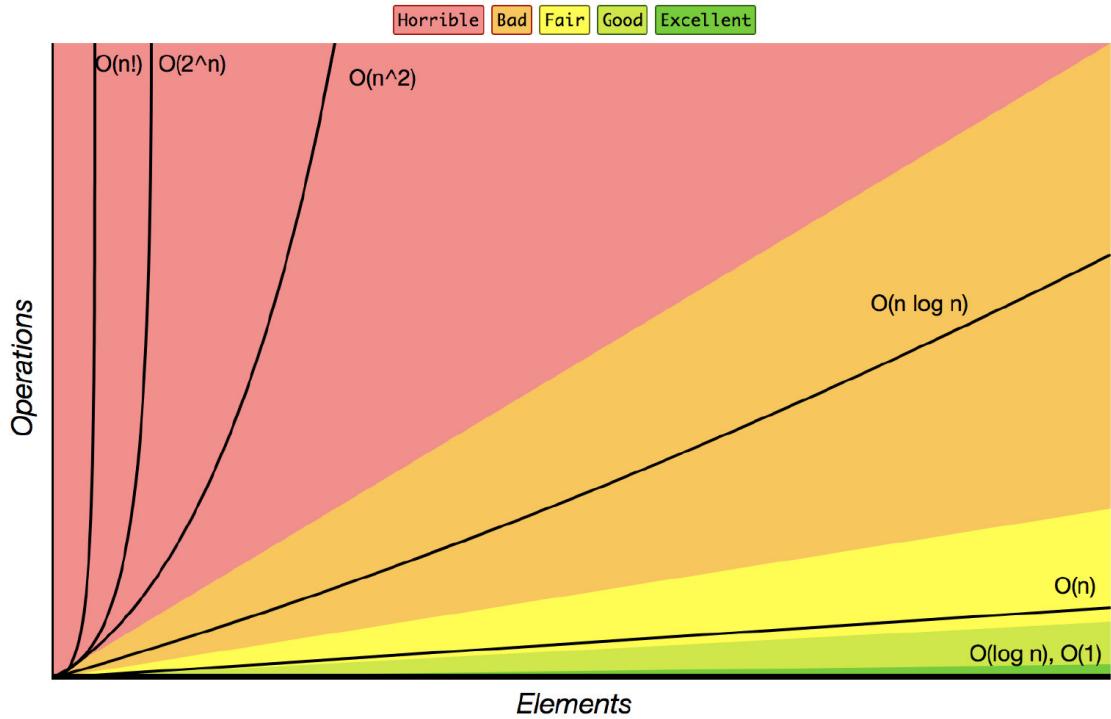


Big-O Complexity Chart



排序算法比较

常用的排序算法

各排序算法的复杂度

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定

算法	稳定性	时间复杂度	空间复杂度	备注
选择排序	✗	N^2	1	
冒泡排序	✓	N^2	1	
插入排序	✓	$N \sim N^2$	1	时间复杂度和初始顺序有关
希尔排序	✗	N 的若干倍乘于递增序列的长度	1	改进版插入排序
快速排序	✗	$N \log N$	$\log N$	
三向切分快速排序	✗	$N \sim N \log N$	$\log N$	适用于有大量重复主键
归并排序	✓	$N \log N$	N	
堆排序	✗	$N \log N$	1	无法利用局部性原理
桶排序	✓	$N * (\log(N/M) + 1)$	$N+M$	均匀分布 $N=M$ 时复杂度为 N , 关键在于桶内排序算法
计数排序	✓	N	N	无法用于浮点数, 不适用于数组最小值和最大值相差过大的情况

快速排序是最快的通用排序算法, 它的内循环的指令很少, 而且它还能利用缓存, 因为它总是顺序地访问数据。它的运行时间近似为 $\sim cN \log N$, 这里的 c 比其它线性对数级别的排序算法都要小。

常见数据结构操作比较

Data Structure	Time Complexity								Space Complexity	
	Average				Worst					
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion		
Array	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	
Stack	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	
Queue	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	
Singly-Linked List	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	
Doubly-Linked List	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	
Skip List	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n \log(n))$	
Hash Table	N/A	$O(1)$	$O(1)$	$O(1)$	N/A	$O(n)$	$O(n)$	$O(n)$	$O(n)$	
Binary Search Tree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	
Cartesian Tree	N/A	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	N/A	$O(n)$	$O(n)$	$O(n)$	$O(n)$	
B-Tree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	
Red-Black Tree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	
Splay Tree	N/A	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	N/A	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	
AVL Tree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	
KD Tree	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	

