数据结构课程设计项目说明文档

——10种排序算法的比较案例

作 者 姓 名： 马威

学 号： 2151294

指 导 教 师： 张颖

学院、 专业： 软件学院 软件工程

****

目 录

[1 项目分析 1](#_Toc495668153)

[1.1 项目背景 1](#_Toc495668154)

[1.2 项目要求 1](#_Toc495668154)

[1.3 项目需求分析 2](#_Toc495668155)

[2 项目设计 4](#_Toc495668156)

[2.1 数据结构设计 4](#_Toc495668157)

[2.2 类设计 4](#_Toc495668158)

2.2.1 排序表类（DataList） 4

2.2.2 结点类（QueueNode） 5

2.2.3 链式队列类（LinkedQueue） 6

2.2.4 最大堆类（MaxHeap） 7

2.3 算法设计 8

2.3.1 冒泡排序 8

2.3.2 选择排序 9

2.3.3 直接插入排序 10

2.3.4 折半插入排序 11

2.3.5 希尔排序 12

2.3.6 快速排序 13

2.3.7 改进快速排序 14

2.3.8 堆排序 17

2.3.9 归并排序 19

2.3.10 LSD基数排序 20

2.3.11 各排序算法对比 22

[3 项目测试 2](#_Toc495668161)3

[3.1 10个数的序列测试 2](#_Toc495668162)3

[3.2 100个数的序列测试 2](#_Toc495668166)4

[3.3 1000个数的序列测试 2](#_Toc495668170)5

[3.4 10000个数的序列测试 2](#_Toc495668174)6

[3.5 100000个数的序列测试 2](#_Toc495668178)7

[3.6 1000000个数的序列测试 2](#_Toc495668178)8

[3.7 10000000个数的序列测试 2](#_Toc495668178)9

**1.项目分析**

1.1 项目背景

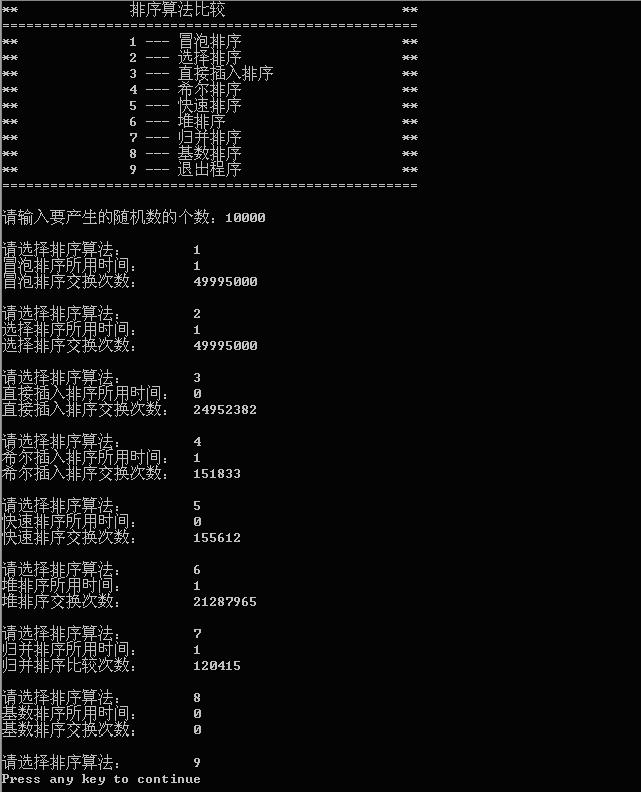
在日常生活中经常需要对所收集到的各种数据进行处理，这些数据处理中经常遇到的核心运算就是排序（sorting）。例如，图书管理员将书籍按照馆藏编号排序放置在书架上，方便读者查找；打开计算机的资源管理器，可以选择按名称、类型、创建日期等来排列图标。排序已被广泛应用在几乎所有的领域。

