# jdk8中数据结构

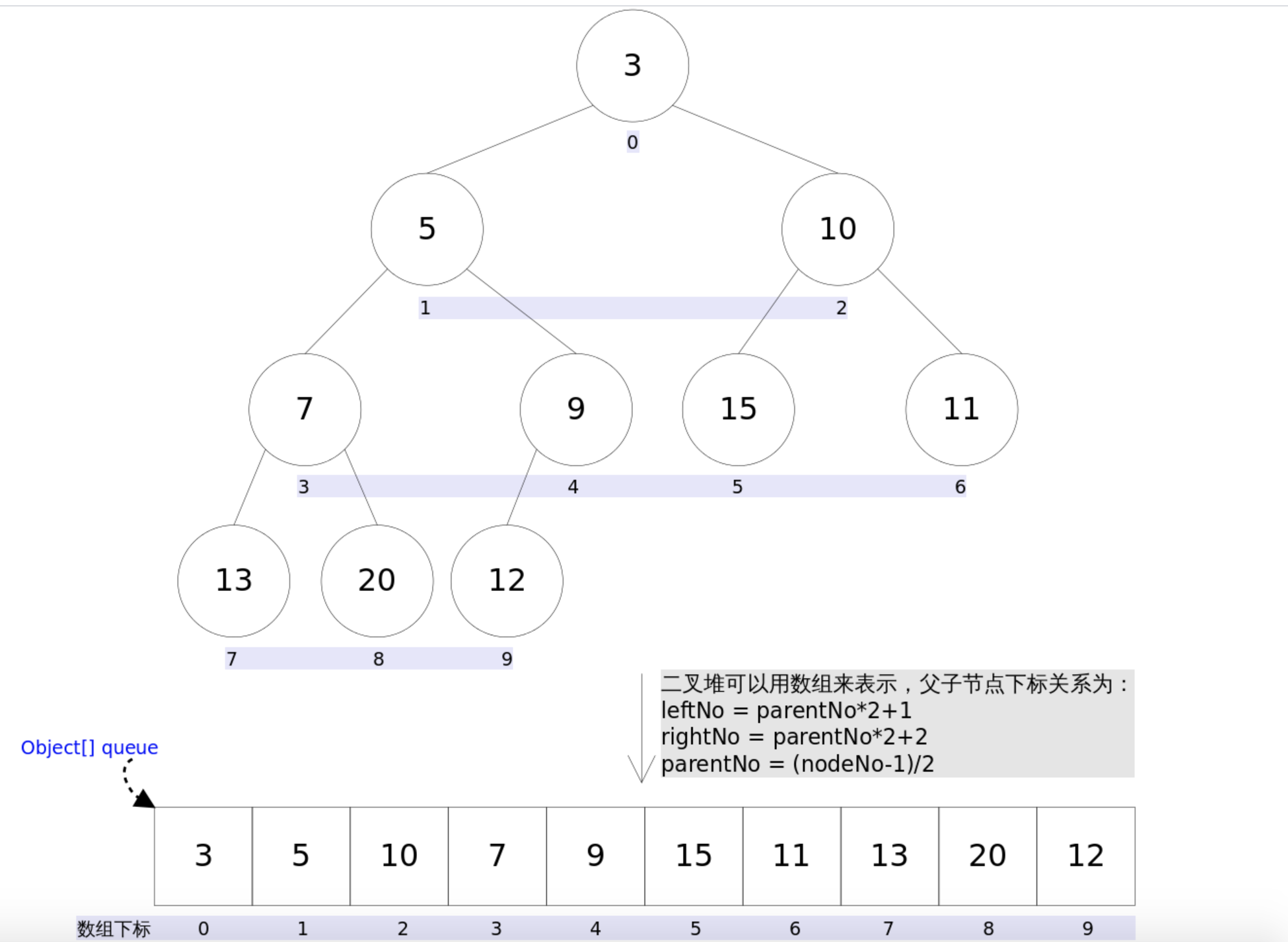
# 对列

## PriorityQueue(优先对列)：底层是数组

特性：有序，父结点大于小于左右孩子。

PriorityQueue的peek()和element操作是常数时间，add(), offer(), 无参数的remove()以及poll()方法的时间复杂度都是log(N)

