```java
		}

		public void receive(int num) {
			if (num < 1) {
				return;
			}
			Node cur = new Node(num);
			headMap.put(num, cur);
			tailMap.put(num, cur);
			if (tailMap.containsKey(num - 1)) {
				tailMap.get(num - 1).next = cur;
				tailMap.remove(num - 1);
				headMap.remove(num);
			}
			if (headMap.containsKey(num + 1)) {
				cur.next = headMap.get(num + 1);
				tailMap.remove(num);
				headMap.remove(num + 1);
			}
			if (headMap.containsKey(lastPrint + 1)) {
				print();
			}
		}

		private void print() {
			Node node = headMap.get(++lastPrint);
			headMap.remove(lastPrint);
			while (node != null) {
				System.out.print(node.num + " ");
				node = node.next;
				lastPrint++;
			}
			tailMap.remove(--lastPrint);
			System.out.println();
		}

	}
```

设计一个没有扩容负担的堆结构

【题目】

堆结构一般是使用固定长度的数组结构来实现的。这样的实现虽然足够经典，但存在扩容的负担，比如不断向堆中增加元素，使得固定数组快耗尽时，就不得不申请一个更大的固定数组，然后把原来数组中的对象复制到新的数组里完成堆的扩容，所以，如果扩容时堆中的元素个数为 N，那么扩容行为的时间复杂度为 $O(N)$。请设计一种没有扩容负担的

堆结构，即在任何时刻有关堆的操作时间复杂度都不超过 $O(\log N)$。

【要求】

1. 没有扩容的负担。
2. 可以生成小根堆，也可以生成大根堆。
3. 包含 getHead 方法，返回当前堆顶的值。
4. 包含 getSize 方法，返回当前堆的大小。
5. 包含 add(x)方法，即向堆中新加元素 x，操作后依然是小根堆/大根堆。
6. 包含 popHead 方法，即删除并返回堆顶的值，操作后依然是小根堆/大根堆。
7. 如果堆中的节点个数为 N，那么各个方法的时间复杂度为：

   getHead：$O(1)$。

   getSize：$O(1)$。

   add：$O(\log N)$。

   popHead：$O(\log N)$。

【难度】

将 ★★★★

【解答】

本题的设计方法有很多，本书提供的方法实际上是实现了完全二叉树结构，并含有堆的调整过程。二叉树的节点类型如下，比经典的二叉树节点多一条指向父节点的 parent 指针：

```
public class Node<K> {
        public K value;
        public Node<K> left;
        public Node<K> right;
        public Node<K> parent;

        public Node(K data) {
                value = data;
        }
}
```

本书实现的堆结构叫 MyHeap 类，MyHeap 中有四个重要的组成部分。

- head：Node 类型的变量，表示当前堆的头节点。

- last：Node 类型的变量，表示当前堆的堆尾节点，也就是最后一排的最右节点。
- size：整型变量，表示当前堆的大小。
- comp：继承了 Comparator 接口的比较器类型的变量。在构造 Myheap 实例时由用户定义，通过定义堆中元素的比较方式，自然可以将堆实现成大根堆或小根堆。comp 变量是在构造时一经设定就不能更改。

所有堆的操作在执行时，变量 head、last 和 size 都能够正确更新是 MyHeap 类实现的重点。其中 getHead 方法和 getSize 方法是很容易实现的，就是直接取值返回即可。那么接下来就重点介绍 add 方法和 popHead 方法的实现细节。

add 方法的实现。如果想要把元素 value 加入到堆中，首先生成二叉树节点类型的实例，即 new Node<value 的类型>(value)，假设生成的节点为 newNode。把 newNode 加到二叉树上的具体过程如下：

1．如果 size==0，说明当前的堆没有节点，三个变量简单赋值即可：

```
if (size == 0) {
        head = newNode;
        last = newNode;
        size++;
        return;
}
```

2．如果 size>0，说明当前的堆有节点，此时想要加上 newNode 的困难在于，不知道 newNode 应该加到二叉树的什么位置。此时利用 last 的位置来找到 newNode 应该加的位置。

