堆结构（Heap）

王红元 coderwhy

目录
content

1 认识堆结构的特性

2 堆结构的设计封装

3 堆结构的插入方法

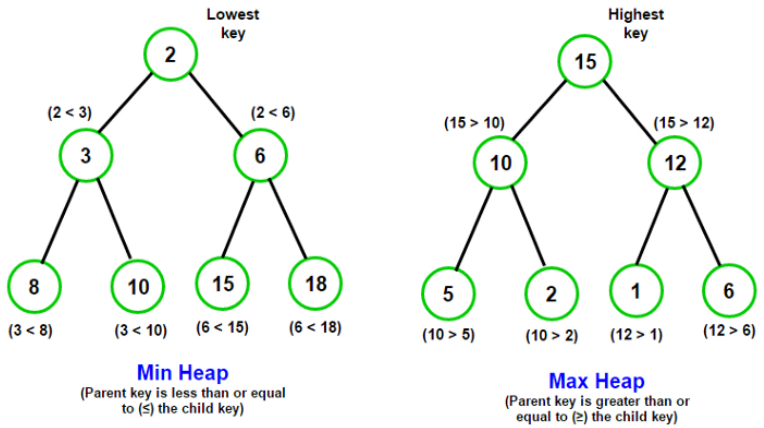
4 堆结构的删除方法

5 堆结构的其他方法

6 数组进行原地建堆

什么是堆（Heap）结构?

- 堆是也是一种非常常见的数据结构，但是相对于前面的数据结构来说，要稍微难理解一点。
- 堆的本质是一种特殊的树形数据结构，使用完全二叉树来实现：
  - 堆可以进行很多分类，但是平时使用的基本都是二叉堆；
  - 二叉堆又可以划分为最大堆和最小堆；
- 最大堆和最小堆：
  - 最小堆：堆中每一个节点都小于等于（<=）它的子节点；
  - 最大堆：堆中每一个节点都大于等于（>=）它的子节点；

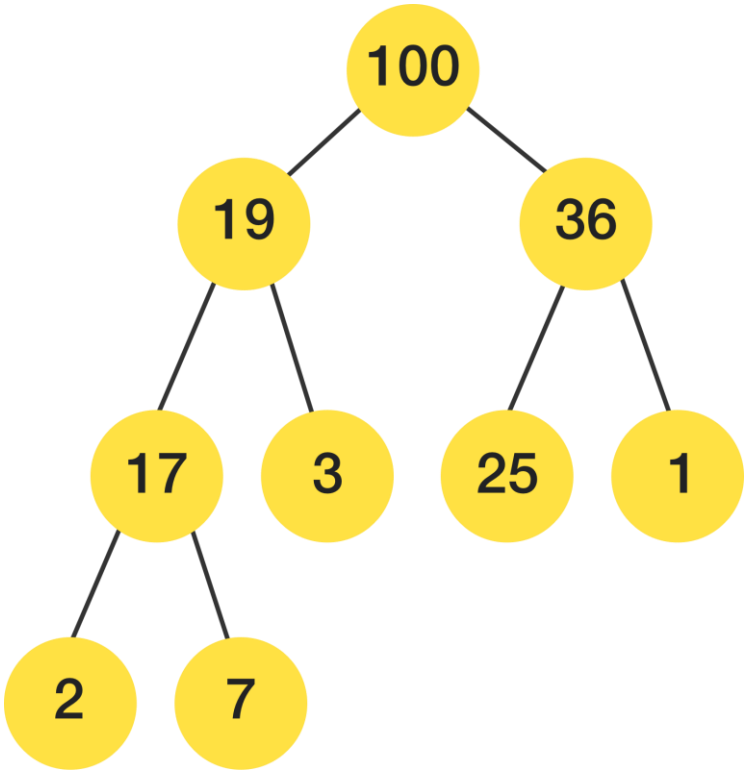


为什么需要堆（Heap）结构?