1.2 项目要求

随机函数产生一百，一千，一万和十万个随机数，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。

请在文档中记录上述数据量下，各种排序的计算时间和存储开销，并且根据实验结果说明这些方法的优缺点。

项目示例：



1.3 项目需求分析

对于寻找迷宫路径的程序，需考虑以下需求：

**·正确性**

程序应当能够按照要求对一个序列进行各类排序，没有错误

**·高效性**

排序有特殊要求，需要用特定的求法解决，这就要求要在尽可能短的时间内处理大量的数据。

**·健壮性**

程序设计到输入序列元素的个数，其需要能对错误的输入进行处理，以免发生意外的结果。

**·可比性**

为了让数据之间更加具有对比性，本项目考虑了10-1000000个随机数的情况，同时分为了随机序列、升序序列和降序序列，还和标准stl库中的排序算法做了对比。

**2.项目设计**

2.1 数据结构设计

由项目分析可以得出，本项目需要用合适的数据结构存放元素序列。由于程序中会对序列中每一个元素进行频繁访问交换，故用数组存储较为合适，可以简单的封装一个动态数组。

堆排序中用到了堆的性质，需要实现一个堆类。

LSD基数排序中，先放进容器内的元素也是最先从容器内出来的，同时为了降低对存储的要求，本系统实现了一个链表为形式的队列。采用struct描述链表结点类（QueueNode），这样使得链式栈类（LinkedQueue）可以直接访问链表结点而不需要定义友元关系。

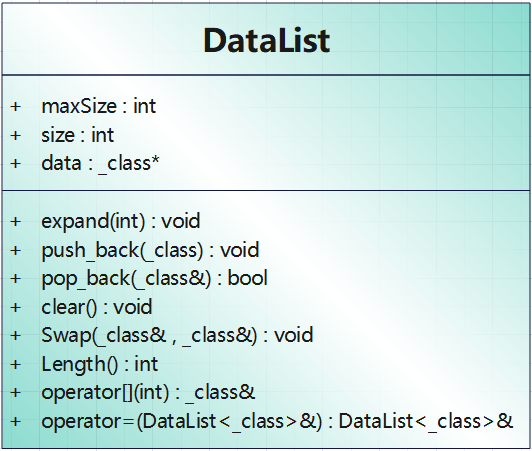
2.2 类设计

**2.2.1 排序表类（DataList）**

排序表类（DataList）是一种容器，其本质上通过动态数组来实现，使得元素按照线性顺序排列，同时可以通过下标快速访问，也提供了在末尾进行插入、删除元素等操作。

由于该数据结构专门用作存放待排序的元素序列，故仅实现了一些排序常用的操作，如交换元素、求元素数量等。

该类的UML图如下所示：



其中，主要函数如下：

//扩大数组空间

void expand(const int \_timesOfExpandingDefaultSize);

//向末尾插入一个元素

void push\_back(const \_class item);

//移走末尾最后一个元素

bool pop\_back(\_class& x);

//清空排序表

void clear();

//交换两个元素

void Swap(\_class& x1, \_class& x2)const;

//取表的长度

int Length()const;

//重载函数：下标访问

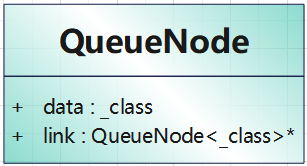
\_class& operator[](const int pos);

//重载函数：赋值

DataList<\_class>& operator=(const DataList<\_class>& list);

**2.2.2 结点类（QueueNode）**

链表结点存储了结点数据、后继节点位置，其UML图如下：

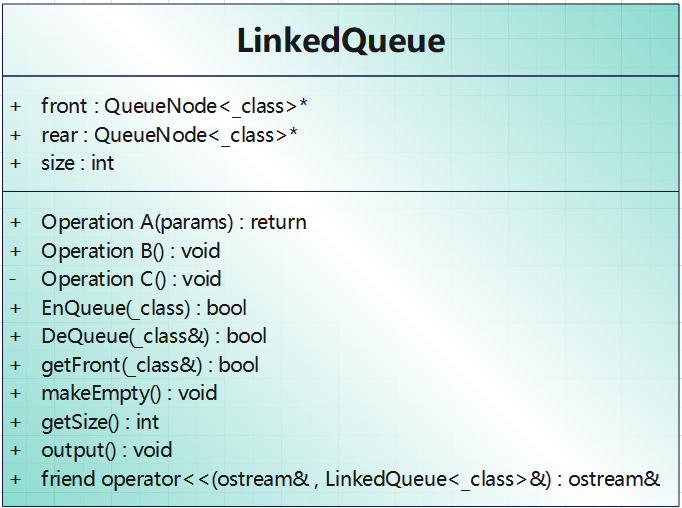


**2.2.3 链式队列类（LinkedQueue）**

链式队列类本质上仍是单链表，保存了链表的基本结构。由于我们在使用队列时仅仅利用其“先进先出”的特性，所以其提供的操作并不像一般链表操作那样齐全，而是仅仅有入队和出队。

