# 3年JAVA技能总结-数据结构篇

1.集合（底层源码）  
 list:  
 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  
 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  
 set:  
 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  
 CopyOnWrite系列

数据结构  
 队列：Queue  
 先入先出FIFO  
 栈：Stack  
 先入后出FILO  
 链表：  
 分类：单向链表、双向链表  
 创建方式：头插法、尾插法  
 树：  
 二叉树：每个结点至多只有二棵子树(不存在度大于2的结点)  
 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  
 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。  
 满二叉树的性质：  
 　　1) 一颗树深度为h，最大层数为k，深度与最大层数相同，k=h;  
 　　2) 叶子数为2h;  
 　　3) 第k层的结点数是2^(k-1);  
 　　4) 总结点数是2^k-1，且总节点数一定是奇数。  
 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  
 二叉查找树(二叉搜索树、二叉排序树)：  
 1)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；  
 　　2) 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；  
 　　3) 左、右子树也分别为二叉排序树；  
 　　4) 没有键值相等的节点。  
 性质：对二叉查找树进行中序遍历，即可得到有序的数列。  
 二叉查找树的时间复杂度：它和二分查找一样，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。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，这就是平衡查找树设计的初衷。  
 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：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  
 红黑树BRT：  
 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）

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));

}

二叉树BindaryTree

红黑树BLTree

二叉平衡树AVL

2-3树

二叉搜索树

完全二叉树

二叉搜索树：一个节点的左子节点的关键字值小于这个节点，右子节点的关键字值大于或等于这个父节点

1. Tree

B+Tree

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）

ArrayDeque{

//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 | ... |