堆的不同实现及其操作

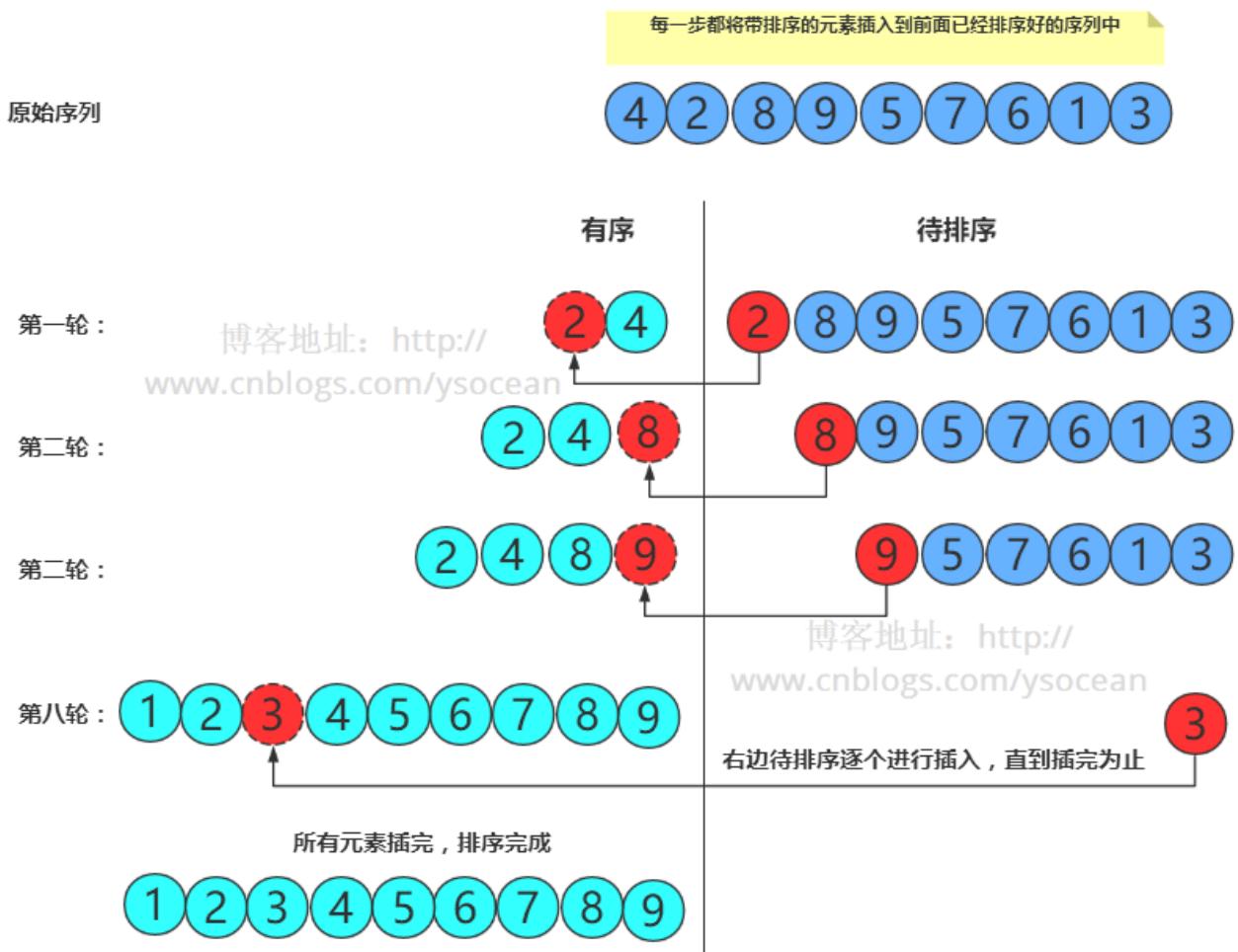
Operation	find-min	delete-min	insert	decrease-key	merge
Binary ^[8]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(n)$
Leftist	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(n)$	$\Theta(\log n)$
Binomial ^[8]	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$ ^[b]	$\Theta(\log n)$	$\Theta(\log n)$ ^[c]
Fibonacci ^{[8][9]}	$\Theta(1)$	$\Theta(\log n)$ ^[b]	$\Theta(1)$	$\Theta(1)$ ^[b]	$\Theta(1)$
Pairing ^[10]	$\Theta(1)$	$\Theta(\log n)$ ^[b]	$\Theta(1)$	$\Theta(\log n)$ ^{[b][d]}	$\Theta(1)$
Brodal ^{[13][e]}	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Rank-pairing ^[15]	$\Theta(1)$	$\Theta(\log n)$ ^[b]	$\Theta(1)$	$\Theta(1)$ ^[b]	$\Theta(1)$
Strict Fibonacci ^[16]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
2-3 heap	?	$\Theta(\log n)$ ^[b]	$\Theta(\log n)$ ^[b]	$\Theta(1)$?

插入排序

适用：小规模数据（小数组优于快排），基本有序

每次都将当前元素插入到左侧已经排序的数组中，使得插入之后左侧数组依然有序。

对于数组 {3, 5, 2, 4, 1}，它具有以下逆序：(3, 2), (3, 1), (5, 2), (5, 4), (5, 1), (2, 1), (4, 1)，插入排序每次只能交换相邻元素，令逆序数量减少 1，因此插入排序需要交换的次数为逆序数量。



```
public static int[] sort(int[] array){
```

```

int j;
//从下标为1的元素开始选择合适的位置插入，因为下标为0的只有一个元素，默认是有序的
for(int i = 1 ; i < array.length ; i++){
    int tmp = array[i];//记录要插入的数据
    j = i;
    while(j > 0 && tmp < array[j-1]){//从已经排序的序列最右边的开始比较，找到比其
        小的数
        array[j] = array[j-1];//向后挪动
        j--;
    }
    array[j] = tmp;//存在比其小的数，插入
}
return array;
}

```

排序，冒泡、选择、插入用大 O 表示法都需要 $O(N^2)$ 时间级别。一般不会选择冒泡排序，虽然冒泡排序书写是最简单的，但是平均性能是没有选择排序和插入排序好的。

选择排序把交换次数降低到最低，但是比较次数还是挺大的。当数据量小，并且交换数据相对于比较数据更加耗时的情况下，可以应用选择排序。

在大多数情况下，假设数据量比较小或基本有序时，插入排序是三种算法中最好的选择。

1	2	3	4	5	6	7	9	8
---	---	---	---	---	---	---	---	---

插入排序：

1. 第一趟比较 $n-1$ 次
2. 第二趟比较 8 和 7，发现 8 大，直接插入

冒泡排序：

1. 第一趟比较 $n-1$ 次
2. 第二趟 end 指向 8，再比较 $n-2$ 次

发现是一个升序序列（12345678），
并没有交换，则直接跳出循环

希尔排序

对于大规模的数组，插入排序很慢，因为它只能交换相邻的元素，每次只能将逆序数量减少 1。

针对插入排序缺点改进：大规模无序数据

希尔排序的出现就是为了解决插入排序的这种局限性，它通过交换不相邻的元素，每次可以将逆序数量减少大于 1。

希尔排序使用插入排序对间隔 h 的序列进行排序。通过不断减小 h ，最后令 $h=1$ ，就可以使得整个数组是有序的。

```
private static void shellSort(int[] arr) {  
    int N = arr.length;  
    // 进行分组，最开始时的增量(gap)为数组长度的一半  
    for (int gap = N/2; gap > 0 ; gap /= 2) {  
        // 对各个分组进行插入排序  
        for (int i = gap; i < N; i++) {  
            // 将 arr[i] 插入到所在分组的正确位置上  
            insertI(arr, gap, i);  
        }  
    }  
}
```

希尔排序的运行时间达不到平方级别，使用递增序列 $1, 4, 13, 40, \dots$ 的希尔排序所需要的比较次数不会超过 N 的若干倍乘于递增序列的长度。后面介绍的高级排序算法只会比希尔排序快两倍左右。

归并排序

该方法采用了分治的思想，比较适用于处理较大规模的数据，但比较耗内存（因为在合并的时候需要辅助数组）

时间复杂度：一般为 $O(N\log N)$ 。

```

/**
 * 将数组 arr[left] --> arr[right] 进行归并排序
 * @param arr 要排序的数组
 * @param temp 辅助数组
 * @param left 左边界
 * @param right 右边界
 */
private static void mergeSort(int[] arr, int[] temp, int left, int right) {
    // left == right 的时候，就递归到只有一个元素 --> 终止条件
    if (left < right) {
        // [分]: 将数组一分为二
        int center = (left+right)/2;
        // [治]: 将左边的数组排序(left --> center)
        mergeSort(arr, temp, left, center);
        // [治]: 将右边的数组排序(center+1 --> right)
        mergeSort(arr, temp, left: center+1,right);
        // [合]: 合并两个有序数组
        merge(arr, temp, left, center, right);
    }
}

/*
 * 将 arr[left...center] 和 arr[center+1...right] 两个有序数组合并为一个有序数组
 */
private static void merge(int[] arr, int [] temp, int left, int center, int right) {
    int i = left, j = center + 1;
    // 先通过比较将两个有序数组合并为一个有序数组，结果暂放到 temp 数组
    for (int k = left; k <= right;k++) {
        // 如果左边数组arr[left...center]中的元素取完[即比较完](i>center),
        // 则直接copy右边数组的元素到辅助数组, 右边数组同理
        if (i > center) { temp[k] = arr[j++]; }
        else if (j > right) { temp[k] = arr[i++]; }

        else if (arr[i] <= arr[j]) { temp[k] = arr[i++]; }
        else { temp[k] = arr[j++]; }
    }
    // 再将已经排好序的辅助数组中的值复制到原数组 arr 中
    for (int k = left; k <= right; k++) {
        arr[k] = temp[k];
    }
}

