

数据结构课程设计

——八种排序算法以及比较

项目说明文档

作 者 姓 名： 韦世贸

学 号： 2351131

指 导 老 师： 软件学院

专 业： 软件工程

同济大学

Tongji University

1 分析

1.1 背景分析

排序算法是计算机科学与技术领域上广泛被使用的术语。所谓排序，就是使一串记录，按照其中的某个或某些关键字的大小，递增或递减的排列起来的操作。排序是程序设计中非常重要的内容，其算法种类繁多，排序算法的选择会直接影响到计算的效率。由于实际工作中处理的数据量巨大，所以排序算法对算法本身的速度要求很高。

1.2 功能分析

排序的主要性能指标有算法执行时间、算法本身的复杂度和算法的稳定度。常用的排序算法主要有：插入排序、选择排序、冒泡排序、快速排序和堆排序等，本文主要介绍这几种常用的排序算法原理、算法实现和排序性能，并比较他们的异同，最后根据分析的结果总结出在不同情况选择不同的排序算法以提高排序效率。

2 设计

2.1 数据结构设计

如上功能分析所述，排序算法一般使用数组存储较多也较为常见，本文中涉及的算法均是使用动态分配内存的数组进行排序。

2.2 类结构设计

这个算法比较程序中包括一个算法比较类和较多的函数——定义该类为排序类（Sort）。可执行全部八种排序操作，在排序系统里按照用户输入进行调用

2.3 成员与操作设计

Sort类的定义

class Sort

{

public:

Sort() : randomNumber(0) {} // 构造函数，初始化随机数个数

~Sort() {} // 析构函数

void initialInterface(); // 初始化界面，获取随机数个数

// 各种排序算法函数声明

long long bubbleSort(int\* data, double& sortTime);

long long selectionSort(int\* data, double& sortTime);

long long directInsertionSort(int\* data, double& sortTime);

long long shellSort(int\* data, double& sortTime);

long long quickSort(int\* data, double& sortTime, int left, int right);

long long heapSort(int\* data, double& sortTime);

long long mergeSort(int\* data, double& sortTime);

int radixSort(int\* data, double& sortTime);

// 归并排序的辅助函数

// 递归和合并

void merge(int\* array, int\* temp, int left, int mid, int right, long long& exchangeCount);

void mergeSortRecursive(int\* array, int\* temp, int left, int right, long long& exchangeCount);

// 辅助函数声明

long long siftDown(int parent, int\* data, int number);

int getMaxBit(int\* data, int n);

int randomNumber; // 随机数个数

};

3 实现

3.1 冒泡排序功能的实现

3.1.1 冒泡排序功能简介

在每一趟的冒泡排序中，依次比较相邻两个关键字大小，若为反序则进行交换，经过一次冒泡排序以后最小的数据已经被排好了，再进行同样的方式进行n-1趟直到全部排好序。

3.1.2 冒泡排序功能核心代码

// 冒泡排序，返回交换次数和排序时间

long long Sort::bubbleSort(int\* data, double& sortTime)

{

int i, j, temp;

long long exchangeCount = 0;

int\* array = new int[randomNumber];

for (i = 0; i < randomNumber; i++) {

array[i] = data[i];

}

clock\_t startTime = clock();

for (i = 1; i < randomNumber; i++) {

for (j = randomNumber - 1; j >= i; j--) {

if (array[j] < array[j - 1]) {

temp = array[j];

array[j] = array[j - 1];

array[j - 1] = temp;

exchangeCount++;

}

}

}

clock\_t endTime = clock();

sortTime = (double)(endTime - startTime) / CLOCKS\_PER\_SEC;

delete[] array;

return exchangeCount;

}

3.2 选择排序功能的实现

3.2.1 选择排序功能简介

第一次从待排序的数据元素中选出最小（或最大）的一个元素，存放在序列的起始位置，然后再从剩余的未排序元素中寻找到最小（大）元素，然后放到已排序的序列的末尾。以此类推，直到全部待排序的数据元素的个数为零。选择排序是不稳定的排序方法。

3.2.2 选择排序功能核心代码

// 选择排序，返回交换次数和排序时间

long long Sort::selectionSort(int\* data, double& sortTime)

{

int i, j, minIndex, temp;

long long exchangeCount = 0;

int\* array = new int[randomNumber];

for (i = 0; i < randomNumber; i++) {

array[i] = data[i];

}

clock\_t startTime = clock();

for (i = 0; i < randomNumber - 1; i++) {

minIndex = i;

for (j = i + 1; j < randomNumber; j++) {

if (array[j] < array[minIndex]) {

minIndex = j;

}

}

if (minIndex != i) {

temp = array[i];

array[i] = array[minIndex];

array[minIndex] = temp;

exchangeCount++;

}

}

clock\_t endTime = clock();

sortTime = (double)(endTime - startTime) / CLOCKS\_PER\_SEC;

delete[] array;

return exchangeCount;

