目录

[**数据结构与算法** 2](#_Toc24027535)

[**一、十大经典排序算法** 2](#_Toc24027536)

[**（一）排序算法说明** 2](#_Toc24027537)

[**（二）冒泡排序（Bubble Sort）** 3](#_Toc24027538)

[**（三）选择排序（Selection Sort）** 3](#_Toc24027539)

[**（四）插入排序（Insertion Sort）** 4](#_Toc24027540)

[**（五）希尔排序（Shell Sort）** 4](#_Toc24027541)

[**（六）归并排序（Merge Sort）** 5](#_Toc24027542)

[**（七）快速排序（Quick Sort）** 6](#_Toc24027543)

[**（八）堆排序（Heap Sort）** 7](#_Toc24027544)

[**（九）计数排序（Counting Sort）** 8](#_Toc24027545)

[**（十）桶排序（Bucket Sort）** 9](#_Toc24027546)

[**（十一）基数排序（Radix Sort）** 10](#_Toc24027547)

[**二、树** 11](#_Toc24027548)

**数据结构与算法**

**一、[十大经典排序算法](https://www.cnblogs.com/guoyaohua/p/8600214.html)**

**（一）排序算法说明**

**1、排序的定义**

对一序列对象根据某个关键字进行排序。

**2、 术语说明**

**稳定**：如果a原本在b前面，而a=b，排序之后a仍然在b的前面；

**不稳定**：如果a原本在b的前面，而a=b，排序之后a可能会出现在b的后面；

**内排序**：所有排序操作都在内存中完成；

**外排序**：由于数据太大，因此把数据放在磁盘中，而排序通过磁盘和内存的数据传输才能进行；

**时间复杂度：** 一个算法执行所耗费的时间。

**空间复杂度**：运行完一个程序所需内存的大小。

**3、算法总结**



**图片名词解释：**

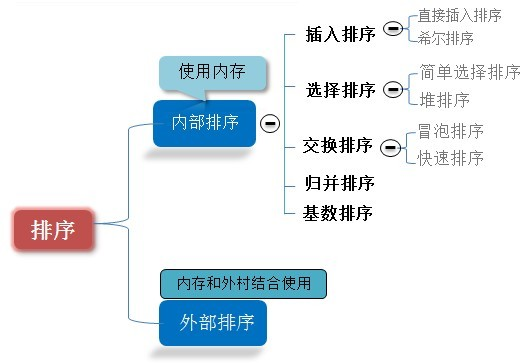
n: 数据规模

k: “桶”的个数

In-place: 占用常数内存，不占用额外内存

Out-place: 占用额外内存

**5、 算法分类**



**6 、比较和非比较的区别**

常见的快速排序、归并排序、堆排序、冒泡排序等属于比较排序。在排序的最终结果里，元素之间的次序依赖于它们之间的比较。每个数都必须和其他数进行比较，才能确定自己的位置。

在冒泡排序之类的排序中，问题规模为n，又因为需要比较n次，所以平均时间复杂度为O(n²)。在归并排序、快速排序之类的排序中，问题规模通过分治法消减为logN次，所以时间复杂度平均O(nlogn)。

比较排序的优势是，适用于各种规模的数据，也不在乎数据的分布，都能进行排序。可以说，比较排序适用于一切需要排序的情况。

计数排序、基数排序、桶排序则属于非比较排序。非比较排序是通过确定每个元素之前，应该有多少个元素来排序。针对数组arr，计算arr[i]之前有多少个元素，则唯一确定了arr[i]在排序后数组中的位置。

非比较排序只要确定每个元素之前的已有的元素个数即可，所有一次遍历即可解决。算法时间复杂度O(n)。  
**非比较排序时间复杂度底，但由于非比较排序需要占用空间来确定唯一位置。所以对数据规模和数据分布有一定的要求。**

**（二）冒泡排序（Bubble Sort）**

冒泡排序是一种简单的排序算法。它重复地走访过要排序的数列，一次比较两个元素，如果它们的顺序错误就把它们交换过来。走访数列的工作是重复地进行直到没有再需要交换，也就是说该数列已经排序完成。这个算法的名字由来是因为越小的元素会经由交换慢慢“浮”到数列的顶端。

**1、算法描述**

比较相邻的元素。如果第一个比第二个大，就交换它们两个；

对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对，这样在最后的元素应该会是最大的数；

针对所有的元素重复以上的步骤，除了最后一个；

重复步骤1~3，直到排序完成。

**2、代码实现**

1 　　/\*\*

2 \* 冒泡排序

6 \*/

7 public static int[] bubbleSort(int[] array) {

8 if (array.length == 0)

9 return array;

10 for (int i = 0; i < array.length; i++)

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

12 if (array[j + 1] < array[j]) {

13 int temp = array[j + 1];

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

15 array[j] = temp;

16 }

17 return array;