- **但是这个堆东西有什么意义呢?**
  - 对于每一个新的数据结构，我们都需要搞清楚为什么需要它，这是我们能够记住并且把握它的关键。
  - 它到底帮助我们解决了什么问题?
- **如果有一个集合，我们希望获取其中的最大值或者最小值，有哪些方案呢?**
  - 数组/链表：获取最大或最小值是O(n)级别的；
    - ✓ 可以进行排序，但是我们只是获取最大值或者最小值而已
    - ✓ 排序本身就会消耗性能；
  - 哈希表：不需要考虑了；
  - 二叉搜索树：获取最大或最小值是O(logn)级别的；
    - ✓ 但是二叉搜索树操作较为复杂，并且还要维护树的平衡时才是O(logn)级别；
- **这个时候需要一种数据结构来解决这个问题，就是堆结构。**

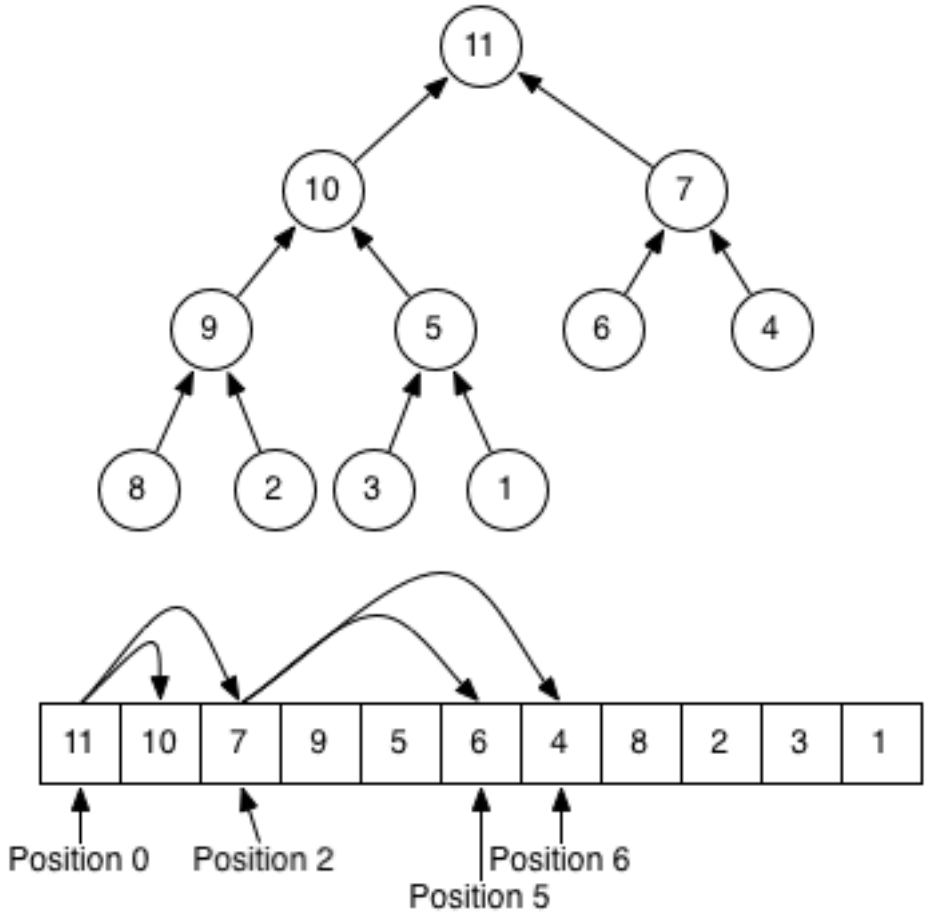
认识堆（Heap）结构

- **堆结构通常是用来解决Top K问题的：**
  - Top K问题是指在一组数据中，找出最前面的K个最大/最小的元素；
  - 常用的解决方案有使用排序算法、快速选择算法、堆结构等；
- **但是我们还是不知道具体长什么样子，以及它是如何实现出来的：**
  - 二叉堆用树形结构表示出来是一颗完全二叉树；
  - 通常在实现的时候我们底层会使用数组来实现；
- **每个节点在数组中对应的索引i（index）有如下的规律：**
  - 如果 i = 0 ，它是根节点；
  - 父节点的公式：floor((i – 1) / 2)
  - 左子节点：2i + 1
  - 右子节点：2i + 2



