# 3年JAVA技能总结-数据结构篇

1. 常用的List集合

ArrayList：有序可重复；底层使用数组；查询快增删慢；线程不安全；扩容机制：当前容量 \* 1.5  
LinkedList：有序可重复；底层使用双向链表；查询慢增删快；线程不安全  
Vector：有序可重复；底层使用数组；查询快增删慢；线程安全效率低；扩容机制：容量一倍

1. 常用的Set集合

HashSet：排列无序，不可重复；底层使用hash表实现；存取速度快；内部是HashMap  
LinkedHashSet：采用hash表存储，并用双向链表记录插入顺序；内部是LinkedHashMap  
TreeSet：排列无序，不可重复；底层使用二叉树实现；排序存储；内部是TreeMap的SortedSet

1. 常用的Map

HashMap：key不重复value可重复；底层hash表+链表+红黑树（jdk1.8），hash表+单向链表（jdk1.7）；线程不安全；允许key=null，value=null；1倍扩容  
 如果需要满足线程安全，可以借助Collections工具类的synchroinzedMap方法具有线程安全的能力，或者使用ConcurrentHashMap  
 HashMap的底层数据结构是什么  
 HashMap的put过程  
 HashMap的扩容机制  
 并发环境下HashMap会带来什么致命问题  
 LinkedHashMap：继承HashMap  
 TreeMap：key不重复value可重复；底层二叉树;  
 Hashtable：key不重复value可重复；底层hash表+链表+红黑树（jdk1.8）；线程安全；key和value都不能是null  
 ConcurrentHashMap：  
 锁分段技术：  
 读是否要加锁：  
 迭代器是强一致性的迭代器还是弱一致性的迭代器

CopyOnWrite系列

队列：Queue  
 先入先出FIFO

栈：Stack  
 先入后出FILO

1. JDK1.8 ArrayDeque源码

public class ArrayDeque<E> extends AbstractCollection<E> implements Deque<E>, Cloneable{

//ArrayDeque是一个用循环数组实现的双端队列

//不允许存null（抛NullPointerException），计算容量算法，扩容算法

//初始化时计算容量：大于默认容量或用户指定容量的最小的2的幂次方

//新增元素时扩容：双倍

transient Object[] elements;

transient int head; //头部节点序号

transient int tail; //尾部节点序号，（指向最后一点节点的后一个位置）

private static final int MIN\_INITIAL\_CAPACITY = 8; //允许最小的elements的容量，限制用户自定义

//当head==tail时扩容

public ArrayDeque() {

elements = new Object[16]; //16是比8大的最小的2的幂次方

}

public ArrayDeque(int numElements){

allocateElements(numElements); //计算elements容量并初始化elements

}

public void addLast(E e) {

if(e == null) throw new NullPointerException();

elements[tail] = e;

if( (tail = (tail + 1) & (elements.size - 1)) == head) {

doubleCapacity(); //双倍扩容

}

}

//转成补码计算：正数原码反码补码相同，负数反码为符号位置1，其余位按位取反，补码为反码+1

public void addFirst(E e){

if (e == null) throw new NullPointerException();

elements[head = (head - 1) & (elements.length - 1)] = e; //注意：head是-1

if (head == tail)

doubleCapacity();

}

因为elements.length是2的幂次方，所以减一后就变成了掩码，tail如果记录的是最后一个位置，即 elements.length - 1，tail + 1 则等于elements.length，与 elements.length - 1 做与操作后，就变成了0，嗯，没错，这样就变成了一个循环数组，如果tail与head相等，则表示没有剩余空间可以存放更多元素了，则调用doubleCapacity进行扩容：

private void doubleCapacity() {

assert head == tail;

int p = head;

int n = elements.length();

int r = n - p;

int newCapacity = n << 1; //新的容量，有符号位左移，低位置0

if (newCapacity < 0) {

throw new IllegalStateException(“Sorry,deque is too big!”);

}

Object[] newElements = new Object[newCapacity];

//将老数组的[head, elements.size()]放在新数组的[0, element.size() - head]上

System.copyarray(elements, p, newElements, 0, r);

//将老数组的[0, head]放在新数组的[element.size() - head, elements.size()]上

System.copyarray(elements, 0, newElements, r, p);

//置换原因：addFirst时head-1，addLast时tail+1，位置循环

elements = newElements;

head = 0;

tail = n;

}

public int size() {

return (tail - head) & (elements.length() - 1);

}

public boolean isEmpty() {

return head == tail;

}

//计算elements的初始容量:大于numElements的最小的2的幂次方

//调用：初始化ArrayDeque实例时、反序列化时

//原理：

\* 只需要关注二进制中第一个高位为1的位即可，其余为可以忽略

\* 1.假设在第n位（...... n n-1 n-2 ...... 2 1）

\* 2.无符号右移1位再按位或运算，则n n-1都是1 -->有2位为1

\* 3.无符号右移2位再按位或运算，则n ...... n-3都是1 -->有4位为1

\* 4.无符号右移4位再按位或运算，则n ...... n-7都是1 -->有8位为1

\* 5.无符号右移8位再按位或运算，则n ...... n-15都是1 -->有16位为1

\* 6.无符号右移16位再按位或运算，则n ...... n-31都是1 -->有32位为1

\* 7.java int类型4字节，32位，正数符号位为0，且无符号右移时最多31位为1，即最大生成Interger.MAX\_VALUE=0x7FFFFFFF

@Test

public void testArrayDeque() {

System.out.println(Integer.MAX\_VALUE);

System.out.println(Integer.MAX\_VALUE + 1);

allocateElements(Integer.MAX\_VALUE);

