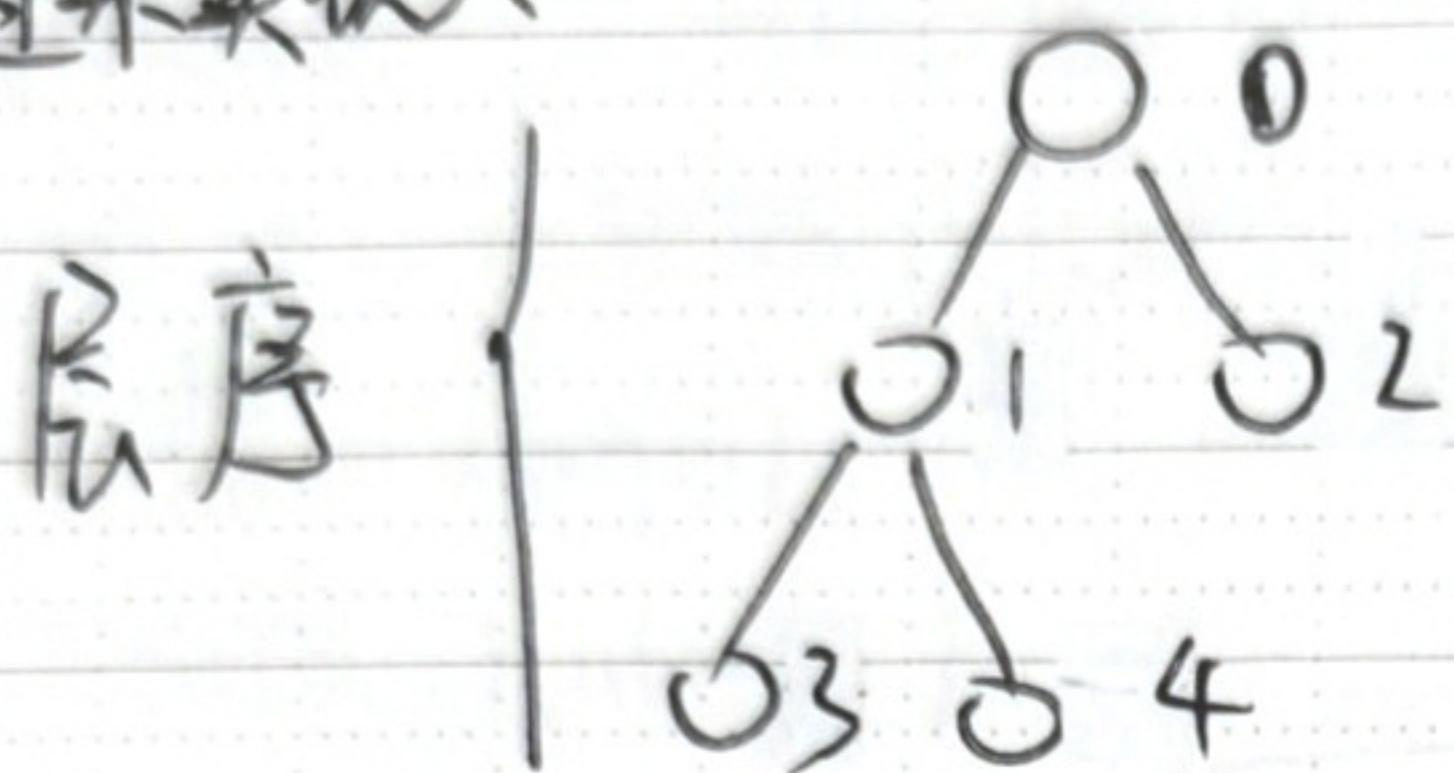


Heap:

二叉堆: Binary Heap 是一棵完全二叉树(但不一定足满二叉树)

堆中某个节点总是不大于其父节点的值。(最大堆)

- 可以通过定义有左结点和右结点来实现
- 也可以通过数组来实现:



对于任意一个结点 i 来说:

