

“数据结构”

课程设计报告

**设计题目**  排序算法及性能对比

**姓 名**  **贺洋**

**学 号**  2021218148

**专 业**  计算机科学与技术

**班 级**  计算机21-3班

**完成日期**

课程设计成绩评定表

|  |  |  |  |
| --- | --- | --- | --- |
| **设计题目** | 排序算法及性能对比 | **成绩** |  |
| **内容简介** | 首先是在各个头文件中实现希尔排序、冒泡排序、快速排序、归并排序、堆排序。在堆中开辟空间，以宏定义来确定数组的大小。同时排序数组为int类型的。  数组的初始化：以随机数初始化数组，随机数的范围为0🡪MAX(数组的长度)。  排序函数所用时间：使用QueryPerformanceCounter（）等进行计时，不用clock（）是因为精度不够。  比较次数和交换次数记录：在各算法实现的头文件中设置变量，在函数实现的过程中出现交换或者比较进行++即可。  特殊情况的测试：调用STL的sort函数来对数组进行升序或者降序的排序。在用这个有序数组去测试各个算法的性能。  更改数据量：直接修改宏定义MAX的值即可。 | | |
| **评语** | **教师签名：** | | |

**（一） 需求和规格说明**

编程实现快速排序、堆排序、希尔排序、冒泡排序、归并排序算法，学生可增加其它排序算法，并完成不同排序算法的性能对比分析。

**要求：**

① 时间性能包括平均时间性能、最好情况下的时间性能、最差情况下的时间性能等。

② 实验数据应具有说服力，包括：

规模范围要大（如从100到10000）

数据的初始特性类型要多，因而需要具有随机性；

实验数据的组数要多，即同一规模的数组要多选几种不同类型的数据来实验。

③ 算法所用时间必须是机器时间，也可以包括比较和交换元素的次数。

④ 实验分析及其结果要能以清晰的方式来描述，如数学公式或图表等。

⑤ 要给出实验的方案及其分析。

**（二） 设计**

首先分别创建各个算法的头文件，在头文件中实现各个算法。然后再各个算法对应的头文件中增加记录该算法比较次数和交换次数的变量。

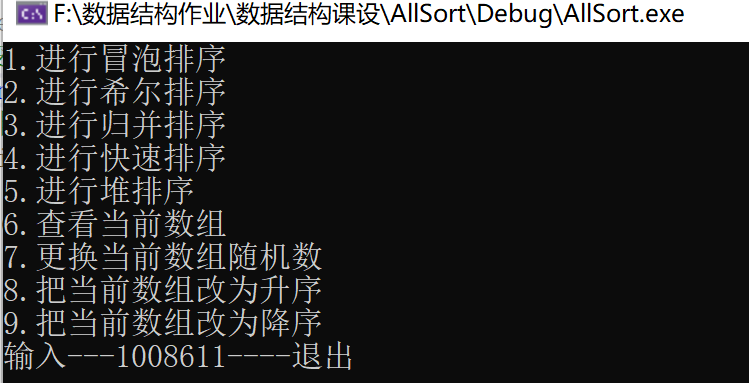
在主函数中先在堆中开辟两个以宏定义MAX大小的数组，用随机数初始化其中一个数组作为根源数组。

为了保证误差的减小，需要让各个排序算法堆同一个数组进行排序。所以不断的复制根源数组给各个排序算法排序以到达减小误差的目的。

记录各个排序算法花费的时间：刚开始是使用clock（）函数，但是其精度为10ms，明显不够，无法准确的记录各个算法进行排序的时间。故改用QueryPerformanceCounter()来记录排序算法所需要的时间。（以微妙为单位计时）。