}

//计算时转成补码

private void allocateElements(int numElements) {

int initialCapacity = numElements;

if (initialCapacity > MIN\_INITIAL\_CAPACITY) { //限制最小容量为8

initialCapacity |= initialCapacity >>> 1; //无符号右移1位再按位或

initialCapacity |= initialCapacity >>> 2; //无符号右移2位再按位或

initialCapacity |= initialCapacity >>> 4; //无符号右移4位再按位或

initialCapacity |= initialCapacity >>> 8; //无符号右移8位再按位或

initialCapacity |= initialCapacity >>> 16; //无符号右移16位再按位或

initialCapacity ++;

if(initialCapacity < 0) {

initialCapacity >>>= 1;

}

}

elements = new Object[initialCapacity]; //生成null数组

}

}

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | ... | i | ... | head | ... | tail | ... |

1. JDK1.8 ArrayList源码

public class ArrayList<E> extends AbstractList<E> implements List<E>, RandomAccess, Cloneable{  
 int DEFAULT\_CAPACITY = 10;  
 Object[] EMPTY\_ELEMENTDATA = {};  
 Object[] DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA = {};  
  
 //\*\*\*\*\*\*\*\*\*底层是用数组来存放元素的，扩容方式：3/2倍  
 //底层数组决定了ArrayList是有序的，  
 transient Object[] elementData;  
 int size; //\*\*\*\*\*\*\*\*\*实际大小

//记录该ArrayList被修改的次数，用于fast-fail快速失败机制，该属性在AbstractList中被定义  
 protected transient int modCount = 0;   
  
 //默认构造器，初始化elementData={}  
 public ArrayList() {  
 this.elementData = DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA;  
 }  
  
 //指定底层数组的默认大小  
 public ArrayList(int initialCapacity) {  
 if (initialCapacity > 0) {  
 this.elementData = new Object[initialCapacity];  
 } else if (initialCapacity == 0) {  
 this.elementData = EMPTY\_ELEMENTDATA;  
 } else {  
 throw new IllegalArgumentException("Illegal Capacity: "+ initialCapacity);  
 }  
 }  
  
 //ArrayList新增数据的方法 ，元素可以为null  
 //1.由于底层是用数组来存放元素的，所以需要判断是否对数组进行扩容  
 //2.更新modCount++  
 public boolean add(E e) {  
 ensureCapacityInternal(size + 1);  
 elementData[size++] = e;  
 return true;  
 }  
  
 //在指定位置插入元素  
 //1.下标校验 [0,size]就是合法的下标  
 //2.是否考虑需要扩容  
 //3.index后元素全部后移  
 //4.在index位置插入元素  
 //5.更新size  
 public void add(int index, E element) {  
 rangeCheckForAdd(index);  
 ensureCapacityInternal(size + 1);  
 System.arraycopy(elementData, index, elementData, index + 1, size - index);  
 elementData[index] = element;  
 size++;  
 }  
  
 //将底层数组缩放到size大小，因为每次扩容后数组默认被null填充，该方法就是去掉null  
 public void trimToSize() {  
 modCount++;  
 if (size < elementData.length) {  
 elementData = (size == 0) ? EMPTY\_ELEMENTDATA : Arrays.copyOf(elementData, size);  
 }  
 }  
  
 //判断该list是否包含指定元素  
 public boolean contains(Object o) {  
 return indexOf(o) >= 0;  
 }  
  
 //返回该list指定元素的下标，如果有重复的就返回从左到右第一个  
 public int indexOf(Object o) {  
 if (o == null) {  
 for (int i = 0; i < size; i++)  
 if (elementData[i]==null)  
 return i;  
 } else {  
 for (int i = 0; i < size; i++)  
 if (o.equals(elementData[i]))  
 return i;  
 }  
 return -1;  
 }  
  
 public E get(int index) {  
 rangeCheck(index);  
 return elementData(index);  
 }  
  
 //移除指定位置的元素  
 public E remove(int index) {  
 rangeCheck(index);  
 modCount++;  
 E oldValue = elementData(index);  
 int numMoved = size - index - 1;  
 if (numMoved > 0) //数据前移  
 System.arraycopy(elementData, index+1, elementData, index, numMoved);  
 elementData[--size] = null;

return oldValue;  
 }  
  
 private void ensureCapacityInternal(int minCapacity) {  
 ensureExplicitCapacity(calculateCapacity(elementData, minCapacity));  
 }  
  
 private static int calculateCapacity(Object[] elementData, int minCapacity) {  
 if (elementData == DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA) {  
 return Math.max(DEFAULT\_CAPACITY, minCapacity);  
 }  
 return minCapacity;  
 }  
  
 private void ensureExplicitCapacity(int minCapacity) {  
 modCount++;  
 if (minCapacity - elementData.length > 0)  
 grow(minCapacity);  
 }  
  
 private void grow(int minCapacity) {  
 int oldCapacity = elementData.length; //当前数组的容量  
 int newCapacity = oldCapacity + (oldCapacity >> 1); //新容量= 3/2倍的老容量  
 if (newCapacity - minCapacity < 0)  
 newCapacity = minCapacity;  
 if (newCapacity - MAX\_ARRAY\_SIZE > 0)  
 newCapacity = hugeCapacity(minCapacity);  
 // minCapacity is usually close to size, so this is a win:  
 elementData = Arrays.copyOf(elementData, newCapacity); //将老数组中的元素全部复制到新数组中  
 }  
  
 private static int hugeCapacity(int minCapacity) {  
 if (minCapacity < 0) throw new OutOfMemoryError();  
 return (minCapacity > MAX\_ARRAY\_SIZE) ? Integer.MAX\_VALUE : MAX\_ARRAY\_SIZE;  
 }  
}