由于对队列的插入、删除元素等操作的位置比较固定，为了节省空间，并未采用附加头结点。

其UML图如下：



其中，主要函数如下：

//入队

bool EnQueue(const \_class x);

//出队

bool DeQueue(\_class& x);

//取队列第一个元素

bool getFront(\_class& x)const;

//清空队列

void makeEmpty();

//判断队列是否为空

bool IsEmpty()const;

//取队列中元素个数

int getSize()const;

//输出队列中所有元素

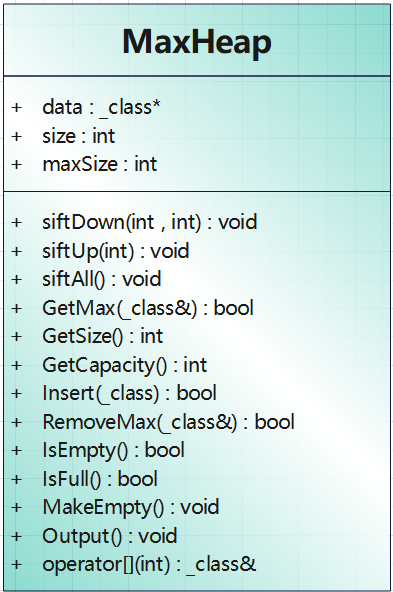
void output()const;

**2.2.4 最大堆类（MaxHeap）**

最大堆（MaxHeap）是一种优先级队列，其逻辑上为二叉树结构，每个结点的值都比其子女大，根结点就是最大值。按照这样的顺序排列，使得最大堆的插入、取出最大值等操作的时间复杂度仅为O(log2n)，性能良好。

每次进行插入或删除操作时，为了保持最大堆的性质，需要频繁地对元素进行访问、比较和交换，而并不关心每个元素的具体位置。因此，虽然最大堆逻辑上是树形结构，但本质上通过数组实现，通过层次往下给出元素的下标，标号为n的左子女（若存在）标号为2n+1，右子女（若存在）标号为2n+2，由此可知，每次进行元素的比较、交换时只需通过下标即可访问元素，非常方便。

其UML图如下：



其中，主要函数如下：

//取堆顶元素，即最大元素

bool GetMax(\_class& x)const;

//取堆的当前大小

int GetSize()const;

//取堆的最大容量

int GetCapacity()const;

//将x插入最大堆中

bool Insert(const \_class x);

//删除堆顶上的最大元素

bool RemoveMax(\_class& x);

//判断堆是否为空

bool IsEmpty()const;

//判断堆是否已满

bool IsFull()const;

//置空堆

void MakeEmpty();

//按数组内存放顺序输出堆内的元素

void Output()const;

//重载函数：下标访问

\_class& operator[](const int pos)const;

2.3 算法设计

**2.3.1 冒泡排序**

·算法思路

设待排序的序列中有n个元素，第1趟先比较第n-2个和第n-1个，若前一个大于后一个，则将这两个元素交换；接着再对第n-3个和第n-2个作相同处理；重复同样过程直至第0个和第1个处理完，至此最小的元素已在序列第一个。重复n趟后整个序列就有序了。

·代码实现

template<class \_class>

void BubbleSort(DataList<\_class>& list, const int num)

{

    for (int i = 0; i < num; i++) {

        for (int j = num - 1; j >= i; j--) {

            if (Compare++, list[j - 1] > list[j]) {

                list.Swap(list[j - 1], list[j]);

                Move += 3;

            }

        }

    }

}

·算法分析

由于每一次比较、交换都涉及到相邻元素，在关键码相等时不进行交换，因此冒泡排序是一种稳定的排序算法。

**2.3.2 选择排序**

·算法思路

此程序中选择排序通过直接选择排序实现，其步骤为：

（1）在第i趟中，从第i个到第n-1个元素中选择最小的元素

（2）若它不是i~n-1这一部分的第一个，则将它与第一个元素对调

（3）重复（1）（2）至完成n趟检查

·代码实现

template<class \_class>

void SelectSort(DataList<\_class>& list, const int num)

{

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

        int k = i;