```

快速排序

算法改进

因为快速排序在小数组中也会递归调用自己，对于小数组，插入排序比快速排序的性能更好，因此在小数组中可以切换到插入排序。

三数取中

最好的情况下是每次都能取数组的中位数作为切分元素，但是计算中位数的代价很高。一种折中方法是取 3 个元素，并将大小居中的元素作为切分元素。

三向切分

对于有大量重复元素的数组，可以将数组切分为三部分，分别对应小于、等于和大于切分元素。

三向切分快速排序对于有大量重复元素的随机数组可以在线性时间内完成排序。

指针移动顺序问题

如果pivot选择数组最左边元素，则一定是right指针先左移，找小于pivot的，left指针再右移找大于pivot

这个移动顺序是为什么：

选择的是最左边元素作为pivot，而最后希望得到的是左边小于pivot，右边大于pivot

且pivot的位置就是right==left的位置，那么要求这个位置的数一定要是小于pivot的（因为要和pivot交换）

而如果先移动left的话，那么最后碰撞就可能会是两种情况

1. left碰撞到上一轮已交换后的right位置，但上一轮交换后right指向的大于pivot！！！
2. right碰撞到这一轮的未交换前left位置，但这一轮未交换前left指向的是大于pivot！！！

所以，以上两种情况都会导致大于pivot的数和pivot交换，同时pivot又是数组最左元素，即大于pivot元素被交换到了左数组，显然发生了错误

基于切分的快速选择算法 (Quick Select)

快速排序的 partition() 方法，会返回一个整数 j 使得 a[l..j-1] 小于等于 a[j]，且 a[j+1..h] 大于等于 a[j]，此时 a[j] 就是数组的第 j 大元素。

可以利用这个特性找出数组的第 k 个元素。

该算法是线性级别的，假设每次能将数组二分，那么比较的总次数为 $(N+N/2+N/4+..)$ ，直到找到第 k 个元素，这个和显然小于 $2N$ 。

```
public T select(T[] nums, int k) {  
    int left = 0, right = nums.length - 1;  
    while (right > left) {  
        int pivotIndex = partition(nums, left, right);  
        if (pivotIndex == k) {  
            return nums[k];  
        } else if (pivotIndex > k) {  
            right = pivotIndex - 1;  
        } else {  
            left = pivotIndex + 1;  
        }  
    }  
    return nums[k];
```

}

计数排序（简化版的桶排序）

时间复杂度：O(M+N)

缺点：不适用于浮点数，不适用于数组最小值和最大值相差过大的情况

计数排序需要根据原始数列的取值范围，创建一个统计数组，用来统计原始数列中每一个可能的整数值所出现的次数。

原始数列中的整数值，和统计数组的下标是一一对应的，以数列的最小值作为偏移量，最大值-最小值作为数组长度（节约空间）。比如原始数列的最小值是90，那么整数95对应的统计数组下标就是 $95 - 90 = 5$ 。

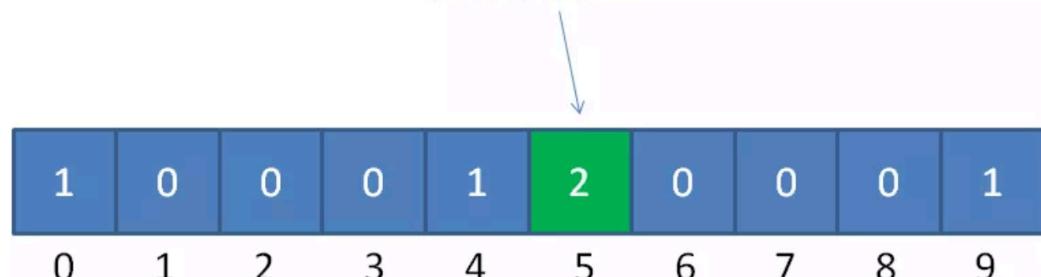
改进版计数排序

姓名	成绩
小灰	90
大黄	99
小红	95
小白	94
小绿	95

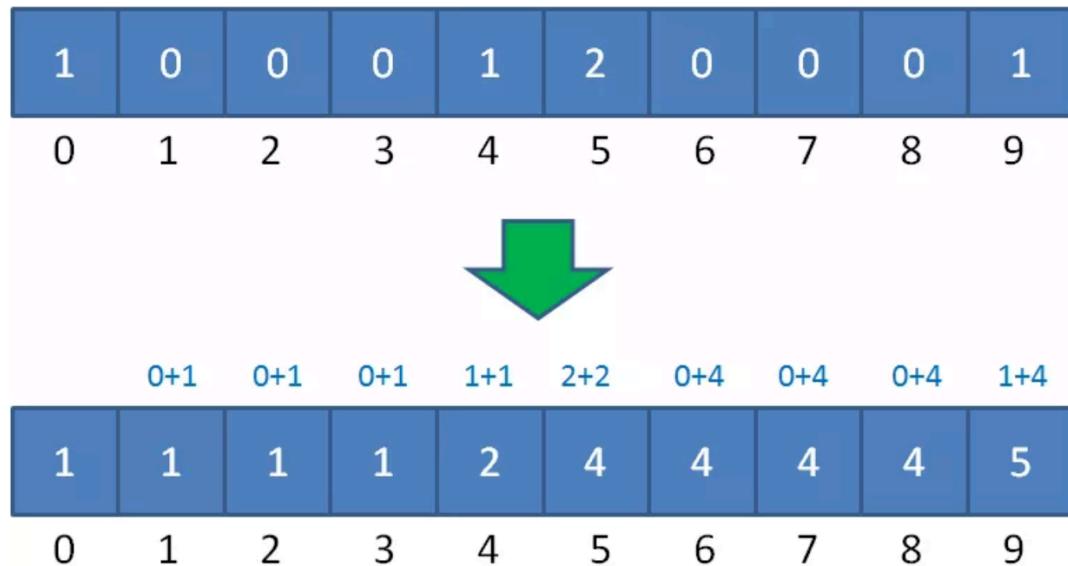
给定一个学生的成绩表，要求按成绩从低到高排序，如果成绩相同，则遵循原表固有顺序。

