数据结构

Heap

最大最小堆，建堆时间(部分数据 topk+海量数据 topk)

建堆时间： 建个 n 结点的堆，需要 O（n），插入的 average 是 O(1), worst case 是 O（logn）
删除复杂度是 O(logn)

最大堆建堆，插入和删除

```
class maxheap {
        private int[] data;
        private int size; //当前堆大小
        private int capability;//堆容量
        maxheap(int max){
            this.data = new int[max+1];
            this.capability = max ;
            this.size=0;
        }
        //增

        public boolean insert(int d) {

            if(size == capability) {
                return false;
            }

            data[size+1]=d;
            size = size+1 ;
            if(shiftup(d))
                return true;
            else
                return false;
        }

        public boolean shiftup(int d) {

            int i  = size;
            while(i>1&&data[i]>data[i/2]) {
                int temp = data[i];
                data[i] = data[i/2];
                data[i/2]= temp;
                i =  i/2;
            }
            return true;
        }
        //删 （取）

        public int delete() {

            if(size==0) {
                return -1;
            }
            int elements = data[1];
            data[1] = data[size];
            size --;
            shiftdown(1);
            return elements;
        }

        public void shiftdown(int i ) {

            if(size ==i ) {
                return;
            }
            while(i*2 <= size) {
                int j=  i*2;

        if(j+1<=size&&data[j+1]>data[j]) {
                    j=j+1;
                }
                //如果子都小于结点  就结束 ，  不继续下面的交换循环
                if(data[i]>data[j])
```

```
                break;                                          }
            int temp = data[i];                             }
            data[i]= data[j];
            data[j]=temp;                               }
            i =j;
```

海量数据 topk

在大规模数据处理中，经常会遇到的一类问题：在海量数据中找出出现频率最好的前 k 个数，或者从海量数据中找出最大的前 k 个数，这类问题通常被称为 top K 问题。例如，在搜索引擎中，统计搜索最热门的 10 个查询词；在歌曲库中统计下载最高的前 10 首歌等。

针对 top K 类问题，通常比较好的方案是分治+Trie 树/hash+小顶堆（就是上面提到的最小堆），即先将数据集按照 Hash 方法分解成多个小数据集，然后使用 Trie 树或者 Hash 统计每个小数据集中的 query 词频，之后用小顶堆求出每个数据集中出现频率最高的前 K 个数，最后在所有 top K 中求出最终的 top K。

树

平衡二叉树（AVL）

判断树的深度：

```
public int TreeDepth(TreeNode root) {
        if(root==null) return 0;
        return Math.max(1+TreeDepth(root.left),1+TreeDepth(root.right));
    }
```

判断一个树是否是平衡二叉树（左右两边最大最小深度一样）：

```
public int nodedepth(TreeNode root) { //最大深度
    if(root == null) return 0;
    return Math.max(1+nodedepth(root.left),1+nodedepth(root.right));
}
public int nodemindepth(TreeNode root) { //最小深度
    if(root == null) return 0;
    return Math.min(1+nodedepth(root.left),1+nodedepth(root.right));
}
// 判断是否是平衡二叉树

public boolean IsBalanced_Solution(TreeNode root) {
    if(root==null) return true;
    return nodedepth(root)- nodemindepth(root) >1 ? false : true;
}
```

