

# 猿灯塔，做程序员的引导者

www.vuandenta.com

## 1. 说一下几种常见的排序算法和分别的复杂度

### a. 快速排序：

i. 原理：快速排序采用的是一种分治的思想,它先找一个基准数（一般选择第一个值）,然后将比这个基准数小的数字都放到它的左边,然后再递归调用,分别对左右两边快速排序,直到每一边只有一个数字,整个排序就完成了。

- 1.选定一个合适的值（理想情况中值最好，但实现中一般使用数组第一个值），称为“枢轴”(pivot)。
- 2.基于这个值，将数组分为两部分，较小的分在左边，较大的分在右边。
- 3.可以肯定，如此一轮下来，这个枢轴的位置一定在最终位置上。
- 4.对两个子数组分别重复上述过程，直到每个数组只有一个元素。
- 5.排序完成。

ii. 复杂度： $O(n)$

iii. 特点：快速排序是我们平常最常使用的一种排序算法,因为它速度快,效率高,是最优秀的一种排序算法。

### b. 冒泡排序：

i. 原理：冒泡排序其实就是逐一比较交换,进行里外两次循环,外层循环为遍历所有数字,逐个确定每个位置,里层循环为确定了位置后,遍历所有后面没有确定位置的数字,与该位置的数字进行比较,只要比该位置的数字小,就和该位置的数字进行交换。

ii. 复杂度： $O(n^2)$ ，最佳时间复杂度为 $O(n)$

iii. 特点：冒泡排序在我们实际开发中,使用的还是比较少的.它更加适合数据规模比较少的时候,因为它的效率是比较低的,但是优点是逻辑简单,容易让我们记得。

### c. 直接插入排序：

i. 原理：直接插入排序是将从第二个数字开始,逐个拿出来,插入到之前排好序的数列里。

ii. 复杂度： $O(n^2)$ ，最佳时间复杂度为 $O(n)$

iii. 特点：

### d. 直接选择排序：

i. 原理：直接选择排序是从第一个位置开始遍历位置,找到剩余未排序的数据里最小的,找到最小的后,再做交换

ii. 复杂度： $O(n^2)$

iii. 特点：和冒泡排序一样,逻辑简单,但是效率不高,适合少量的数据排序

## 2. 用java写一个冒泡排序算法

```
1 public class Test {
2     public static void bubbleSort() {
3         int a[] = {49, 38, 65, 97, 76, 13, 27, 49, 78, 34, 12, 64, 5, 4, 62, 99, 98, 54, 56, 11};
4         int temp;
5         for (int i = 0; i < a.length - 1; i++) {
6             for (int j = 0; j < a.length - 1 - i; j++) {
7                 if (a[j] > a[j + 1]) {
8                     temp = a[j];
9                     a[j] = a[j + 1];
10                    a[j + 1] = temp;
11                }
12            }
13        }
14        for (int i = 0; i < a.length; i++)
15            System.out.println(a[i]);
16    }
17
18    public static void main(String[] args) {
19        bubbleSort();
20    }
21 }
```

## 3. 描述一下链式存储结构

a. 线性结构的优点是可以实现随机读取，时间复杂度为 $O(1)$ ，空间利用率高，缺点是进行插入和删除操作时比较麻烦，时间复杂度为 $O(n)$ ，同时容量受限制，需要事先确定容量大小，容量过大，浪费空间资源，过小不能满足使用要求，会

# 猿灯塔，做程序员的引导者

www.vuandenata.com

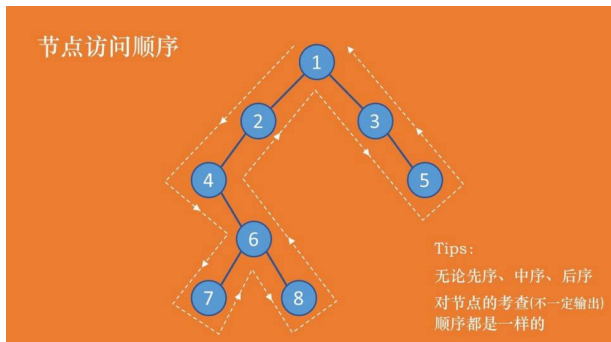
产生溢出问题。

b. 链式存储结构的优点主要是插入和删除非常简单，前提条件是知道操作位置，时间复杂度是 $O(1)$ ，但如果不知道操作位置则要定位元素，时间复杂度为 $O(n)$ ，没有容量的限制，可以使用过程中动态分配的分配内存空间，不用担心溢出问题，但是它并不能实现随机读取，同时空间利用率不高。