为了测试极端的情况，增加已经升序或者降序的数组。让各个算法对其进行排序，记录时间。

**（三） 用户手册**

****

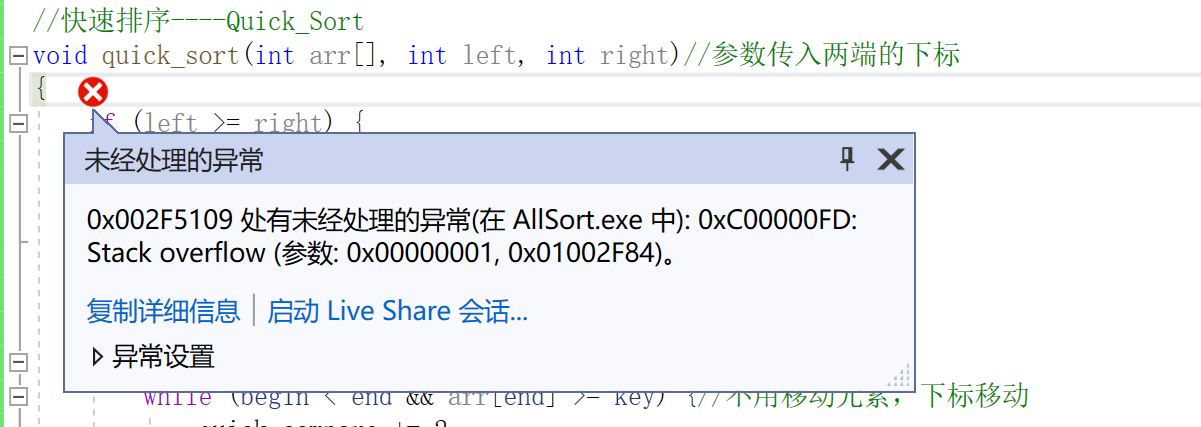
**第一次排序数组已经准备好，可以直接开始排序。**

****

**更改数组的长度即改变MAX的值**

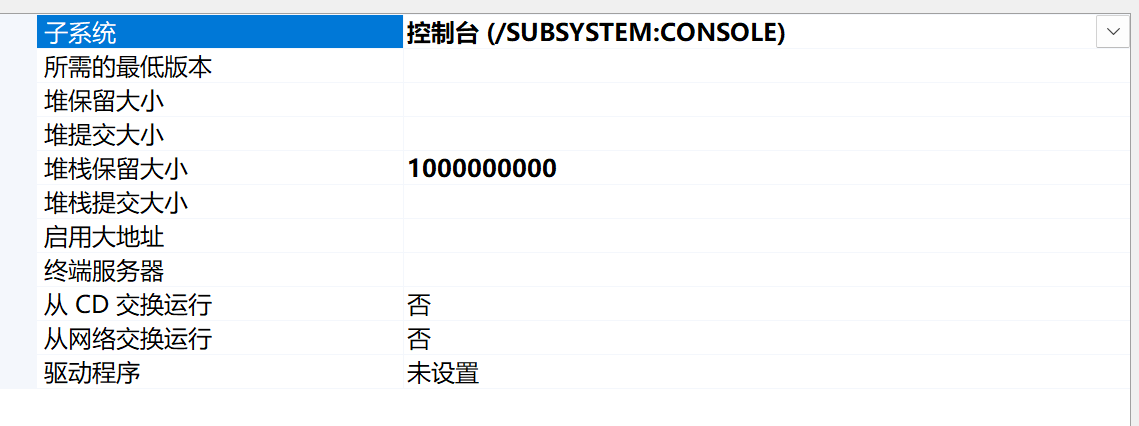