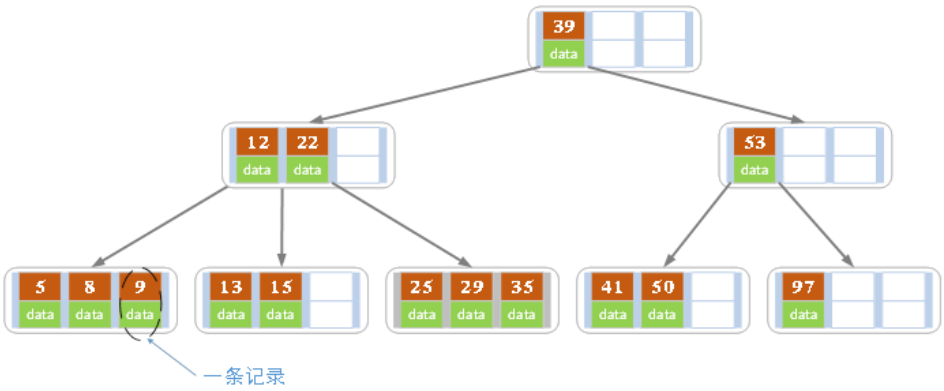
与红黑树性质比较

下一小节

红黑树 （旋转调整）

红黑树性质：
一般的，红黑树，满足以下性质，即只有满足以下全部性质的树，我们才称之为红黑树:
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）是黑的。
4）如果一个结点是红的，那么它的俩个儿子都是黑的。
5)对于任一结点而言,其到叶结点树尾端 NIL 指针的每一条路径都包含相同数目的黑结点。
红黑树的各种操作的时间复杂度是多少？
能保证在最坏情况下，基本的动态几何操作的时间均为 O（lgn）
红黑树相比于 BST 和 AVL 树有什么优点？
红黑树是牺牲了严格的高度平衡的优越条件为代价，它只要求部分地达到平衡要求，降低了对旋转的要求，从而提高了性能。红黑树能够以 O(log2 n)的时间复杂度进行搜索、插入、删除操作。此外，由于它的设计，任何不平衡都会在三次旋转之内解决。当然，还有一些更好的，但实现起来更复杂的数据结构能够做到一步旋转之内达到平衡，但红黑树能够给我们一个比较“便宜”的解决方案。
相比于 BST，因为红黑树可以能确保树的最长路径不大于两倍的最短路径的长度，所以可以看出它的查找效果是有最低保证的。在最坏的情况下也可以保证 O(logN)的，这是要好于二叉查找树的。因为二叉查找树最坏情况可以让查找达到 O(N)。
红黑树的算法时间复杂度和 AVL 相同，但统计性能比 AVL 树更高，所以在插入和删除中所做的后期维护操作肯定会比红黑树要耗时好多，但是他们的查找效率都是 O(logN)，所以红黑树应用还是高于 AVL 树的. 实际上插入 AVL 树和红黑树的速度取决于你所插入的数据. 如果你的数据分布较好,则比较宜于采用 AVL 树(例如随机产生系列数),但是如果你想处理比较杂乱的情况,则红黑树是比较快的

B 树

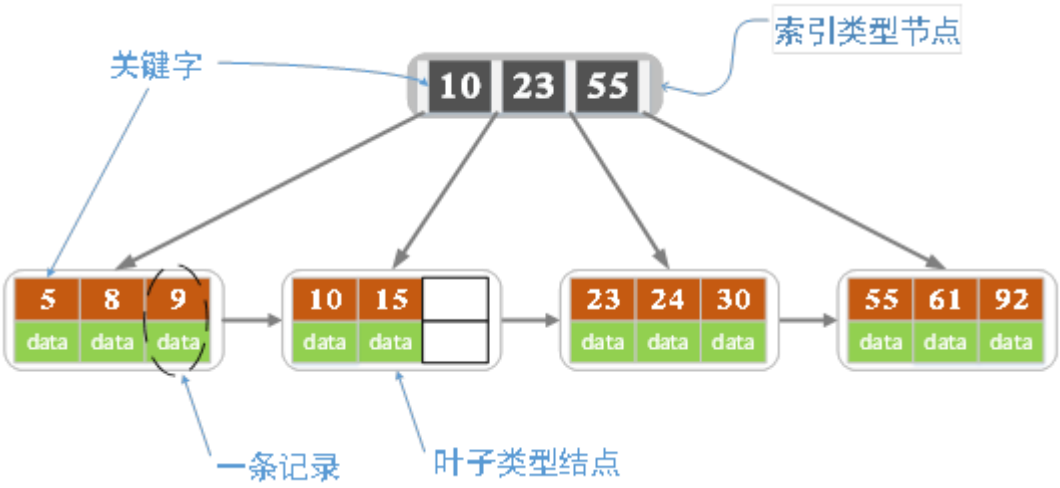


B 树也称 B-树,它是一颗多路平衡查找树。我们描述一颗 B 树时需要指定它的阶数，阶数表示了一个结点最多有多少个孩子结点，一般用字母 m 表示阶数。当 m 取 2 时，就是我们常