1）last 具体在堆中的什么位置特别关键，具体有如下三种情况：

情况一，last 是当前层的最后一个节点，也就是当前层已经满，无法再加新的节点，那么 newNode 应该加在新一层最左的位置。

情况二，如果 last 是 last 父节点的左孩子，那么 newNode 应该加在 last 父节点的右孩子的位置。

情况三，如果 last 既不是情况一，也不是情况二，则参见图 9-7。

图 9-7 代表情况三，即当前层并没有添加满，但是 last 的父节点（比如图中的 D 节点）已经添加满，此时需要一个向上寻找的过程。先以 last 作为当前节点，然后看看当前节点是不是当前节点的父节点的左孩子，如果不是，就一直向上。比如图 9-7 中的节点 I，它不是其父节点的左孩子，那么向上寻找开始，节点 D 成为当前节点。此时发现节点 D 是其父节点（即节点 B）的左孩子，此时寻找结束。新节点 newNode 应该加在节点 B 的右子树的最左节点的左孩子的位置上，即节点 E 的左孩子位置。下面再举一例，如图 9-8 所示。

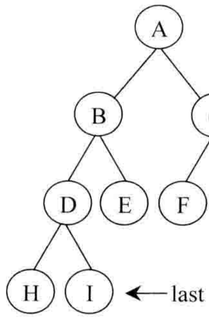

图 9-7

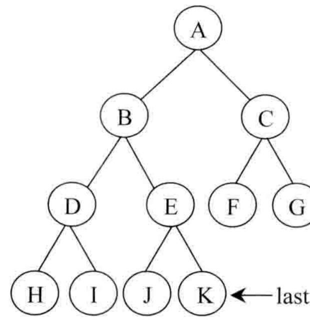

图 9-8

图 9-8 中 last 节点是节点 K，如何找到 newNode 应该加的位置呢？和图 9-7 的方式相同，也是往上寻找的过程。开始时当前节点为节点 K，发现它不是其父节点（E）的左孩子，那么节点 E 变成当前节点，发现也不是其父节点（B）的左孩子，那么节点 B 变成当前节点，发现节点 B 是其父节点 A 的左孩子，此时向上的过程停止。新节点 newNode 应该加在节点 A 的右子树的最左节点的左孩子的位置上，即节点 F 的左孩子位置。

2）加完 newNode 之后，newNode 就成为新的 last，令 last=newNode，同时 size++。

3）此时的 last 节点就是新加节点，虽然加在了二叉树上，但还没有经历建堆的调整过程。比如，如果整个堆是大根堆，而新加节点的值又很大，按道理，这个节点应该经历向上交换的过程，所以最后应该从 last 节点向上经历堆的调整过程，即 heapInsert 过程。同时需要特别注意的是，在交换的过程中，last 和 head 的值可能会变化，如图 9-9 所示。

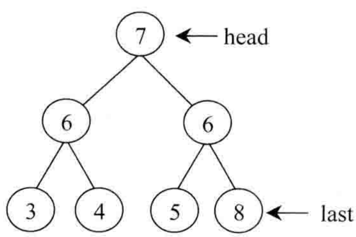

图 9-9

假设加上新节点（值为 8 的节点）之后的完全二叉树如图 9-9 所示，很明显，last 节点需要往上调整的过程。调整之后的二叉树应该为图 9-10。

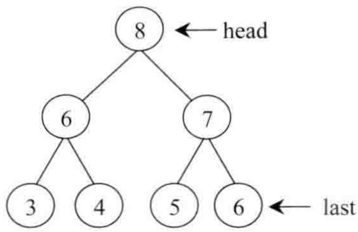

图 9-10

如果在经历调整之后，新加的节点最后没有占据头节点的位置，那么 head 的值当然是不用改变的，但如果最后占据了头节点的位置，则 head 的值应该调整，比如图 9-10 中 head 的值应该变为节点 8。同理，如果在经历调整时发现，新加的节点并不比它的父节点大，说明新加的节点不需要向上移动，那么 last 的值当然还是新加的节点，但如果新加的节点需要向上移动，比如图 9-10，那么 last 的值也需要调整，应该设为新加的节点的父节点（图 9-10 中的节点 6）。只有 head 和 last 在调整的每一步都正确地更新，整个设计才能不出错。具体请参看如下代码中 MyHeap 类实现的 heapInsertModify 方法。

popHead 方法的实现。删除堆顶节点并返回堆顶的值，具体过程如下：

1. 如果 size==0，说明当前堆为空，直接返回 null，也不需要任何调整。

2. 如果 size==1，说明当前堆里只有一个节点，返回节点值并将堆清空，即如下代码：

```
Node<K> res = head;
if (size == 1) {
        head = null;
        last = null;
        size--;
        return res.value;
}
```