18 }

**3、 算法分析**

**最佳情况：T(n) = O(n)   最差情况：T(n) = O(n2)   平均情况：T(n) = O(n2)**

**（三）选择排序（Selection Sort）**

表现**最稳定的排序算法之一**，因为**无论什么数据进去都是O(n2)的时间复杂度**，所以用到它的时候，数据规模越小越好。唯一的好处可能就是不占用额外的内存空间了吧。理论上讲，选择排序可能也是平时排序一般人想到的最多的排序方法了吧。

选择排序(Selection-sort)是一种简单直观的排序算法。它的工作原理：首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。

**1、算法描述**

n个记录的直接选择排序可经过n-1趟直接选择排序得到有序结果。具体算法描述如下：

初始状态：无序区为R[1..n]，有序区为空；

第i趟排序(i=1,2,3…n-1)开始时，当前有序区和无序区分别为R[1..i-1]和R(i..n）。该趟排序从当前无序区中-选出关键字最小的记录 R[k]，将它与无序区的第1个记录R交换，使R[1..i]和R[i+1..n)分别变为记录个数增加1个的新有序区和记录个数减少1个的新无序区；

n-1趟结束，数组有序化了。

**2、代码实现**

　　/\*\*

\* 选择排序

\*/

public static int[] selectionSort(int[] array) {

if (array.length == 0)

return array;

for (int i = 0; i < array.length; i++) {

int minIndex = i;

for (int j = i; j < array.length; j++) {

if (array[j] < array[minIndex]) //找到最小的数

minIndex = j; //将最小数的索引保存

}

int temp = array[minIndex];

array[minIndex] = array[i];

array[i] = temp;

}

return array;

}

**3、算法分析**

**最佳情况：T(n) = O(n2)  最差情况：T(n) = O(n2)  平均情况：T(n) = O(n2)**

**（四）插入排序（Insertion Sort）**

插入排序（Insertion-Sort）的算法描述是一种简单直观的排序算法。它的工作原理是通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。插入排序在实现上，通常采用in-place排序（即只需用到O(1)的额外空间的排序），因而在从后向前扫描过程中，需要反复把已排序元素逐步向后挪位，为最新元素提供插入空间。

**1、算法描述**

一般来说，插入排序都采用in-place在数组上实现。具体算法描述如下：

从第一个元素开始，该元素可以认为已经被排序；

取出下一个元素，在已经排序的元素序列中从后向前扫描；

如果该元素（已排序）大于新元素，将该元素移到下一位置；

重复步骤3，直到找到已排序的元素小于或者等于新元素的位置；

将新元素插入到该位置后；

重复步骤2~5。

**2、代码实现**

　　/\*\*

\* 插入排序

\*/

public static int[] insertionSort(int[] array) {

if (array.length == 0)

return array;

int current;

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

current = array[i + 1];

int preIndex = i;

while (preIndex >= 0 && current < array[preIndex]) {

array[preIndex + 1] = array[preIndex];

preIndex--;

}

array[preIndex + 1] = current;

}

return array;

}

**4、算法分析**

最佳情况：T(n) = O(n)   最坏情况：T(n) = O(n2)   平均情况：T(n) = O(n2)

**（五）希尔排序（Shell Sort）**

希尔排序是希尔（Donald Shell）于1959年提出的一种排序算法。希尔排序也是一种插入排序，它是简单插入排序经过改进之后的一个更高效的版本，也称为缩小增量排序，同时该算法是冲破O(n2）的第一批算法之一。它与插入排序的不同之处在于，它会优先比较距离较远的元素。希尔排序又叫缩小增量排序。

**希尔排序是把记录按下表的一定增量分组，对每组使用直接插入排序算法排序；随着增量逐渐减少，每组包含的关键词越来越多，当增量减至1时，整个文件恰被分成一组，算法便终止。**

**1、算法描述**

我们来看下希尔排序的基本步骤，在此我们选择增量gap=length/2，缩小增量继续以gap = gap/2的方式，这种增量选择我们可以用一个序列来表示，**{n/2,(n/2)/2...1}**，称为**增量序列**。希尔排序的增量序列的选择与证明是个数学难题，我们选择的这个增量序列是比较常用的，也是希尔建议的增量，称为希尔增量，但其实这个增量序列不是最优的。此处我们做示例使用希尔增量。