**数组中随机数为 0 – MAX**

**（四） 调试及测试**

****

**当数组中数据已经有序的极端情况时出现栈溢出的情况**

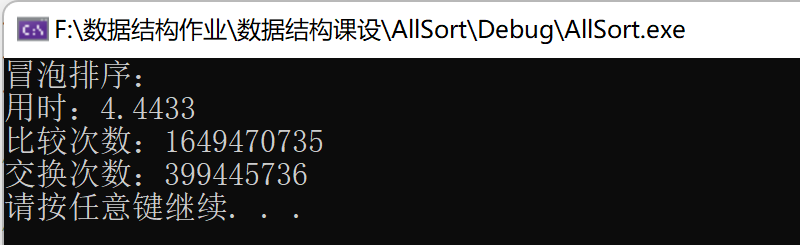
**解决方法：**

****

**手动增加栈保留的大小**

**（五） 运行实例：**

**程序运行的结果：**

****

**通过不断的更改MAX到达得到不同数据量的数组**

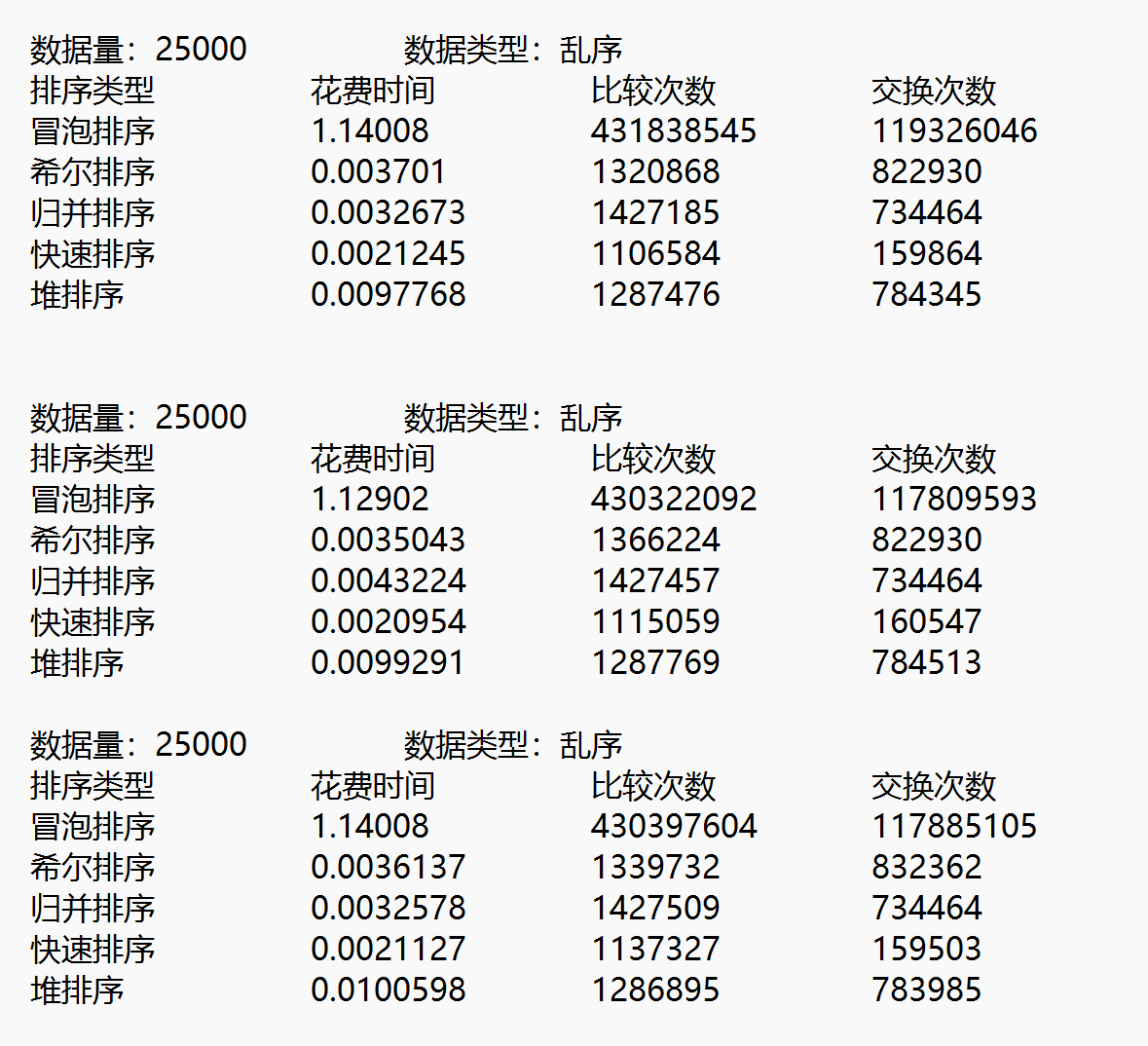
**同时同一个类型的数据可以反复的测试**

**以下给出多次数据结果的归纳：**

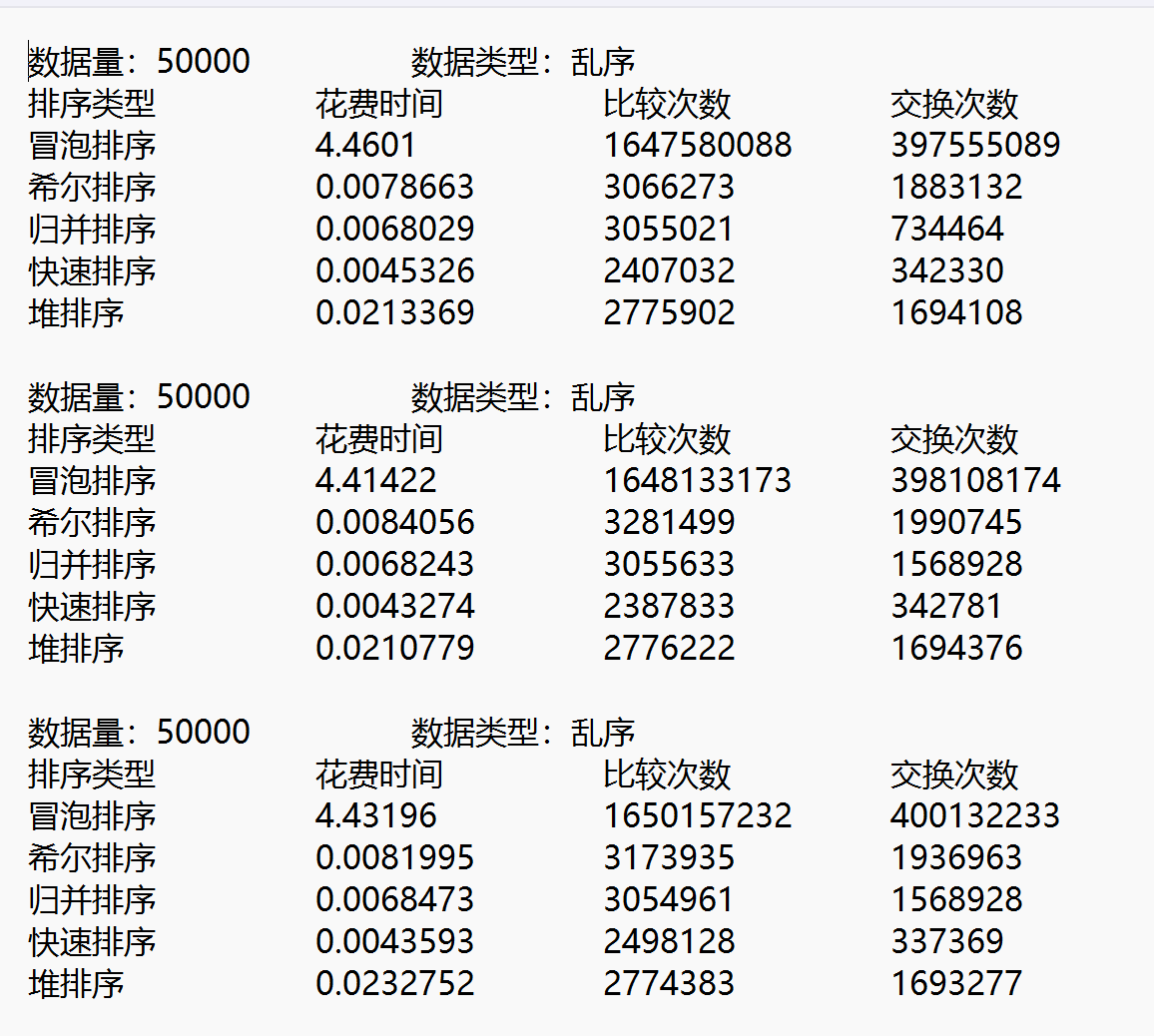
**首先是数组长度为一万时，且数组中元素为乱序各排序算法排序所消耗的时间、需要的比较次数、进行的交换次数：**

