项目说明文档

数据结构课程设计

——8种排序算法的比较案例

作 者 姓 名： 张诚睿

学 号： 2150998

指 导 教 师： 张颖

学院、 专业： 软件学院 软件工程

同济大学

Tongji University

目录

[1 分析 1](#_Toc122645392)

[1.1 项目功能分析 1](#_Toc122645393)

[2 设计 1](#_Toc122645394)

[2.1 数据结构设计 1](#_Toc122645395)

[2.2 类具体成员与操作设计 1](#_Toc122645396)

[2.4 main函数设计 2](#_Toc122645397)

[3 排序实现与分析 3](#_Toc122645398)

[3.1 排序情况总览 3](#_Toc122645399)

[3.2 冒泡排序 3](#_Toc122645400)

[3.2.1 排序方法 3](#_Toc122645401)

[3.2.2 核心代码 3](#_Toc122645402)

[3.3选择排序 4](#_Toc122645403)

[3.3.1 排序方法 4](#_Toc122645404)

[3.3.2 核心代码 4](#_Toc122645405)

[3.4插入排序 5](#_Toc122645406)

[3.4.1 排序方法 5](#_Toc122645407)

[3.4.2 核心代码 5](#_Toc122645408)

[3.5希尔排序 5](#_Toc122645409)

[3.5.1 排序方法 5](#_Toc122645410)

[3.5.2 核心代码 6](#_Toc122645411)

[3.6快速排序 6](#_Toc122645412)

[3.6.1 排序方法 6](#_Toc122645413)

[3.6.2 核心代码 6](#_Toc122645414)

[3.7堆排序 7](#_Toc122645415)

[3.7.1 排序方法 7](#_Toc122645416)

[3.7.2 核心代码 8](#_Toc122645417)

[3.8归并排序 8](#_Toc122645418)

[3.8.1 排序方法 8](#_Toc122645419)

[3.8.2 核心代码 8](#_Toc122645420)

[3.9基数排序 9](#_Toc122645421)

[3.9.1 排序方法 9](#_Toc122645422)

[3.9.2 核心代码 9](#_Toc122645423)

[4 测试 11](#_Toc122645424)

[4.1 功能测试 11](#_Toc122645425)

[4.2 出错测试 11](#_Toc122645426)

# 1 分析

## 1.1 项目功能分析

随机函数产生一百，一千，一万和十万个随机数，用8种不同的排序方法排序，并统计每种排序所花费的排序时间和交换次数。其中，随机数的个数由用户定义，系统产生随机数。

# 2 设计

## 2.1 数据结构设计



对整个排序系统的核心数据结构设计如上，通过类-Rank实现控制排序系统，用data申请length长度的动态数组，用于初始化随机待排序数据，times记录交换元素的次数，begin和end用于统计排序所用时间。虽然堆排序涉及堆的结构，但本次实现选择直接在函数内完成，不再单独设计一个类。

## 2.2 类具体成员与操作设计

class Rank {

private:

int length; // 数据数量

int\* data; // 待排序数组

long long int times;

time\_t begin, end;

public:

Rank() :data(NULL), length(0), begin(0), end(0), times(0) { srand(unsigned int(time(NULL))); };

~Rank() { delete[]data; };

void StartTime();

void EndTime();

void Choose();

void InitData();

void Ranking();

void BubbleSort();

void SelectSort();

void InsertionSort();

void ShellSort();

void QuickSort(); // 总函数

void QuickSort(int startPos, int endPos); // 实现函数

void HeapSort();

void MergeSort();

void RadixSort();

void out();

void initHeap(int\* content);

void siftUp(int start, int\* content); // 上滑调整

void siftDown(int start, int end, int\* content); // 下滑调整

void Merge(int\* L1, int\* L2, int left, int right);

void MergeCalc(int\* L1, int\* L2, int left, int mid, int right);

void Radix(int\* List, int left, int right, int Weigh);

int Digit(int num, int Weigh);

};

## 2.4 main函数设计

首先建立Rank对象，之后获取总的数据量，进入排序系统根据指令不断排序，退出排序系统后，返回主函数，主函数返回0结束程序。

# 3 排序实现与分析

## 3.1 排序情况总览

以下是对各种排序的交换次数和效率情况进行的统计。

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 数据量  方式 | 100 | 1000 | 10000 | 100000 | 稳定性 | 平均 | 最好 | 最坏 | 空间复杂度 |
| 冒泡 | 2218 | 248717 | 24916370 | 2498709610（20s） | √ | O(n2) | O(n) | O(n2) | O(1) |
| 选择 | 93 | 994 | 9982 | 99899（5s） | × | O(n2) | O(n2) | O(n2) | O(1) |
| 插入 | 2373 | 253822 | 24541408 | 2490698756（4s） | √ | O(n2) | O(n) | O(n2) | O(1) |
| 希尔 | 2335 | 184264 | 16907014 | 2049217848（4s） | × | O(nlogn) | O(nlog2n) | O(nlog2n) | O(1) |
| 快速 | 225 | 3534 | 42444 | 437738（0s） | × | O(nlogn) | O(nlogn) | O(n2) | O(nlogn) |
| 堆 | 624 | 9618 | 129348 | 1626692（0s） | × | O(nlogn) | O(nlogn) | O(nlogn) | O(1) |
| 归并 | 672 | 9976 | 133616 | 1668928（0s） | √ | O(nlogn) | O(nlogn) | O(nlogn) | O(n) |
| 基数 | 0 | 0 | 0 | 0 | √ | O(nk) | O(nk) | O(nk) | O(n+k) |

