排序

本文档需要用到的编程相关知识点：

C语言：struct（结构体）、typedef、malloc（free）、函数做参数。

本文档的参考教材：

数据结构（C语言版） 严蔚敏

一. 归并排序

归并排序的实现参考教材283页。

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

#define OK 1
#define TRUE 1
#define FALSE 0
#define ERROR 0
#define OVERFLOW -1
#define LIST_INIT_SIZE 10  // 线性表存储空间初始分配容量
#define LISTINCREMENT 10   //线性表存储空间分配增量
#define NOTEXIT 0
typedef int Status;
typedef int ElemType;

struct SqList {
    ElemType *elem;  // 存储空间基址
    int length;      // 当前长度
    int listsize;    // 当前分配的存储容量
};

/**
 * 构造一个空的线性表L
 */
Status InitList_Sq(struct SqList &L)
{
    L.elem = (ElemType *)malloc(LIST_INIT_SIZE * sizeof(struct SqList));
    if (!L.elem)
        return (OVERFLOW);         //存储分配失败
    L.length = 0;                  // 空表长度为0
    L.listsize = LIST_INIT_SIZE;  // 初始存储容量
    return OK;
}  // InitList_Sq

/**
 * 在第i个位置之前插入数据元素e, L的长度加1
 * 要求线性表存在，1<=i<=ListLength_Sq(L)+1
 */
void ListInsert_Sq(struct SqList &L, int i, ElemType e)
```

```
{
    int j;
    ElemType *newbase;
    if (L.length + 2 > L.listsize) {  // 因为0号元素未用，所以是加2
        newbase = (ElemType *)realloc(
            L.elem, (L.listsize + LISTINCREMENT) * sizeof(ElemType));
        if (!newbase)
            exit(OVERFLOW);           // 存储分配失败
        L.elem = newbase;             // 新基址
        L.listsize += LISTINCREMENT;  // 增加存储容量
    }
    for (j = L.length + 1; j >= i; j--) {
        *(L.elem + j) = *(L.elem + j - 1);
    }
    *(L.elem + i - 1) = e;
    L.length++;  // 表长加1
}  // ListInsert_Sq

/**
 * 依次对L的每个元素调用函数visit(),
 * 一旦visit()失败，则操作失败，返回FALSE,否则返回TRUE 要求线性表存在
 */
Status ListTraverse_Sq(struct SqList L, Status (*visit)(ElemType))
{
    int i;
    for (i = 1; i <= L.length; i++) {
        if (!visit(*(L.elem + i)))
            return FALSE;
    }
    return TRUE;
}  // ListTraverse_Sq

Status visit_display_Sq(ElemType e)
{
    printf("%d ", e);
    return TRUE;
}  // visit_display_Sq

/**
 * 算法10.12，将有序的SR[i...m]和SR[m+1...n]归并为有序的TR[i...n]
 */
void Merge(int SR[], int TR[], int i, int m, int n)
{
    int j, k;
    for (j = m + 1, k = i; i <= m && j <= n; k++) {
        if (SR[i] <= SR[j]) {
            TR[k] = SR[i];
            i++;
        } else {
            TR[k] = SR[j];
            j++;
        }
    }
    if (i <= m) {
        for (j = i; j <= m; j++) {
            TR[k] = SR[j];
            k++;
        }
```

```
    }
    if (j <= n) {
        for (i = j; i <= n; i++) {
            TR[k] = SR[i];
            k++;
        }
    }
}

/**
 * 算法10.13，将SR[s...t]归并排序为TR1[s...t]
 */
void MSort(int SR[], int TR1[], int s, int t)
{
    int m;
    int TR2[100];
    if (s == t) {
        TR1[s] = SR[s];
    } else {
        m = (s + t) / 2;
        MSort(SR, TR2, s, m);
        MSort(SR, TR2, m + 1, t);
        Merge(TR2, TR1, s, m, t);
    }
}

/**
 * 算法10.14
 */
void MergeSort(SqList &L) { MSort(L.elem, L.elem, 1, L.length); }
```

归并排序测试程序：（测试程序代码放在实现代码下方即可）

```
int main()
{
    int arr[8] = {33, 12, 75, 0, 49, 67, 8, 999};
    SqList list1;
    InitList_Sq(list1);
    for (int i = 1; i < 9; i++)  // 将arr数组中的8个元素插入顺序表中
        ListInsert_Sq(list1, i + 1, arr[i - 1]);

    printf("初始线性表为：\n");
    ListTraverse_Sq(list1, visit_display_Sq);