}

3.3 直接插入排序功能的实现

3.3.1 直接插入排序功能简介

第一趟比较前两个数，然后把第二个数按大小插入到有序表中； 第二趟把第三个数据与前两个数从后向前扫描，把第三个数按大小插入到有序表中；依次进行下去，进行了(n-1)趟扫描以后就完成了整个排序过程。

直接插入排序是由两层嵌套循环组成的。外层循环标识并决定待比较的数值。内层循环为待比较数值确定其最终位置。直接插入排序是将待比较的数值与它的前一个数值进行比较，所以外层循环是从第二个数值开始的。当前一数值比待比较数值大的情况下继续循环比较，直到找到比待比较数值小的并将待比较数值置入其后一位置，结束该次循环。

3.3.2 直接插入排序功能核心代码

// 直接插入排序，返回交换次数和排序时间

long long Sort::directInsertionSort(int\* data, double& sortTime)

{

int i, j, temp;

long long exchangeCount = 0;

int\* array = new int[randomNumber];

for (i = 0; i < randomNumber; i++) {

array[i] = data[i];

}

clock\_t startTime = clock();

for (i = 1; i < randomNumber; i++) {

temp = array[i];

j = i - 1;

while (j >= 0 && array[j] > temp) {

array[j + 1] = array[j];

j--;

exchangeCount++;

}

array[j + 1] = temp;

}

clock\_t endTime = clock();

sortTime = (double)(endTime - startTime) / CLOCKS\_PER\_SEC;

delete[] array;

return exchangeCount;

}

3.4 希尔排序功能的实现

3.4.1 希尔排序功能简介

先取一个小于n的整数d1作为第一个增量，把文件的全部记录分组。所有距离为d1的倍数的记录放在同一个组中。先在各组内进行直接插入排序；然后，取第二个增量d2 。该方法实质上是一种分组插入方法

比较相隔较远距离（称为增量）的数，使得数移动时能跨过多个元素，则进行一次比较就可能消除多个元素交换。算法先将要排序的一组数按某个增量d分成若干组，每组中记录的下标相差d.对每组中全部元素进行排序，然后再用一个较小的增量对它进行，在每组中再进行排序。当增量减到1时，整个要排序的数被分成一组，排序完成。

3.4.2 希尔排序功能核心代码

// 希尔排序，返回交换次数和排序时间

long long Sort::shellSort(int\* data, double& sortTime)

{

int gap, i, j, temp;

long long exchangeCount = 0;

int\* array = new int[randomNumber];

for (i = 0; i < randomNumber; i++) {

array[i] = data[i];

}

clock\_t startTime = clock();

for (gap = randomNumber / 2; gap > 0; gap /= 2) {

for (i = gap; i < randomNumber; i++) {

temp = array[i];

j = i;

while (j >= gap && array[j - gap] > temp) {

array[j] = array[j - gap];

j -= gap;

exchangeCount++;

}

array[j] = temp;

}

}

clock\_t endTime = clock();

sortTime = (double)(endTime - startTime) / CLOCKS\_PER\_SEC;

delete[] array;

return exchangeCount;

}

3.5 快速排序功能的实现

3.5.1 快速排序功能简介

快速排序算法通过多次比较和交换来实现排序，其排序流程如下：

(1)首先设定一个分界值，通过该分界值将数组分成左右两部分。

(2)将大于或等于分界值的数据集中到数组右边，小于分界值的数据集中到数组的左边。此时，左边部分中各元素都小于或等于分界值，而右边部分中各元素都大于或等于分界值。

(3)然后，左边和右边的数据可以独立排序。对于左侧的数组数据，又可以取一个分界值，将该部分数据分成左右两部分，同样在左边放置较小值，右边放置较大值。右侧的数组数据也可以做类似处理。

(4)重复上述过程，可以看出，这是一个递归定义。通过递归将左侧部分排好序后，再递归排好右侧部分的顺序。当左、右两个部分各数据排序完成后，整个数组的排序也就完成了。

3.5.2 快速排序功能核心代码

// 快速排序，返回交换次数和排序时间

long long Sort::quickSort(int\* data, double& sortTime, int left, int right)

{

int i = left, j = right, temp;

int pivot = data[left];

long long exchangeCount = 0;

clock\_t startTime = clock();

while (i < j) {

while (i < j && data[j] >= pivot) {

j--;

}

if (i < j) {

data[i] = data[j];

exchangeCount++;

}

while (i < j && data[i] < pivot) {

i++;

}

if (i < j) {

data[j] = data[i];

exchangeCount++;

}

}

data[i] = pivot;

if (left < i - 1) {

exchangeCount += quickSort(data, sortTime, left, i - 1);

}

if (i + 1 < right) {

exchangeCount += quickSort(data, sortTime, i + 1, right);

}

clock\_t endTime = clock();

sortTime = (double)(endTime - startTime) / CLOCKS\_PER\_SEC;

return exchangeCount;

}

3.6 堆排序系统的实现

3.6.1 堆排序系统简介

在堆的数据结构中，堆中的最小值位于根节点。堆中定义以下几种操作：最小堆调整：将堆的末端子节点作调整，使得子节点永远大于父节点。创建最小堆：将堆中的所有数据重新排序。堆排序：移除位在第一个数据的根节点，并做最小堆调整的递归运算