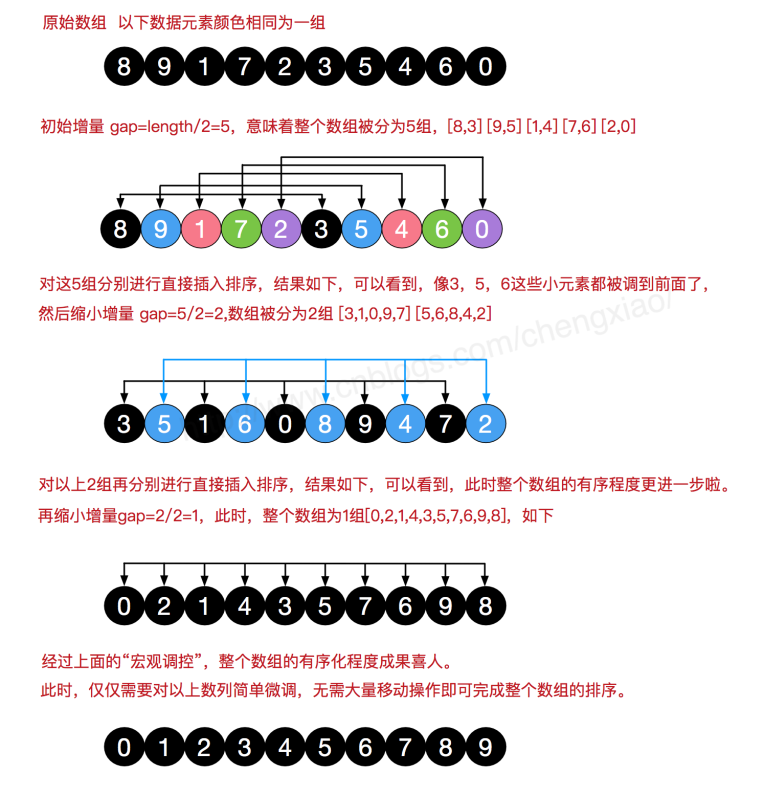
先将整个待排序的记录序列分割成为若干子序列分别进行直接插入排序，具体算法描述：

选择一个增量序列t1，t2，…，tk，其中ti>tj，tk=1；

按增量序列个数k，对序列进行k 趟排序；

每趟排序，根据对应的增量ti，将待排序列分割成若干长度为m 的子序列，分别对各子表进行直接插入排序。仅增量因子为1 时，整个序列作为一个表来处理，表长度即为整个序列的长度。

**2、过程演示**



**3、代码实现**

　/\*\*

\* 希尔排序

\*/

public static int[] ShellSort(int[] array) {

int len = array.length;

int temp, gap = len / 2;

while (gap > 0) {

for (int i = gap; i < len; i++) {

temp = array[i];

int preIndex = i - gap;

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

array[preIndex + gap] = array[preIndex];

preIndex -= gap;

}

array[preIndex + gap] = temp;

}

gap /= 2;

}

return array;

}

**4、算法分析**

最佳情况：T(n) = O(nlog2 n)  最坏情况：T(n) = O(nlog2 n)  平均情况：T(n) =O(nlog2n)

**（六）归并排序（Merge Sort）**

和选择排序一样，归并排序的性能不受输入数据的影响，但表现比选择排序好的多，因为始终都是O(n log n）的时间复杂度。代价是需要额外的内存空间。

归并排序是建立在归并操作上的一种有效的排序算法。该算法是采用分治法（Divide and Conquer）的一个非常典型的应用。归并排序是一种稳定的排序方法。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2-路归并。

**1、 算法描述**

把长度为n的输入序列分成两个长度为n/2的子序列；

对这两个子序列分别采用归并排序；

将两个排序好的子序列合并成一个最终的排序序列。

**2、代码实现**

　　/\*\*

\* 归并排序

\*/

public static int[] MergeSort(int[] array) {

if (array.length < 2) return array;

int mid = array.length / 2;

int[] left = Arrays.copyOfRange(array, 0, mid);

int[] right = Arrays.copyOfRange(array, mid, array.length);

return merge(MergeSort(left), MergeSort(right));

}

/\*\*

\* 归并排序——将两段排序好的数组结合成一个排序数组

\*/

public static int[] merge(int[] left, int[] right) {

int[] result = new int[left.length + right.length];

for (int index = 0, i = 0, j = 0; index < result.length; index++) {

if (i >= left.length)

result[index] = right[j++];

else if (j >= right.length)

result[index] = left[i++];

else if (left[i] > right[j])

result[index] = right[j++];

else

result[index] = left[i++];

}

return result;

}

**3、 算法分析**

最佳情况：T(n) = O(n)  最差情况：T(n) = O(nlogn)  平均情况：T(n) = O(nlogn)

**（七）快速排序（Quick Sort）**

快速排序的基本思想：通过一趟排序将待排记录分隔成独立的两部分，其中一部分记录的关键字均比另一部分的关键字小，则可分别对这两部分记录继续进行排序，以达到整个序列有序。

**1、算法描述**

快速排序使用分治法来把一个串（list）分为两个子串（sub-lists）。具体算法描述如下：

从数列中挑出一个元素，称为 “基准”（**pivot**）；

重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区（partition）操作；

递归地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序。

**2、 代码实现**

/\*\*

\* 快速排序方法

\*/

public static int[] QuickSort(int[] array, int start, int end) {

if (array.length < 1 || start < 0 || end >= array.length || start > end) return null;

int smallIndex = partition(array, start, end);

if (smallIndex > start)