3. 如果 size>1，把当前堆顶节点记为 res，把最后一个元素（last）放在堆顶位置作为新的头，同时从头部开始进行堆的调整，使其继续是大根/小根堆，最后返回 res.value 即可。话虽如此，但是这个过程还是要保证 head 和 last 的正确更新，具体细节如下：

1）先把堆中最后一个节点（last）和整个堆结构断开，记为 oldLast。因为 oldLast 要放在头节点的位置，所以 last 的值应该变成 oldLast 节点之前的那个节点，同样有三种情况。

情况一，如果 oldLast 在断开之前是其所在层的最左节点，那么在断开之后，last 应该变为上一层的最右节点。

情况二，如果 oldLast 在断开之前是 oldLast 的父节点的右孩子，那么在断开之后，last

应该变为 oldLast 的父节点的左孩子。

情况三，除情况一和情况二外，还有一种情况，如图 9-11 所示。

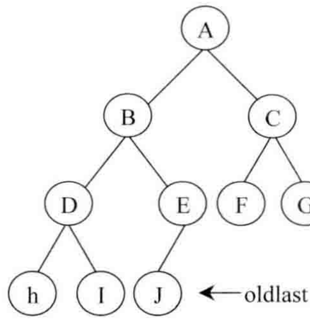

图 9-11

图 9-11 代表了情况三，即 oldLast 并不是当前层的最左节点，也不是其父节点的右孩子，此时需要一个向上寻找的过程。先以 oldLast 作为当前节点，然后看当前节点是不是当前节点的父节点的右孩子，如果不是，就一直向上。比如，图 9-11 中的节点 J，它不是其父节点的右孩子，那么向上寻找开始，节点 E 成为当前节点，此时发现节点 E 是其父节点（即节点 B）的右孩子，寻找结束。last 节点应该设成节点 B 的左子树的最右节点（即节点 I）。我们再举一例，如图 9-12 所示。

图 9-12 中的 oldLast 节点是节点 L，如何设置 last 节点的值呢？和图 9-11 的方式相同，也是往上寻找的过程。开始时当前节点为节点 L，发现它不是其父节点（F）的右孩子，那么节点 F 变成当前节点，发现也不是其父节点（C）的右孩子，那么节点 C 变成当前节点，发现节点 C 是其父节点 A 的右孩子，此时向上的过程停止。Last 节点应该设成节点 A 的左子树的最右节点，即节点 K。步骤 1）的具体过程请参看 MyHeap 类实现的 popLastAndSetPreviousLast 方法。

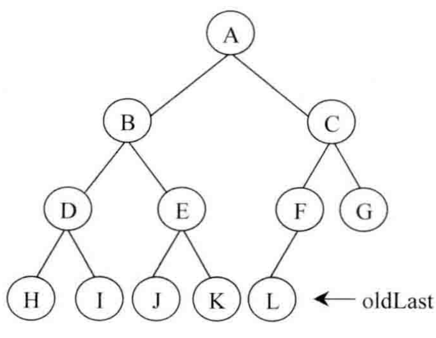

图 9-12

2）断开 oldLast 节点后，堆中的元素少了一个，所以 size 减 1。如果 size 在减 1 之后有 size==1，说明一开始堆的大小为 2，断开 oldLast 之后堆中只剩一个头节点。那么此时令 oldLast 作为新的头节点，并返回旧的头节点的值即可，代码如下：

```
Node<K> res = head;
Node<K> oldLast = popLastAndSetPreviousLast();
if (size == 1) {
        head = oldLast;
        last = oldLast;
        return res.value;
}
```