那么，当我们填充统计数组以后，我们只知道有两个成绩并列95分的小伙伴，却不知道哪一个是小红，哪一个是小绿：

有两个95分的学生，
究竟小红在前还是
小绿在前呢？



下面的讲解会有一些烧脑，请大家扶稳坐好。我们仍然以刚才的学生成绩表为例，把之前的统计数组变形形成下面的样子：



这是如何变形的呢？统计数组从第二个元素开始，每一个元素都加上前面所有元素之和。

为什么要相加呢？初次看到的小伙伴可能会觉得莫名其妙。

这样相加的目的，是让统计数组存储的元素值，等于相应整数的最终排序位置。比如下标是9的元素值为5，代表原始数列的整数9，最终的排序是在第5位。

第一步，我们遍历成绩表最后一行的小绿：

小绿是95分，我们找到countArray下标是5的元素，值是4，代表小绿的成绩排名位置在第4位。

同时，我们给countArray下标是5的元素值减1，从4变成3，，代表着下次再遇到95分的成绩时，最终排名是第3。

姓名	成绩
小灰	90
大黄	99
小红	95
小白	94
小绿	95



桶排序

桶元素分布均匀：当n=m时间复杂度可以达到O(n)

桶元素分布极不均匀：退化成O(nlogn)，相当于全在一个桶里排序，且浪费了很多桶空间

1.得到数列的最大值和最小值，并算出差值d

2.初始化桶

3.遍历原始数组，将每个元素放入桶中

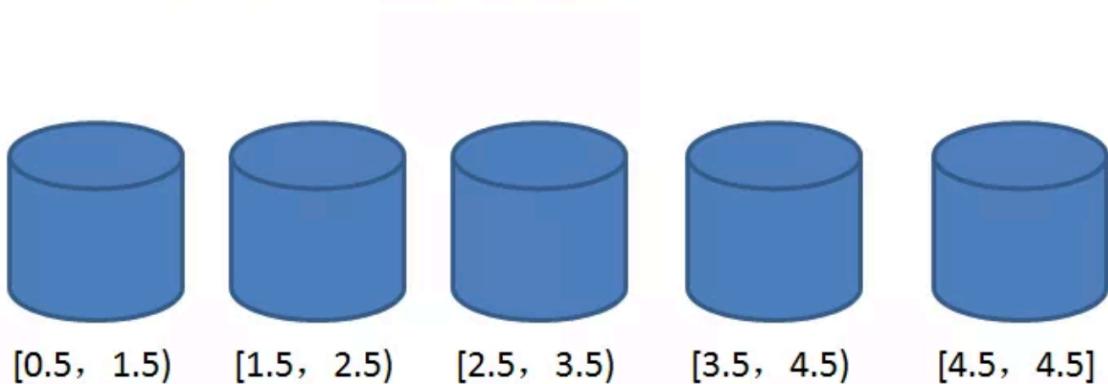
```
1. int num = (int)((array[i] - min) * (bucketNum-1) / d);  
2. bucketList.get(num).add(array[i]);
```

4.对每个桶内部进行排序（内部可以用Collections.sort()//JDK底层采用了归并排序或归并的优化版本）

5.输出全部元素

每一个桶 (bucket) 代表一个区间范围，里面可以承载一个或多个元素。桶排序的第一步，就是创建这些桶，确定每一个桶的区间范围：

4.5, 0.84, 3.25, 2.18, 0.5



具体建立多少个桶，如何确定桶的区间范围，有很多不同的方式。我们这里创建的桶数量等于原始数列的元素数量，除了最后一个桶只包含数列最大值，前面各个桶的区间按照比例确定。

区间跨度 = (最大值-最小值) / (桶的数量 - 1)

多源最短路径Floyd算法

时间复杂度：O(n^3)

用一句话概括就是：从 i 号顶点到 j 号顶点只经过前 k 号点的最短路程，其实这是一种“动态规划”的思想

```
for(k=1;k<=n;k++)
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
            if(e[i][j]>e[i][k]+e[k][j])
                e[i][j]=e[i][k]+e[k][j];
```

单源最短路径Dijkstra算法

时间复杂度：O(n^2)，若每次用堆来找离源点的最近节点，复杂度为 O($n \log n$)

图是稀疏图时：用邻接矩阵比邻接表块很多

算法的基本思想：每次找到离源点（上面例子的源点就是 1 号顶点）最近的一个顶点，然后以该顶点为中心进行扩展，最终得到源点到其余所有点的最短路径。基本步骤如下：

- 将所有的顶点分为两部分：已知最短路程的顶点集合 P 和未知最短路程的顶点集合 Q 。最开始，已知最短路程的顶点集合 P 中只有源点一个顶点。我们这里用一个 $book[i]$ 数组来记录哪些点在集合 P 中。例如对于某个顶点 i ，如果 $book[i]$ 为 1 则表示这个顶点在集合 P 中，如果 $book[i]$ 为 0 则表示这个顶点在集合 Q 中。
- 设置源点 s 到自己的最短路径为 0 即 $dis=0$ 。若存在源点有能直接到达的顶点 i ，则把 $dis[i]$ 设为