见的二叉搜索树。
一颗 m 阶的 B 树定义如下：
1）每个结点最多有 m-1 个关键字。
2）根结点最少可以只有 1 个关键字。
3）非根结点至少有 Math.ceil(m/2)-1 个关键字。
4）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。
5）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+树

B+树是应文件系统所需而产生的一种 B 树的变形树(文件的目录一级一级索引,只有最底层的叶子节点(文件)保存数据.),非叶子节点只保存索引,不保存实际的数据,数据都保存在叶子节点中.这不就是文件系统文件的查找吗?我们就举个文件查找的例子:有 3 个文件夹,a,b,c, a 包含 b,b 包含 c,一个文件 yang.c, a,b,c 就是索引(存储在非叶子节点), a,b,c 只是要找到的 yang.c 的 key,而实际的数据 yang.c 存储在叶子节点上. 所有的非叶子节点都可以看成索引部分



B 树 B+树应用场景比较（典型：数据库索引）

B 和 **B+**树主要用在文件系统以及数据库做索引.比如 **Mysql;(**为什么**)**
1、 B+树的磁盘读写代价更低：B+树的内部节点并没有指向关键字具体信息的指针，因此其内部节点相对 B 树更小，如果把所有同一内部节点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多，一次性读入内存的需要查找的关键字也就越多，相对 IO 读写次数就降低了。
2、B+树的查询效率更加稳定：由于非终结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。
3、由于 B+树的数据都存储在叶子结点中，分支结点均为索引，方便扫库，只需要扫一遍叶

子结点即可，但是 B 树因为其分支结点同样存储着数据，我们要找到具体的数据，需要进行一次中序遍历按序来扫，所以 B+树更加适合在区间查询的情况，所以通常 B+树用于数据库索引。 数据库索引

栈

定义栈的数据结构，请在该类型中实现一个能够得到栈中所含最小元素的 min 函数（时间复杂度应为 O（1））

```
import java.util.Stack;

public class Solution {


    Stack<Integer> re = new Stack<Integer>();
    Stack<Integer> min = new Stack<Integer>();
    int mintmp= Integer.MAX_VALUE;
    public void push(int node) {
        re.push(node);
        if(node<mintmp) {
            mintmp=node;
            min.push(node);
        }else {
            min.push(mintmp);
        }
    }

    public void pop() {
        re.pop();
        min.pop();
    }

    public int top() {
        return re.peek();
    }

    public int min() {
        return min.peek();
    }
}
```

队列

优先队列(底层是堆)， 改写比较器，可用于 topk 题目。

链表

1. 在 O(1)时间删除链表节点
   下一个结点数据覆盖需要删除的结点，但是不能是尾节点
2. 单链表的转置
   用三个临时指针 pre、head、next 在链表上循环一遍即可。

```
public ListNode ReverseList(ListNode head) {
        ListNode next ,pre;
        pre = null;
        if (head == null)
                return head;
        while (head!=null) {
            next = head .next;
            head.next = pre;
            pre = head;
            head = next ;
        }
        return pre;
    }
```

3. 找倒数 k 个结点
   快慢指针， 距离为 k
4. 判断是不是环
   快慢指针 两倍速相遇
5. 判断环入口
   两倍速， 相遇后， 快指针从头开始变一倍速

Map

Hashtable:

HashTable 的主要方法的源码实现逻辑，与 HashMap 中非常相似，有一点重大区别就是所有的操作都是通过 **synchronized** 锁保护的。只有获得了对应的锁，才能进行后续的读写等操作。

Hashmap:

底层实现：在 JDK1.6，JDK1.7 中，HashMap 采用位桶+链表实现，即使用链表处理冲，而 JDK1.8 中，HashMap 采用位桶+链表+红黑树实现，当链表长度超过阈值（8）时，将链表转换为红黑树。
通过 hash 的方法，通过 put 和 get 存储和获取对象。如果发生碰撞的时候，Hashmap 通过链表将产生碰撞冲突的元素组织起来，在 Java 8 中，如果一个 bucket 中碰撞冲突的元素超过某个限制(默认是 8)，则使用红黑树来替换链表，从而提高速度。
Hashmap 可能存在的安全问题:
如果多个线程同时使用 put 方法添加元素，而且假设正好存在两个 put 的 key 发生了碰撞（根据 hash 值计算的 bucket 一样），那么根据 HashMap 的实现，这两个 key 会添加到数组的同一个位置，这样最终就会发生其中一个线程 put 的数据被覆盖
如果多个线程同时检测到元素个数超过数组大小 * loadFactor，这样就会发生多个线程同时对 Node 数组进行扩容，都在重新计算元素位置以及复制数据，但是最终只有一个线程扩容后的数组会赋给 table，也就是说其他线程的都会丢失，并且各自线程 put 的数据也丢失 *（put 多了 rehash, get 的时候死循环）

Concurrenthashmap 以及使用场景

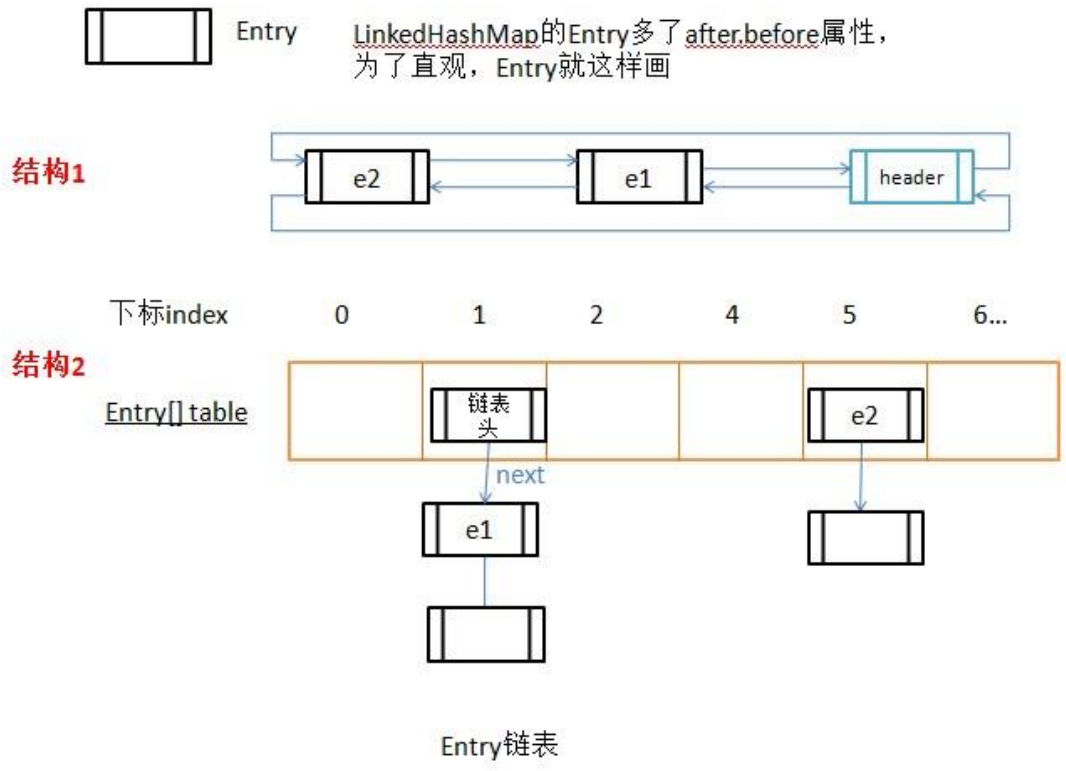
ConcurrentHashMap 是 J.U.C(java.util.concurrent 包)的重要成员，它是 HashMap 的一个线程安全的、支持高效并发的版本。ConcurrentHashMap 在默认并发级别下会创建 16 个 Segment 对象的数组，如果键能均匀散列，每个 Segment 大约守护整个散列表中桶总数的 1/16。Segment 类继承于 ReentrantLock 类，从而使得 Segment 对象能充当锁的角色。每个 segment 实际上还是存储的哈希表，写入的时候，先找到对应的 segment，然后锁这个 segment，写完，解锁。锁 segment 的时候，其他 segment 还可以继续工作。分段锁。
使用场景多线程并发 MAP 的时候。