QuickSort(array, start, smallIndex - 1);

if (smallIndex < end)

QuickSort(array, smallIndex + 1, end);

return array;

}

/\*\*

\* 快速排序算法——partition

\*/

public static int partition(int[] array, int start, int end) {

int pivot = (int) (start + Math.random() \* (end - start + 1));

int smallIndex = start - 1;

swap(array, pivot, end);

for (int i = start; i <= end; i++)

if (array[i] <= array[end]) {

smallIndex++;

if (i > smallIndex)

swap(array, i, smallIndex);

}

return smallIndex;

}

/\*\*

\* 交换数组内两个元素

\*/

public static void swap(int[] array, int i, int j) {

int temp = array[i];

array[i] = array[j];

array[j] = temp;

}

**3、 算法分析**

最佳情况：T(n) = O(nlogn)   最差情况：T(n) = O(n2)   平均情况：T(n) = O(nlogn)

**（八）堆排序（Heap Sort）**

堆排序（Heapsort）是指利用堆这种数据结构所设计的一种排序算法。堆积是一个近似完全二叉树的结构，并同时满足堆积的性质：即子结点的键值或索引总是小于（或者大于）它的父节点。

**1、算法描述**

将初始待排序关键字序列(R1,R2….Rn)构建成大顶堆，此堆为初始的无序区；

将堆顶元素R[1]与最后一个元素R[n]交换，此时得到新的无序区(R1,R2,……Rn-1)和新的有序区(Rn),且满足R[1,2…n-1]<=R[n]；

由于交换后新的堆顶R[1]可能违反堆的性质，因此需要对当前无序区(R1,R2,……Rn-1)调整为新堆，然后再次将R[1]与无序区最后一个元素交换，得到新的无序区(R1,R2….Rn-2)和新的有序区(Rn-1,Rn)。不断重复此过程直到有序区的元素个数为n-1，则整个排序过程完成。

**2、代码实现**

注意：这里用到了完全二叉树的部分性质

//声明全局变量，用于记录数组array的长度；

static int len;

/\*\*

\* 堆排序算法

\*/

public static int[] HeapSort(int[] array) {

len = array.length;

if (len < 1) return array;

//1.构建一个最大堆

buildMaxHeap(array);

//2.循环将堆首位（最大值）与末位交换，然后在重新调整最大堆

while (len > 0) {

swap(array, 0, len - 1);

len--;

adjustHeap(array, 0);

}

return array;

}

/\*\*

\* 建立最大堆

\*/

public static void buildMaxHeap(int[] array) {

//从最后一个非叶子节点开始向上构造最大堆

for (int i = (len/2 - 1); i >= 0; i--) { //感谢 @让我发会呆 网友的提醒，此处应该为 i = (len/2 - 1)

adjustHeap(array, i);

}

}

/\*\*

\* 调整使之成为最大堆

\*/

public static void adjustHeap(int[] array, int i) {

int maxIndex = i;

//如果有左子树，且左子树大于父节点，则将最大指针指向左子树

if (i \* 2 < len && array[i \* 2] > array[maxIndex])

maxIndex = i \* 2;

//如果有右子树，且右子树大于父节点，则将最大指针指向右子树

if (i \* 2 + 1 < len && array[i \* 2 + 1] > array[maxIndex])

maxIndex = i \* 2 + 1;

//如果父节点不是最大值，则将父节点与最大值交换，并且递归调整与父节点交换的位置。

if (maxIndex != i) {

swap(array, maxIndex, i);

adjustHeap(array, maxIndex);

}

}

**4、算法分析**

最佳情况：T(n) = O(nlogn) 最差情况：T(n) = O(nlogn) 平均情况：T(n) = O(nlogn)

**（九）计数排序（Counting Sort）**

计数排序的核心在于将输入的数据值转化为键存储在额外开辟的数组空间中。 作为一种线性时间复杂度的排序，计数排序要求输入的数据必须是有确定范围的整数。

计数排序(Counting sort)是一种稳定的排序算法。计数排序使用一个额外的数组C，其中第i个元素是待排序数组A中值等于i的元素的个数。然后根据数组C来将A中的元素排到正确的位置。它只能对整数进行排序。

**1、 算法描述**

找出待排序的数组中最大和最小的元素；

统计数组中每个值为i的元素出现的次数，存入数组C的第i项；

对所有的计数累加（从C中的第一个元素开始，每一项和前一项相加）；

反向填充目标数组：将每个元素i放在新数组的第C(i)项，每放一个元素就将C(i)减去1。

**2 、代码实现**

/\*\*

\* 计数排序

\*/

public static int[] CountingSort(int[] array) {

if (array.length == 0) return array;

int bias, min = array[0], max = array[0];

for (int i = 1; i < array.length; i++) {

if (array[i] > max)

max = array[i];

if (array[i] < min)

min = array[i];