$e[s][i]$ 。同时把所有其它（源点不能直接到达的）顶点的最短路径为设为 ∞ 。

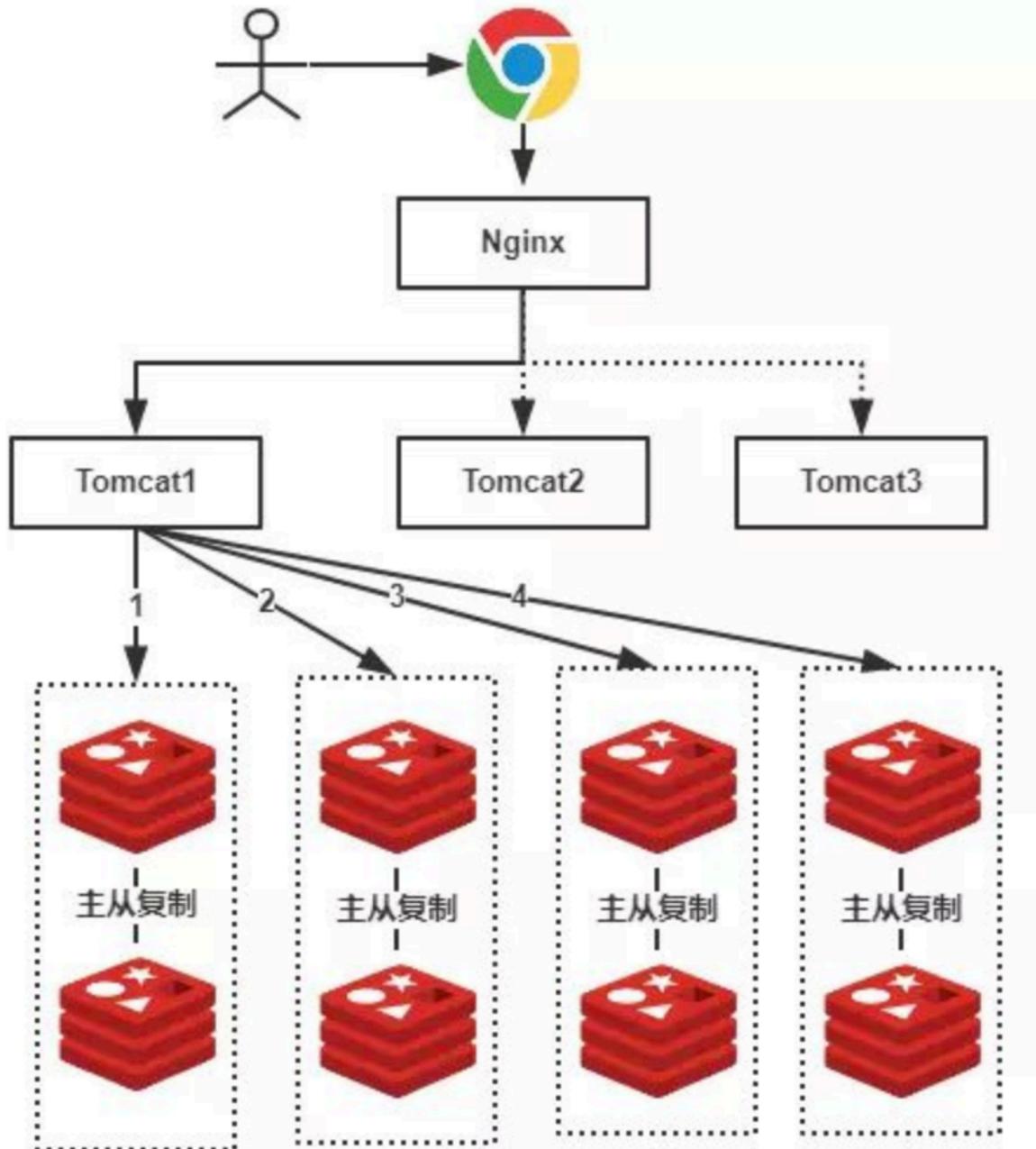
- 在集合 Q 的所有顶点中选择一个离源点 s 最近的顶点 u (即 $dis[u]$ 最小) 加入到集合 P。并考察所有以点 u 为起点的边，对每一条边进行松弛操作。例如存在一条从 u 到 v 的边，那么可以通过将边 $u \rightarrow v$ 添加到尾部来拓展一条从 s 到 v 的路径，这条路径的长度是 $dis[u] + e[u][v]$ 。如果这个值比目前已知的 $dis[v]$ 的值要小，我们可以用新值来替代当前 $dis[v]$ 中的值。
- 重复第 3 步，如果集合 Q 为空，算法结束。最终 dis 数组中的值就是源点到所有顶点的最短路径。

```
//Dijkstra算法核心语句
book[1]=1;
for(i=1;i<=n-1;i++) //最开始集合P只有源点到自己的最短距离是确认的，其他剩余节点都需要进行操作
{
    //找到离1号顶点最近的顶点
    min=Integer.MAX_VALUE;
    for(j=1;j<=n;j++)
    {
        if(book[j]==0 && dis[j]<min)
        {
            min=dis[j];
            u=j;
        }
    }
    book[u]=1;
    for(v=1;v<=n;v++) //对最近节点的所有边进行判断，用来松弛1号顶点到其他顶点的距离
    {
        if(e[u][v]<inf)
        {
            if(dis[v]>dis[u]+e[u][v])
                dis[v]=dis[u]+e[u][v];
        }
    }
}
```