****

**数据量为两万五时：**

****

**数据量为五万时：**

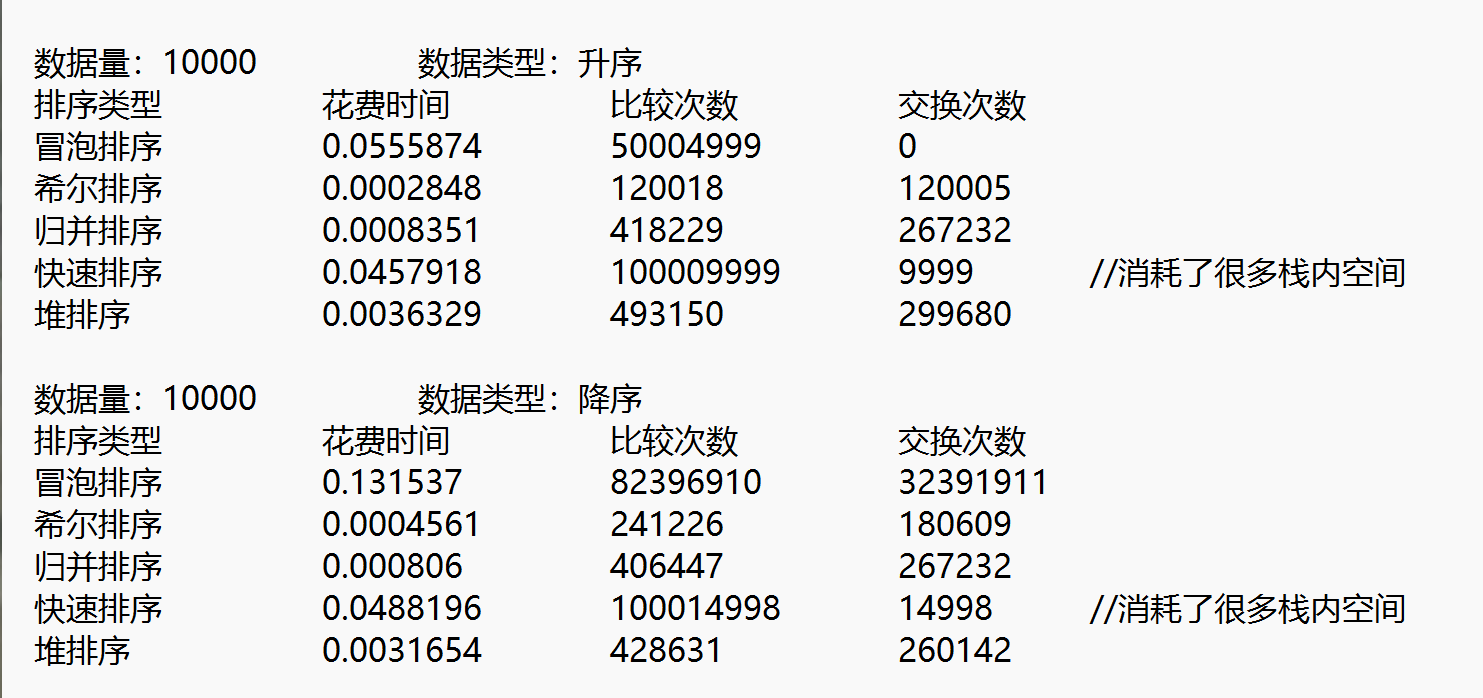
****

**以上数据均为乱序数据进行排序。**

**对乱序数据排序时各算法性能分析：**

1. 从表中可以很明显的看出，随着数据量的增大，各种排序算法所需要的时间也同样增加了，并且其交换次数、比较次数也明显增加。
2. 由表中可知，希尔排序、归并排序、快速排序、堆排序的性能都是比较好的。远远好于冒牌排序。其中很明显的看出，快速排序的性能时最优的。从比较次数和交换次数可以看出，冒泡排序的次数远远多于其他各个排序，所以冒泡排序所花费的时间是最多的。同时快速排序的交换次数和比较次数是最少的、也可以得到快速排序的花费的时间是最少的。（但不能以此为评估算法性能的唯一标准）

**对特殊有序数组的排序时算法性能分析：**

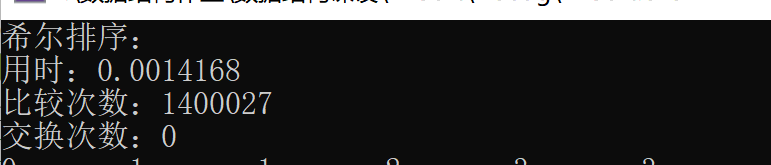
****

****

****

**分析如下：**

1. 对于已经有序的序列来说，冒泡排序可以优化对于升序序列快速退出的方案。进行优化之后时间性能极佳，超过所有排序，但这是已经排好序的情况，是极少数情况。
2. 对于归并排序来说，已经有序的数组排序在合并子表的过程几乎是一样的。单调的两个有序表的不断合并。
3. 对于希尔排序来说，是只有对比的，没有发生元素的交换。这里出现比较次数和交换次数几乎相同的原因是因为程序中将写入直接插入排序的最后确定插入元素时写入数据的情况也记录为发生了元素的交换。增加了下标i和j的下标相同则不交换后结果为



1. 对于快速排序来说，遇到了明显的大问题🡪栈溢出：快速排序是递归的，理想情况下是不断的二分结束。但是在极端的已经有序的序列中，即使不发生任何交换的情况下，也会进行n次递归。（交换次数的显示同希尔排序，没有增加判断条件）。而当n很大时，比如五万，进行五万次递归需要的内层可想而知。这是递归实现快速排序的一个弊端。可以用非递归的形式进行代替。同时快速排序的交换次数也达到了顶峰，甚至比冒泡排序的交换次数还要多，超出了int类型的最大限度，记录快速排序的变量需要改为long型的才能记录。
2. 堆排序、快速排序、归并排序都是递归实现的，但是堆排序和归并排序的递归不会像快速排序这样受到序列的很大的影响。他们的递归调用多是收到数组元素的多少的影响。

**算法优化：**

1. 由于冒泡排序花费的时间多，原因在于其比较次数和交换次数的繁琐，所以对其进行优化：1.1 首先是极端情况下，已经有序了不用再排序的情况，或者再排序中出现了已经使得数组有序的情况，此时设置布尔变量控制，若是一趟循环下来没有交换任何元素的值，则说明序列已经有序，可以直接结束循环结束排序。1.2 在编写冒泡排序的算法（因为一直都是这样写的），即为每一次是不同的多个元素一起冒泡，一次内层循环后开头元素位置确定，优化后内层循环为：
2. 由于快速排序出现了stack over的情况，再已经有序的情况数据使用快速排序进行排序时都会大量的浪费空间。例如当我给出的数据为五万时，如果数据无序，快速排序的优势依然是明显；但是要是数据是有序的，没有优化的快速排序就会递归调用五万次，栈溢出是可想而知的。所以堆快速排序的空间复杂度经行优化：减少递归调用的次数，再快速排序中将区间长度长的子列经行迭代的代替递归，区间长度短的进行递归调用。如此一来大大的减小了快速排序的时间复杂度，同时也解决了当数据量很大，数据又有序时栈溢出的问题。

