**数据结构课程设计《8种排序算法的比较案例》项目说明文档**

2253893 苗君文 软件工程

**1. 项目简介**

**1.1. 项目背景**

排序算法是计算机科学中的基础概念，对数据的快速、有效排序是许多应用的关键步骤。在实际软件开发中，选择适当的排序算法对于提高程序性能至关重要。许多应用场景，包括数据库查询、搜索引擎、图形渲染等，要求对海量数据进行高效排序。排序算法的性能直接关系到应用程序的响应速度和效率。

**1.2. 项目要求**

随机函数产生10000个随机数，用快速排序，直接插入排序，冒泡排序，选择排序的排序方法排序，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。并且显示他们的比较次数。

**1.3. 输入格式**

本项目首先要求用户输入要产生的随机数的个数，接着用户可以选择不同的排序算法选项，不断选择，直到选择退出程序。

**1.4. 输出格式**

本项目针对输入有详细的输入错误处理，若用户输入错误，则程序会输出相关提示信息，并要求重新输入。针对用户选择的某一排序算法，程序会输出此排序算法所用的时间，比较次数和移动次数。

**2. 设计思路**

**2.1. 数据结构设计**

本项目涉及的排序算法涵盖了冒泡排序，选择排序，直接插入排序，希尔排序，快速排序，堆排序，归并排序，基数排序。程序中采用数据表存放产生的随机数。dataList能够动态调整数组的大小，适应不同规模的排序问题。排序算法中可能需要频繁地插入、删除和交换元素，而数据表具有灵活性，能够高效地进行这些操作。数据表也能方便访问其中的元素。

在基数排序中，使用了静态链表作为存储方式，用于将元素按照某一位数字进行分组。静态链表staticLinkedList在插入和删除方面的效率较高，也符合了基数排序的使用特性。

在堆排序的算法中需要使用最大堆的数据结构，因为每次可以高效地取出对顶元素。最大堆maxHeap的性质使得堆排序具有较小的时间复杂度，并且不需要额外的空间。而且，最大堆对于元素的插入具有较高的效率，也加快了堆排序中反复插入元素的效率。

**2.2. 类设计**

**2.2.1. Element类**

Element类用来表示排序算法中的元素，包含关键码，其他数据成员和链接指针。

成员变量：

* T key: 元素的关键码。
* char otherdata: 元素的其他数据成员。
* int link: 链接指针，用于静态链表。

成员函数：

* Element<T>(): 无参构造函数，初始化成员变量。
* Element<T>(T x, int next = 0): 带参构造函数，根据给定的关键码和链接指针创建元素。
* Element<T>& operator=(const Element<T>& x): 赋值运算符，实现元素之间的赋值操作。
* int operator==(Element<T>& x): 判断两个元素的关键码是否相等。
* int operator<=(Element<T>& x): 判断当前元素的关键码是否小于等于另一个元素的关键码。
* T getKey(): 获取元素的关键码。
* void setKey(const T x): 设置元素的关键码。
* int getLink(): 获取元素的链接指针。
* void setLink(const int l): 设置元素的链接指针。

**2.2.2. dataList类**

dataList类用于表示数据表，存储排序算法中的元素。

成员变量：

* Element<T>\* Vector：存储向量，即元素数组。
* int maxSize：最大元素个数。
* int currentSize：当前元素个数。

成员函数：

* dataList(int sz = DefaultSize): 构造函数，初始化向量大小。
* void randList(int num): 随机生成元素，用于排序算法测试。
* dataList<T>& operator=(const dataList<T>& other): 赋值运算符，实现动态数组的赋值操作。

**2.2.3. staticLinkedList类**

staticLinkedList类用于表示静态链表，用在基数排序中的“分配”和“收集”。

成员变量：

* Element<T>\* Vector: 存储链表节点的数组。
* int maxSize: 最大节点个数。
* int currentSize: 当前节点个数。

成员函数：

* staticLinkedList(int sz = DefaultSize): 构造函数，初始化链表大小。
* Element<T>& operator[](int i): 重载下标运算符，用于访问链表节点。