一致性Hash算法

产生背景

一般来说单表数据量多于500w的时候，使用分库分表，采用主从复制，读写分离的策略。在来了一个请求的时候，比如a.png，

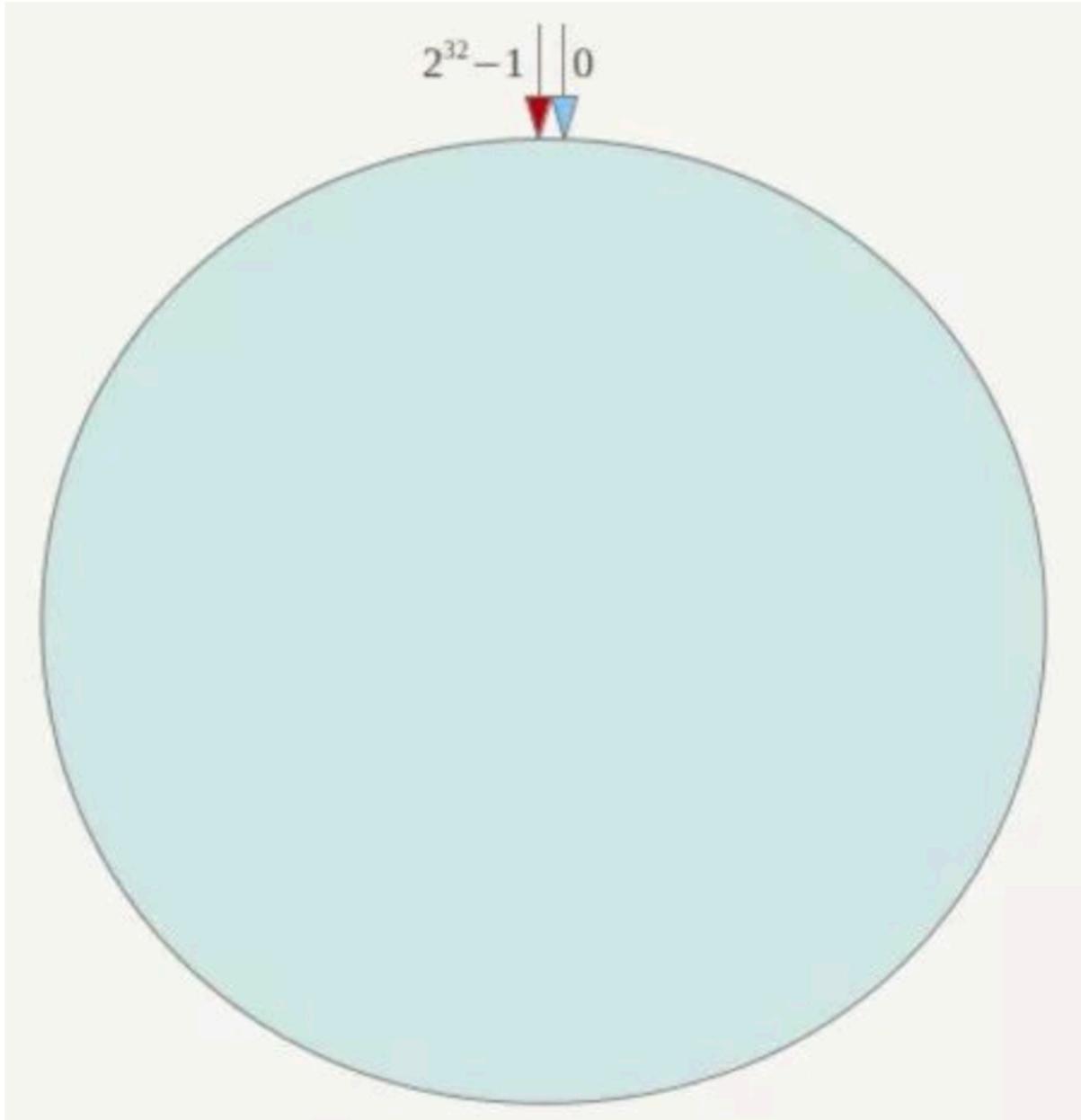


Hash取模来判断属于哪一台服务器： $\text{hash(a.png)} \% 4 = 2$ ，则说明在第2台服务器上。

缺陷：服务器的故障，或者集群扩增，都会导致Hash取模结果变化，导致请求找不到缓存，直接访问后端数据库

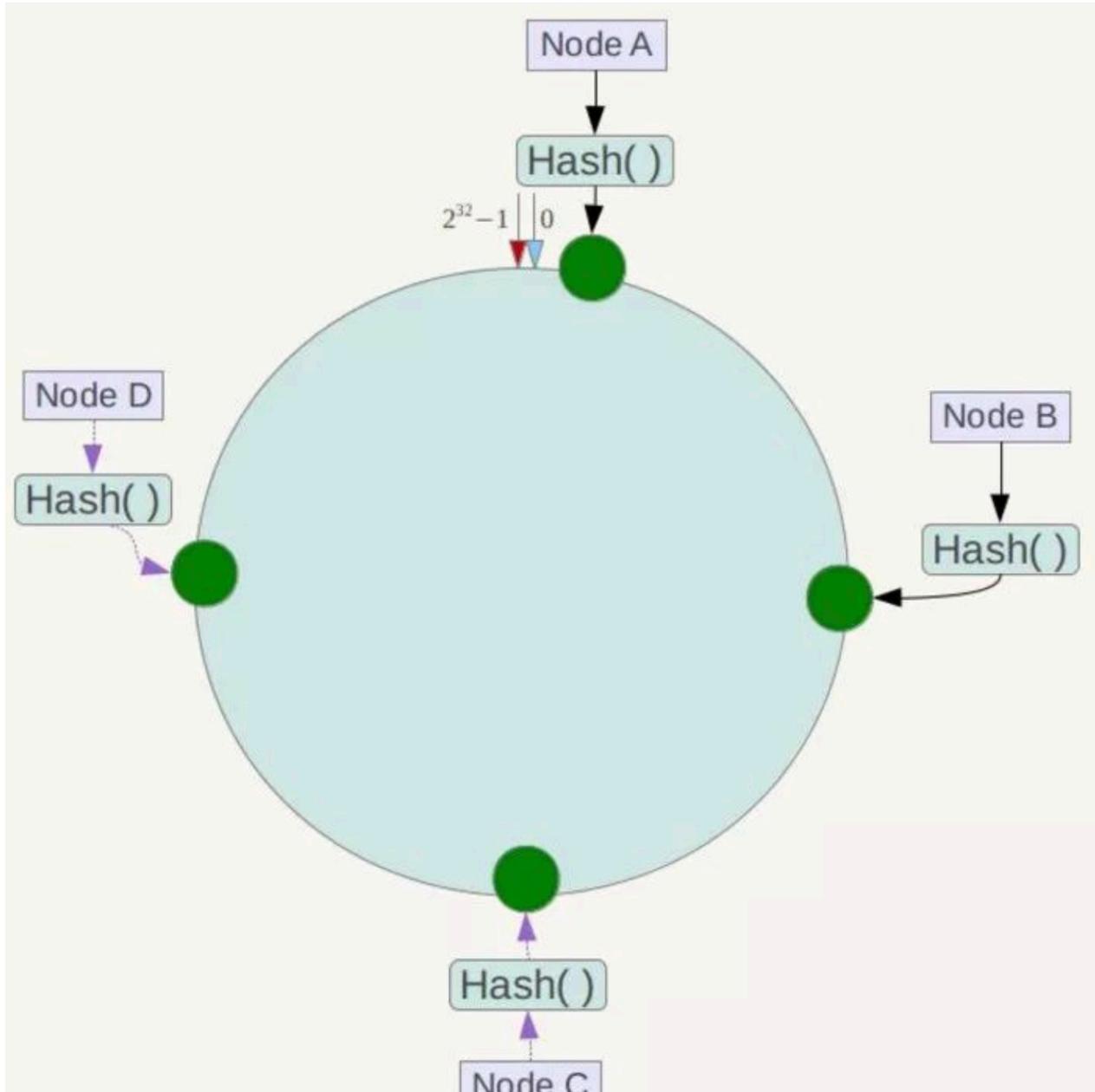
一致性Hash

一致性Hash算法也是使用取模的方法，只是，刚才描述的取模法是对服务器的数量进行取模，而一致性Hash算法是对 2^{32} 取模，什么意思呢？简单来说，**一致性Hash算法将整个哈希值空间组织成一个虚拟的圆环**，如假设某哈希函数H的值空间为0- $2^{32}-1$ （即哈希值是一个32位无符号整形），整个哈希环如下：



整个空间按顺时针方向组织，圆环的正上方的点代表0，0点右侧的第一个点代表1，以此类推，2、3、4、5、6.....直到 $2^{32}-1$ ，也就是说0点左侧的第一个点代表 $2^{32}-1$ ，0和 $2^{32}-1$ 在零点中方向重合，我们把这个由 2^{32} 个点组成的圆环称为**Hash环**。