## 3.2 冒泡排序

### 3.2.1 排序方法

设数组中全体数据量为length

一共排序length趟，第i趟对0-length-1-i进行逐个比较，若前一项比后一项大，则交换。如此做法使得每一趟排序后在末尾放置了当前排序数组内最大的值，重复操作即可获得从小到大排序的数组。

冒泡排序的优点在于简单直观且稳定，是最易于编写代码的排序算法之一，而且其空间复杂度仅为O(1)，开销较小。但缺点是时间效率较低，平均时间复杂度达到O(n2)

### 3.2.2 核心代码

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

{

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

{

if (data[j] > data[j + 1])

{

int tmp = data[j];

data[j] = data[j + 1];

data[j + 1] = tmp;

}

}

}

## 3.3选择排序

### 3.3.1 排序方法

选择排序是在待排序数组中遍历一遍寻找到最小值，将最小值放在第一位，再对剩余数组内容重复上述过程，直到整个数组从小到大有序排列为止的排序算法。

他的比较次数很多，但交换次数约等于数据量。总体来看，选择排序不具有稳定性，它的优点是空间复杂度仅为O(1)，缺点是需要不断遍历，时间效率低，在任何情况下时间复杂度都是O(n2)。

### 3.3.2 核心代码

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

{

int minAddress = i; // 当前处理位记为最小值

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

{

if (data[j] < data[minAddress])

{

minAddress = j;

}

}

if (minAddress != i) // 需要交换

{

times++;

int tmp = data[i];

data[i] = data[minAddress];

data[minAddress] = tmp;

}

}

## 3.4插入排序

### 3.4.1 排序方法

插入排序一般从第一个元素开始，认为该元素已经被排序，然后取出下一个元素，在已经排序的元素序列中从后向前扫描，若该元素大于新元素，将该元素移到下一位置。重复上述步骤，直到找到已排序的元素小于或者等于新元素的位置，即将新元素插入到该位置后。当所有元素都在已排序序列中时，插入排序就完成了。

插入排序的优点是空间复杂度仅为O(1)且具有稳定性，但缺点是时间效率一般，平均时间复杂度为O(n2)，并且往往在算法执行过程中要频繁交换元素。

### 3.4.2 核心代码

for (int i = 1; i < length; i++)

{ // 第一个元素默认已排序，从第二个元素开始遍历

int j = i - 1;

int tmp = data[i];

for (; j >= 0; j--)

{

if (data[j] > tmp)

{ // 还需向前找，原位置向后挪

times++;

data[j + 1] = data[j];

}

else

{

break;

}

}

times++;

data[j + 1] = tmp;

}

## 3.5希尔排序

### 3.5.1 排序方法

希尔排序是对插入排序的一种优化。执行时先将待排记录序列分割成为若干子序列分别进行插入排序，不断缩短子序列每个元素间隔长度，直到最后对全体进行一次直接插入排序。

希尔排序的优点是时间复杂度为O(n log n)，效率较高，但缺点是不稳定。

### 3.5.2 核心代码

for (int step = length / 2; step > 0; step /= 2)

{

for (int i = step; i < length; i += step)

{

int j = i - step;

int tmp = data[i];

for (; j >= 0; j-=step)

{

if (data[j] > tmp)

{ // 还需向前找，原位置向后挪

data[j + step] = data[j];

times++;

}

else

{

break;

}

}

times++;

data[j + step] = tmp;

}

}

## 3.6快速排序

### 3.6.1 排序方法

快速排序要从待排序列中任意选取一个元素(一般选取第一个)作为基准值，然后将整个数据中中关键字比它小的元素都安置在它的位置之前，将元素中关键字比它大的元素都安置在它的位置之后。这样，以该基准值为分界线，将待排序列分成的两个子序列。再对两个子序列做同样的操作直到序列近一个元素。

快速排序往往较快，比其他时间复杂度为O(nlogn)的排序算法更适合处理极大量的数据，但缺点是不具有稳定性。

### 3.6.2 核心代码

void Rank::QuickSort(int startPos, int endPos)

{

if (startPos >= endPos)

{

return;

}

int key = data[startPos]; // 选取第一个元素为基准值

int keyPos = startPos; // 记录基准值所在位置

int left = startPos + 1, right = endPos; // 记录操作数组左右位置

while (left <= right)

{

if (right>keyPos && data[right] >= key) // 顺序正确，不排

{

right--;

}

else if (right>keyPos) // 小的在前，交换

{

times++;

data[keyPos] = data[right];

keyPos = right;

right--;