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

            if (Compare++, list[j] < list[k]) {

                k = j;

            }

        if (i != k) {

            list.Swap(list[i], list[k]);

            Move += 3;

        }

    }

}

·算法分析

在一趟选择，如果a比当前元素b小，而a又出现在一个和当前元素相等的元素b\*后面，那么a势必要与b交换，则b在b\*后面了，稳定性就被破坏了。因此，选择排序是一个不稳定的排序算法。

**2.3.3 直接插入排序**

·算法思路

当插入第i个元素时，前面0~n-1的元素已经有序。这时，用第i个元素的排序码和前面元素的排序码进行比较，找出插入位置将第i个元素插入，其余元素往后移一位。

·代码实现

template<class \_class>

void InsertSort(DataList<\_class>& list, const int num)

{

    \_class temp;

    for (int i = 1; i < num; i++) {

        if (Compare++, list[i] < list[i - 1]) {

            temp = list[i];

            Move++;

            int j = i - 1;

            do {

                list[j + 1] = list[j];

                Move++;

                j--;

            } while (j >= 0 && (Compare++, temp < list[j]));

            list[j + 1] = temp;

            Move++;

        }

    }

}

·算法分析

设有相等元素b、b\*，在插入b\*时b已经在有序序列中，而相等时循环会停止，因此插入位置会在b后面，直接插入排序是一种稳定的排序算法。

**2.3.4 折半插入排序**

·算法思路

折半插入排序的思想和直接插入排序一致。区别是在有序区中查找新元素插入位置时，为了减少元素比较次数提高效率，采用折半搜索进行插入位置的确定。

·代码实现

template<class \_class>

void BinaryInsertSort(DataList<\_class>& list, const int num)

{

    \_class temp;

    for (int i = 1; i < num; i++) {  /\*从1开始：后num-1个数排好，第1个数也排好\*/

        temp = list[i];

        Move++;

        int low = 0, high = i - 1;

        while (low <= high) {

            int middle = (low + high) / 2;

            if (Compare++, temp < list[middle])

                high = middle - 1;

            else

                low = middle + 1;

        }

        for (int j = i - 1; j >= low; j--) {

            list[j + 1] = list[j];

            Move++;

        }

        list[low] = temp;

        Move++;

    }

}

·算法分析

折半插入排序是一种稳定的排序算法，其速度会比直接插入排序快一些。虽然降低了比较次数，但并没有降低移动次数，因此折半插入排序的时间复杂度仍为O(n**2**)。

**2.3.5 希尔排序**

·算法思路

设待排序元素序列有n个元素，取一个整数gap<n作为间隔，将全部元素划分为gap个子序列，所有距离为gap的元素放在同一个子序列中，在每一个子序列中分别进行直接插入排序。然后缩小间隔gap（缩小方式不唯一），重复上述步骤直至gap=1，将所有元素放在同一个子序列中为止。

·代码实现

template<class \_class>

void ShellSort(DataList<\_class>& list, const int num)

{

    int gap = num;

    \_class temp;

    do {

        gap = gap / 3 + 1;  /\*缩小gap的方式为gap=[gap/3+1]\*/

        for (int i = gap; i <= num - 1; i++) {

            if (Compare++, list[i] < list[i - gap]) {

                temp = list[i];

                Move++;

                int j = i - gap;

                do {

                    list[j + gap] = list[j];

                    Move++;

                    j = j - gap;

                } while (j >= 0 && (Compare++, temp < list[j]));

                list[j + gap] = temp;

                Move++;

            }

        }

    } while (gap > 1);  /\*gap等于1时停止循环，排序结束\*/

}

·算法分析

由于开始时gap较大，每个子序列中的元素很少，排序速度很快；到排序后期，gap变小，子序列元素变多，但由于之前的排序，多数元素已基本有序，所以排序速度也很快。稳定性方面，设有相等元素b、b\*，在gap等于某个值时它们分别在两个子序列中，若b\*在子序列中较小，移动到了b的前面，则稳定性就会被破坏，故希尔排序是一种不稳定的排序算法。

**2.3.6 快速排序**

·算法思路

快速排序算法采用分治法进行排序。任取待排序序列的序列中的某个元素作为基准，按照该元素的排序码大小，将整个序列分为左右两个子列：左子列元素排序码都小于基准的排序码，右子列元素排序码都大于基准的排序码。然后再对左右子列重复上述方法，直到所有元素都排在相应的位置。