**（七）心得体会**

实现了这五个排序，收获很多。首先是冒泡排序：原来我一直书写的冒泡排序都是程序和为了方便我们理解的冒泡排序的动画图是不一样的。就像在实现希尔排序的时候发现，并不是一组序列一组序列的进行直接插入排序的，而是轮换交替的实现，直到最后一轮，实现所有组的直接插入排序都实现。这里学习到了，表面的理解和实际书写代码的区别和差异。

归并排序的主要思想是先不断的划分子表，直到子表的序列长度为1，即为有序。在不断的通过两个有序表的合并的这种操作来把有序的值不断的写入到临时保留值的数组当中，最后在把这个临时数组的值写入到排序的数组当中完成排序。

快速排序因为是递归实现的，所以实现的过程并不复杂。不断的通过中间轴来调整整个表的元素的位置。再通过中间轴不断的划分左表和右表递归实现调整表的元素的位置，最后实现排序的目的。但是要注意特殊的情况下会导致多次的递归情况。这里我用迭代减少了很大部分的递归调用空间，减少了递归的调用次数，同时解决了栈溢出的问题

堆排序的思想十分的有趣。虽然是顺序表的存储结构，但是却能利用堆这个结构来实现排序。堆排序算法实现过程中需要三个函数。一个函数为基础的调整堆，即为给出堆的最大编号的值，同时给出要调整的堆顶，调整该堆顶，同时交换元素后若是下面的堆不符合堆的定义则继续交换，保证该堆顶往下的调整不会改变原来堆的性质。另一个函数来实现如何创建这个堆，因为堆的结构可以看出完全二叉树的结构，所以建成一个堆应该从完全二叉树的最右下的一个孩子的父节点开始调整堆，同时对每一个结点都编号。调整完这个堆之后，调整堆顶的变量减一，继续调整，直到调整到根结点的位置，整个变成一个堆。最后是不断的输出堆顶的元素，把数据在顺序表中变为有序数列。大根堆对应的是升序，小根堆对应的是降序。

**（八）对课程设计的建议**

希望课程设计的时间可以安排得更合理。

**（九）附录⎯⎯源程序**

Shell\_sort.h

#pragma once

#include <iostream>

//全局变量记录比较次数，数据交换次数

int shell\_compare, shell\_swap;

//希尔排序----shell\_sort

void shell\_sort(int arr[], int n)

{

int i, j, inc, key;

for (inc = n / 2; inc > 0; inc /= 2)//每组间距---直到间距为1

{

shell\_compare++;

//每一趟都插入排序

for (i = inc; i < n; i++)//不是一次完成一组，而是根据间隔来。

{

shell\_compare++;

key = arr[i];

for (j = i; j >= inc && key < arr[j - inc]; j -= inc)

{

arr[j] = arr[j - inc];//找出空位

shell\_compare += 2;

shell\_swap++;

}

arr[j] = key;//写入值

shell\_swap++;

}

}

}

Bubble\_sort.h

#pragma once

#include <iostream>

//全局变量记录比较次数，数据交换次数

int bubble\_compare, bubble\_swap;

//冒泡排序---bubble\_sort

void bubble\_sort(int arr[], int n)

{

bool flage = false;

for (int i = 0; i < n - 1; i++)

{

flage = false;

bubble\_compare++;//

for (int j = i + 1; j < n; j++)

{

bubble\_compare++;//

if (arr[i] > arr[j])

{

flage = true;

bubble\_compare++;//

bubble\_swap++;//

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

if (flage == false) return;

}

}

Merge\_sort.h

#pragma once

#include <iostream>

//归并排序---merge\_sort

//全局变量记录比较次数，数据交换次数

int merge\_compare, merge\_swap;

//合并表

void merge(int arr[], int tempArr[], int left, int mid, int right)

{

int L\_m = left, R\_m = mid + 1, temp = left;//两表的第一个元素的下标， temp记录写入新表的元素的下标

//合并递增有序的两个表，写入新的数组tempArr中

while (L\_m <= mid && R\_m <= right)

{

if (arr[L\_m] < arr[R\_m])tempArr[temp++] = arr[L\_m++];

else

tempArr[temp++] = arr[R\_m++];

merge\_compare += 3;

merge\_swap++;

}

//左表未完

while (L\_m <= mid)

{

tempArr[temp++] = arr[L\_m++];

merge\_swap++;

merge\_compare++;

}

//右表未完

while (R\_m <= right)

{

tempArr[temp++] = arr[R\_m++];

merge\_swap++;

merge\_compare++;

}

//把数据移动到数组中

while (left <= right)

{

merge\_swap++;

merge\_compare++;

arr[left] = tempArr[left];

left++;

}

}

//递归划分子表

void msort(int arr[], int tempArr[], int left, int right)