1. JDK1.8 LinkedList源码

public class LinkedList<E> extends AbstractSequentialList<E> implements List<E>, Deque<E>, Cloneable{  
  
 transient int size = 0;  
  
 //底层维护了一个双向链表，快速删除和插入，查询比较慢  
 transient Node<E> first; //双向链表的头结点  
 transient Node<E> last; //双向链表的尾节点  
  
 protected transient int modCount = 0; //fast-fail机制  
  
 private static class Node<E> { //双向链表  
 E item;  
 Node<E> next;  
 Node<E> prev;  
 }  
  
 //核心方法：新增元素，链表尾部新增节点  
 public boolean add(E e) {  
 linkLast(e);  
 return true;  
 }  
  
 void linkLast(E e) {  
 final Node<E> l = last;  
 final Node<E> newNode = new Node<>(l, e, null); //新增节点  
 last = newNode; //更新尾节点  
 if (l == null)  
 first = newNode; //初始化头结点  
 else  
 l.next = newNode;  
 size++;  
 modCount++;  
 }  
  
 //获取链表中指定位置的元素  
 public E get(int index) {  
 checkElementIndex(index);  
 return node(index).item;  
 }  
  
 //获取指定索引位置的节点  
 Node<E> node(int index) {  
 if (index < (size >> 1)) {  
 Node<E> x = first;  
 for (int i = 0; i < index; i++)  
 x = x.next;  
 return x;  
 } else {  
 Node<E> x = last;  
 for (int i = size - 1; i > index; i--)  
 x = x.prev;  
 return x;  
 }  
 }  
  
 public void add(int index, E element) {  
 checkPositionIndex(index);  
 if (index == size)  
 linkLast(element); //直接在链表尾部新增节点  
 else  
 linkBefore(element, node(index)); //链表中间插入节点  
 }  
  
 void linkBefore(E e, Node<E> succ) {  
 final Node<E> pred = succ.prev;  
 final Node<E> newNode = new Node<>(pred, e, succ);  
 succ.prev = newNode;  
 if (pred == null)  
 first = newNode;  
 else  
 pred.next = newNode;  
 size++;  
 modCount++;  
 }  
}

1. JDK1.8 Vector源码

public class Vector<E>{  
 protected Object[] elementData; //底层使用数组来存储元素  
 protected int elementCount; //等同于size  
 protected int capacityIncrement; //数组扩增的增量大小  
  
 public Vector() {  
 this(10); //默认初始化底层数组大小为10，增量为0  
 }  
  
 public Vector(int initialCapacity) {  
 this(initialCapacity, 0);  
 }  
  
 public Vector(int initialCapacity, int capacityIncrement) {  
 super();  
 this.elementData = new Object[initialCapacity]; //初始化底层数组  
 this.capacityIncrement = capacityIncrement; //初始化底层数组的扩容增量  
 }  
  
 public synchronized boolean add(E e) { //同步，效率低下  
 modCount++;  
 ensureCapacityHelper(elementCount + 1); //考虑是否扩容数组  
 elementData[elementCount++] = e;  
 return true;  
 }  
  
 private void ensureCapacityHelper(int minCapacity) {  
 if (minCapacity - elementData.length > 0) //考虑是否需要扩容  
 grow(minCapacity);  
 }  
  
 private void grow(int minCapacity) { //minCapacity是数组的最小容量  
 int oldCapacity = elementData.length;  
 //默认的扩容规则：当前容量 + 增量  
 int newCapacity = oldCapacity + ((capacityIncrement > 0) ? capacityIncrement : oldCapacity);  
 if (newCapacity - minCapacity < 0)  
 newCapacity = minCapacity;  
 if (newCapacity - MAX\_ARRAY\_SIZE > 0)  
 newCapacity = hugeCapacity(minCapacity);  
 elementData = Arrays.copyOf(elementData, newCapacity);  
 }  
}

1. 单向链表

定义

private static class ListNode<T> {

T val;

ListNode<T> next;

public ListNode(T val) {

this.val = val;

}

}

头插法创建单向链表

//头插法创建单向链表：每次生成的新节点设置为头节点

private static ListNode<Integer> headLink() {

ListNode<Integer> header = null;

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

ListNode<Integer> newNode = new ListNode<>(random.nextInt(10) + 1);

if (header == null) {

header = newNode;

} else {

//将新节点放在header的前面

newNode.next = header;

//更新新节点就是头节点

header = newNode;

}

}

return header;

}

//头插法  
 private static ListNode headInsert() {

ListNode head;  
 if (head == null) {  
 head = new ListNode(); //特殊，头结点不存储任何数据  
 }  
 for (int i = 0; i < 10; i++) {  
 ListNode newNode = new ListNode(random());  
 newNode.next = head.next;  
 head.next = newNode;  
 }

return head;  
}

尾插法创建单向链表

//尾插法创建单向链表：每次生成的新节点设置为尾节点  
private static ListNode<Integer> tailLink() {  
 ListNode<Integer> header = null, tailer = null;  
 for (int i = 0; i < 10; i++) {  
 ListNode<Integer> newNode = new ListNode<>(random.nextInt(10) + 1);  
 if (header == null) {  
 header = tailer = newNode;  
 } else {  
 //尾节点后面插入新节点  
 tailer.next = newNode;  
 //更新新节点就是尾节点  
 tailer = newNode;  
 }  
 }  
 return header;  
}