·代码实现

//划分（快速排序专用）

template<class \_class>

int Partition(DataList<\_class>& list, const int low, const int high)

{

    int pivotpos = low;

    \_class pivot = list[low];

    for (int i = low + 1; i <= high; i++) {

        if (Compare++, list[i] < pivot) {

            pivotpos++;

            if(pivotpos != i) {

                list.Swap(list[pivotpos], list[i]);

                Move += 3;

            }

        }

    }

    list[low] = list[pivotpos];

    list[pivotpos] = pivot;

    Move += 2;

    return pivotpos;

}

//快速排序

template<class \_class>

void QuickSort(DataList<\_class>& list, const int left, const int right)

{

    if (left < right) {

        int pivotpos = Partition(list, left, right);

        QuickSort(list, left, pivotpos - 1);

        QuickSort(list, pivotpos + 1, right);

    }

}

·算法分析

由于快速排序使用了递归方式，需要一个递归栈存放每层递归调用时的指针和参数，存储开销为O(log**2**n)。

由于每一次都选择第一个元素作为基准，算法存在最坏情况，即待排序列已经有序的情况，比较次数将达到O(n**2**)。

由于涉及到远距离交换元素，快速排序算法是一个不稳定的排序算法。

**2.3.7 改进快速排序**

·算法思路

改进快速排序的思想和快速排序一致。区别是每一次取基准时，在序列的左端点、右端点和中点位置取中间值，然后把基准元素交换到序列右端点，最后进行划分。

·代码实现

//在表前端、尾端与中间点三者取中值，交换到尾端（改进快速排序专用）

template<class \_class>

\_class& Partition\_median3(DataList<\_class>& list, const int left, const int right)

{

    int mid = (left + right) / 2;

    int k = left;

    if (Compare++, list[mid] < list[k])

        k = mid;

    if (Compare++, list[right] < list[k])

        k = right;

    if (k != left) {

        list.Swap(list[k], list[left]);

        Move += 3;

    }

    if (mid != right && (Compare++, list[mid] < list[right])) {

        list.Swap(list[mid], list[right]);

        Move += 3;

    }

    return list[right];

}

//划分（改进快速排序专用）

template<class \_class>

int PartitionX(DataList<\_class>& list, const int left, const int right)

{

    int i = left, j = right - 1;

    if (left < right) {

        \_class pivot = Partition\_median3(list, left, right);

        while (1) {

            while (i < j && (Compare++, list[i] < pivot))

                i++;

            while (i < j && (Compare++, pivot < list[j]))

                j--;

            if (i < j) {

                list.Swap(list[i], list[j]);

                Move += 3;

                i++; j--;

            }

            else

                break;

        }

        if (Compare++, list[i] > pivot) {

            list[right] = list[i];

            list[i] = pivot;

            Move += 2;

        }

    }

    return i;

}

//改进快速排序

template<class \_class>

void QuickSortX(DataList<\_class>& list, const int left, const int right)

{

    if (left < right) {

        int pivotpos = PartitionX(list, left, right);

        QuickSortX(list, left, pivotpos - 1);

        QuickSortX(list, pivotpos + 1, right);

    }

}

·算法分析

由于涉及到远距离交换元素，故该算法仍然是一个不稳定的排序算法。

由于取基准元素时选择了较为“中间”的元素，避开了元素已经有序的最坏情况，故基本不会产生算法退化情况，在一般情况下也比改进前的快速排序速度要快（时间复杂度不变）。

**2.3.8 堆排序**

·算法思路

堆排序利用堆的优先级队列特性进行排序。先根据初始序列形成堆，每次再从堆中拿出一个最值并调整，直至所有元素都被取出，取出的顺序就是元素的一个有序序列了。

为了减少存储开销，本算法使用最大堆，形成堆时使用的空间与排序表的是同一段，每次取出最大的元素放在数组末尾，排序完成后数组内的元素也就是有序的了，不需额外的存储空间。

·代码实现

//给定数据和大小的构造函数

template<class \_class>

MaxHeap<\_class>::MaxHeap(\_class arr[], const int n)

{

    maxSize = (MAXHEAP\_DEFAULT\_SIZE < n) ? n : MAXHEAP\_DEFAULT\_SIZE;

    data = new \_class[maxSize];

    if (data == NULL) {

        cerr << "存储分配失败！" << endl;

        exit(1);