#### 4. 如何遍历一颗二叉树

a. 树节点：

```
1 class TreeNode {
2     int val;
3     //左子树
4     TreeNode left;
5     //右子树
6     TreeNode right;
7     //构造方法
8     TreeNode(int x) {
9         val = x;
10    }
11 }
```



b. 递归先序遍历：先输出节点的值，再递归遍历左右子树。中序和后序的递归类似，改变根节点输出位置即可。

```
1 // 递归先序遍历
2 public static void recursionPreorderTraversal(TreeNode root) {
3     if (root != null) {
4         System.out.print(root.val + " ");
5         recursionPreorderTraversal(root.left);
6         recursionPreorderTraversal(root.right);
7     }
8 }
9 //1 2 4 6 7 8 3 5
```

c. 递归中序遍历：过程和递归先序遍历类似

```
1 // 递归中序遍历
2 public static void recursionMiddleorderTraversal(TreeNode root) {
3     if (root != null) {
4         recursionMiddleorderTraversal(root.left);
5         System.out.print(root.val + " ");
6         recursionMiddleorderTraversal(root.right);
7     }
8 }
9 //4 7 6 8 2 1 3 5
```

d. 递归后序遍历：

```
1 // 递归后序遍历
```

# 猿灯塔，做程序员的引导者

www.vuandenata.com

```
2 public static void recursionPostorderTraversal(TreeNode root) {
3     if (root != null) {
4         recursionPostorderTraversal(root.left);
5         recursionPostorderTraversal(root.right);
6         System.out.print(root.val + " ");
7     }
8 }
```

## 5. 倒排一个LinkedList

```
1 Collections.reverse(linkedList);
```

## 6. 用java写一个递归遍历目录下面的所有文件 (directory.listFiles())

```
1 void listAll(File directory) {
2     if (!(directory.exists() && directory.isDirectory())) {
3         throw new RuntimeException("目录不存在");
4     }
5
6     File[] files = directory.listFiles();
7
8     for (File file : files) {
9         System.out.println(file.getPath() + file.getName());
10        if (file.isDirectory()) {
11            listAll(file);
12        }
13    }
14 }
```

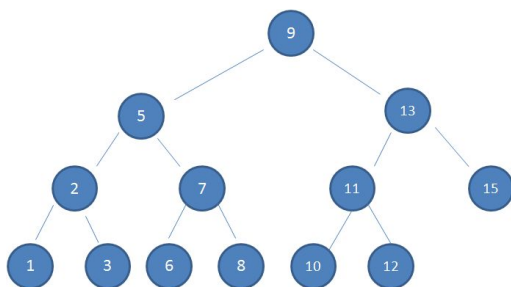
## 7. 二叉树与红黑树:

### 1. 二叉树:

#### a. 特性:

- 左子树上所有结点的值均小于或等于它的根结点的值。
- 右子树上所有结点的值均大于或等于它的根结点的值。
- 左、右子树也分别为二叉排序树。

#### b. 图例:

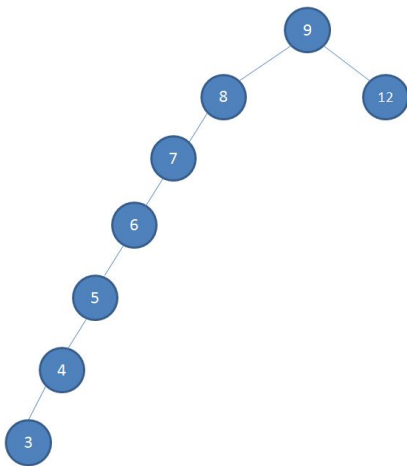


c. 查找: 二分查找 (通过一层一层的比较大小来查找位置): 如查找值为10的节点: 9--13--11--10

d. 缺陷: 插入容易变成线性形态, 查找性能大打折扣, 这时需要引入红黑树来解决

# 猿灯塔，做程序员的引导者

www.vuandenata.com



## 2. 红黑树：

a. 特点：是一种自平衡的二叉查找树，除了符合二叉树的特点之外，还符合以下几点：

- 节点是红色或黑色。
- 根节点是黑色。
- 每个叶子节点都是黑色的空节点（NIL节点）。
- 每个红色节点的两个子节点都是黑色。（从每个叶子到根的所有路径上不能有两个连续的红色节点）
- 从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。