//尾插法  
private static ListNode tailInsert() {  
 ListNode tail = null;  
 if (head == null) {  
 head = new ListNode(); //特殊，头结点不存储任何数据  
 tail = head;  
 }  
 for (int i = 0; i < 10; i++) {  
 ListNode newNode = new ListNode(random());  
 tail.next = newNode;  
 tail = newNode;  
 }

return head;  
}

正序输出单向链表

private static void positivePrint(ListNode head) {  
 if (head != null) {  
 System.out.println();  
 ListNode newHead = head;  
 while (newHead != null) {  
 System.out.print(newHead.val + "->");  
 newHead = newHead.next;  
 }  
 }  
}

单向链表逆序输出

//将单向链表逆序，改变原链表，参考https://blog.csdn.net/alpgao/article/details/86509265  
//借助头结点  
private static ListNode reverse(ListNode head) {

if (head == null || head.next == null) {  
 return head;  
 }  
 ListNode p = head.next, q = p.next, next = null;  
 p.next = null;  
 while (q != null) {  
 next = q.next;  
 q.next = p;  
 p = q;  
 q = next;  
 }  
 head.next = p;  
 return head;  
}

// 在不使用额外存储节点的情况下使一个单链表的所有节点逆序  
// 循环式 https://blog.csdn.net/xiao\_ma\_CSDN/article/details/80550092  
private static ListNode reverse1(ListNode head) {  
 if (head == null || head.next == null) {  
 return head;  
 }  
 ListNode curHead = head, prev = null, next = null;  
 while (curHead != null) {  
 //记录下当前节点的下一个节点  
 next = curHead.next;  
 //逆序  
 curHead.next = prev;  
 //后移  
 prev = curHead;  
 //后移  
 curHead = next;  
 }  
 return prev;  
}

//在不使用额外存储节点的情况下使一个单链表的所有节点逆序  
//递归方式https://blog.csdn.net/xiao\_ma\_CSDN/article/details/80550092  
private static ListNode reverse2(ListNode head) {  
 if (head == null || head.next == null) {  
 return head;  
 }  
 ListNode newHead = reverse2(head.next);  
 //另当前节点的下一个节点指向当前节点，形成闭合回路  
 head.next.next = head;  
 //断开闭合回路，实现逆序  
 head.next = null;  
 return newHead;  
}

//利用栈的先入后出特性实现倒序打印，并没有改变原链表  
private static void reverse3(ListNode head) {  
 ListNode newNode = head;  
 Stack<ListNode> stack = new Stack<>(); //利用栈的FILO特性  
 while (newNode != null) {  
 stack.push(newNode); //入栈  
 newNode = newNode.next;  
 }  
 System.out.println();  
 while (stack.size() > 0) {  
 newNode = stack.pop(); //出栈  
 System.out.print(newNode.val + "->");  
 }  
}

//链表入栈  
private static void reverse4(ListNode head) {  
 if (head == null || head.next == null) {  
 return;  
 }  
 ListNode curNode = head, next = null;  
 Stack<ListNode> stack = new Stack<>();  
 while (curNode != null) {  
 //将当前节点入栈  
 stack.push(curNode);  
 //找到next节点  
 next = curNode.next;  
 //断开下一个节点  
 curNode.next = null;  
 //后移  
 curNode = next;  
 }  
  
 head = stack.pop();  
 curNode = head;  
 while (!stack.isEmpty()) {  
 curNode.next = stack.pop();  
 curNode = curNode.next;  
 }  
}

1. 双向链表

定义

private static class DoubleListNode<T> {  
 T val;  
 DoubleListNode next;  
 DoubleListNode prev;  
 public DoubleListNode(T val) {  
 this.val = val;  
 }  
 }

头插法创建双向链表

private static void headInsert() {  
 DoubleListNode next = null, head = null;  
 for (int i = 0; i < 10; i++) {  
 DoubleListNode<Integer> newNode = new DoubleListNode(i);  
 if (head == null) {  
 head = newNode;  
 } else {  
 next = head.next;  
 if (next != null) {  
 next.prev = newNode;  
 }  
 newNode.next = next;  
 head.next = newNode;  
 newNode.prev = head;  
 }  
 }

return head;  
}

尾插法创建双向链表

private static void tailInsert() {  
 DoubleListNode tail = null, head = null;  
 for (int i = 0; i < 10; i++) {  
 DoubleListNode<Integer> newNode = new DoubleListNode(i);  
 if (head == null) {  
 head = tail = newNode;  
 } else {  
 tail.next = newNode;  
 newNode.prev = tail;  
 tail = newNode;  
 }  
 }

return head;  
}

LinkedHashMap

TreeMap

ConcurrentHashMap

HashSet

TreeSet

LinkedHashSet

HashMap

//计算大于目标容量最小的2次幂，最大2^30

static final int tableSizeFor(int cap) {

int n = cap - 1;

n |= n >>> 1;

n |= n >>> 2;

n |= n >>> 4;

n |= n >>> 8;

n |= n >>> 16;

return (n < 0) ? 1 : (n >= MAXIMUM\_CAPACITY) ? MAXIMUM\_CAPACITY : n + 1;

}

//计算key的hash值，用于散列分布到数组索引上

//计算方式：高16位与低16位做异或运算

//目的：int index = (n - 1) & hash(key);较少碰撞，散列均匀

//如果直接使用key的hashcode()作为hash很容易发生碰撞。比如，在n - 1为15(0x1111)时，散列值真正生效的只是低4位。当新增的键的hashcode()是2，18，34这样恰好以16的倍数为差的等差数列，就产生了大量碰撞。

static int hash(Object k){

int h;

return k == null ? 0 : ((h = k.hashCode()) ^ (h >>> 16));