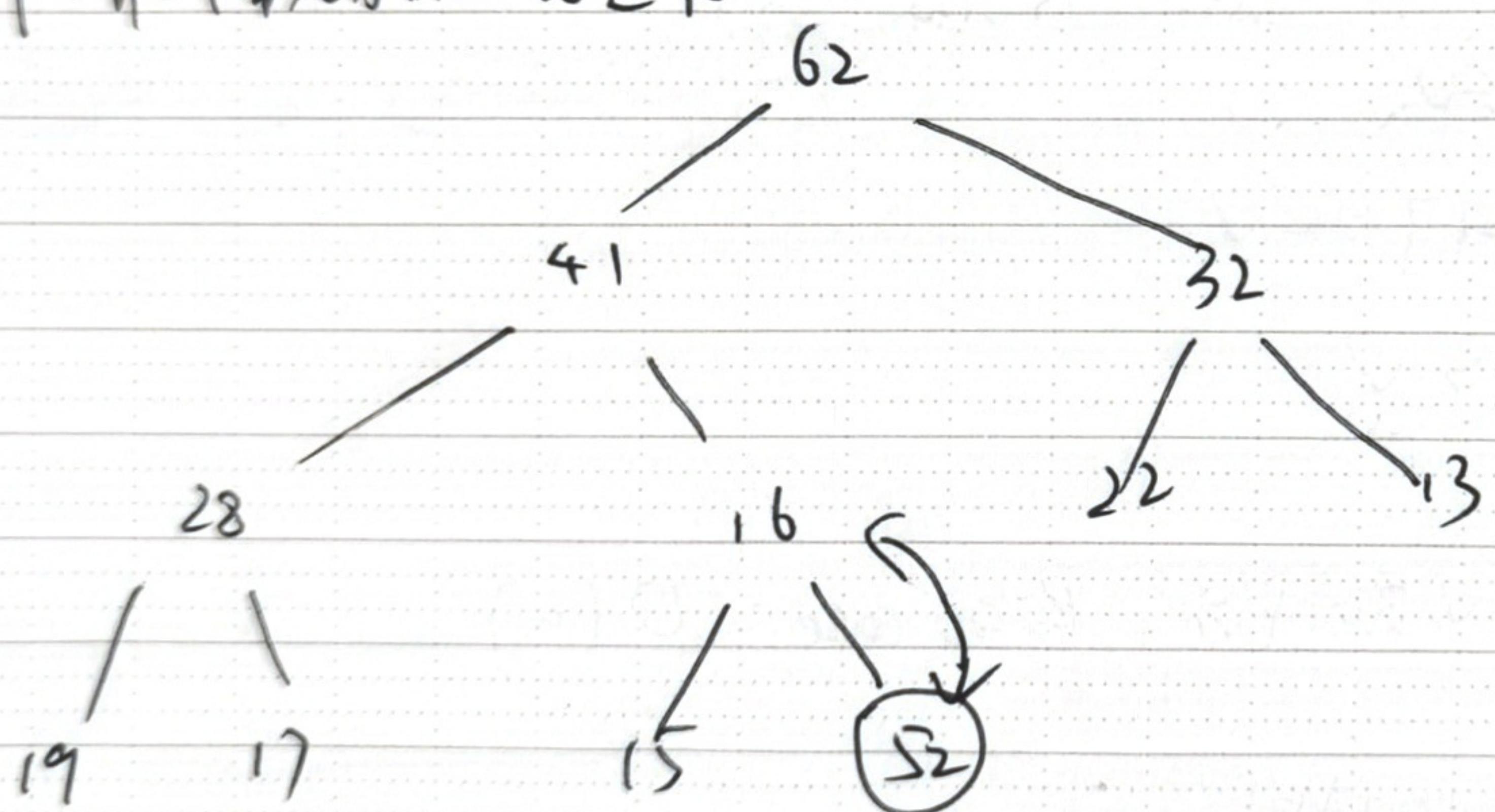
$$\text{parent}(i) = \frac{i-1}{2} \text{ (整数除法)}$$

$$\text{left child}(i) = 2 \cdot i + 1$$

$$\text{right child}(i) = 2 \cdot i + 2$$

完全二叉树一定不会退化成链表, 所以树高度

Add element: $O(\log n)$
Sift up: 堆中某元素的上浮过程:



```
private void siftUp(int k) {
```

```
    while (k > 0 & & data.get(parent(k)).compareTo(data.get(k)) < 0) {
```

```
        data.swap(k, parent(k));
```

```
        k = parent(k);
```

```
}
```

```
}
```

```
public void add(E e) {
```