0	1	2	3	4	5	6	7	8	9
100	19	36	17	3	25	1	2	7	

堆结构的性质


11
10
7
9
5
6
4
8
2
3
1
11 10 7 9 5 6 4 8 2 3 1
Position 0
Position 2
Position 5
Position 6

堆结构的设计

- **接下来，让我们对堆结构进行设计，看看需要有哪些属性和方法。**
- **常见的属性：**
  - data：存储堆中的元素，通常使用数组来实现。
  - size：堆中当前元素的数量。
- **常见的方法：**
  - insert(value)：在堆中插入一个新元素。
  - extract/delete()：从堆中删除最大/最小元素。
  - peek()：返回堆中的最大/最小元素。
  - isEmpty()：判断堆是否为空。
  - build_heap(list)：通过一个列表来构造堆。
- **那么接下来我们就来实现这个堆结构吧！**

堆结构的封装

- 封装Heap的类：

```
class Heap<T> {
  private data: T[] = []
  private size: number = 0
}
```

- 这个堆结构里面只包含了两个属性：data和size
  - data是一个泛型数组，存储堆中的元素；
  - size是当前堆中元素的数量。

```
private swap(i: number, j: number) {
  const temp = this.data[i]
  this.data[i] = this.data[j]
  this.data[j] = temp
}
```

insert插入元素

- **如果你想实现一个最大堆，那么可以从实现“insert”方法开始。**
  - 因为每次插入元素后，需要对堆进行重构，以维护最大堆的性质。
  - 这种策略叫做**上滤**（percolate up， percolate [ˈpɜːkəleɪt] 是过滤的意思）。

```
/** 插入方法 */
insert(value: T) {
  this.data.push(value)
  this.size++
  this.heapify_up()
}
```

```
private heapify_up() {
  let index = this.size - 1
  while (index > 0) {
    let parentIndex = Math.floor((index - 1) / 2)
    if (this.data[index] <= this.data[parentIndex]) {
      break
    }
    this.swap(index, parentIndex)
    index = parentIndex
  }
}
```

数据模拟

```
[19, 100, 36, 17, 3, 25, 1, 2, 7]
```

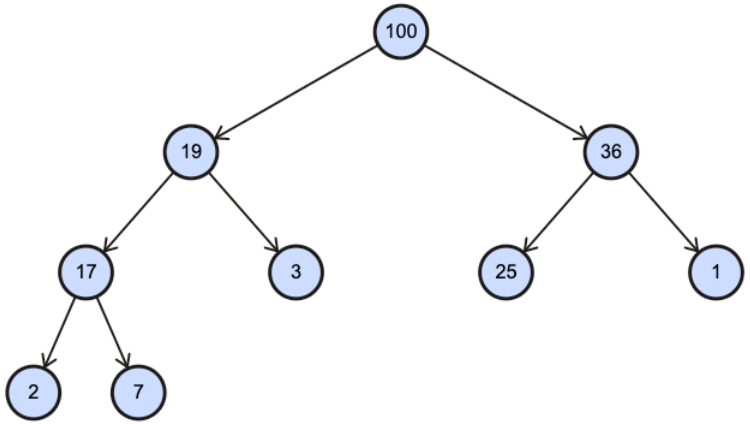
```
const heap = new Heap<number>()
for (const item of arr) {
  heap.insert(item)
}
heap.traverse()
```