}

1. 树

**二叉树**

定义：每个结点至多只有二棵子树(不存在度大于2的结点)  
性质： 二叉树的第i层至多有2^(i-1)个结点；

深度为k的二叉树至多有2^k-1个结点

public static class TreeNode<T> {

private T val;

private TreeNode left;

private TreeNode right;

}

**满二叉树**

定义：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。  
性质： 一颗树深度为h，最大层数为k，深度与最大层数相同，k=h;  
 叶子数为2h;  
 第k层的结点数是2^(k-1);  
 总结点数是2^k-1，且总节点数一定是奇数。

**完全二叉树**

定义：

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。

排序二叉树BST（二叉查找树、二叉搜索树Binary Search Tree）

性质：

若左子树不空，则左子树上所有结点的值均小于它的根结点的值；  
 若右子树不空，则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；  
性质：

对二叉查找树进行中序遍历，即可得到有序的数列。  
缺点：

在按顺序插入数据的情况下，二叉树会退化成链表结构，时间复杂度为O(n)，相当于一个有序链表

时间复杂度：

和二分查找一样，插入和查找的时间复杂度均为O(logn)，但是在最坏的情况下仍然会有O(n)的时间复杂度（退化成链表了）。原因在于插入和删除元素的时候，树没有保持平衡。我们追求的是在最坏的情况下仍然有较好的时间复杂度，这就是AVL设计的初衷。 二叉查找树的高度决定了二叉查找树的查找效率

**平衡二叉树AVL**

定义：

它是一棵空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树

平衡因子bf：节点的左右子树的深度差

特点：

不管我们是执行插入还是删除操作，只要不满足条件（**所有结点的左右子树高度差不超过1**），就要通过旋转来保存平衡，而因为旋转非常**耗时**，由此我们可以知道**AVL树适合用于插入与删除次数比较少，但查找多的情况。**

局限性：

由于维护这种高度平衡所付出的代价比从中获得的效率收益还大，故而实际的应用不多，更多的地方是用追求局部而不是非常严格整体平衡的红黑树。当然，**如果应用场景中对插入删除不频繁，只是对查找要求较高，那么AVL还是较优于红黑树**  
自平衡：

如果发现任意节点不是子AVL树时（左右高度差大于1），就需要通过旋转来平衡了

生成一颗平衡二叉树：

先按照生成二叉搜索树的方法构造二叉树，直至二叉树变得不平衡，即出现这样的节点：左子树与右子树的高度差大于1。至于如何调整，要看插入的导致二叉树不平衡的节点的位置。主要有四种调整方式：LL（左旋）、RR（右旋）、LR（先左旋再右旋）、RL（先右旋再左旋）

**红黑树BRT**：一种含有红黑结点并能自平衡的二叉查找树  
性质： 每个节点要么是黑色，要么是红色

根节点是黑色

每个叶子节点（NULL）是黑色

每个红色结点的两个子结点一定都是黑色

任意一结点到每个叶子结点的路径都包含数量相同的黑结点

从性质4和5又可以推出：

如果一个结点A存在黑子结点，那么该结点A肯定有两个子结点

红黑树的最长路径不超过最短路径的两倍

红黑树自平衡的三种操作：左旋、右旋和变色

左旋：以某个结点作为支点(旋转结点)，其右子结点变为旋转结点的父结点，右子结点的左子结点变为旋转结点的右子结点，左子结点保持不变。

右旋：以某个结点作为支点(旋转结点)，其左子结点变为旋转结点的父结点，左子结点的右子结点变为旋转结点的左子结点，右子结点保持不变。

变色：结点的颜色由红变黑或由黑变红。

红黑树和AVL树的区别

由于AVL树需要满足：所有结点的左右子树高度差不超过1。不管我们是执行插入还是删除操作，只要不满足上面的条件，就要通过旋转来保存平衡，而因为旋转非常耗时，由此我们可以知道AVL树适合用于插入与删除次数比较少，但查找多的情况。

**二叉树的遍历方式**

前序（根、左子树、右子树）、中序（左子树、根、右子树）、后序（左子树、右子树、根）、层次（从上到下，从左到右）

**B树（或者B-tree）**

M阶的B树指任意节点最多有M个子树，当M=2时是二叉树，当M=3时是三叉树

关键字树：枝节点的关键字数量>=ceil(M/2)-1，且<=M-1个

排序方式：所有树节点的关键字都是递增排列，遵循左小右大原则

子节点数：非叶子节点的子树范围(1,M]，且M>=2

**2-3树**

一个父节点可以有两个子节点，也可以有三个子节点，并且其也满足类似二叉搜索树的定义（父节点的值大于左子树，但小于右子树），所有叶子节点都在同一层

B+Tree

哈夫曼树

二叉树遍历（

前序（根左右）、中序（左根右）、后序（左右根）、层次、深度优先、广度优先遍历）

1. JDK1.8 HashMap源码

public class HashMap<K,V> {

//默认初始化容量 16。容量必须为2的次方。默认的hashmap大小为16

static final int DEFAULT\_INITIAL\_CAPACITY = 1 << 4;

static final int MAXIMUM\_CAPACITY = 1 << 30;

//默认加载因子

//即实际数量超过总数DEFAULT\_LOAD\_FACTOR的数量即会发生resize动作

static final float DEFAULT\_LOAD\_FACTOR = 0.75f;