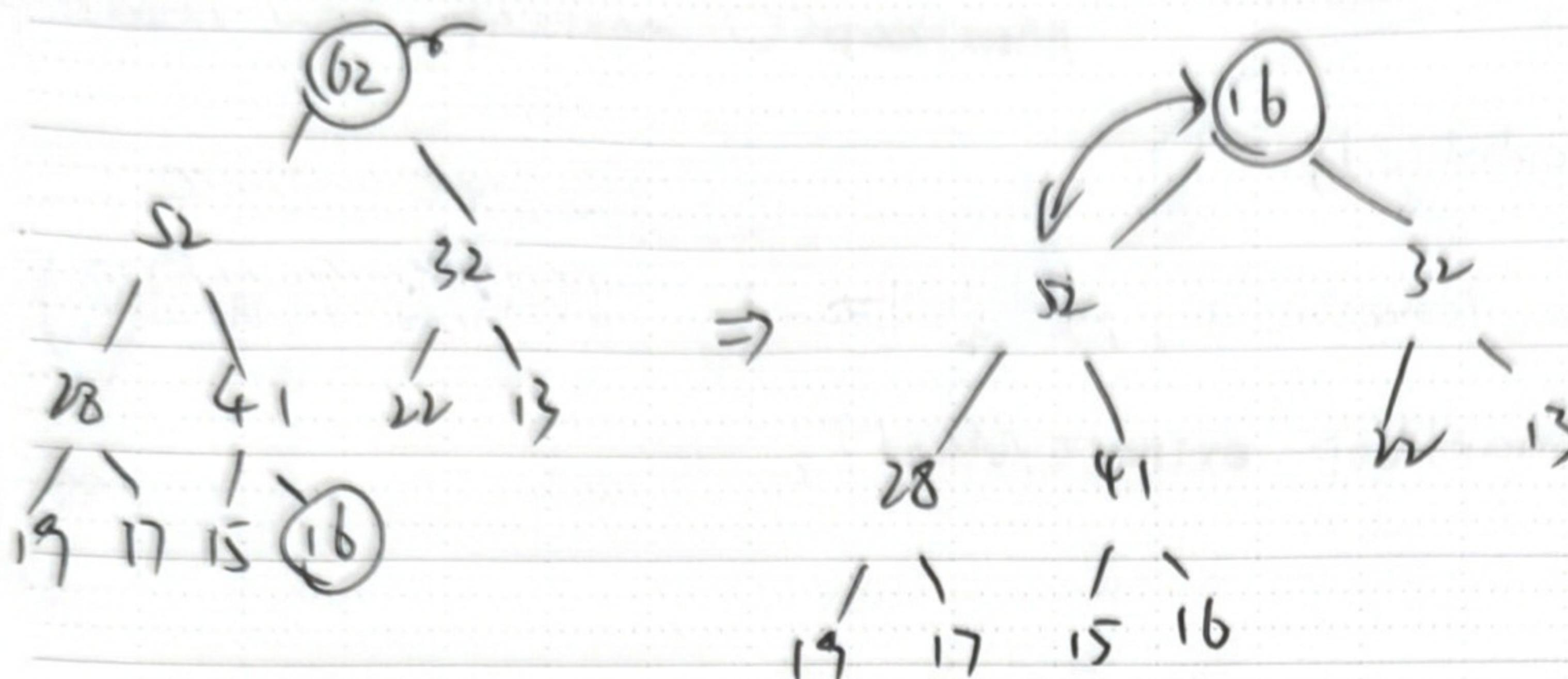
```
    data.addLast(e);
```

```
    siftUp(data.getSize() - 1);
```

Delete element: $O(\log n)$

Shift down:

`extractMax()`: 删除根节点，将最后一个节点放置堆顶，删除最后节点。



`public E extractMax()`

```
E ret = findMax();
data.swap(0, data.getSize() - 1);
data.removeLast();
shiftDown(0);
return ret;
```

↳

`private void shiftDown(int k)`

`while (leftChild(k) < data.getSize()) {`

`int j = leftChild(k);`

`if (j + 1 < data.getSize() && data.get(j + 1).compareTo(data.get(j)) >= 0)`

`j = rightChild(k);`

`data[j]` 是 `leftChild` 和 `rightChild` 中的最大值。

`if (data.get(k).compareTo(data.get(j)) >= 0)`

`break;`

`data.swap(k, j);`

`k = j;`

↳

}

Heap Sort:

- 最直观的 Heap Sort: 对于用户传来的数组 `data[]`:

```

TC: O(nlogn)   for (E e: data)   maxHeap > maxHeap = new MaxHeap()
                  maxHeap.add(e);      // 不是 Sort in place.

for (int i = data.length - 1; i >= 0; i--)   (需要额外的空间)
    data[i] = maxHeap.extractMax();
  