}

bias = 0 - min;

int[] bucket = new int[max - min + 1];

Arrays.fill(bucket, 0);

for (int i = 0; i < array.length; i++) {

bucket[array[i] + bias]++;

}

int index = 0, i = 0;

while (index < array.length) {

if (bucket[i] != 0) {

array[index] = i - bias;

bucket[i]--;

index++;

} else

i++;

}

return array;

}

**3、 算法分析**

当输入的元素是n 个0到k之间的整数时，它的运行时间是 O(n + k)。计数排序不是比较排序，排序的速度快于任何比较排序算法。由于用来计数的数组C的长度取决于待排序数组中数据的范围（等于待排序数组的最大值与最小值的差加上1），这使得计数排序对于数据范围很大的数组，需要大量时间和内存。

最佳情况：T(n) = O(n+k)  最差情况：T(n) = O(n+k)  平均情况：T(n) = O(n+k)

**（十）桶排序（Bucket Sort）**

桶排序是计数排序的升级版。它利用了函数的映射关系，高效与否的关键就在于这个映射函数的确定。

桶排序 (Bucket sort)的工作的原理：假设输入数据服从均匀分布，将数据分到有限数量的桶里，每个桶再分别排序（有可能再使用别的排序算法或是以递归方式继续使用桶排序进行排

**1、 算法描述**

人为设置一个BucketSize，作为每个桶所能放置多少个不同数值（例如当BucketSize==5时，该桶可以存放｛1,2,3,4,5｝这几种数字，但是容量不限，即可以存放100个3）；

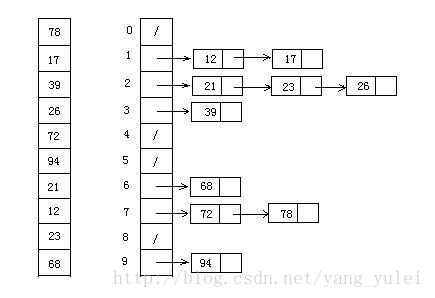
遍历输入数据，并且把数据一个一个放到对应的桶里去；

对每个不是空的桶进行排序，可以使用其它排序方法，也可以递归使用桶排序；

从不是空的桶里把排好序的数据拼接起来。

注意，如果递归使用桶排序为各个桶排序，则当桶数量为1时要手动减小BucketSize增加下一循环桶的数量，否则会陷入死循环，导致内存溢出。

**2 、图片演示**



**3、 代码实现**

/\*\*

\* 桶排序

\*/

public static ArrayList<Integer> BucketSort(ArrayList<Integer> array, int bucketSize) {

if (array == null || array.size() < 2)

return array;

int max = array.get(0), min = array.get(0);

// 找到最大值最小值

for (int i = 0; i < array.size(); i++) {

if (array.get(i) > max)

max = array.get(i);

if (array.get(i) < min)

min = array.get(i);

}

int bucketCount = (max - min) / bucketSize + 1;

ArrayList<ArrayList<Integer>> bucketArr = new ArrayList<>(bucketCount);

ArrayList<Integer> resultArr = new ArrayList<>();

for (int i = 0; i < bucketCount; i++) {

bucketArr.add(new ArrayList<Integer>());

}

for (int i = 0; i < array.size(); i++) {

bucketArr.get((array.get(i) - min) / bucketSize).add(array.get(i));

}

for (int i = 0; i < bucketCount; i++) {

if (bucketSize == 1) { // 如果带排序数组中有重复数字时 有错误

for (int j = 0; j < bucketArr.get(i).size(); j++)

resultArr.add(bucketArr.get(i).get(j));

} else {

if (bucketCount == 1)

bucketSize--;

ArrayList<Integer> temp = BucketSort(bucketArr.get(i), bucketSize);

for (int j = 0; j < temp.size(); j++)

resultArr.add(temp.get(j));

}

}

return resultArr;

}

**4、算法分析**

桶排序最好情况下使用线性时间O(n)，桶排序的时间复杂度，取决与对各个桶之间数据进行排序的时间复杂度，因为其它部分的时间复杂度都为O(n)。很显然，桶划分的越小，各个桶之间的数据越少，排序所用的时间也会越少。但相应的空间消耗就会增大。

最佳情况：T(n) = O(n+k)   最差情况：T(n) = O(n+k)   平均情况：T(n) = O(n2)

**（十一）基数排序（Radix Sort）**

基数排序也是非比较的排序算法，对每一位进行排序，从最低位开始排序，复杂度为O(kn),为数组长度，k为数组中的数的最大的位数；

基数排序是按照低位先排序，然后收集；再按照高位排序，然后再收集；依次类推，直到最高位。有时候有些属性是有优先级顺序的，先按低优先级排序，再按高优先级排序。最后的次序就是高优先级高的在前，高优先级相同的低优先级高的在前。基数排序基于分别排序，分别收集，所以是稳定的。