//树化阈值 8。当单个segment的容量超过阈值时，将链表转化为红黑树

static final int TREEIFY\_THRESHOLD = 8;

//链表化阈值 6。当resize后或者删除操作后单个segment的容量低于阈值时，将红黑树转化为链表

static final int UNTREEIFY\_THRESHOLD = 6;

//最小树化容量 64。当桶中的bin被树化时最小的hash表容量，低于该容量时不会树化

static final int MIN\_TREEIFY\_CAPACITY = 64;

//底层维护了一个table存放Node节点

transient Node<K, V>[] table;

transient Set<Map.Entry<K, V>> entrySet;

transient int size;

transient int modCount; //fast-fail机制

//装载因子：用来衡量HashMap满的程度。loadFactor的默认值为0.75f。

//计算HashMap的实时装载因子的方法为：size/capacity，而不是占用桶的数量去除以capacity

final float loadFactor;

//扩容阈值：当HashMap的size值大于threshold时会执行resize操作

int threshold;

//单向链表

static class Node<K, V> implements Map.Entry<K, V> {

final int hash;

final K key;

V value;

Node<K, V> next;

}

static final class TreeNode<K, V> extends Entry<K, V> {

TreeNode<K, V> parent; // red-black tree links

TreeNode<K, V> left;

TreeNode<K, V> right;

TreeNode<K, V> prev; // needed to unlink next upon deletion

boolean red;

红黑树自平衡核心方法balanceInsertion

static <K, V> TreeNode<K, V> balanceInsertion(TreeNode<K, V> root, TreeNode<K, V> x) {

x.red = true; //插入结点是红色。理由：红色在父结点（如果存在）为黑色结点时，

//红黑树的黑色平衡没被破坏，不需要做自平衡操作。但如果插入结点是黑色，

//那么插入位置所在的子树黑色结点总是多1，必须做自平衡。(所有插入操作都是在叶子结点进行的，叶子节点一定是黑色)

//xp：当前节点x的父节点

//xpp：当前节点x的爷爷节点

//xppl：当前节点x的爷爷节点的左子节点

//xppr：当前节点x的爷爷节点的右子节点

for (TreeNode<K, V> xp, xpp, xppl, xppr; ; ) {

if ((xp = x.parent) == null) { //如果当前节点没有父节点，代表x就是根节点，根节点为黑色

x.red = false;

return x;

} else if (!xp.red || (xpp = xp.parent) == null)

//情况1：父节点xp为黑色，当前节点x为红色。因此红黑树已经自平衡，直接返回就可以了

//情况2：父节点xp为红色，根节点xp为黑色，当前节点x为红色，已经自平衡了（满足5大特性）

return root;

//此时当前节点x为红色，父节点xp为红色，祖父节点xpp一定是黑色

if (xp == (xppl = xpp.left)) {//如果父节点是祖父节点的左子节点xppl

if ((xppr = xpp.right) != null && xppr.red) { //如果存在叔叔节点（右），并且还是红色

xppr.red = false;

xp.red = false;

xpp.red = true;

x = xpp;

} else { //如果不存在叔叔节点 或者叔叔节点还是黑色

if (x == xp.right) {

root = rotateLeft(root, x = xp); //左旋父节点xp，

xpp = (xp = x.parent) == null ? null : xp.parent;

}

if (xp != null) {

xp.red = false;

if (xpp != null) {

xpp.red = true;

root = rotateRight(root, xpp);

}

}

}

} else { //如果xp是xpp的右子节点

if (xppl != null && xppl.red) {

xppl.red = false;

xp.red = false;

xpp.red = true;

x = xpp;

} else {

if (x == xp.left) {

root = rotateRight(root, x = xp);

xpp = (xp = x.parent) == null ? null : xp.parent;

}

if (xp != null) {

xp.red = false;

if (xpp != null) {

xpp.red = true;

root = rotateLeft(root, xpp);

}

}

}

}

}

}

}

计算key的hash值，用于散列分布到数组索引上

//hash函数大概的作用就是：高16bit不变，低16bit和高16bit做了一个异或，目的是减少碰撞

//目的：int index = (n - 1) & hash(key);较少碰撞，散列均匀

//如果直接使用key的hashcode作为hash很容易发生碰撞。比如在n - 1为15(0x1111)时，散列值真正生效的只是低4位。

//当新增的键的hashcode()是2，18，34这样恰好以16的倍数为差的等差数列，就产生了大量碰撞。

static final int hash(Object key) {

int h;

return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);

}

//hashmap的核心方法：新增键值对

public V put(K key, V value) {

return putVal(hash(key), key, value, false, true);

}

final V putVal(int hash, K key, V value, boolean onlyIfAbsent, boolean evict) {  
 Node<K, V>[] tab;  
 Node<K, V> p;  
 int n, i;  
 //1.看table数组是否已经初始化  
 if ((tab = table) == null || (n = tab.length) == 0) //还没有初始化table  
 n = (tab = resize()).length; //初始化table，分配内存  
  