{

if (left < right)//表的长度大于1

{

merge\_compare++;

int mid = (left + right) / 2;//二分划分

msort(arr, tempArr, left, mid);//左半区

msort(arr, tempArr, mid + 1, right);//右半区划分

merge(arr, tempArr, left, mid, right);//合并子表

}

}

Heap\_sort

#pragma once

#include <iostream>

//堆排序-----Heap\_Sort

//全局变量记录比较次数，数据交换次数

int heap\_compare, heap\_swap;

//交换值的函数

void swap(int arr[], int i, int j)

{

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

//调整堆的操作

void heapify(int tree[], int n, int i)//n为结点数目，亦为数组元素数目。i为从这个顶点开始调整堆

{

if (i >= n)

{

heap\_compare++;

return;//递归出口

}

int c1 = 2 \* i + 1;//第一个单元仍然存储元素

int c2 = 2 \* i + 2;

int imax = i;//最大值的下标记录

if (c1 < n && tree[c1] > tree[imax])

{

imax = c1;

heap\_compare += 2;

heap\_swap++;

}

if (c2 < n && tree[c2] > tree[imax])

{

imax = c2;//找出最大值的下标

heap\_compare += 2;

heap\_swap++;

}

if (imax != i)

{

heap\_compare++;

heap\_swap++;

swap(tree, imax, i);

heapify(tree, n, imax);

}

}

//创建堆

void build\_heap(int tree[], int n)

{

int last\_node = n - 1;

int parent = (last\_node - 1) / 2;

for (int i = parent; i >= 0; i--)

{

heap\_compare++;

heapify(tree, n, i);//从i开始倒着往上减着去做堆调整

}

}

//排序堆

void heap\_sort(int tree[], int n)

{

build\_heap(tree, n);//首先创建堆

for (int i = n - 1; i >= 0; i--)

{

heap\_compare++;

swap(tree, i, 0);//交换堆顶值到末尾

heapify(tree, i, 0);//调整堆顶

}

}

Quick\_sort

#pragma once

#include <iostream>

//全局变量记录比较次数，数据交换次数

long long quick\_compare, quick\_swap;

//快速排序----Quick\_Sort

void quick\_sort(int arr[], int left, int right)//参数传入两端的下标

{

if (left >= right) {

quick\_compare++;

return;

}

//int key = arr[left];

//int begin = left;

//int end = right;//代替左右下标，因为需要保留值

while (left < right) {//新加对长子列经行迭代实现快速排序的部分

int key = arr[left];

int begin = left;

int end = right;//代替左右下标，因为需要保留值

while (begin != end) {

while (begin < end && arr[end] >= key) {//不用移动元素，下标移动

quick\_compare += 2;

end--;

}

if (end > begin) {

quick\_compare++;

quick\_swap++;

arr[begin] = arr[end];

}

while (begin < end && arr[begin] <= key) {//不用移动元素，下标移动

quick\_compare += 2;

begin++;

}

if (begin < end) {

quick\_compare++;

quick\_swap++;

arr[end] = arr[begin];

}

quick\_compare++;

}

quick\_swap++;

arr[begin] = key;//左右下标相遇的位置，存储中间轴的值

if (begin - left < right - begin)

{

quick\_sort(arr, left, begin - 1);//子表快速排序递归

left = begin + 1;

}

else

{

quick\_sort(arr, begin + 1, right);//子表快速排序递归

right = begin - 1;

}

}

}

DubSort.cpp

#include "bubble\_sort.h"

#include "heap\_sort.h"

#include "merge\_sort.h"

#include "quick\_sort.h"

#include "shell\_sort.h"

#include <cstdlib>

#include <iomanip> //精度控制函数位于该库中

#include <algorithm> //sort排序函数

#include <windows.h>//更换计时

#define MAX 50000

using namespace std;

void myRandom(int arr[], int wide, int n);

void MyPrintf(int arr[], int n);

void copyArr(int arr[], int useArr[], int n);

void init(void);

bool cmp2(int a, int b);

bool cmp1(int a, int b);

int main(void)

{

int \*arr = new int[MAX], \*useArr = new int[MAX];//开辟空间

clock\_t star, finish; //>= 10ms

LARGE\_INTEGER t1, t2, tc;

myRandom(arr, MAX, MAX);//随机数初始化数组

//quick\_sort(arr, 0, MAX\_FOUR - 1);

int select = -1;

while (select != 1008611)

{

init();

cin >> select;

system("cls");

switch (select)

{

case 1://冒泡

{

star = 0;

finish = 0;

bubble\_compare = 0;

bubble\_swap = 0;

copyArr(arr, useArr, MAX);

//star = clock();

QueryPerformanceFrequency(&tc);

QueryPerformanceCounter(&t1);

bubble\_sort(useArr, MAX);

//finish = clock();

