;

    int temp[l];        // 用于储存结果

    k = k % l;          // 防止k大于数组长度

    if (k < 0) k += l;  // 将负数转换为正数，简化处理

    for (int i = 0; i < l; i++) {

        temp[(i + k) % l] = s->data[i];  // 计算每个元素移动后的新位置

    }

    for (int i = 0; i < l; i++) {

        s->data[i] = temp[i];  // 将临时数组的内容复制回原数组

    }

}

// 5.合并两个线性表

Slist Combination\_Slist(Slist s, Slist t) {  // 默认为s在前面，t在后面

    int l = s.length;

    Slist new;

    for (int i = 0; i < l; i++) {

        new.data[i] = s.data[i];

    }

    for (int j = 0; j < t.length; j++) {

        new.data[l + j] = t.data[j];

    }

    new.length = l + t.length;

    return new;

}

int main() {

    Slist s1 = {{0, 1, 2, 3, 4, 5, 6}, 7};

    Slist s2 = {{7, 8, 9}, 3};

    Print\_Data(s1);

    Print\_Data(s2);

    // Print\_Data(s1);

    // if (Delete\_Slist(&s1, 3) == OK) {

    //     Print\_Data(s1);

    //     printf("成功删除元素3\n");

    // } else {

    //     printf("未找到相应元素");

    // }

    // if (Delete\_Slist\_index(&s1, 8) == OK) {

    //     Print\_Data(s1);

    //     printf("成功删除8号元素\n");

    // } else {

    //     printf("未找到相应元素");

    // }

    // if (Delete\_Slist\_repeat(&s1) == OK) Print\_Data(s1);

    // Inversion\_Slist(&s1);

    // R\_L\_Move\_Slist(&s1, 9);

    Slist new = Combination\_Slist(s1, s2);

    Print\_Data(new);

    return 0;

}

链式储存代码：

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define OK 1

#define ERROR 0

typedef int ElemType;

typedef int Status;

typedef struct NODE {  // 若自身调用，则需要用结构体的显示名称

    ElemType data;

    struct NODE \*next;

} NODE;

// 创建链表

NODE \*createNODE(int data) {

    NODE \*newNODE = (NODE \*)malloc(sizeof(NODE));

    if (newNODE == NULL) {

        printf("Error: memory allocation failed\n");

        exit(1);

    }

    newNODE->data = data;

    newNODE->next = NULL;

    return newNODE;

}

// 插入节点

void insertNODE(NODE \*\*head, int data) {

    NODE \*newNODE = createNODE(data);

    if (\*head == NULL) {

        \*head = newNODE;

    } else {

        NODE \*temp = \*head;

        while (temp->next != NULL) {

            temp = temp->next;

        }

        temp->next = newNODE;

    }

}

// 打印链表

void Print\_List(NODE \*\*head) {

    NODE \*temp = \*head;

    while (temp != NULL) {

        printf("%d -> ", temp->data);

        temp = temp->next;

    }

    printf("NULL\n");

}

// 1.1删除给定元素

void Delete\_Llist(NODE \*\*head, int Data) {

    NODE \*temp = \*head;

    NODE \*prev = NULL;

    int flag = 0;

    if (temp == NULL) printf("该链表为空\n");

    // 如果头节点本身就需要被删除

    if (temp != NULL && temp->data == Data) {

        \*head = temp->next;  // 改变头指针

        free(temp);          // 释放旧的头节点

        flag = 1;

        return;

    }

    while (temp != NULL) {

        while (temp != NULL && temp->data != Data) {

            prev = temp;

            temp = temp->next;

        }

        if (temp == NULL) {

            if (flag == 0) printf("未找到该节点\n");

            return;

        }

        prev->next = temp->next;  // 删除该节点

        NODE \*toDelete = temp;

        temp = temp->next;  // 更新临时节点,无需回退

        free(toDelete);     // 释放被删除节点的内存

        flag = 1;

    }

}

// 1.2删除指定位置的元素

void Delete\_Llist\_index(NODE \*\*head, int index) {

    NODE \*temp = \*head;