这些规则保证了红黑树的自平衡。

红黑树从根到叶子的最长路径不会超过最短路径的2倍。

提高寻址效率。

b. 添加删除：通过自旋来保证平衡

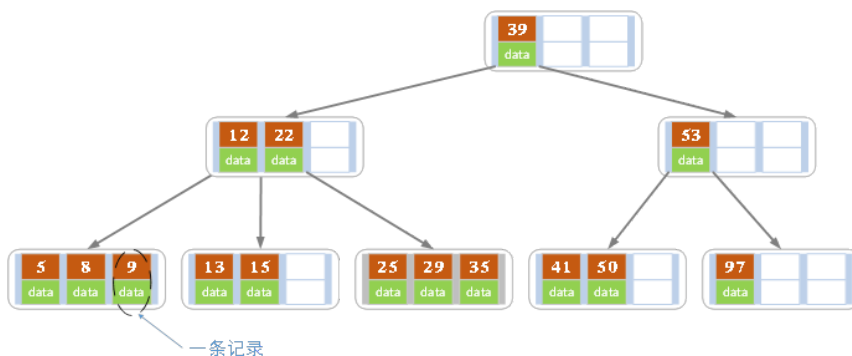
## 8. b-tree、b+tree多叉树：

<https://www.cnblogs.com/nullzx/p/8729425.html>

1. b-tree（文件系统）：B树也称B-树,它是一颗多路平衡查找树。我们描述一颗B树时需要指定它的阶数，阶数表示了一个结点最多有多少个孩子结点，一般用字母m表示阶数。当m取2时，就是我们常见的二叉搜索树。

a. 定义：

- 每个结点最多有m-1个关键字。
- 根结点最少可以只有1个关键字。
- 非根结点至少有 $\text{Math.ceil}(m/2)-1$ 个关键字。
- 每个结点中的关键字都按照从小到大的顺序排列，每个关键字的左子树中的所有关键字都小于它，而右子树中的所有关键字都大于它。
- 所有叶子结点都位于同一层，或者说根结点到每个叶子结点的长度都相同。



b. 插入数据，向兄弟节点借，兄弟节点不够则向父节点借；

## 2. b+tree（mysql索引）：

a. 定义：

- B+树包含2种类型的结点：内部结点（也称索引结点）和叶子结点。根结点本身即可以是内部结点，也可以

# 猿灯塔，做程序员的引导者

www.vuandentata.com

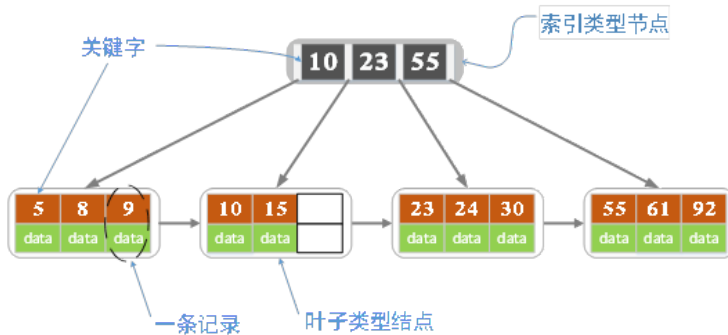
是叶子结点。根结点的关键字个数最少可以只有1个。

2) B+树与B树最大的不同是内部结点不保存数据，只用于索引，所有数据（或者说记录）都保存在叶子结点中。

3) m阶B+树表示了内部结点最多有m-1个关键字（或者说内部结点最多有m个子树），阶数m同时限制了叶子结点最多存储m-1个记录。

4) 内部结点中的key都按照从小到大的顺序排列，对于内部结点中的一个key，左树中的所有key都小于它，右子树中的key都大于等于它。叶子结点中的记录也按照key的大小排列。