QueryPerformanceCounter(&t2);

cout << "冒泡排序：" << endl;

cout << "用时：" << (double)(t2.QuadPart - t1.QuadPart) / (double)tc.QuadPart << endl;

cout << "比较次数：" << bubble\_compare << endl;

cout << "交换次数：" << bubble\_swap << endl;

//MyPrintf(useArr, MAX);

break;

}

case 2://希尔排序

{

star = 0;

finish = 0;

shell\_compare = 0;

shell\_swap = 0;

copyArr(arr, useArr, MAX);

//star = clock();

QueryPerformanceFrequency(&tc);

QueryPerformanceCounter(&t1);

shell\_sort(useArr, MAX);

//finish = clock();

QueryPerformanceCounter(&t2);

cout << "希尔排序：" << endl;

cout << "用时：" << (double)(t2.QuadPart - t1.QuadPart) / (double)tc.QuadPart << endl;

cout << "比较次数：" << shell\_compare << endl;

cout << "交换次数：" << shell\_swap << endl;

//MyPrintf(useArr, MAX);

break;

}

case 3://归并

{

star = 0;

finish = 0;

merge\_compare = 0;

merge\_swap = 0;

copyArr(arr, useArr, MAX);

//star = clock();

QueryPerformanceFrequency(&tc);

QueryPerformanceCounter(&t1);

merge\_sort(useArr, MAX);

//finish = clock();

QueryPerformanceCounter(&t2);

cout << "归并排序：" << endl;

cout << "用时：" << (double)(t2.QuadPart - t1.QuadPart) / (double)tc.QuadPart << endl;

cout << "比较次数：" << merge\_compare << endl;

cout << "交换次数：" << merge\_swap << endl;

//MyPrintf(useArr, MAX);

break;

}

case 4://快速

{

star = 0;

finish = 0;

quick\_compare = 0;

quick\_swap = 0;

copyArr(arr, useArr, MAX);

//star = clock();

QueryPerformanceFrequency(&tc);

QueryPerformanceCounter(&t1);

quick\_sort(useArr, 0, MAX - 1);

QueryPerformanceCounter(&t2);

//finish = clock();

cout << "快速排序：" << endl;

cout << "用时：" << (double)(t2.QuadPart - t1.QuadPart) / (double)tc.QuadPart << endl;

cout << "比较次数：" << quick\_compare << endl;

cout << "交换次数：" << quick\_swap << endl;

//MyPrintf(useArr, MAX);

break;

}

case 5:

{

star = 0;

finish = 0;

heap\_compare = 0;

heap\_swap = 0;

copyArr(arr, useArr, MAX);

//star = clock();

QueryPerformanceFrequency(&tc);

QueryPerformanceCounter(&t1);

heap\_sort(useArr, MAX);

//finish = clock();

QueryPerformanceCounter(&t2);

cout << "堆排序：" << endl;

cout << "用时：" << (double)(t2.QuadPart - t1.QuadPart) / (double)tc.QuadPart << endl;

cout << "比较次数：" << heap\_compare << endl;

cout << "交换次数：" << heap\_swap << endl;

//MyPrintf(useArr, MAX);

break;

}

case 6:

{

MyPrintf(arr, MAX);

break;

}

case 7:

{

myRandom(arr, MAX, MAX);

cout << "OK" << endl;

break;

}

case 8:

{

sort(arr, arr + MAX, cmp2);

cout << "OK" << endl;

break;

}

case 9:

{

sort(arr, arr + MAX, cmp1);

cout << "OK" << endl;

break;

}

default:

break;

}

system("pause");

system("cls");

}

delete[] arr;

delete[] useArr;

return 0;

}

//排序函数

bool cmp1(int a, int b)

{

return a > b;

}

bool cmp2(int a, int b)

{

return a < b;

}

//随机数生成

void myRandom(int arr[], int wide, int n)

{

srand((int)time(0)); // 产生随机种子 把0换成NULL也行

for (int i = 0; i < n; i++)

arr[i] = rand() % wide;

}

//打印数组元素

void MyPrintf(int arr[], int n)

{

long long c = 1;

for (int i = 0; i < n; i++)

{

cout << arr[i] << "\t";

if (c % 9 == 0) cout << endl;

c++;

}

}

//复制数组

void copyArr(int arr[], int useArr[], int n)

{

for (int i = 0; i < n; i++)

useArr[i] = arr[i];

}

//打印提示信息

void init(void)

{

cout << "1.进行冒泡排序" << endl;

cout << "2.进行希尔排序" << endl;

cout << "3.进行归并排序" << endl;

cout << "4.进行快速排序" << endl;

cout << "5.进行堆排序" << endl;

cout << "6.查看当前数组" << endl;

cout << "7.更换当前数组随机数" << endl;

cout << "8.把当前数组改为升序" << endl;

cout << "9.把当前数组改为降序" << endl;

cout << "输入---1008611----退出" << endl;

}