}

if (left < keyPos && data[left] <= key)

{

left++;

}

else if (left<keyPos)

{

times++;

data[keyPos] = data[left];

keyPos = left;

left++;

}

}

data[keyPos] = key;

QuickSort(startPos, keyPos - 1); // 左序列排序

QuickSort(keyPos + 1, endPos); // 右序列排序

}

## 3.7堆排序

### 3.7.1 排序方法

堆排序需要首先建立最大堆，每次交换头尾元素，再对0到n-1的元素重新自顶向下调整为最大堆，重复上述过程，每次都会使得堆顶最大的元素放到数组最后，形成从小到大的有序序列。

堆排序的优点在于时间复杂度O(nlogn)，效率较高，缺点在于不具有稳定性。

### 3.7.2 核心代码

int\* heap = new int[length]; // 新建堆数组

initHeap(heap); // 生成最大堆

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

{

times++;

int tmp = heap[0];

heap[0] = heap[i];

heap[i--] = tmp;

// 交换头尾

siftDown(0, i, heap); // 重新调整为最大堆

}

EndTime();

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

{

data[i] = heap[i];

}

delete[]heap;

## 3.8归并排序

### 3.8.1 排序方法

归并排序是采用分治法（Divide and Conquer）的一个非常典型的应用。它将已有序的子序列合并，得到完全有序的序列，对子序列也是同样的原理，因此这是一个递归过程，只需拆分子序列再重新合成有序序列即可。

归并排序的缺点是需要额外的内存空间O(n)，但优点在于算法时间复杂度为O(n log n)，具有较高的效率，同时具有稳定性。

### 3.8.2 核心代码

if (left >= right)

{

return;

}

int mid = (left + right) / 2;

Merge(L1, L2, left, mid);

Merge(L1, L2, mid + 1, right);

MergeCalc(L1, L2, left, mid, right);

## 3.9基数排序

### 3.9.1 排序方法

基数排序是一种非比较型整数排序算法，因此在表格中它的交换次数永远是0。它将整数（本项目中设定为三位数）按位数切割成不同的数字，然后按每个位数分别放入不同的桶之中。例如三位数中，首先按照百位放入不同的桶，再对每个桶做同样的操作（按十位分割），直到最低位个位也被排序完成，最后整个序列都是有序的。

基数排序的时间复杂度为O(n+k)，效率非常高，并且具有稳定性，但缺点是需要额外的存储空间O(n+k)。

### 3.9.2 核心代码

if (left >= right || Weigh > 2)

{

return;

}

int\* auxArray = new int[right - left + 1];

int count[10];

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

{

count[i] = 0;

}

for (int i = left; i <= right; i++)

{

count[Digit(List[i], Weigh)]++;

}

/\* 计算桶的地址 \*/

int posit[10];

posit[0] = 0;

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

{

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

}

for (int i = left; i <= right; i++)

{

int j = Digit(List[i], Weigh);

auxArray[posit[j]++] = List[i]; // 放到桶里

}

for (int i = left, j = 0; i <= right; i++, j++)

{

List[i] = auxArray[j];

}

delete[]auxArray;

int pos1 = left;

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

{

int pos2 = pos1 + count[i] - 1;

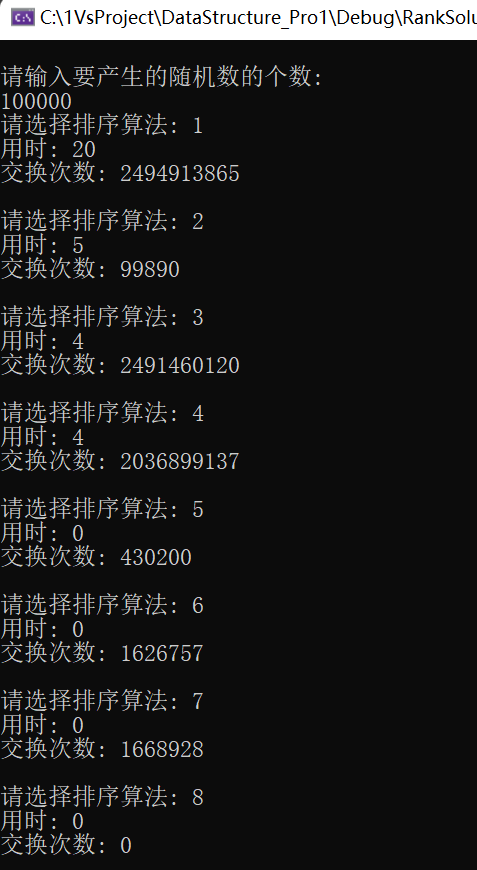
Radix(List, pos1, pos2, Weigh + 1);

pos1 = pos2 + 1;

}

# 4 测试

## 4.1 功能测试



## 4.2 出错测试

**测试用例：**总数据量输入非100、1000、10000、100000的数字或字符

**预期结果：**输出错误提示，要求重新输入，程序不崩溃

**结果展示：**