```

Replaceable Heap Sort:

`replace`: 取出最大元素后，放入一个新元素

实现: 可以先 `extractMax`, 再 `add`. \Rightarrow 次数 $O(\log n)$

实现: 可以直接将堆顶元素替换以后 `sift Down`. \Rightarrow 次数 $O(1 \log n)$

```

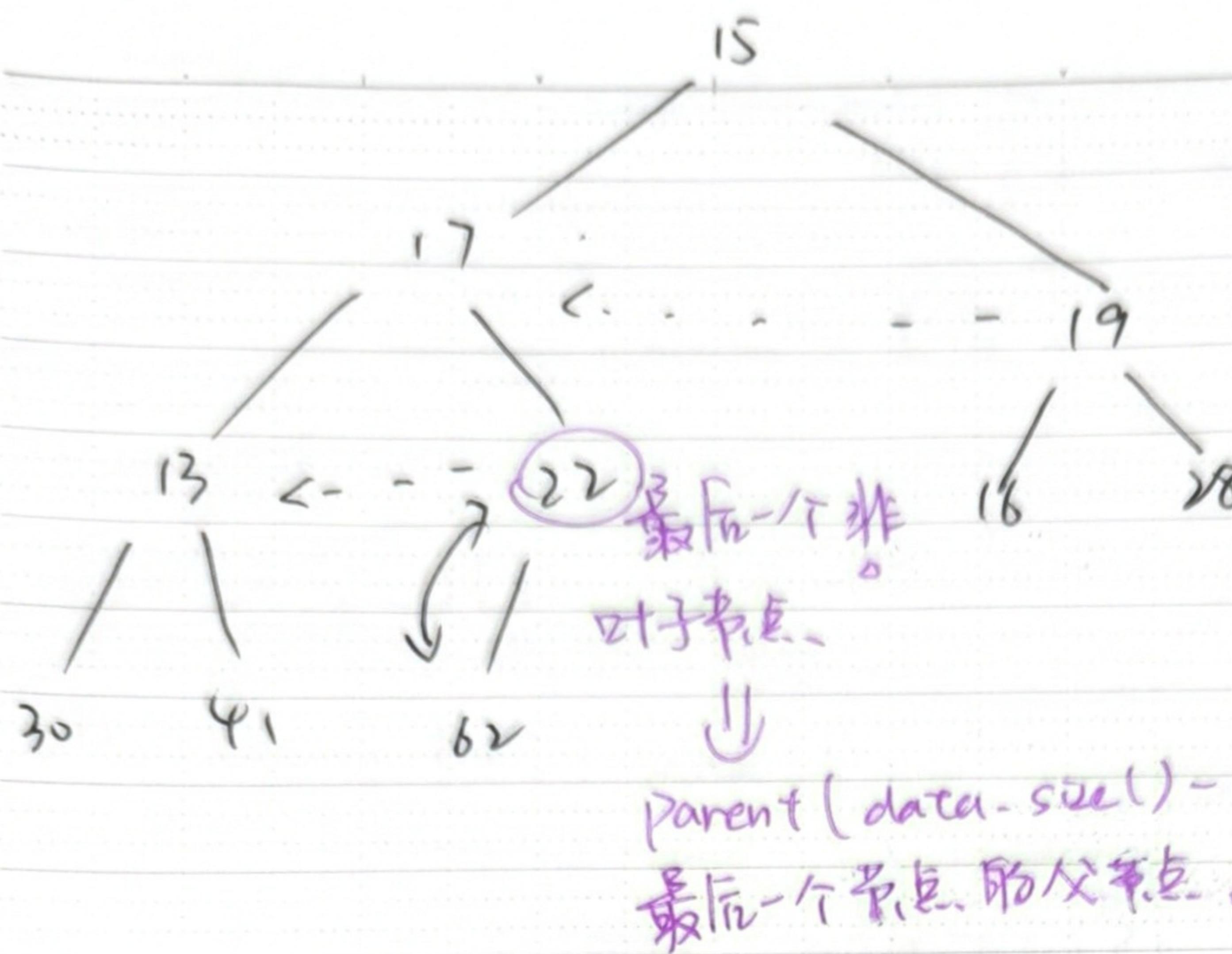
public E replace (E e) {
    E ret = findMax();
    data.set (0, e);
    siftDown (0);
    return ret;
}
  
```

`Heapify`: 将任意数组整理成堆的形状: 比用 `add()` 更快

对于任意数组, 从最后一个非叶子节点开始, 从后往前进行 `sift Down`.