**2.2.4. maxHeap类**

* maxHeap类表示的是最大堆，用在堆排序算法中。

成员变量：

* Element<T>\* heap: 存储最大堆元素的数组。
* int currentSize: 当前堆元素个数。
* int maxHeapSize: 最大堆容量。

成员函数：

* maxHeap(int sz = DefaultSize): 构造函数，初始化最大堆容量。
* ~maxHeap(): 析构函数，释放动态分配的内存。
* bool IsEmpty(): 判断最大堆是否为空。
* bool IsFull(): 判断最大堆是否已满。
* void HeapSort(int& compare, int& move): 堆排序算法。
* void insert(Element<T> item): 插入元素到最大堆。
* void siftDown(int start, int m, int& compare, int& move): 将堆的某个分支向下调整为最大堆的过程。
* void siftUp(int start, int& compare, int& move): 将堆的某个叶结点向上调整至合适位置的过程。

**2.3. 主程序设计**

首先展示排序算法比较的菜单，并让用户输入随机数的个数，生成包含随机数的dataList对象。根据用户的选择，将调用相应的排序算法函数，包括冒泡排序、选择排序、直接插入排序、希尔排序、快速排序、堆排序、归并排序和基数排序。每种排序算法的调用都会记录比较次数、移动次数，并计算排序所用的时间，并将这些性能参数输出，而后循环回到选项菜单，等待用户进一步选择或退出程序。其中，随机数的生成由randList函数产生；时间的测量用系统时钟clock记录排序算法的运行时间。

**2.4. 输入错误处理设计**

**2.4.1. 单个参数的输入错误处理**

函数dealInputError使用 while 循环不断尝试获取用户输入，直到输入满足要求。使用 cin.fail() 及min、max来判断输入是否出错，检查输入是否在有效范围内。如果输入无效，则输出错误信息，清除输入缓冲区并忽略之后的字符，并重新输出输入的提示信息。通过一个字符来获取输入一个数之后的字符以检查输入字符的个数，若不正确，也输出错误信息，并重新输出输入的提示信息，执行相应清除操作。如果输入有效，则跳出循环。

此错误处理可以用在输入操作选项以及输入要产生的随机数的个数。

**3. 功能实现**

本程序的主要内容就是各种排序算法，因此接下来将要介绍各种排序算法的功能实现。

**3.1. 冒泡排序**

程序中用BubbleSort函数实现冒泡排序。该算法反复遍历列表，比较相邻元素，并在它们的顺序错误时进行交换。通过重复通过列表直到排序完成。外部循环控制对列表的遍历。排序过程会持续，直到列表完全排序或在一次遍历中没有进行任何交换。内部循环遍历未排序部分的列表，从末尾到当前遍历的元素。在内部循环内，比较相邻元素，如果它们的顺序不正确，则使用Swap函数进行交换，相应地增加move计数；并且如果出现了比较，增加compare的计数。如果在一次遍历中进行了任何交换，将exchange设置为1。如果没有任何交换，则排序结束，可以提前终止排序。在每次遍历后，最大的未排序元素“冒泡”到其正确的位置，下一次遍历集中在剩余的未排序部分上。这是一种时间复杂度在最坏情况下为O(n^2)的简单排序算法。冒泡排序是一种稳定的排序算法。

**3.2. 选择排序**

程序中用SelectSort函数实现选择排序。该算法通过不断选择列表中的最小元素并将其放置在已排序部分的末尾来完成排序。外部循环从列表的开头开始，依次选择未排序部分的每个元素。内部循环遍历未排序部分的列表，寻找具有最小关键码的对象，并将其索引存储在k中。在内部循环中，对每次比较进行计数。

如果找到具有更小关键码的对象，则将其索引赋给k。在内部循环结束后，如果k不等于i，表示找到了比当前选择的元素更小的元素，则进行交换，并相应地增加move的计数。外部循环继续，将当前选择的元素移动到已排序部分的末尾。这是一种简单的排序算法，其时间复杂度为O(n^2)。选择排序是一个不稳定的排序算法。