**1、算法描述**

取得数组中的最大数，并取得位数；

arr为原始数组，从最低位开始取每个位组成radix数组；

对radix进行计数排序（利用计数排序适用于小范围数的特点）；

**2、代码实现**

　　public static int[] RadixSort(int[] array) {

if (array == null || array.length < 2)

return array;

// 1.先算出最大数的位数；

int max = array[0];

for (int i = 1; i < array.length; i++) {

max = Math.max(max, array[i]);

}

int maxDigit = 0;

while (max != 0) {

max /= 10;

maxDigit++;

}

int mod = 10, div = 1;

ArrayList<ArrayList<Integer>> bucketList = new ArrayList<ArrayList<Integer>>();

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

bucketList.add(new ArrayList<Integer>());

for (int i = 0; i < maxDigit; i++, mod \*= 10, div \*= 10) {

for (int j = 0; j < array.length; j++) {

int num = (array[j] % mod) / div;

bucketList.get(num).add(array[j]);

}

int index = 0;

for (int j = 0; j < bucketList.size(); j++) {

for (int k = 0; k < bucketList.get(j).size(); k++)

array[index++] = bucketList.get(j).get(k);

bucketList.get(j).clear();

}

}

return array;

}

**4、算法分析**

最佳情况：T(n) = O(n \* k)   最差情况：T(n) = O(n \* k)   平均情况：T(n) = O(n \* k)

基数排序有两种方法：

MSD 从高位开始进行排序 LSD 从低位开始进行排序

**5、基数排序、计数排序、桶排序比较**

这三种排序算法都利用了桶的概念，但对桶的使用方法上有明显差异：

基数排序：根据键值的每位数字来分配桶

计数排序：每个桶只存储单一键值

桶排序：每个桶存储一定范围的数值

**二、树**

**二叉树**

二叉树：每个节点最多含有两个子树的树称为二叉树。（我们一般在书中试题中见到的树是二叉树，但并不意味着所有的树都是二叉树。）

在二叉树的概念下又衍生出满二叉树和完全二叉树的概念

满二叉树：除最后一层无任何子节点外，每一层上的所有结点都有两个子结点。也可以这样理解，除叶子结点外的所有结点均有两个子结点。节点数达到最大值，所有叶子结点必须在同一层上

完全二叉树：若设二叉树的深度为h，除第 h 层外，其它各层 (1～(h-1)层) 的结点数都达到最大个数，第h层所有的结点都连续集中在最左边，这就是完全二叉树。

算法实现（

二叉树：

 private static class TreeNode {

        int val;

        TreeNode left;

        TreeNode right;

TreeNode(int x) { val = x; }

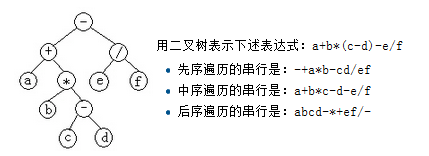
}

二叉树的遍历方式

先序遍历：先根节点->遍历左子树->遍历右子树

中序遍历：遍历左子树->根节点->遍历右子树

后序遍历：遍历左子树->遍历右子树->根节点



深度优先搜索（DFS）与广度优先搜索（BFS）

实现：bfs＝队列，入队列，出队列 一次访问一条路径；dfs=栈，压栈，出栈 一次访问多条路径（来自知乎）

关系：用DFS解决的问题都可以用BFS解决。DFS易于编写（递归），时间消耗较少但是容易发生爆栈，而BFS可以控制队列的长度。

**2.动态查找树**

2.1 二叉查找树

二叉查找树是二叉树的衍生概念：

二叉查找树（英语：Binary Search Tree），也称为二叉搜索树、有序二叉树（ordered binary tree）或排序二叉树（sorted binary tree），是指一棵空树或者具有下列性质的二叉树：

    1.若任意节点的左子树不空，则左子树上所有节点的值均小于它的根节点的值；

   2. 若任意节点的右子树不空，则右子树上所有节点的值均大于它的根节点的值；

    3.任意节点的左、右子树也分别为二叉查找树；

    4.没有键值相等的节点。

二叉查找树相比于其他数据结构的优势在于查找、插入的时间复杂度较低为 O ( log ⁡ n ) 。二叉查找树是基础性数据结构，用于构建更为抽象的数据结构，如集合、多重集、关联数组等。

2.2 平衡二叉树（AVL树）

平衡二叉树：当且仅当任何节点的两棵子树的高度差不大于1的二叉树；

其中AVL树是最先发明的自平衡二叉查找树，是最原始典型的平衡二叉树。

平衡二叉树是基于二叉查找树的改进。由于在某些极端的情况下（如在插入的序列是有序的时），二叉查找树将退化成近似链或链，此时，其操作的时间复杂度将退化成线性的，即O(n)。所以通过自平衡操作（即旋转）构建两个子树高度差不超过1的平衡二叉树。