3）如果断开 oldLast 节点后，size 依然大于 1。那么将 oldLast 设成新的头节点，然后从堆顶开始往下调整堆结构，即 heapify 的过程，此时依然要注意 head 和 last 可能改变的情况，因为调整的过程中新的头节点（即 oldLast）还可能会移动，使得 head 和 last 位置上的节点发生变化，具体过程请参看 MyHeap 类实现的 heapify 方法。

MyHeap 类的设计就介绍完了，与经典堆结构是一个数组结构不同的是，MyHeap 类是一个完全二叉树结构，所以两个相邻节点在交换位置时的处理会更复杂，都考虑彼此的拓扑关系，才能做到正确地进行交换。具体请参看 MyHeap 类实现的 swapClosedTwoNodes 方法。当然也可以不进行结构上的交换，而只是交换两个节点的值，即 Node.value。

add 和 popHead 方法的所有操作都是在完全二叉树的一条或两条路径上进行的操作，所以每一个操作的代价都是完全二叉树的高度级别，一个节点数为 N 的完全二叉树高度为 $O(\log N)$，所以 add 和 popHead 方法的时间复杂度为 $O(\log N)$。MyHeap 类的全部实现如下：

```
public class MyHeap<K> {
        private Node<K> head; // 堆头节点
        private Node<K> last; // 堆尾节点
        private long size; // 当前堆的大小
        private Comparator<K> comp; // 大根堆或小根堆

        public MyHeap(Comparator<K> compare) {
                head = null;
                last = null;
                size = 0;
                comp = compare; // 基于比较器决定是大根堆还是小根堆
        }

        public K getHead() {
                return head == null ? null : head.value;
        }

        public long getSize() {
```

```
		return size;
	}

	public boolean isEmpty() {
		return size == 0 ? true : false;
	}

	// 添加一个新节点到堆中
	public void add(K value) {
		Node<K> newNode = new Node<K>(value);
		if (size == 0) {
			head = newNode;
			last = newNode;
			size++;
			return;
		}
		Node<K> node = last;
		Node<K> parent = node.parent;
		// 找到正确的位置并插入到新节点
		while (parent != null && node != parent.left) {
			node = parent;
			parent = node.parent;
		}
		Node<K> nodeToAdd = null;
		if (parent == null) {
			nodeToAdd = mostLeft(head);
			nodeToAdd.left = newNode;
			newNode.parent = nodeToAdd;
		} else if (parent.right == null) {
			parent.right = newNode;
			newNode.parent = parent;
		} else {
			nodeToAdd = mostLeft(parent.right);
			nodeToAdd.left = newNode;
			newNode.parent = nodeToAdd;
		}
		last = newNode;
		// 建堆过程及其调整
		heapInsertModify();
		size++;
	}

	public K popHead() {
		if (size == 0) {
			return null;
		}
		Node<K> res = head;
		if (size == 1) {
			head = null;
			last = null;
			size--;
			return res.value;
```

```java
		}
		Node<K> oldLast = popLastAndSetPreviousLast();
		// 如果弹出堆尾节点后，堆的大小等于 1 的处理
		if (size == 1) {
			head = oldLast;
			last = oldLast;
			return res.value;
		}
		// 如果弹出堆尾节点后，堆的大小大于 1 的处理
		Node<K> headLeft = res.left;
		Node<K> headRight = res.right;
		oldLast.left = headLeft;
		if (headLeft != null) {
			headLeft.parent = oldLast;
		}
		oldLast.right = headRight;
		if (headRight != null) {
			headRight.parent = oldLast;
		}
		res.left = null;
		res.right = null;
		head = oldLast;
		// 堆 heapify 过程
		heapify(oldLast);
		return res.value;
	}

	// 找到以 node 为头的子树中，最左的节点
	private Node<K> mostLeft(Node<K> node) {
		while (node.left != null) {
			node = node.left;
		}
		return node;
	}

	// 找到以 node 为头的子树中，最右的节点
	private Node<K> mostRight(Node<K> node) {
		while (node.right != null) {
			node = node.right;
		}
		return node;
	}

	// 建堆及调整的过程
	private void heapInsertModify() {
	  Node<K> node = last;
	  Node<K> parent = node.parent;
	  if (parent != null && comp.compare(node.value, parent.value) < 0) {
		last = parent;
	  }
	  while (parent != null && comp.compare(node.value, parent.value) < 0) {
		swapClosedTwoNodes(node, parent);
```

```
            parent = node.parent;
        }
        if (head.parent != null) {
            head = head.parent;
        }
    }

    // 堆 heapify 过程
private void heapify(Node<K> node) {
    Node<K> left = node.left;
    Node<K> right = node.right;
    Node<K> most = node;
    while (left != null) {
        if (left != null && comp.compare(left.value, most.value) < 0) {
                most = left;
            }
        if (right != null && comp.compare(right.value, most.value) < 0) {
                most = right;
        }
        if (most != node) {
                swapClosedTwoNodes(most, node);
        } else {
                break;
        }
        left = node.left;
        right = node.right;
        most = node;
    }
    if (node.parent == last) {
        last = node;
    }
    while (node.parent != null) {
        node = node.parent;
    }
    head = node;
}

    // 交换相邻的两个节点
    private void swapClosedTwoNodes(Node<K> node, Node<K> parent) {
            if (node == null || parent == null) {
                    return;
            }
            Node<K> parentParent = parent.parent;
            Node<K> parentLeft = parent.left;
            Node<K> parentRight = parent.right;
            Node<K> nodeLeft = node.left;
            Node<K> nodeRight = node.right;
            node.parent = parentParent;
            if (parentParent != null) {
                    if (parent == parentParent.left) {
                            parentParent.left = node;
                    } else {
```

```
                        parentParent.right = node;
                }
        }
        parent.parent = node;
        if (nodeLeft != null) {
                nodeLeft.parent = parent;
        }
        if (nodeRight != null) {
                nodeRight.parent = parent;
        }
        if (node == parent.left) {
                node.left = parent;
                node.right = parentRight;
                if (parentRight != null) {
                        parentRight.parent = node;
                }
        } else {
                node.left = parentLeft;
                node.right = parent;
                if (parentLeft != null) {
                        parentLeft.parent = node;
                }
        }
        parent.left = nodeLeft;
        parent.right = nodeRight;
}

// 在树中弹出堆尾节点后，找到原来的倒数第二个节点设置成新的队尾节点
private Node<K> popLastAndSetPreviousLast() {
        Node<K> node = last;
        Node<K> parent = node.parent;
        while (parent != null && node != parent.right) {
                node = parent;
                parent = node.parent;
        }
        if (parent == null) {
                node = last;
                parent = node.parent;
                node.parent = null;
                if (node == parent.left) {
                        parent.left = null;
                } else {
                        parent.right = null;
                }
                last = mostRight(head);
        } else {
                Node<K> newLast = mostRight(parent.left);
                node = last;
                parent = node.parent;
                node.parent = null;
                if (node == parent.left) {
                        parent.left = null;
```

```
                    } else {
                        parent.right = null;
                    }
                    last = newLast;
                }
                size--;
                return node;
            }

        }
```