通过完全二叉树实现的小顶堆(任意一个非叶子节点的权值，都不大于器左右子结点的权值)。底层是数组



### 1 字段queue

底层数组对象

|  |
| --- |
| */\*\*  \* Priority queue represented as a balanced binary heap: the two  \* children of queue[n] are queue[2\*n+1] and queue[2\*(n+1)]. The  \* priority queue is ordered by comparator, or by the elements'  \* natural ordering, if comparator is null: For each node n in the  \* heap and each descendant d of n, n <= d. The element with the  \* lowest value is in queue[0], assuming the queue is nonempty.  \*/*  *// 优先对列是平衡二叉堆，queue[n]的两个孩子节点分别是queue[2\*n+1],*  *// queue[2\*(n+1)]。优先对列通过比较器使其有序，或者按自然顺序。* transient Object[] queue; // non-private to simplify nested class access |

### 2 方法add和offer：入对列

|  |
| --- |
| public boolean add(E e) {  return offer(e); }  public boolean offer(E e) {  if (e == null)  throw new NullPointerException();  modCount++;  int i = size;  if (i >= queue.length)  // 扩容操作  grow(i + 1);  size = i + 1;  if (i == 0)  queue[0] = e;  else  // 进行调整  siftUp(i, e);  return true; }  private void siftUp(int k, E x) {  if (comparator != null)  // 通过比较器  siftUpUsingComparator(k, x);  else  // 按照自然顺序  siftUpComparable(k, x); }  private void siftUpComparable(int k, E x) {  Comparable<? super E> key = (Comparable<? super E>) x;  while (k > 0) {  // 无符号右移1位，相当于除以2，计算出父结点的索引  int parent = (k - 1) >>> 1;  // 父结点的元素  Object e = queue[parent];  // 如果果key（即目标元素x）刚好大于等于parent，就退出循环  if (key.compareTo((E) e) >= 0)  break;  // 走到这里说明，key(即目标元素x)，小于e(parent),那么就需要交换  queue[k] = e;  // 以父结点索引继续向上寻找  k = parent;  }  queue[k] = key; }  这个方法和上个方法一样的，只是比较器变成立自定义的比较器  private void siftUpUsingComparator(int k, E x) {  while (k > 0) {  int parent = (k - 1) >>> 1;  Object e = queue[parent];  if (comparator.compare(x, (E) e) >= 0)  break;  queue[k] = e;  k = parent;  }  queue[k] = x; }  如下图 |

### 3 poll 出队方法

|  |
| --- |
| public E poll() {  if (size == 0)  return null;  int s = --size;  modCount++;  // 取出对列头部元素  E result = (E) queue[0];  // 取出对列尾部元素  E x = (E) queue[s];  queue[s] = null;  if (s != 0)  // 调整对列  siftDown(0, x);  return result; }  private void siftDown(int k, E x) {  if (comparator != null)  siftDownUsingComparator(k, x);  else  siftDownComparable(k, x); }  private void siftDownComparable(int k, E x) {  Comparable<? super E> key = (Comparable<? super E>)x;  // 求出2分之一索引  int half = size >>> 1; // loop while a non-leaf  while (k < half) {  // 这是求出左孩子的索引，假设左孩子是最小的那个  int child = (k << 1) + 1; // assume left child is least  // 左孩子元素  Object c = queue[child];  // 右孩子  int right = child + 1;  // 若是左孩子大于右孩子  // 那就将右孩子的值赋给c  if (right < size &&  ((Comparable<? super E>) c).compareTo((E) queue[right]) > 0)  c = queue[child = right];  if (key.compareTo((E) c) <= 0)  break;  // 左孩子(实际是左右孩子中小的那个值)移到父结点  queue[k] = c;  // 在以小的那个索引开始往下递归，逐层比对左右孩子，将小的那个提升到父结点位置  k = child;  }  queue[k] = key; } |

核心方法就是siftUp和siftDown。

## DelayQueue(延迟对列)：底层是PriorityQueue

## ArrayBlockingQueue(有界阻塞对列)：底层是数组

## ArrayBlockingQueue(有界阻塞对列)：底层是数组

## LinkedBlockingQueue(无界阻塞对列):底层是链表