 //2.找到该key对应数组中的位置，(n - 1) & hash就是下标，前提：n是2的幂次方  
 //在算index时，用位运算（n-1） & hash而不是模运算 hash % n的好处  
 //2.1位运算消耗资源更少，更有效率  
 //2.2避免了hashcode为负数的情况  
 //为什么capcity是2的幂？  
 //因为 算index时用的是（n-1） & hash，这样就能保证n -1是全为1的二进制数，如果不全为1的话，存在某一位为0，那么0，1与0与的结果都是0，这样便有可能将两个hash不同的值最终装入同一个桶中，造成冲突。所以必须是2的幂。  
 if ((p = tab[i = (n - 1) & hash]) == null)  
 //当前桶无元素，直接放就行了  
 tab[i] = newNode(hash, key, value, null);  
 else {  
 //桶内有元素  
 Node<K, V> e;  
 K k;  
 //key是否存在  
 if (p.hash == hash &&  
 ((k = p.key) == key || (key != null && key.equals(k))))  
 //桶内第一个元素的key等于待放入的key，用  
 e = p;  
 else if (p instanceof TreeNode) //如果节点类型是树，说明此时已经把链表转出红黑树了  
 e = ((TreeNode<K, V>) p).putTreeVal(this, tab, hash, key, value);  
 else {//桶内还是一个链表，则插入链尾（尾插）  
 //此时还是链表结构的，没有转成红黑树，此时要做的事情有：  
 //准备在链表上查找key是否已经存在，如果存在直接用新value值覆盖老value值  
 //如果不存在，就需要在链表上插入节点，如果节点数超过8个，就将链表转成红黑树  
 for (int binCount = 0; ; ++binCount) {  
 if ((e = p.next) == null) { //访问到链表尾部了，此时key没有重复，可以在链表尾部插入新节点了  
 p.next = newNode(hash, key, value, null);  
 if (binCount >= TREEIFY\_THRESHOLD - 1) // -1 for 1st  
 treeifyBin(tab, hash); //开始把链表转红黑树了  
 break;  
 }  
 if (e.hash == hash &&  
 ((k = e.key) == key || (key != null && key.equals(k))))  
 //发现链表中key重复了，直接break，重复的节点保存在e中  
 break;  
 p = e;  
 }  
 }  
 if (e != null) { // existing mapping for key  
 V oldValue = e.value;  
 if (!onlyIfAbsent || oldValue == null)  
 e.value = value;  
 afterNodeAccess(e);  
 return oldValue;  
 }  
 }  
 ++modCount;  
 if (++size > threshold) //检查是否应该扩容  
 resize();  
 afterNodeInsertion(evict);  
 return null;  
 }

计算大于等于cap的最小2的幂次方的数

//原理：只需要关注二进制中第一个高位为1的位即可，其余为可以忽略

\*1.假设在第n位（......n n-1n-2 ......2 1）

\*2.无符号右移1位再按位或运算，则n n-1都是1 -->有2位为1

\*3.无符号右移2位再按位或运算，则n ......n-3都是1 -->有4位为1

\*4.无符号右移4位再按位或运算，则n ......n-7都是1 -->有8位为1

\*5.无符号右移8位再按位或运算，则n ......n-15都是1 -->有16位为1

\*6.无符号右移16位再按位或运算，则n ......n-31都是1 -->有32位为1

\*7.java int类型4字节，32位，正数符号位为0，且无符号右移时最多31位为1，即最大生成Interger.MAX\_VALUE=0x7FFFFFFF

\*/

static final int tableSizeFor(int cap) {

int n = cap - 1;//如果cap已经是2的幂，又没有执行这个减1操作，则执行完后面的几条无符号右移操作之后，返回的capacity将是这个cap的2倍

n |= n >>> 1;

n |= n >>> 2;

n |= n >>> 4;

n |= n >>> 8;

n |= n >>> 16;

return (n < 0) ? 1 : (n >= MAXIMUM\_CAPACITY) ? MAXIMUM\_CAPACITY : n + 1;

}

底层数组扩容及重新散列

/\*\*

\* resize扩容操作，主要用在两处：

\* 向一个空的HashMap中执行put操作时，会调用resize()进行初始化，要么默认初始化，capacity为16，要么根据传入的值进行初始化

\* put操作后，检查到size已经超过threshold，那么便会执行resize，进行扩容，如果此时capcity已经大于了最大值，那么便把threshold置为int最大值，否则，对capcity,threshold进行扩容操作。

\* 发生了扩容操作，那么必须Map中的所有的数进行再散列，重新装入。

\* <p>

\* 扩容后需要重新散列的原因:

\* 扩容前：某hash(key)=1000,n = 8,此时 (n-1) & hash = 0111 & 1000 = 0

\* 扩容后： n = 16,此时(n-1) & hash = 1111 & 1000 = 1

\* 散列位置发生了变化，如果扩容后不重新散列，相同的hash(key)可能会散列到不同的索引上

\* <p>

\* 如果cap由8扩容16，

\* 8-1= 00111

\* 16-1=01111

\*/

final Node<K, V>[] resize() {

Node<K, V>[] oldTab = table;

int oldCap = (oldTab == null) ? 0 : oldTab.length; //当前的桶数

int oldThr = threshold; //当前的扩容阈值

int newCap, newThr = 0;

if (oldCap > 0) {

if (oldCap >= MAXIMUM\_CAPACITY) {

threshold = Integer.MAX\_VALUE;

return oldTab;

} else if ((newCap = oldCap << 1) < MAXIMUM\_CAPACITY && //2倍扩容

oldCap >= DEFAULT\_INITIAL\_CAPACITY)

newThr = oldThr << 1; // double threshold

} else if (oldThr > 0) {

newCap = oldThr;

} else {

newCap = DEFAULT\_INITIAL\_CAPACITY;

newThr = (int) (DEFAULT\_LOAD\_FACTOR \* DEFAULT\_INITIAL\_CAPACITY);

}

if (newThr == 0) {

float ft = (float) newCap \* loadFactor;

newThr = (newCap < MAXIMUM\_CAPACITY && ft < (float) MAXIMUM\_CAPACITY ?