    MergeSort(list1);
    printf("\n排序后的线性表为：\n");
    ListTraverse_Sq(list1, visit_display_Sq);

    return 0;
}
```

测试程序运行结果为：

初始线性表为：
33 12 75 0 49 67 8 999
排序后的线性表为：
0 8 12 33 49 67 75 999

二. 快速排序

快速排序的实现参考教材274页。

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

#define OK 1
#define TRUE 1
#define FALSE 0
#define ERROR 0
#define OVERFLOW -1
#define LIST_INIT_SIZE 10  // 线性表存储空间初始分配容量
#define LISTINCREMENT 10   //线性表存储空间分配增量
#define NOTEXIT 0
typedef int Status;
typedef int ElemType;

struct SqList {
    ElemType *elem;  // 存储空间基址
    int length;      // 当前长度
    int listsize;    // 当前分配的存储容量
};

/**
 * 构造一个空的线性表L
 */
Status InitList_Sq(struct SqList &L)
{
    L.elem = (ElemType *)malloc(LIST_INIT_SIZE * sizeof(struct SqList));
    if (!L.elem)
        return (OVERFLOW);         //存储分配失败
    L.length = 0;                  // 空表长度为0
    L.listsize = LIST_INIT_SIZE;  // 初始存储容量
    return OK;
}  // InitList_Sq

/**
 * 在第i个位置之前插入数据元素e, L的长度加1
 * 要求线性表存在，1<=i<=ListLength_Sq(L)+1
 */
void ListInsert_Sq(struct SqList &L, int i, ElemType e)
{
    int j;
    ElemType *newbase;
    if (L.length + 2 > L.listsize) {  // 因为0号元素未用，所以是加2
        newbase = (ElemType *)realloc(
            L.elem, (L.listsize + LISTINCREMENT) * sizeof(ElemType));
        if (!newbase)
            exit(OVERFLOW);           // 存储分配失败
        L.elem = newbase;             // 新基址
        L.listsize += LISTINCREMENT;  // 增加存储容量
    }
    for (j = L.length + 1; j >= i; j--) {
        *(L.elem + j) = *(L.elem + j - 1);
    }
```

```
    *(L.elem + i - 1) = e;
    L.length++;  // 表长加1
}  // ListInsert_Sq

/**
 * 依次对L的每个元素调用函数visit(),
 * 一旦visit()失败，则操作失败，返回FALSE,否则返回TRUE 要求线性表存在
 */
Status ListTraverse_Sq(struct SqList L, Status (*visit)(ElemType))
{
    int i;
    for (i = 1; i <= L.length; i++) {
        if (!visit(*(L.elem + i)))
            return FALSE;
    }
    return TRUE;
}  // ListTraverse_Sq

Status visit_display_Sq(ElemType e)
{
    printf("%d ", e);
    return TRUE;
}  // visit_display_Sq

/**
 * 交换两个数i, j
 */
void swap(int &i, int &j)
{
    int temp;
    temp = i;
    i = j;
    j = temp;
}

/**
 * 算法10.6（a），交换顺序表L中子表L.r[low...high]的记录，使枢轴记录到位，
 * 并返回其所在位置，此时在它之前(之后）的记录不大（小）于它。
 */
int Partition_a(SqList &L, int low, int high)
{
    int pivotkey;
    pivotkey = L.elem[low];  //第一个记录作枢轴记录
    while (low < high) {
        while (low < high && L.elem[high] >= pivotkey)
            --high;
        swap(L.elem[low], L.elem[high]);
        while (low < high && L.elem[low] <= pivotkey)
            ++low;
        swap(L.elem[low], L.elem[high]);
    }
    return low;
}

/**
 * 算法10.6(b)，对算法10.6(a)的改进算法，减少了交换的步骤
 */
int Partition(SqList &L, int low, int high)
```

```
{
    int pivotkey;
    L.elem[0] = L.elem[low];
    pivotkey = L.elem[low];  //第一个记录作枢轴记录
    while (low < high) {
        while (low < high && L.elem[high] >= pivotkey) {
            --high;
        }
        L.elem[low] = L.elem[high];
        while (low < high && L.elem[low] <= pivotkey) {
            ++low;
        }
        L.elem[high] = L.elem[low];
    }
    L.elem[low] = L.elem[0];
    return low;
}

/**
 * 算法10.7，递归法对顺序表L中的子序列L.r[low...high]作快速排序
 */
void QSort(SqList &L, int low, int high)
{
    int pivotloc;
    if (low < high) {
        pivotloc = Partition(L, low, high);
        QSort(L, low, pivotloc - 1);
        QSort(L, pivotloc + 1, high);
    }
}

/**
 * 算法10.8，快速排序
 */
void QuickSort(SqList &L) { QSort(L, 1, L.length); }
```

快速排序测试程序：（测试程序代码放在实现代码下方即可）

```
int main()
{
    int arr[8] = {33, 12, 75, 0, 49, 67, 8, 999};
    SqList list1;

    InitList_Sq(list1);
    for (int i = 1; i < 9; i++)  // 将arr数组中的8个元素插入顺序表中
        ListInsert_Sq(list1, i + 1, arr[i - 1]);
    printf("初始线性表为：\n");
    ListTraverse_Sq(list1, visit_display_Sq);

    QuickSort(list1);
    printf("\n排序后的线性表为：\n");
    ListTraverse_Sq(list1, visit_display_Sq);

    return 0;
}
```

测试程序运行结果为：

初始线性表为：
33 12 75 0 49 67 8 999
排序后的线性表为：
0 8 12 33 49 67 75 999

Author: XF

Finished time: 2019/12/8