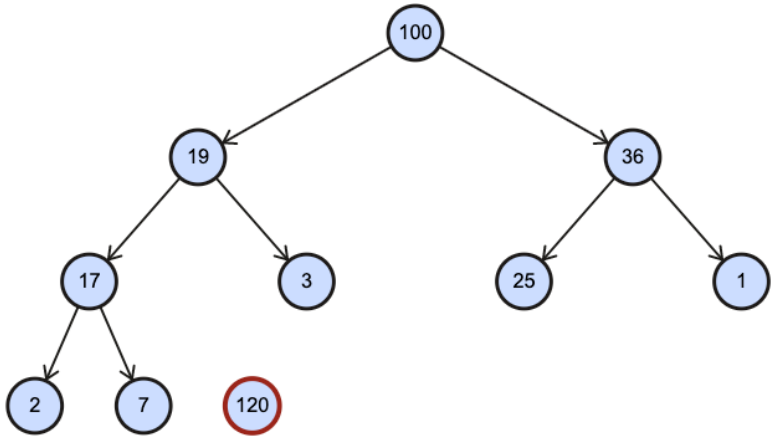
数据结构可视化

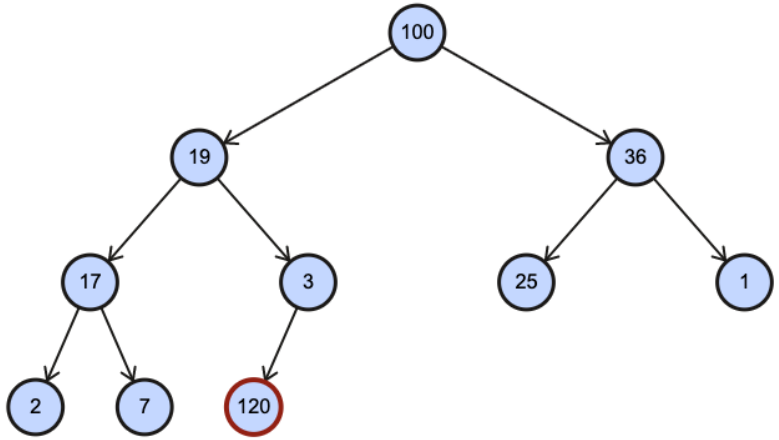
- **在线数据结构演练：**
  - https://www.cs.usfca.edu/~galles/visualization/Algorithms.html（加利福尼亚州的旧金山大学）
  - https://visualgo.net/en/heap?slide=1
  - http://btv.melezinek.cz/binary-heap.html