5) 每个叶子结点都存有相邻叶子结点的指针，叶子结点本身依关键字的大小自小而大顺序链接。

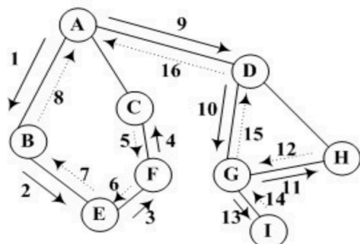


9. 谈谈数据结构，比如TreeMap：

TreeMap实现了红黑树的结构。

10. 图的深度遍历和广度遍历

1、深度优先遍历：

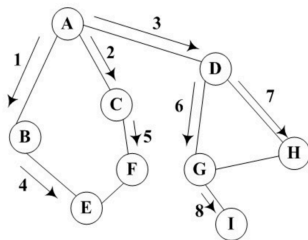


图G<sub>6</sub>的深度优先遍历过程

深度优先遍历结果是：A B E F C D G H I

深度优先遍历尽可能优先往深层次进行搜索

2、广度优先遍历：



广度优先遍历结果是：A B C D E F G H I

广度优先遍历按层次优先搜索最近的结点，一层一层往外搜索。

11. 介绍一下红黑树、二叉平衡树

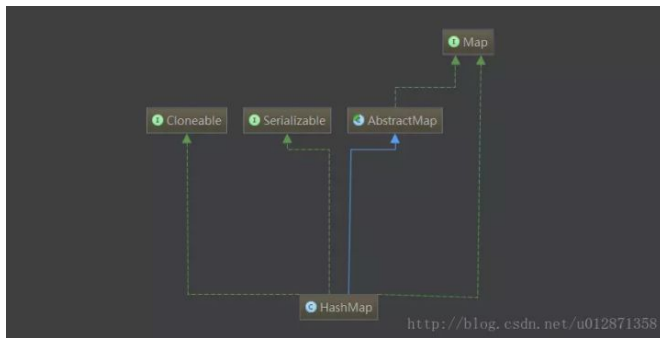
12. 说说java集合，每个集合下面有哪些实现类，及其数据结构。

a. HashMap：

i. 概念：

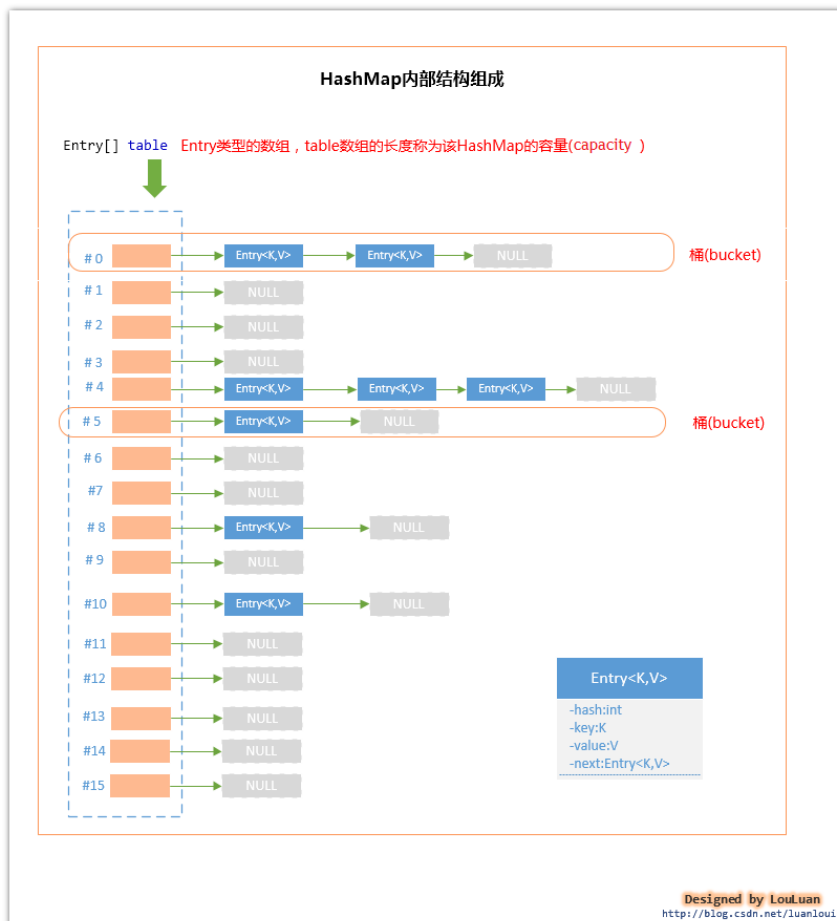
# 猿灯塔，做程序员的引导者

www.vuandenta.com



1. 在HashMap内部，采用了**数组+链表**的形式来组织键值对Entry<Key,Value>（利用数组的查询快+链表的插入删除快）
2. HashMap在存储键值对Entry<Key,Value>的时候，会根据Key的hashCode值，以某种映射关系，决定应当将这键值对Entry<Key,Value>存储在HashMap中的什么位置上
3. 在JDK1.7进行多线程put操作，之后遍历，直接死循环，CPU飙到100%，在JDK 1.8中进行多线程操作会出现节点和value值丢失，为什么JDK1.7与JDK1.8多线程操作会出现很大不同，是因为JDK 1.8的作者对resize方法进行了优化不会产生链表闭环。

ii. 结构：



iii. HashMap扩容：

# 猿灯塔，做程序员的引导者

www.vuandenta.com

阈值 (threshold) = 容量 (capacity) \* 加载因子 (load factor)

容量(capacity): 是指HashMap内部Entry[] table线性数组的长度

加载因子 : 是一个经验值, 一般为0.75

阈值(threshold): 当HashMap大小超过了阈值, HashMap将扩充2倍, 并且rehash

Designed by LouLuan  
<http://blog.csdn.net/luanlouis>

1、很简单的计算：由于默认的加载因子是0.75，那么，此时map的阈值是  $16 * 0.75 = 12$ ，即添加第13个键值对<Key, Value>的时候，map的容量会扩充一倍。