**3.3. 直接插入排序**

程序中用InsertSort函数实现直接插入排序。该算法通过逐个将未排序部分的元素插入到已排序部分的正确位置来完成排序。外部循环从列表的第二个元素开始，逐个选择未排序部分的元素。在每次迭代中，将当前选择的元素存储在temp中，并增加move的计数。内部循环向前顺序比较，找到合适的插入位置。对每次比较进行计数，并在找到插入位置时，将比temp大的元素依次向后移动，为temp腾出插入位置。然后将temp 插入到正确的位置，并增加move的计数。外部循环继续，每次迭代都将一个元素插入到已排序部分的正确位置。这是一种简单的排序算法，其时间复杂度为O(n^2)。直接插入排序是一种稳定的排序算法。

**3.4. 希尔排序**

程序中用ShellSort函数实现希尔排序。该算法是插入排序的一种改进版本，通过使用不同的间隔gap来对列表的子序列进行排序，最终实现整体的有序。外部循环通过不同的间隔gap，对列表进行多轮排序。在每轮排序中，从当前gap索引开始，逐个选择元素。将当前选择的元素存储在temp中，并增加move的计数。内部循环向前顺序比较，找到合适的插入位置。对每次比较进行计数，并在找到插入位置时，将比temp大的元素依次向后移动，为temp腾出插入位置。

将temp插入到正确的位置，并增加move的计数。外部循环继续，每次迭代都缩小gap直至为0。希尔排序是一种基于插入排序的高效排序算法，通过逐渐减小间隔来提高插入排序的效率，最终使整个列表达到有序状态。其最坏情况下的时间复杂度为O(n^2)，但平均情况下通常为O(nlogn)。希尔排序是种不稳定的排序算法。

**3.5. 快速排序**

程序中用QuickSort函数和Partition函数实现快速排序。Partition函数实现了快速排序算法中的分区操作。选择left作为基准元素的位置，并将其存储在pivot中。从left + 1到right的范围内遍历元素，将小于pivot的元素交换到基准元素的左侧。在遍历过程中，记录比较次数和移动次数。最后，将基准元素放置在最终的位置，并返回该位置。QuickSort函数则利用该函数完成快速排序。通过调用 Partition 函数，获取基准元素的最终位置pivotpos。对基准元素左右两侧的子区间分别递归调用QuickSort函数。在递归过程中，更新比较次数和移动次数。当left大于等于right或者超出合法范围时，终止递归。通过递归的方式，QuickSort函数将列表分割为越来越小的子列表，并通过不断调整基准元素的位置，最终实现整个列表的排序。快速排序的平均时间复杂度是O(nlog2n)，是一种不稳定的排序算法。

**3.6. 堆排序**

程序中用HeapSort函数实现堆排序。首先，需要建堆，从最后一个非叶子节点开始，依次向前进行siftDown操作，保证每个节点都满足最大堆的性质。经过上述步骤后，整个数组变成一个最大堆。然后进行堆排序，将当前节点值保存在临时变量temp中。比较左右子节点的值，选择较大的子节点与当前节点比较。如果当前节点的值大于等于较大的子节点值，则结束siftDown否则，将较大子节点的值上移，当前节点下沉到较大子节点的位置。递归调用上述过程，直到满足最大堆的性质。堆排序是一种选择排序，其中构建初始堆的复杂度为O(n)，堆排序时间复杂度一般认为就是O(nlogn)，堆排序是一种不稳定的排序算法。