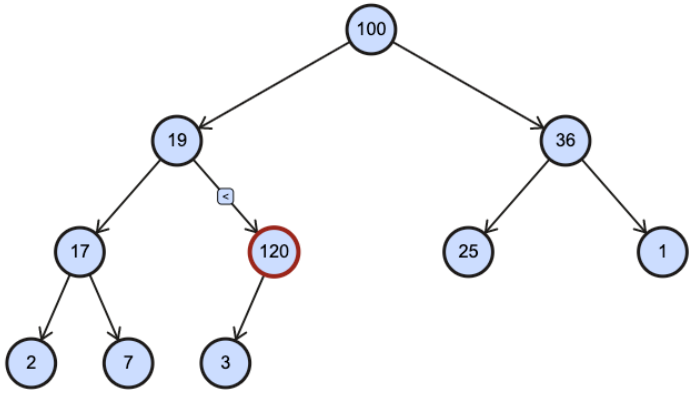
插入元素 insert

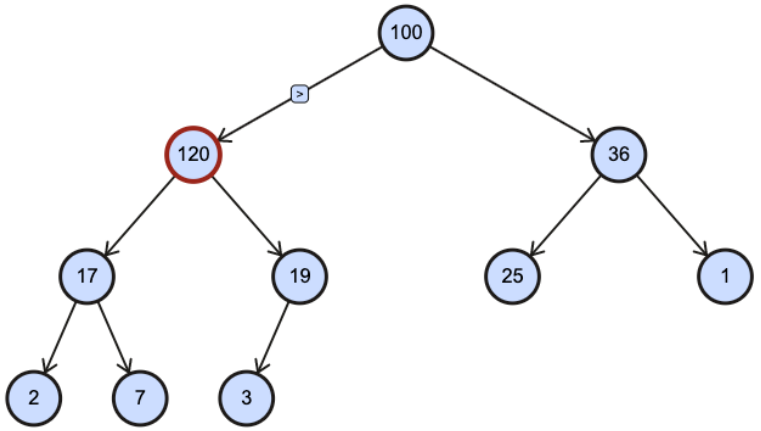
- 如果我们现在有这样一个结构的最大堆：插入120

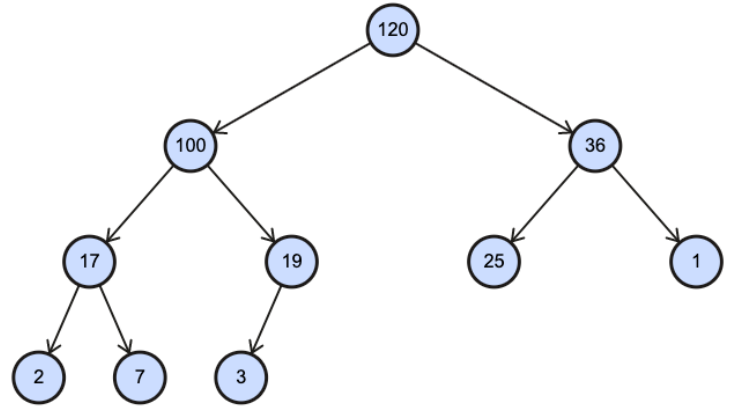












delete删除元素

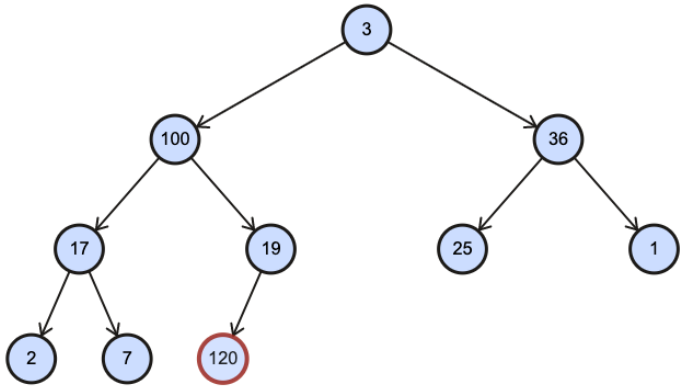
- **删除操作也需要考虑在删除元素后的操作：**
  - 因为每次删除元素后，需要对堆进行重构，以维护最大堆的性质。
  - 这种向下替换元素的策略叫作**下滤**（percolate down）。

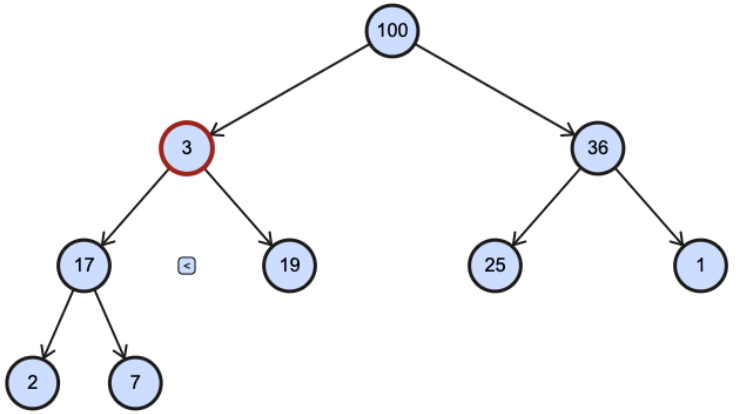
```ts
/** 提取操作 */
delete(): T | null {
  // 1.没有元素或者一个元素，直接返回null
  if (this.size === 0) return null
  if (this.size === 1) {
    this.size--
    return this.data.pop()!
  }

  // 2.获取到要删除的元素
  const max = this.data[0]
  // 将最后一个位置的元素，放到第一个为止
  this.data[0] = this.data.pop()!
  this.size--

  return max
}
```