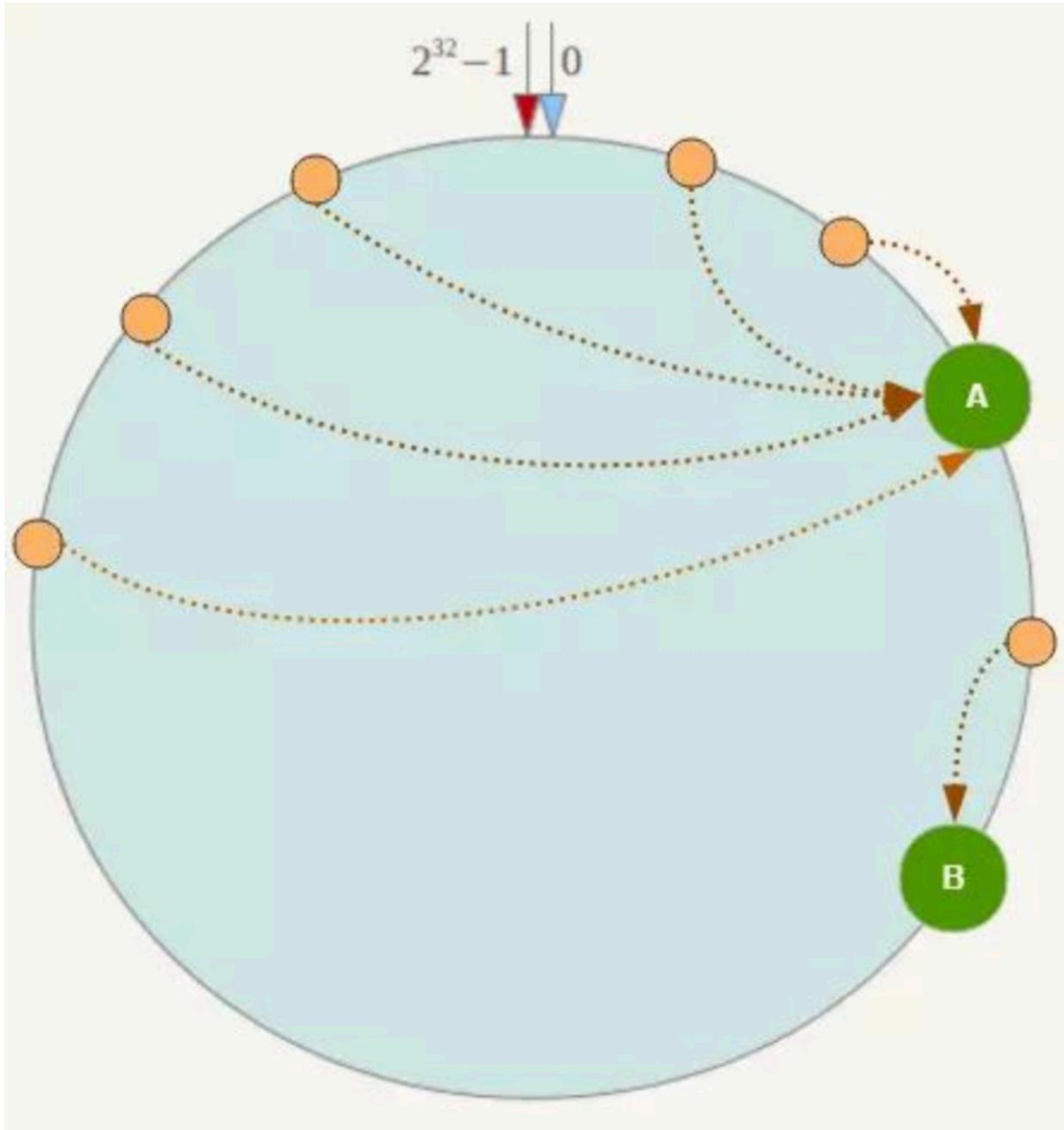
下一步将各个服务器使用Hash进行一个哈希，具体可以选择服务器的IP或主机名作为关键字进行哈希，这样每台机器就能确定其在哈希环上的位置，这里假设将上文中四台服务器使用IP地址哈希后在环空间的位置如下：



接下来使用如下算法定位数据访问到相应服务器：将数据key使用相同的函数Hash计算出哈希值，并确定此数据在环上的位置，从此位置沿环顺时针“行走”，第一台遇到的服务器就是其应该定位到的服务器！

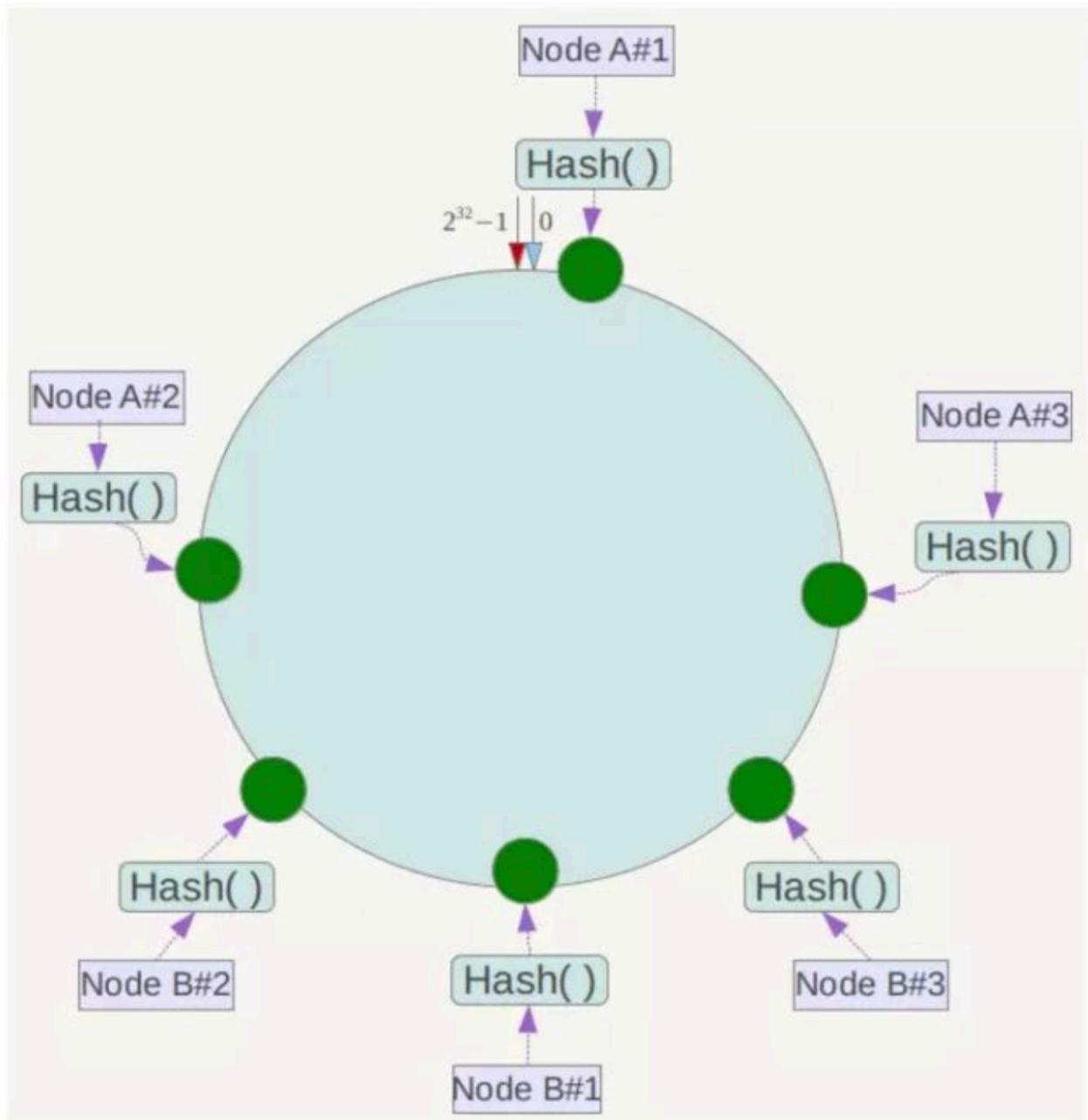
Hash环数据倾斜（构造虚拟节点解决）

一致性Hash算法在服务节点太少时，容易因为节点分布不均匀而造成数据倾斜（被缓存的对象大部分集中缓存在某一台服务器上）问题，例如系统中只有两台服务器，其环分布如下：