**2.3 红黑树**

红黑树也是一种自平衡的二叉查找树。

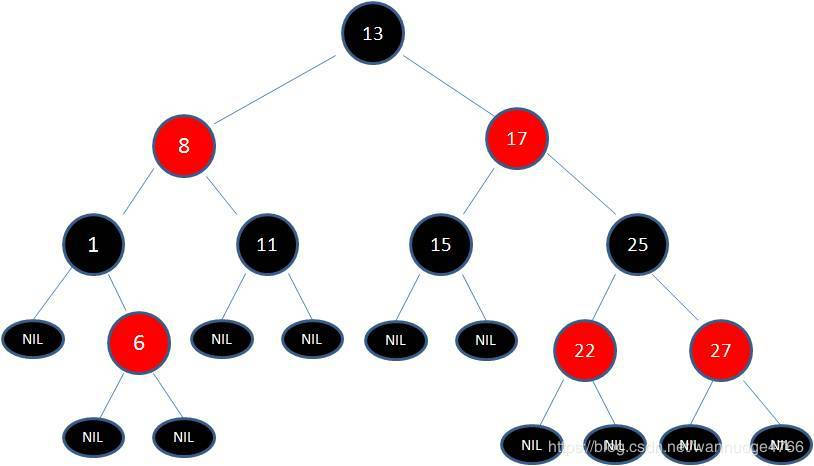
    1.每个结点要么是红的要么是黑的。（红或黑）

    2.根结点是黑的。  （根黑）

    3.每个叶结点（叶结点即指树尾端NIL指针或NULL结点）都是黑的。  （叶黑）

    4.如果一个结点是红的，那么它的两个儿子都是黑的。  （红子黑）

     5.对于任意结点而言，其到叶结点树尾端NIL指针的每条路径都包含相同数目的黑结点。（路径下黑相同）



如图就是一棵典型的红黑树。保证红黑树满足它的基本性质，就是在调整数据结构自平衡。

而红黑树自平衡的调整操作方式就有旋转和变色两种。

红黑树是一种应用很广的数据结构，如在Java集合类中TreeSet和TreeMap的底层，C++STL中set与map，以及linux中虚拟内存的管理。

2.4 哈夫曼树（Huffman Tree）

哈夫曼树是一种带权路径长度最短的二叉树，也称为最优二叉树。

一般可以按下面步骤构建：

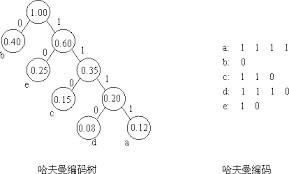
1，将所有左，右子树都为空的作为根节点。

2，在森林中选出两棵根节点的权值最小的树作为一棵新树的左，右子树，且置新树的附加根节点的权值为其左，右子树上根节点的权值之和。注意，左子树的权值应小于右子树的权值。

3，从森林中删除这两棵树，同时把新树加入到森林中。

4，重复2，3步骤，直到森林中只有一棵树为止，此树便是哈夫曼树。

大家可能更多听说的是哈夫曼编码，其实就是哈夫曼树的应用。即如何让电文中出现较多的字符采用尽可能短的编码且保证在译码时不出现歧义。



**3.多路查找树**

大规模数据存储中，实现索引查询这样一个实际背景下，树节点存储的元素数量是有限的（如果元素数量非常多的话，查找就退化成节点内部的线性查找了），这样导致二叉查找树结构由于树的深度过大而造成磁盘I/O读写过于频繁，进而导致查询效率低下。

**3.1 B树**

B树（英语：B-tree）是一种自平衡的树，能够保持数据有序。这种数据结构能够让查找数据、顺序访问、插入数据及删除的动作，都在对数时间内完成。B树，概括来说是一个一般化的二叉查找树（binary search tree），可以拥有最多2个子节点。与自平衡二叉查找树不同，B树适用于读写相对大的数据块的存储系统，例如磁盘。