**3.7. 归并排序**

程序中的归并排序算法主要包括归并排序函数MergeSort、一趟两路归并函数MergePass以及一次两路归并函数merge。其中，MergeSort函数实现了归并排序算法，使用自底向上的迭代方式。函数内先初始化一个临时数组tempList，用于在归并过程中暂存中间结果。设置归并的初始子数组大小len为1，然后不断将子数组大小翻倍，进行归并操作，直到整个数组有序。在每次归并中，调用 MergePass函数，将相邻的子数组进行合并。MergePass函数实现了一趟两路归并，从左到右，调用将相邻的子数组进行合并。使用了两个数组initList和mergedList，其中initList存储待合并的子数组，mergedList存储合并后的结果。Merge函数实现了一次两路归并，它使用i和j分别指向左右子数组的起始位置。依次比较initList[i]和initList[j]的大小，将较小的元素放入mergedList中。若其中一个子数组元素已经全部放入mergedList中，则将另一个子数组的剩余元素直接放入。重复上述步骤，直到两个子数组均已放入 mergedList。通过上述步骤，归并排序实现了将数组分割、排序和合并的过程，最终得到有序的序列。归并排序无论什么情况下是假的花销是不变的，其时间复杂度始终为O(nlogn)，是一种稳定的排序算法。

**3.8. 基数排序**

程序中用RadixSort函数实现基数排序。它将整数按照位数切割成不同的数字，然后按照每个位数分别比较。基数排序从最低有效位（个位）到最高有效位（最高位）依次排序，每次排序使用稳定的计数排序算法。基数排序的时间复杂度为O(d(N+radix)),是一种稳定的排序算法。

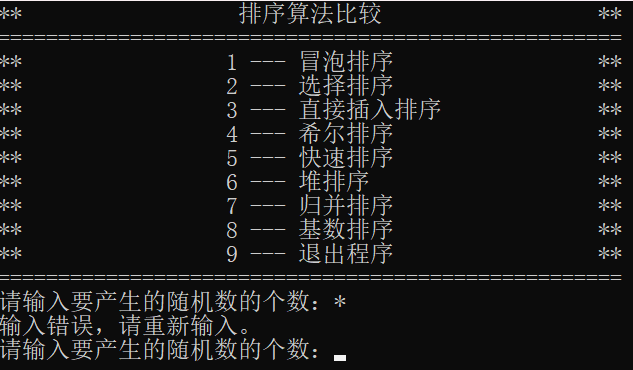
**3.9. 8种排序算法的比较**

|  |  |  |  |
| --- | --- | --- | --- |
| **算法** | **时间复杂度** | **空间复杂度** | **稳定性** |
| 冒泡排序 | O(N²) | O(1) | 是 |
| 选择排序 | O(N²) | O(1) | 否 |
| 直接插入排序 | O(N²) | O(1) | 是 |
| 希尔排序 |  | O(1) | 否 |
| 堆排序 | O(NlogN) | O(1) | 否 |
| 快速排序 | O(NlogN) | O(logN) | 否 |
| 归并排序 | O(NlogN) | O(N) | 是 |
| 基数排序 | O(d(N+radix)) | O(N) | 是 |

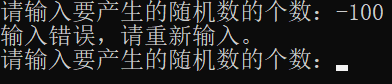
**4. 测试结果（包括边界测试）**

**4.1. 输入要产生的随机数的个数**

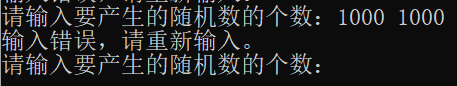
（1）输入的类型非法



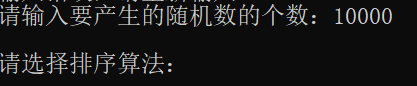
（2）输入的范围非法



（3）输入的个数超过一个

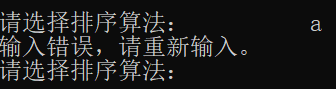


（4）输入正确即可进入下一环节

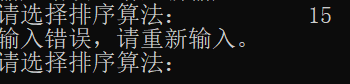


**4.2. 输入选择的排序算法编号**

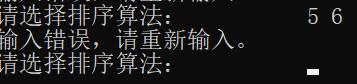
（1）输入的类型非法



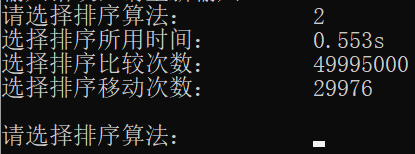
（2）输入的范围非法



（3）输入的个数超过一个



（4）输入正确即可进入对应的排序算法



**4.3. 程序的测试结果总览（此处仅展示输入正确的情况）**