3.6.2 堆排序系统核心代码

// 堆排序，返回交换次数和排序时间

long long Sort::heapSort(int\* data, double& sortTime)

{

int i, temp;

long long exchangeCount = 0;

int\* array = new int[randomNumber + 1]; // 堆从下标1开始

for (i = 0; i < randomNumber; i++) {

array[i + 1] = data[i];

}

clock\_t startTime = clock();

// 建堆

for (i = randomNumber / 2; i >= 1; i--) {

exchangeCount += siftDown(i, array, randomNumber);

}

// 排序

for (i = randomNumber; i >= 2; i--) {

temp = array[1];

array[1] = array[i];

array[i] = temp;

exchangeCount += siftDown(1, array, i - 1);

}

clock\_t endTime = clock();

sortTime = (double)(endTime - startTime) / CLOCKS\_PER\_SEC;

delete[] array;

return exchangeCount;

}

3.7 归并排序系统的实现

3.7.1 归并排序系统简介

归并排序是建立在归并操作上的一种有效，稳定的排序算法，该算法是采用分治法的一个非常典型的应用。将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。若将两个有序表合并成一个有序表，称为二路归并。

3.7.2 归并排序系统核心代码

// 归并排序，返回交换次数和排序时间

long long Sort::mergeSort(int\* data, double& sortTime)

{

int i;

long long exchangeCount = 0;

int\* array = new int[randomNumber];

for (i = 0; i < randomNumber; i++) {

array[i] = data[i];

}

clock\_t startTime = clock();

int\* temp = new int[randomNumber];

mergeSortRecursive(array, temp, 0, randomNumber - 1, exchangeCount);

clock\_t endTime = clock();

sortTime = (double)(endTime - startTime) / CLOCKS\_PER\_SEC;

delete[] temp;

delete[] array;

return exchangeCount;

}

3.8 基数排序系统的实现

3.8.1 基数排序系统简介

基数排序属于“分配式排序”，又称“桶子法”。顾名思义，它是透过关键值，将要排序的元素分配至某些“桶”中，藉以达到排序的作用，基数排序法是属于稳定性的排序，其时间复杂度为O (nlog(r)m)，其中r为所采取的基数，而m为堆数，在某些时候，基数排序法的效率高于其它的稳定性排序法。

3.8.2 基数排序系统核心代码

// 基数排序，返回交换次数和排序时间

int Sort::radixSort(int\* data, double& sortTime)

{

int i, j, k, digit, temp;

int\* array = new int[randomNumber];

for (i = 0; i < randomNumber; i++) {

array[i] = data[i];

}

clock\_t startTime = clock();

int maxDigit = getMaxBit(array, randomNumber);

int\* count = new int[10];

int\* tempArray = new int[randomNumber];

for (digit = 1; digit <= maxDigit; digit++) {

// 清零计数器

for (i = 0; i < 10; i++) {

count[i] = 0;

}

// 统计每个桶中的元素个数

for (i = 0; i < randomNumber; i++) {

k = (array[i] / digit) % 10;

count[k]++;

}

// 计算每个桶的起始位置

for (i = 1; i < 10; i++) {

count[i] += count[i - 1];

}

// 将元素放入临时数组中

for (i = randomNumber - 1; i >= 0; i--) {

k = (array[i] / digit) % 10;

tempArray[count[k] - 1] = array[i];

count[k]--;

}

// 将临时数组复制回原数组

for (i = 0; i < randomNumber; i++) {

array[i] = tempArray[i];

}

}

clock\_t endTime = clock();

sortTime = (double)(endTime - startTime) / CLOCKS\_PER\_SEC;

delete[] tempArray;

delete[] count;

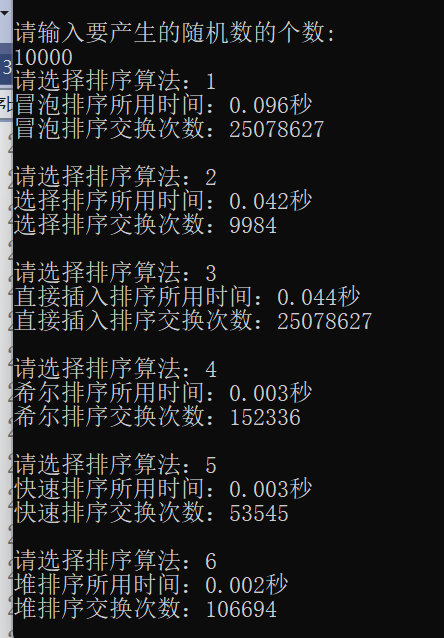
delete[] array;

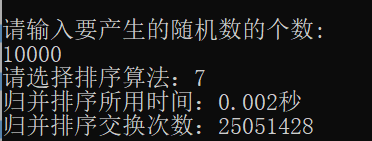
return 0;

}

4.总体功能截屏

测试一：随机数10000：





5.设计小结：

通过这个程序的设计，我清晰的看出了八种排序方式的区别，同时还把这八种排序都复习了一遍，也完成了用代码的实现