TC: $O(n)$

(堆底见下页)



对于满二叉树：一共有 $1 + 2 + 4 + 8 + \dots + 2^{h-1}$ 节点。
 ↓
 第一层节点.

$$\frac{1(1-2^h)}{1-2} = 2^h - 1$$

非叶子节点: $2^{h-1} - 1$

叶子节点: 2^{h-1} , 大约是 $\frac{n}{2}$ 个

满二叉树最后一层有 $\frac{n}{2}$ 个节点.: heapify: $\frac{n}{2} \cdot 0$ (不需要 siftDown)

倒数第二层有 $\frac{n}{4}$ 个节点.: heapify: $\frac{n}{4} \cdot 1$ (最多进行一次 siftDown)

如果某一层下面有 h 层节点: 则这一层有 $\frac{n}{2^{h+1}}$ 个节点, , heapify: $\frac{n}{2^{h+1}} \cdot h$

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left[\frac{n}{2^{h+1}} \right] O(h) = O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\right)$$

(最多每个节点 swap h 次)

级数求和: $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$ for $|x| < 1$
 → 这该级数收敛。

Time: $\lceil x \rceil < 1$, 但收敛.

$$x = \frac{1}{2}, k = h$$

$$\Rightarrow \sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{\frac{1}{2}}{(1 - \frac{1}{2})^2} = 2$$

$$\therefore O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right)$$

$$= O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{1}{2^h}\right)$$

$$= O(n)$$

实现Heapify:

`public MaxHeap (E[] arr) {` 动机：数组

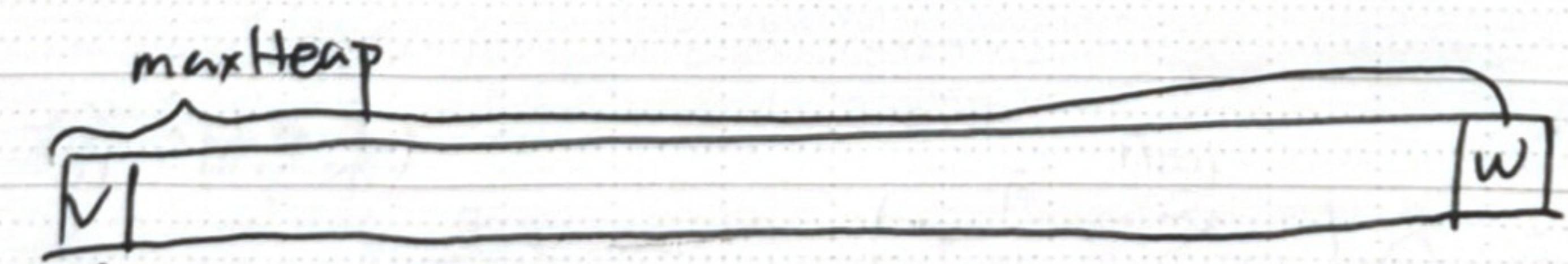
`data = new Array<T>(arr);`

`for (int i = parent(arr.length - 1); i >= 0; i--)
 sifeDown(i);`

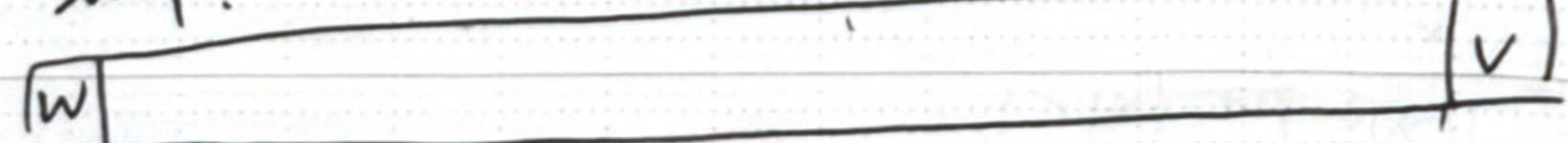
4

将堆排序优化为原地排序: 一个数组本身就可以表示一个堆.

