经典的 7 种排序算法 原理 C++实现

作者：草根 IT 网 来源：未知 人气：2288 标签：
导读：排序是编程过程中经常遇到的操作，它在很大程度上影响了程序的执行效率。目前关于排序的算法有很多，其中不乏非常精妙的算法。但是总体来说，作为一个计算机相关专业的学习者来说，必须要知道而且会亲自动手去实现 7 种常见 的算法。这不管对自己编程能力的提高还是日后的实习就业都会有莫大的帮助。7 种常见的排序算法大致可以...

经典的 7 种排序算法 原理 C++实现

排序是编程过程中经常遇到的操作，它在很大程度上影响了程序的执行效率。目前关于排序的算法有很多，其中不乏非常精妙的算法。但是总体来说，作为一 个计算机相关专业的学习者来说，必须要知道而且会亲自动手去实现 7 种常见的算法。这不管对自己编程能力的提高还是日后的实习就业都会有莫大的帮助。

7 种常见的排序算法大致可以分为两类：第一类是低级排序算法，有选择排序、冒泡排序、插入排序；第二类是高级排序算法，有堆排序、排序树、归并排序、快速排序。下面就分别介绍一下这几种排序算法，并会给出 c++的实现，实现代码均经过测试。

一、低级排序算法

1. 选择排序

排序过程：给定一个数值集合，循环遍历集合，每次遍历从集合中选择出最小或最大的放入集合的开头或结尾的位置，下次循环从剩余的元素集合中遍历找出最小的并如上操作，最后直至所有原集合元素都遍历完毕，排序结束。

实现代码：

```
//选择排序法
template
void Sort::SelectSort(T* array, int size)
{
	int
minIndex;

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

	{
		minIndex =
```

```
i;

                for(int j = i + 1; j < size; j++)
                {
                        if(array[minIndex] >
array[j])

                        {
                                minIndex =
j;

                        }
                }
                if(minIndex !=
i)

                {
                        Swap(array, i, minIndex);
                }
        }
}
```

分析总结：选择排序时间复杂度比较高，达到了 O（n^2），每次选择都要遍历一遍无序区间。选择排序对一类重要的元素序列具有较好的效率，就是元素规模很大，而排序码却比较小的序列。另外要说明的是选择排序是一种不稳定的排序方法。

2. 冒泡排序

排序过程：冒泡排序的过程形如其名，就是依次比较相邻两个元素，优先级高（或大或小）的元素向后移动，直至到达序列末尾，无序区间就会相应地缩小。下一次再从无序区间进行冒泡操作，依此循环直至无序区间为 1，排序结束。

实现代码：

```
//冒泡排序法
template
void Sort::BubbleSort(T* array, int size)
{
        for(int i = 0; i < size; i++)
        {
                for(int j = 1; j < size - i; j++)
                {
                        if(array[j] < array[j - 1])
```

```
                        {
                                Swap(array, j, j - 1);
                        }
                }
        }
}
```

分析总结：冒泡排序的时间复杂度也比较高，达到 O（n^2），每次遍历无序区间都将优先级高的元素移动到无序区间的末尾。冒泡排序是一种**稳定**的排序方式。

3. 插入排序

排序过程：将前面的区间（初始区间为 1，包含第一个元素）视作有序区间，然后将有序区间的后一元素插入到前面有序区间的适当位置。直至有有序区间扩展到原区间的大小，排序结束。

实现代码：

```
//插入排序
template
void Sort::InsertSort(T* array, int size)
{
        for(int i = 1; i < size; i++)
        {
                for(int j = i; j > 0; j--)
                {
                        if(array[j] < array[j - 1])
                        {
                                Swap(array, j, j-1);
                        }
                }
        }
}
```

分析总结：插入排序的时间复杂度达到 O（n^2），排序的运行时间和待排序元素的原始排列顺序密切相关。插入排序是一种**稳定**的排序方法。

二、高级排序算法

1. 快速排序

排序过程：快速排序应该是应用最广泛的排序算法，它是采用了分治的思想（这种思想很重要）。其基本的思想就是任 取待排序序列中的某个元素（元素

的选取方式在一定程序上会影响实现过程和排序效率）作为标杆，将待排序序列划分为左右两个子序列，左侧元素小于标杆元素， 右侧元素大于标杆元素，标杆元素则排在这两个子序列的中间，然后再对这两个子序列重复上述的方法，直至排序结束。