(int) ft : Integer.MAX\_VALUE);

}

threshold = newThr;

Node<K, V>[] newTab = (Node<K, V>[]) new Node[newCap]; //准备扩容数组了

table = newTab;

if (oldTab != null) {

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

Node<K, V> e;

if ((e = oldTab[j]) != null) {

oldTab[j] = null; //gc

if (e.next == null) //当前桶上就一个节点

newTab[e.hash & (newCap - 1)] = e;

else if (e instanceof TreeNode) //当前桶已经树化

((TreeNode<K, V>) e).split(this, newTab, j, oldCap);

else { //当前桶还是链表

Node<K, V> loHead = null, loTail = null;

Node<K, V> hiHead = null, hiTail = null;

Node<K, V> next;

do {

next = e.next;

//扩容前计算索引位置index = (n - 1) & hash，此时n = oldCap

// int oldIndex = (oldCap - 1) & hash;

//扩容后

//int newIndex = (2 \* oldCap - 1) & hash;

//eg1：如果扩容前n = 4,扩容后就是n = 8

// 0011 & hash 0111 & hash 扩容前后索引位置是否发生变化取决于hash的第低三位是否是0还是1

//eg2：如果扩容前n = 8,扩容后就是n = 16

// 0111 & hash 1111 & hash 取决于第低四位是否是0还是1

//如果是0,0按位与任何数都是0，此时扩容后位置没有发生变化

//如果是1,此时扩容后位置会发生变化：此时位置如何变化呢？

//索引位置在高位多了一个1，就是增加了oldCap

//oldCap可能是 0010,0100,1000,...

if ((e.hash & oldCap) == 0) { //该位置处扩容后位置不会变化

if (loTail == null)

loHead = e;

else

loTail.next = e;

loTail = e;

} else { //该位置处扩容后位置发生改变

if (hiTail == null)

hiHead = e;

else

hiTail.next = e;

hiTail = e;

}

} while ((e = next) != null);

if (loTail != null) {

loTail.next = null;

newTab[j] = loHead; //2倍扩容后索引位置不变

}

if (hiTail != null) {

hiTail.next = null;

newTab[j + oldCap] = hiHead;//2倍扩容后索引位置增加oldCap

}

}

}

}

}

return newTab;

}

//将hash对应table下标处的单向链表转成双向链表，然后再将这一个双向链表转成红黑树

final void treeifyBin(Node<K, V>[] tab, int hash) {

int n, index;

Node<K, V> e;

if (tab == null || (n = tab.length) < MIN\_TREEIFY\_CAPACITY) //如果table数组还没有被初始化，或者数组长度小于64

resize(); //

else if ((e = tab[index = (n - 1) & hash]) != null) { //该hash对应table位置上不为空

TreeNode<K, V> hd = null, tl = null;

//do while循环目的：将单向链表转成双向链表

do {

TreeNode<K, V> p = replacementTreeNode(e, null); //将链表节点转成树节点

if (tl == null)

hd = p;

else {

p.prev = tl;

tl.next = p;

}

tl = p;

} while ((e = e.next) != null);

if ((tab[index] = hd) != null) //将双向链表头结点地址存入该hash值对应的下标处

hd.treeify(tab); //着手开始将这一个双向链表转成红黑树了

}

}

}

JDK1.8 ConcurrentHashMap源码

计算key的hash值

static final int spread(int h) { //h = key.hashCode();

return (h ^ (h >>> 16)) & 0x7fffffff;

}

CAS原子操作

//获取Node[].class的首元素地址偏移

private static final long ABASE = U.arrayBaseOffset(Node[].class);

//位移运算

private static final int ASHIFT = 31 - Integer.numberOfLeadingZeros(U.arrayIndexScale(Node[].class));

//unsafe获取数组tab的下标i位置处的元素

static final <K,V> Node<K,V> tabAt(Node<K,V>[] tab, int i) {  
 return (Node<K,V>)U.getObjectVolatile(tab, ((long)i << ASHIFT) + ABASE);  
}

//尝试cas更新数组tab的下标i位置处元素为v，可能会失败  
static final <K,V> boolean casTabAt(Node<K,V>[] tab, int i, Node<K,V> c, Node<K,V> v) {  
 return U.compareAndSwapObject(tab, ((long)i << ASHIFT) + ABASE, c, v);  
}

//cas更新数组tab的下标i位置处元素为v  
static final <K,V> void setTabAt(Node<K,V>[] tab, int i, Node<K,V> v) {  
 U.putObjectVolatile(tab, ((long)i << ASHIFT) + ABASE, v);  
}

默认初始化底层table

private transient volatile int sizeCtl;

private final Node<K,V>[] initTable() {  
 Node<K,V>[] tab; int sc;  
 while ((tab = table) == null || tab.length == 0) {  
 if ((sc = sizeCtl) < 0)  
 Thread.yield(); // lost initialization race; just spin  
 else if (U.compareAndSwapInt(this, SIZECTL, sc, -1)) {//尝试cas更新sizeCtl=-1

//sizeCtl=-1，表明当前线程正在初始化table  
 try {  
 if ((tab = table) == null || tab.length == 0) { //双重校验  
 int n = (sc > 0) ? sc : 16; //默认table大小为16  
 Node<K,V>[] nt = (Node<K,V>[])new Node<?,?>[n];  
 table = tab = nt;  
 sc = n - (n >>> 2); *//相当于0.75\*n 设置一个扩容的阈值*  
 }  
 } finally {  
 sizeCtl = sc;  
 }  
 break;  
 }  
 }  
 return tab;  
}

public class ConcurrentHashMap {

}