此时必然造成大量数据集中到Node A上，而只有极少量会定位到Node B上。为了解决这种数据倾斜问题，一致性Hash算法引入了虚拟节点机制，即对每一个服务节点计算多个哈希，每个计算结果位置都放置一个此服务节点，称为虚拟节点。具体做法可以在服务器IP或主机名的后面增加编号来实现。

例如上面的情况，可以为每台服务器计算三个虚拟节点，于是可以分别计算“Node A#1”、“Node A#2”、“Node A#3”、“Node B#1”、“Node B#2”、“Node B#3”的哈希值，于是形成六个虚拟节点：



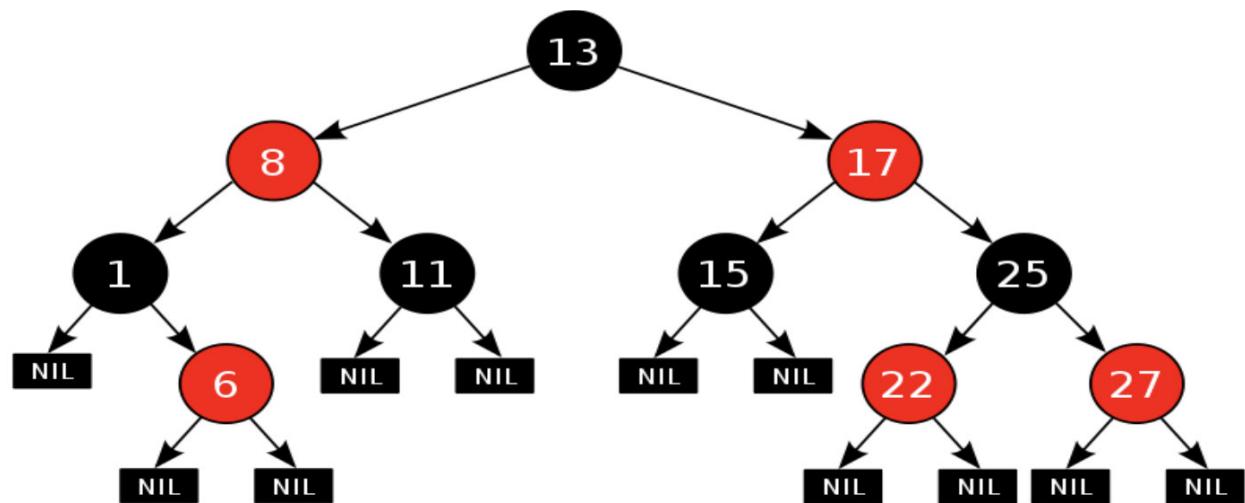
同时数据定位算法不变，只是多了一步虚拟节点到实际节点的映射，例如定位到“Node A#1”、“Node A#2”、“Node A#3”三个虚拟节点的数据均定位到Node A上。这样就解决了服务节点少时数据倾斜的问题。在实际应用中，通常将虚拟节点数设置为32甚至更大，因此即使很少的服务节点也能做到相对均匀的数据分布。

红黑树

相比于AVL树，牺牲了严格的平衡性，来换取查找和插入删除效率的折中

五个性质

1. 每个结点要么是红的要么是黑的。
2. 根结点是黑的。
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。
4. 如果一个结点是红的，那么它的两个儿子都是黑的。
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。



红黑树比AVL树好在哪里

插入node: 引起了树的不平衡, AVL和RB-Tree都是最多只需要2次旋转操作, 即两者都是O(1);

删除node: 引起树的不平衡时, 最坏情况下, AVL需要维护从被删node到root这条路径上所有node的平衡性, 因此需要旋转的量级 $O(\log N)$, 而RB-Tree最多只需3次旋转, 只需要 $O(1)$ 的复杂度

大量插入删除场景，红黑树的效率 > AVL树

大量查找场景，红黑树的效率 < AVL树（因为AVL树更平衡）

当然，红黑树并不适应所有应用树的领域。如果数据基本上是静态的，那么让他们待在他们能够插入，并且不影响平衡的地方会具有更好的性能。如果数据完全是静态的，例如，做一个哈希表，性能可能会更好一些。

跳跃表 (Skip List)

为什么要跳表？

我们在实际开发中经常会有在一堆数据中查找一个指定数据的需求，而常用的支持高效查找算法的实现方式有以下几种：

- 有序数组

优点：支持数据的随机访问，并且可以采用二分查找算法降低查找操作的复杂度。

缺点：插入和删除数据时，为了保持元素的有序性，需要进行大量的移动数据的操作。

- 二叉查找树

优点：既支持高效的二分查找算法，又能快速的进行插入和删除操作的数据结构

缺点：是在某些极端情况下，二叉查找树有可能变成一个线性链表。

- 平衡二叉树

二叉树表示不服，于是基于二叉查找树的优点，对其缺点进行改进，引入了平衡的概念。根据平衡算法的不同，具体实现有AVL树 / B树 (B-Tree) / B+树 (B+Tree) / 红黑树 等等。但是平衡二叉树的实现多数比较复杂，较难理解。

- 跳跃表

优点：同样支持对数据进行高效的查找，插入和删除数据操作也比较简单，最重要的就是实现比较平衡二叉树真是轻量几个数量级。

缺点：就是存在一定数据冗余。

跳跃表实现

链表无法如数组一般使用二分查找，但是可以利用类似索引思想提取出关键节点，先和关键节点进行比较再和原链表比较。

索引一层接一层，直到最高层只有两个节点

当大量新节点插入后，索引会不够用，通过「抛硬币」方式决定新节点是否向上层提拔

跳跃表插入复杂度

1. 新节点和各层索引节点逐一比较，确定原链表的插入位置。O ($\log N$)
2. 把索引插入到原链表。O (1)
3. 利用抛硬币的随机方式，决定新节点是否提升为上一级索引。结果为“正”则提升并继续抛硬币，结果为“负”则停止。O ($\log N$)

总体上，跳跃表插入操作的时间复杂度是O ($\log N$)，而这种数据结构所占空间是 $2N$ ，既空间复杂度是O (N)。

跳跃表删除复杂度

1. 自上而下，查找第一次出现节点的索引，并逐层找到每一层对应的节点。O ($\log N$)
2. 删除每一层查找到的节点，如果该层只剩下1个节点，删除整个一层（原链表除外）。O ($\log N$)

总体上，跳跃表删除操作的时间复杂度是O ($\log N$)。

跳跃表比平衡二叉树优势

维持结构平衡的代价低，靠随机实现，而平衡二叉树需要调整