实现代码：

```
//快速排序
template
void Sort::QuickSort(T *array, int left, int right)
{
	if(left < right)
	{
		int i = left -1, j = right + 1;
		T mid = array[(left + right) / 2];
		while(true)
		{
			while(array[++i] < mid);
			while(array[--j] > mid);
			if(i >= j)
			{
				break;
			}
			Swap(array, i, j);
		}
		QuickSort(array, left, i - 1);
		QuickSort(array, j + 1, right);
	}
}
```

分析总结：快速排序的时间复杂度为 0（nlogn），是一种**不稳定**的排序算法。

2. 归并排序

排序过程：归并排序的原理比较简单，也是基于分治思想的。它将待排序的元素序列分成两个长度相等的子序列，然后为每一个子序列排序，然后再将它们合并成一个序列。

实现代码：

```
//归并排序
template
void Sort::MergeSort(T* array, int left, int right)
{
	if(left < right)
```

```
        {
                int mid = (left + right) / 2;
                MergeSort(array, left, mid);
                MergeSort(array, mid + 1, right);
                Merge(array, left, mid, right);
        }
}
//合并两个已排好序的子链
template
void Sort::Merge(T* array, int left, int mid, int right)
{
        T* temp = new T[right - left + 1];
        int i = left, j = mid + 1, m = 0;
        while(i <= mid && j <= right)
        {
                if(array[i] < array[j])
                {
                        temp[m++] = array[i++];
                }
                else
                {
                        temp[m++] = array[j++];
                }
        }
        while(i <= mid)
        {
                temp[m++] = array[i++];
        }
        while(j <= right)
        {
                temp[m++] = array[j++];

        }
        for(int n = left, m = 0; n <= right; n++, m++)
        {
                array[n] = temp[m];
        }
        delete temp;
}
```

分析总结：归并排序最好、最差和平均时间复杂度都是 O（nlogn），是一种**稳定**的排序算法。

3. 堆排序

排序过程：堆排序的过程分为两个步骤，第一步是根据初始输入数据，建立一个初始堆；第二步是将堆顶元素与当前无序区间的最后一个元素进行交换，然后再从堆顶元素开始对堆进行调整。

实现代码：

```
//堆排序
template
void Sort::HeapSort(T* array, int size)
{
	int lastP = size / 2;
	//从最后一个有孩子的结点开始建初始堆
	for(int i = lastP - 1; i >= 0; i--)
	{
		HeapAjust(array, i, size);
	}
	int j = size;
	//将堆顶元素和无序区间的最后一个元素交换，再调整堆
	while(j > 0)
	{
		Swap(array, 0, j - 1);
		j--;
		HeapAjust(array, 0, j);
	}
}
//调整堆
template
void Sort::HeapAjust(T *array, int toAjust, int size)
{
	int pos = toAjust;
	while((pos * 2 + 1) < size)
	{
		int lChild = pos * 2 + 1;
		if(array[lChild] > array[pos])
		{
			pos = lChild;//左孩子大
		}
		int rChild = lChild + 1;
		if(rChild < size && array[rChild] > array[pos])
		{
			pos = rChild;//右孩子更大
		}
		if(pos != toAjust) //父结点比其中一个孩子小
		{
```

```
                Swap(array, toAjust, pos);
                toAjust = pos;
            }
            else
            {
                break;
            }
        }
}
```

分析总结：堆排序的算法时间复杂度为 O（nlogn），它是一种**不稳定**的排序算法。

4. 排序树

排序过程：排序树算法应用了 AVL 树的原理，只不过排序树不是平衡的，它的特点就是父结点元素总是比左孩子元素 要大却比右孩子元素要小。根据这个特点，可以将原数组元素组织成排序树，然后在对排序树进行中序遍历，中序遍历的结果就是排好序的序列。在算法的实现中采 用的数组的形式来存储排序树，并采用的三个辅助数组来表示元素与元素之间在树中的关系，这三个辅助数组分别是父结点索引表、左孩子索引个、右孩子索引表。 最后采用了递归的方法对排序树进行了中序遍历。