    }

    for (int i = 0; i < n; i++) {

        data[i] = arr[i];

        Move++;

    }

    size = n;

    int current = (size - 2) / 2;

    while (current >= 0) {

        siftDown(current, size - 1);

        current--;

    }

}

//从start到m下滑调整成为最大堆

template<class \_class>

void MaxHeap<\_class>::siftDown(const int start, const int m)

{

    if (start < 0 || start > size || m < 0 || m > size)

        return;

    int i = start, j = 2 \* i + 1;

    \_class temp = data[i];

    while (j <= m) {

        if (j < m && (Compare++, data[j] < data[j + 1]))

            j++;

        if (Compare++, temp >= data[j])

            break;

        else {

            data[i] = data[j];

            Move++;

            i = j;

            j = 2 \* j + 1;

        }

    }

    data[i] = temp;

}

//堆排序

template<class \_class>

void HeapSort(DataList<\_class>& list, const int num)

{

    MaxHeap<\_class> heap(&list[0], num);

    \_class \_max;

    const int size = heap.GetSize();

    for (int i = 0; i < size; i++) {

        heap.ReMoveMax(\_max);

        list[num - 1 - i] = \_max;

        Move++;

    }

}

·算法分析

时间开销上，需要取n个元素，每次取出后须进行调整，调整时间逐次递减，为log**2** n、log**2** (n-1)、log**2** (n-2)，总时间近似看作O(nlog**2**n)。

由于涉及到远距离交换元素，堆排序是一个不稳定的排序算法。

**2.3.9 归并排序**

·算法思路

归并排序基于分治法实现，将待排序列分成长度相等的两个子序列，递归划分到序列长度仅为1，再将序列两两归并（归并的同时排序），最终合成回一个序列，排序就完成了。

·代码实现

//归并排序：将两路进行归并

template<class \_class>

void Merge(DataList<\_class>& L1, DataList<\_class>& L2, const int left, const int mid, const int right)

{

    for (int k = left; k <= right; k++) {

        L2[k] = L1[k];

        Move++;

    }

    int s1 = left, s2 = mid + 1, t = left;

    while (s1 <= mid && s2 <= right) {

        if (Compare++, L2[s1] <= L2[s2]) {

            L1[t++] = L2[s1++];

            Move++;

        }

        else {

            L1[t++] = L2[s2++];

            Move++;

        }

    }

    while (s1 <= mid) {

        L1[t++] = L2[s1++];

        Move++;

    }

    while (s2 <= right) {

        L1[t++] = L2[s2++];

        Move++;

    }

}

//归并排序

template<class \_class>

void MergeSort(DataList<\_class>& L1, DataList<\_class>& L2, const int left, const int right)

{

    if (left >= right)

        return;

    int mid = (left + right) / 2;

    MergeSort(L1, L2, left, mid);

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

    Merge(L1, L2, left, mid, right);

}

·算法分析

空间开销上，有递归工作栈和辅助序列两个来源，需要O(n+log**2** n)，故空间复杂度为O(n)。

由于归并时相等元素不进行交换，故不会破坏元素排列的稳定性，

**2.3.10 LSD基数排序**

·算法思路

设一整数的十进制位数表示为k1,k2...kd（d是排序码最大位数），其中kd为高位，k1为低位。第1趟根据每一个元素的k1将所有元素分到10个容器内，然后从0到9依次把容器内元素取出放回到原序列，这样元素在k1上就是有序的。重复d趟后元素在所有位上都是有序的，排序也就完成了。

·代码实现

//取元素的某一位数（0为最低位）

template<class \_class>

int getDigit(\_class x, const int i)

{

    int num = 1;

    for (int j = 0; j < i; j++)

        num \*= 10;

    if (x < num)

        return 0;

    return (x - x % num) / num % 10;

}

//LSD基数排序

template<class \_class>

void LSDSort(DataList<\_class>& list, int d)

{

    LinkedQueue<\_class> queues[10];

    for (int i = 0; i < d; i++) {

        for (int j = 0; j < list.Length(); j++) {

            const int k = getDigit(list[j], i);

            queues[k].EnQueue(list[j]);

            Move++;

        }

        int current = 0;

        for (int j = 0; j < 10; j++) {

            while (!queues[j].IsEmpty()) {

                queues[j].DeQueue(list[current]);