    NODE \*prev = NULL;

    int i = 0;

    if (temp == NULL) printf("该链表为空\n");

    if (index == 1) {

        \*head = temp->next;

        free(temp);

        return;

    }

    for (i = 2; temp != NULL; i++) {

        prev = temp;

        temp = temp->next;

        if (i == index) {

            prev->next = temp->next;

            printf("成功删除%d号元素\n", index);

            free(temp);

            return;

        }

    }

    printf("该index超出链表长度。");

    return;

}

// 2.对于已经排好的线性表，删除所有重复元素

void Delete\_Llist\_repeat(NODE \*\*head) {

    NODE \*prev = \*head;

    NODE \*temp = NULL;

    while (prev != NULL) {

        for (temp = prev->next; temp != NULL;) {

            if (prev->data == temp->data) {

                NODE \*ToDelete = temp;    // 临时保存重复节点

                prev->next = temp->next;  // 删除重复节点

                temp = temp->next;        // 向后移一个

                free(ToDelete);           // 释放内存

            } else

                break;

        }

        prev = prev->next;

    }

}

// 3.逆置

void Inversion\_Llist(NODE \*\*head) {

    NODE \*temp = \*head;

    NODE \*prev = NULL;

    NODE \*behind = \*head;

    while (temp != NULL) {

        behind = temp->next;

        temp->next = prev;  // temp连到前一个的前面

        prev = temp;        // 往后移一个

        temp = behind;      // 往后移一个

    }

    \*head = prev;  // 改变头结点位置

}

// 4.左移右移

void R\_L\_Move\_Llist(NODE \*\*head, int k) {

    int i = 0;  // k>0左移,k<0右移

    int l = 0;

    if (\*head == NULL || (\*head)->next == NULL) {

        return;  // 如果链表为空或只有一个元素，则不需要移动

    }

    NODE \*temp = \*head;

    NODE \*run = \*head;

    NODE \*prev = NULL;

    while (run != NULL) {

        prev = run;

        run = run->next;

        l++;  // 获得链表长度

    }

    prev->next = temp;  // 成环（循环链表）注意不能写成run=temp

                        // 因为run已经从链表中掉下去了

    for (i = 0; i < (l + k) % l; i++) {

        prev = temp;

        temp = temp->next;

    }

    \*head = temp;       // 改变头结点位置

    prev->next = NULL;  // 断开

}

// 5.合并两个线性表

NODE \*Combination\_Llist(NODE \*\*head, NODE \*\*Head) {

    NODE \*new = NULL;

    NODE \*temp = \*head;

    while (temp != NULL) {

        insertNODE(&new, temp->data);

        temp = temp->next;

    }

    temp = \*Head;

    while (temp != NULL) {

        insertNODE(&new, temp->data);

        temp = temp->next;

    }

    return new;

}

// 释放整个链表的空间

void freeList(NODE \*head) {

    NODE \*temp;

    while (head != NULL) {

        temp = head;

        head = head->next;

        free(temp);

    }

}

int main() {

    NODE \*head = NULL;

    NODE \*Head = NULL;

    int k = 0;

    int i = 0;

    for (k = 0; k < 6; k++) {

        insertNODE(&head, k);

    }

    // insertNODE(&head, 3);

    // insertNODE(&head, 6);

    // insertNODE(&head, 6);

    // insertNODE(&head, 6);

    // insertNODE(&head, 7);

    for (i = 0; i < 3; i++) {

        insertNODE(&Head, i + 6);

    }

    // insertNODE(&head, 3);

    Print\_List(&head);

    Print\_List(&Head);

    // Delete\_Llist(&head, 3);

    // Delete\_Llist\_index(&head, 8);

    // Delete\_Llist\_repeat(&head);

    // Inversion\_Llist(&head);

    // R\_L\_Move\_Llist(&head, 8);

    NODE \*new = Combination\_Llist(&head, &Head);

    Print\_List(&new);

    freeList(head);

    freeList(Head);

    return 0;

}