Hashmap 与 Hashtable 的区别:

这两个类主要有以下几方面的不同:
Hashtable 和 HashMap 都实现了 Map 接口，但是 Hashtable 的实现是基于 Dictionary 抽象类。
在 HashMap 中，null 可以作为键，这样的键只有一个；可以有一个或多个键所对应的值为 null。
当 get()方法返回 null 值时，即可以表示 HashMap 中没有该键，也可以表示该键所对应的值为 null。因此，在 HashMap 中不能由 get()方法来判断 HashMap 中是否存在某个键，而应该用 containsKey()方法来判断。而在 Hashtable 中，无论是 key 还是 value 都不能为 null 。
这两个类最大的不同在于:
1. Hashtable 是线程安全的，它的方法是同步了的，可以直接用在多线程环境中。
2. 而 HashMap 则不是线程安全的。在多线程环境中，需要手动实现同步机制。

怎么保证 map 有序？-（LinkedHashmap）

首先 LinkedHashMap 是非线程安全的。
LinkedHashMap 是 HashMap 的一个子类，它保留插入的顺序，如果需要输出的顺序和输入时的相同，那么就选用 LinkedHashMap。
LinkedHashMap 是 Map 接口的哈希表和链接列表实现，具有可预知的迭代顺序。此实现提供所有可选的映射操作，并允许使用 null 值和 null 键。此类不保证映射的顺序，特别是它不保证该顺序恒久不变。
LinkedHashMap 实现与 HashMap 的不同之处在于，LinkedHashMap 维护着一个运行于所有条目的**双重链接链表**。此链接列表定义了迭代顺序，该迭代顺序可以是插入顺序或者是访问顺序。
下列是 LinkedHashmap 中 Entry 里面有的一些属性：
K key
V value
Entry<K, V> next
int hash
Entry<K, V> before
Entry<K, V> after
其中前面四个，也就是红色部分是从 HashMap.Entry 中继承过来的；后面两个，是 LinkedHashMap 独有的。不要搞错了 next 和 before、After，**next** 是用于维护 HashMap 指定 table 位置上连接的 Entry 的顺序的（从 table 中指向 entry[]数组的），before、After 是用于维护 Entry 插入的先后顺序的。
关系图如下：



Entry链表

说明：
LinkedHashMap 是继承 HashMap，也就继承了 HashMap 的结构，也就是图中的结构 2，在下文中我用"Entry 数组+next 链表"来描述。而 LinkedHashMap 有其自己的变量 header，也就是图中的结构 1，下文中我用"header 链表"来描述。
结构 1 中的 Entry 和结构 2 中的 Entry 本是同一个，结构 1 中应该就只有一个 header，它指向的是结构 2 中的 e1 e2，但这样会使结构图难画。为了说明问题的方便，我把结构 2 里的 e1 e2 在结构 1 中多画一个。

TreeMap

对于 TreeMap 而言，由于它底层采用一颗“红黑树”来保存集合中的 Entry，这意味着 TreeMap 添加元素、取出元素的性能都比 HashMap 低。
该映射根据其键的自然顺序进行排序，或者根据创建时提供的 Comparator 进行排序。
当 TreeMap 添加元素时，需要通过循环找到新增 Entry 的插入位置，因此比较耗性能；当从 TreeMap 中取出元素时，需要通过循环才能找到合适的 Entry，也比较耗性能。但 TreeMap、TreeSet 相比 HashMap、HashSet 的优势在于：TreeMap 中的所有 Entry 总是按 key 根据指定排序规则保存有序状态，TreeSet 中的所有元素总是根据指定排序规则保存有序状态。
红黑树是一种自平衡二叉查找树，树中每个节点的值，都大于或等于在它的左子树中的所有节点的值，并且小于或等于在它的右子树中的所有节点的值，这确保红黑树运行时可以快速地在树中查找和定位的所需节点。