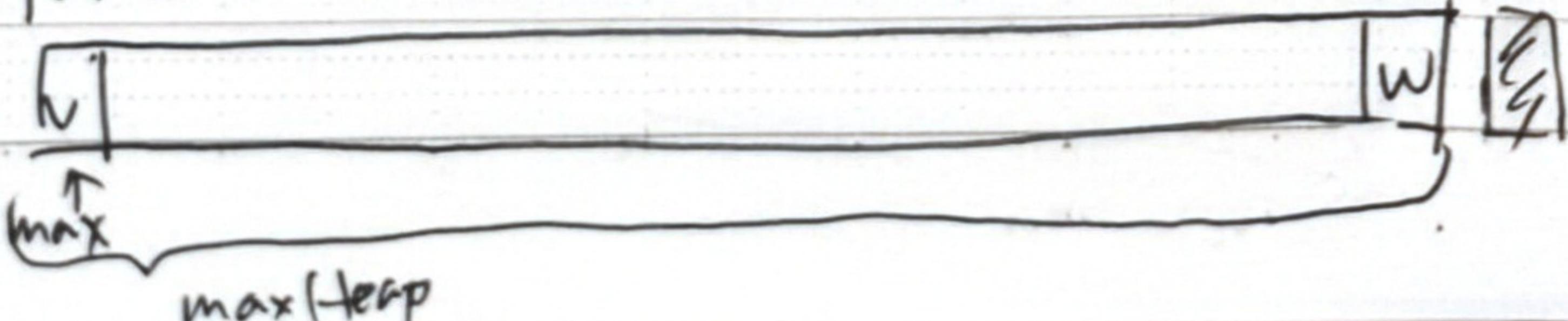
(Sort in place)



swap:



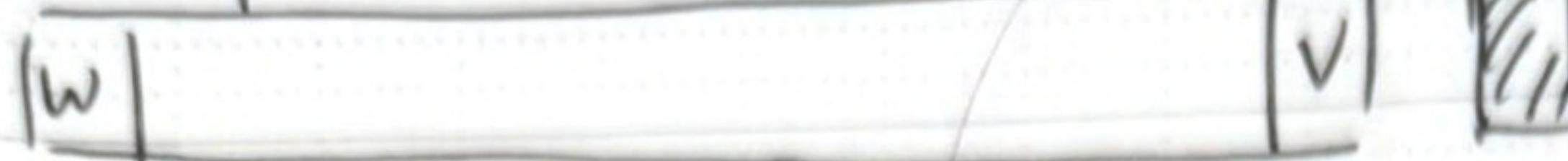
SifeDown:



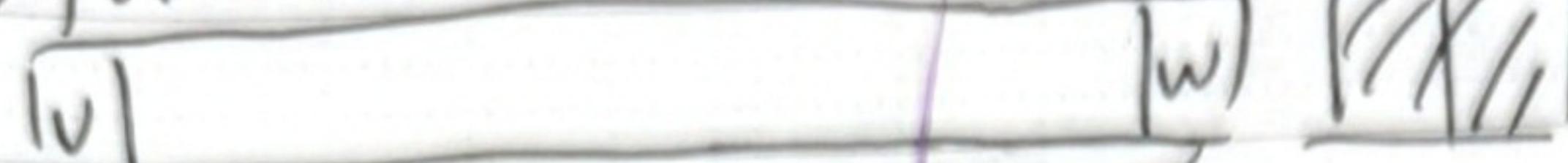
Date _____

数组是 maxHeap 的部分逐渐减小

Swap:



SiftDown:



maxHeap

空间复杂度: $O(1)$

在 HeapSort 中改写 SiftDown(): 不在 MaxHeap 基本中进行 SiftDown
过程,

HeapSort (E[] data) {

if (data.length <= 1) return;

for (int i = (data.length - 2) / 2; i >= 0; i--) {
 Heapify

 SiftDown (data, i, data.length);

 for (int i = data.length - 1; i >= 0; i--) {

 swap (data, 0, i);

 SiftDown (data, 0, i);

}

// 对 data[0, n) 所形成的最大堆中，索引 k 的元素，执行 SiftDown.

SiftDown (E[] data, int k, int n) {

 while (2 * k + 1 < n) {

 int j = 2 * k + 1;

 if (j + 1 < n && data[j + 1].compareTo (data[j]) > 0)

 j++;

 data[j] 是 left child 和 right child
 中的最大值。

 if (data[k].compareTo (data[j]) >= 0)
 break;

 swap (data, k, j);

 k = j;

}

在此轮
循环中，
data[j]
是 data
[k, j] 之
中最
大值。