实现代码：

```
template
void Sort::TreeSort(T* array, int size)
{
    int *parent = new int[size];//父结点子针
    int *lChild = new int[size];//左孩子子针
    int *rChild = new int[size];//右孩子子针
    //将各结点左右子结点指针均置为-1，表示没有左右子结点
    for(int i = 0; i < size; i++)
    {
        lChild[i] = -1;
        rChild[i] = -1;
    }

    parent[0] = -1; //将第一个元素作为根结点，其父结点置为-1
    //从第 2 个数开始构造树
    for(int i = 1; i < size; i++)
    {
        int last = 0;
        while(true)
        {
```

```
                        int compare = array[i] - array[last];
                        if(compare > 0) //比当前值大，进入右子树
                        {
                                if(rChild[last] == -1)
                                {
                                        rChild[last] = i;
                                        parent[i] = last;
                                        break;
                                }
                                else
                                {
                                        last = rChild[last];
                                }
                        }
                        else //比当前值小，进入左子树
                        {
                                if(lChild[last] == -1)
                                {
                                        lChild[last] = i;
                                        parent[i] = last;
                                        break;
                                }
                                else
                                {
                                        last = lChild[last];
                                }
                        }
                }
        }
        //保存排序树的中序遍历结果
        T* midRetrival = new T[size];
        TreeMidRetrival(array, midRetrival, 0, lChild, rChild);
        //将排好序的中序数组复制到原数组
        for(int i = 0; i < size; i++)
        {
                array[i] = midRetrival[i];
        }
}
//中序遍历
template
void Sort::TreeMidRetrival(T *array, T* temp, int pos, T* lChild, T*
rChild)
{
        static int i = 0;
```

```
        if(pos != -1)
        {
                TreeMidRetrival(array, temp, lChild[pos], lChild,
rChild);//遍历左子树
                temp[i++] = array[pos];//保存当前值
                TreeMidRetrival(array, temp, rChild[pos], lChild,
rChild);//遍历右子树
        }
        else
        {
                return;
        }
}
```

分析总结：算法中排序树建立的时间复杂度是 O（nlogn），中序遍历的时间复杂度是 O（n），故排序树排序的时间复杂度为 O（nlogn）。排序树的空间复杂度比较高，因为它使用了几个额外的存储空间的存储排序树元素之间的关系。

模板类定义如下所示：

```
template
class Sort
{
public:
        void SelectSort(T* array, int size);
        void InsertSort(T* array, int size);
        void BubbleSort(T* array, int size);
        void MergeSort(T* array, int left, int right);
        void Merge(T* array, int left, int mid, int right);
        void HeapSort(T *array, int size);
        void HeapAjust(T *array, int toAjust, int size);
        void QuickSort(T *array, int left, int right);
        void TreeSort(T *array, int size);
        void TreeMidRetrival(T* array, T* temp, int pos, T* lChild, T*
rChild);
        void Swap(T* array, int x, int y);
};

template //交换两个元素
void Sort::Swap(T* array, int x, int y)
{
        T temp = array[x];
        array[x] = array[y];
```

```
        array[y] = temp;
}
```

总结：

如上介绍的几种排序算法都是基于算法最基本的思想来实现的，其实对其中一些算法都会有几种可以改进的地方，改进的方法可以在网上找找资料或看看书上面的相关介绍。

以上介绍的排序算法最好是能新自实现实现，这对你找实习和工作都会有莫大的帮助。以本人为例，前几天参加的腾讯暑期实习的招聘，笔试虽然很基础，但 是也没过，主要是基础不够扎实。最后不想放弃机会，去霸面了，霸面时没有面试，被挑过去笔了两道题，其中第二道题是一个对海量数据排序之类的题目，我开始 题目听错了，给的解决方案让负责笔试的人不是很满意，但是我在纸上把要用到的归并算法和快速排序写出来了，而且写得还比较顺利，因为之前我就自己动手实现 过，所以在这里派上了用场。最后为我争取到了一次去腾讯机试的机会，这周四机试，不管结果怎样，能够用自己所学的为自己争取一次机会也没有什么遗憾了。所 以大家一定要动手去编码实现一下，光看懂远远不够，到了实现的时候就会很棘手的。

本博客权当抛砖引玉，只代表了个人的一些看法，只想将自己所掌握的分享一下。本人能力有限，算法实现可能不尽如人意，请大牛轻拍！