                Move++;

                current++;

            }

        }

    }

}

·算法分析

时间开销上，在每趟进行分配和收集时都需将n个元素进行移动，而若排序码有d位时需重复d趟，时间复杂度为O(n\*d)，比较适合元素个数较多而排序码位数较少的情况。

由于每次分配和收集都是按照待排序列的顺序执行，相等元素中，排在前面的总是会被先分配、先收集，故元素排列的稳定性不会被破坏，LSD基数排序是一种稳定的排序算法。

**2.3.11 各排序算法对比**

通过对各种排序算法实现方法的过程分析，可以得出它们的时间复杂度、空间复杂度和稳定性，如下所示：

|  |  |  |  |
| --- | --- | --- | --- |
| 算法 | 时间复杂度 | 空间复杂度 | 稳定性 |
| 冒泡排序 | O(n**2**) | O(1) | √ |
| 选择排序 | O(n**2**) | O(1) | × |
| 直接插入排序 | O(n**2**) | O(1) | √ |
| 折半插入排序 | O(n**2**) | O(1) | √ |
| 希尔排序 | O(n**1.25**) | O(1) | × |
| 快速排序 | O(nlog**2** n) | O(log**2** n) | × |
| 改进快速排序 | O(nlog**2** n) | O(log**2** n) | × |
| 堆排序 | O(nlog**2** n) | O(1) | × |
| 归并排序 | O(nlog**2** n) | O(n) | √ |
| LSD基数排序 | O(n\*d) | O(n) | √ |

**3.项目测试**

3.1 10个数的序列测试

随机序列测试结果：



升序序列测试结果：



降序序列测试结果：



3.2 100个数的序列测试

随机序列测试结果：



升序序列测试结果：



降序序列测试结果：



3.3 1000个数的序列测试

随机序列测试结果：



升序序列测试结果：



降序序列测试结果：



3.4 10000个数的序列测试

随机序列测试结果：



升序序列测试结果：



降序序列测试结果：



3.5 100000个数的序列测试

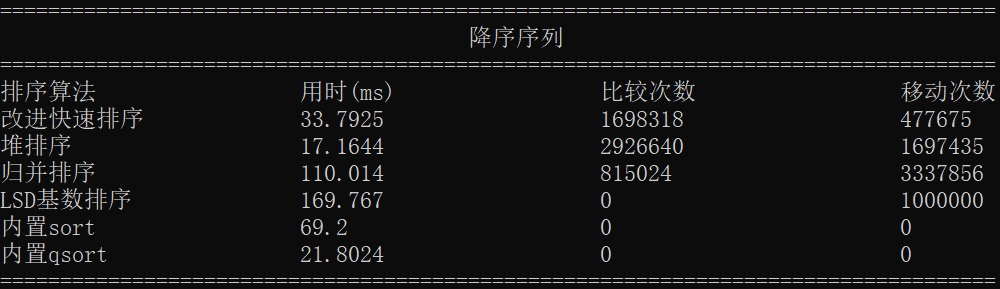
随机序列测试结果：



升序序列测试结果：



降序序列测试结果：



3.6 1000000个数的序列测试

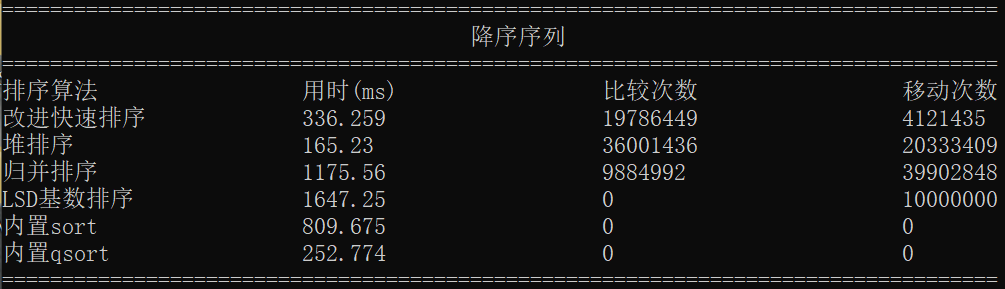
随机序列测试结果：



升序序列测试结果：



降序序列测试结果：



3.7 10000000个数的序列测试

随机序列测试结果：



升序序列测试结果：



降序序列测试结果：