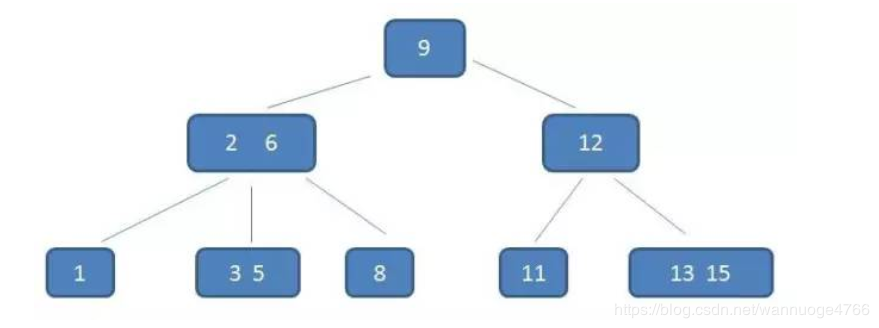
1.根结点至少有两个子女。

2.每个中间节点都包含k-1个元素和k个孩子，其中 m/2 <= k <= m

3.每一个叶子节点都包含k-1个元素，其中 m/2 <= k <= m

4.所有的叶子结点都位于同一层。

5.每个节点中的元素从小到大排列，节点当中k-1个元素正好是k个孩子包含的元素的值域分划。



如图所示就是一颗符合规范的B树，由于相比于磁盘IO的速度，内存中的耗时几乎可以省略，所以只要树的高度足够低，IO次数足够小，就可以提升查询性能。

B树的增加删除同样遵循自平衡的性质，有旋转和换位。

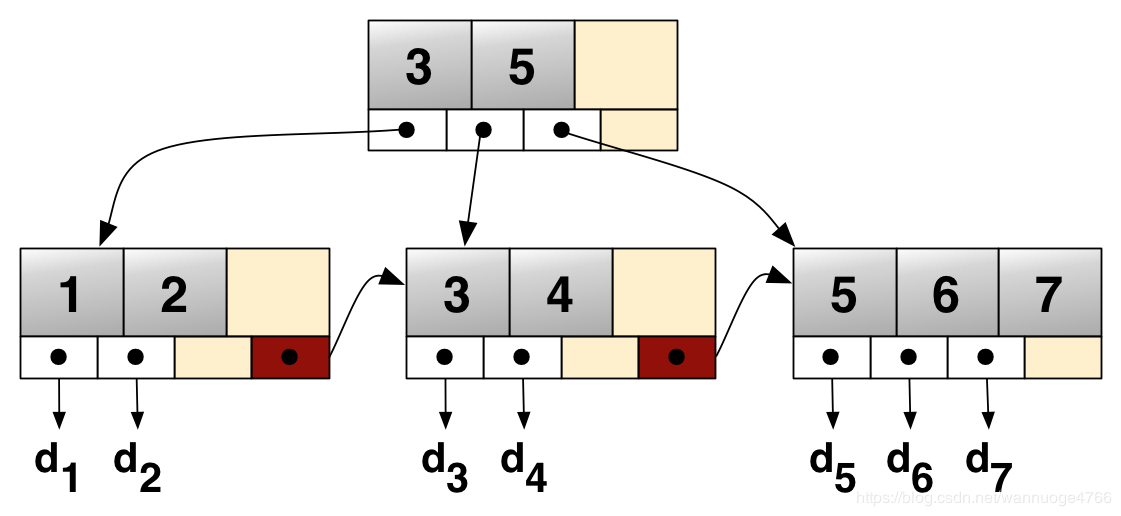
B树的应用是文件系统及部分非关系型数据库索引。

**3.2 B+树**

B+ 树是一种树数据结构，通常用于关系型数据库（如Mysql）和操作系统的文件系统中。B+ 树的特点是能够保持数据稳定有序，其插入与修改拥有较稳定的对数时间复杂度。B+ 树元素自底向上插入，这与二叉树恰好相反。

在B树基础上，为叶子结点增加链表指针（B树+叶子有序链表），所有关键字都在叶子结点 中出现，非叶子结点作为叶子结点的索引；B+树总是到叶子结点才命中。

b+树的非叶子节点不保存数据，只保存子树的临界值（最大或者最小），所以同样大小的节点，b+树相对于b树能够有更多的分支，使得这棵树更加矮胖，查询时做的IO操作次数也更少。



这通常在多数节点在次级存储比如硬盘中的时候出现。通过最大化在每个内部节点内的子节点的数目减少树的高度，平衡操作不经常发生，而且效率增加了。

**3.3 B\*树**

B\*树是B+树的变体，在B+树的非根和非叶子结点再增加指向兄弟的指针

在B+树基础上，为非叶子结点也增加链表指针，将结点的最低利用率从1/2提高到2/3。

**3.4 R树**

R树是用来做空间数据存储的树状数据结构。例如给地理位置，矩形和多边形这类多维数据建立索引。

R树的核心思想是聚合距离相近的节点并在树结构的上一层将其表示为这些节点的最小外接矩形（MBR），这个最小外接矩形就成为上一层的一个节点。因为所有节点都在它们的最小外接矩形中，所以跟某个矩形不相交的查询就一定跟这个矩形中的所有节点都不相交。叶子节点上的每个矩形都代表一个对象，节点都是对象的聚合，并且越往上层聚合的对象就越多。也可以把每一层看做是对数据集的近似，叶子节点层是最细粒度的近似，与数据集相似度100%，越往上层越粗糙。