2、确实如此，但是为了尽可能减少桶中的Entry<Key, Value>链表的长度，以提高HashMap的存取性能，确定的这个经验值。如果读者你对存取效率要求的不是太高，想省点空间的话，你可以new HashMap(int initialCapacity, float loadFactor)构造方法将这个因子设置得大一些也无妨。

## 1. 扩容步骤：

1. 申请一个新的、大小为当前容量两倍的数组；
2. 将旧数组的Entry[] table中的链表重新计算hash值，然后重新均匀地放置到新的扩充数组中；
3. 释放旧的数组；

## 2. 为何扩容为原来的两倍（性能）：

a. 在HashMap通过键的哈希值进行定位桶位置的时候，调用了一个indexFor(hash, table.length);方法。

```
1  /**
2   * Returns index for hash code h.
3   */
4  static int indexFor(int h, int length) {
5      return h & (length-1);
6  }
```

b. 通过限制length是一个2的幂数， $h \& (length-1)$ 和 $h \% length$ 结果是一致的。

c. 如果length是一个2的幂的数，那么length-1就会变成一个mask，它会将hashcode低位取出来，hashcode的低位实际就是余数，和取余操作相比，与操作会将性能提升很多。

## 3.

### iv. put流程：

- 1 a. 获取这个Key的hashCode值，根据此值确定应该将这一对键值对存放在哪一个桶中，即确定要存放桶的索引；
- 2 b. 遍历所在桶中的Entry<Key, Value>链表，查找其中是否已经有了以Key值为Key存储的Entry<Key, Value>对象，
- 3 c1. 若已存在，定位到对应的Entry<Key, Value>，其中的Value值更新为新的Value值；返回旧值；
- 4 c2. 若不存在，则根据键值对<Key, Value> 创建一个新的Entry<Key, Value>对象，然后添加到这个桶的Entry<Key, Value>对象，
- 5 d. 当前的HashMap的大小(即Entry<key, Value>节点的数目)是否超过了阈值，若超过了阈值(threshold)，
- 6 则增大HashMap的容量(即Entry[] table 的大小)，并且重新组织内部各个Entry<Key, Value>排列。

### v. get流程：

- 1 a. 获取这个Key的hashCode值，根据此hashCode值决定应该从哪一个桶中查找；
- 2 b. 遍历所在桶中的Entry<Key, Value>链表，查找其中是否已经有了以Key值为Key存储的Entry<Key, Value>对象，
- 3 c1. 若已存在，定位到对应的Entry<Key, Value>，返回value
- 4 c2. 若不存在，返回null；

13.

猿灯塔，做程序员的引导者

[www.vuandenta.com](http://www.vuandenta.com)