随时找到数据流的中位数

【题目】

有一个源源不断地吐出整数的数据流，假设你有足够的空间来保存吐出的数。请设计一个名叫 MedianHolder 的结构，MedianHolder 可以随时取得之前吐出所有数的中位数。

【要求】

1．如果 MedianHolder 已经保存了吐出的 N 个数，那么任意时刻将一个新数加入到 MedianHolder 的过程，其时间复杂度是 $O(\log N)$。

2．取得已经吐出的 N 个数整体的中位数的过程，时间复杂度为 $O(1)$。

【难度】

将 ★★★★

【解答】

本书设计的 MedianHolder 中有两个堆，一个是大根堆，一个是小根堆。大根堆中含有接收的所有数中较小的一半，并且按大根堆的方式组织起来，那么这个堆的堆顶就是较小一半的数中最大的那个。小根堆中含有接收的所有数中较大的一半，并且按小根堆的方式组织起来，那么这个堆的堆顶就是较大一半的数中最小的那个。

例如，如果已经吐出的数为 6，1，3，0，9，8，7，2。

较小的一半为：0，1，2，3，那么 3 就是这一半的数组成的大根堆的堆顶。

较大的一半为：6，7，8，9，那么 6 就是这一半的数组成的小根堆的堆顶。

因为此时数的总个数为偶数，所以中位数就是两个堆顶相加，再除以 2。