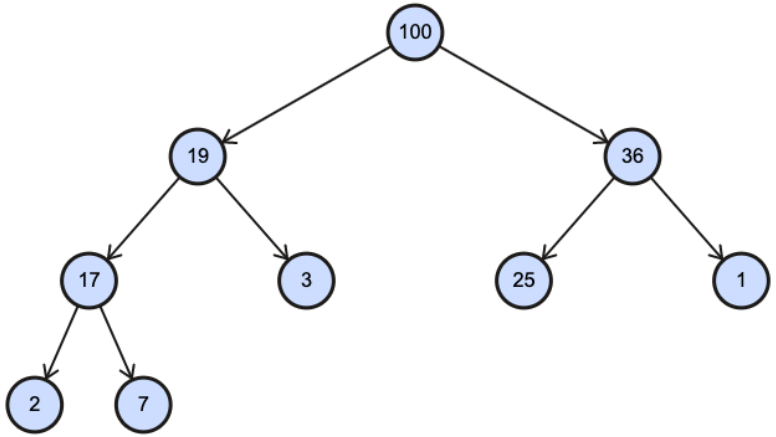
```ts
private heapify_down() {
  let index = 0
  while (2 * index + 1 < this.size) {
    let leftChildIndex = 2 * index + 1
    let rightChildIndex = 2 * index + 2
    // 找到较大的索引
    let largerIndex = leftChildIndex
    if (rightChildIndex < this.size && this.data[rightChildIndex] > this.data[leftChildIndex]) {
      largerIndex = rightChildIndex
    }

    if (this.data[index] >= this.data[largerIndex]) {
      break
    }
    this.swap(index, largerIndex)
    index = largerIndex
  }
}
```

提取操作 extract、delete

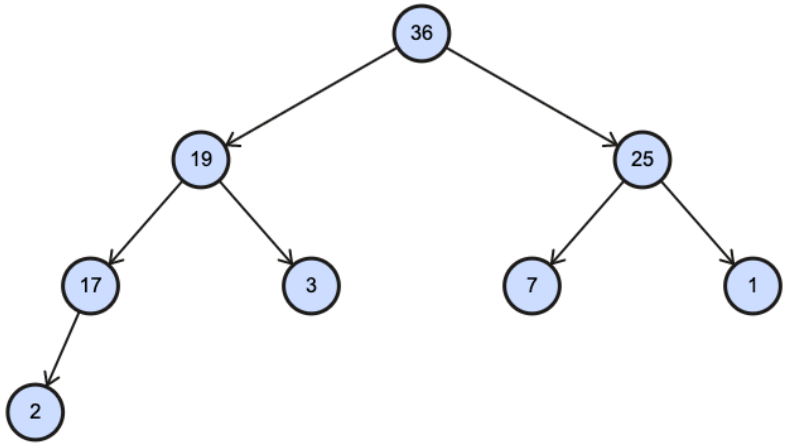
删除120的过程







删除100的结果

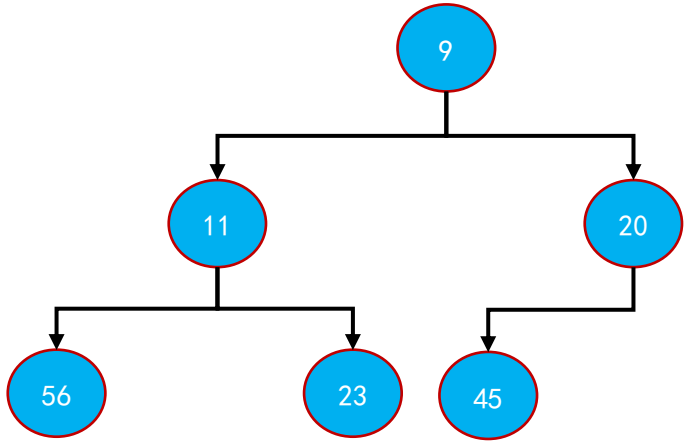


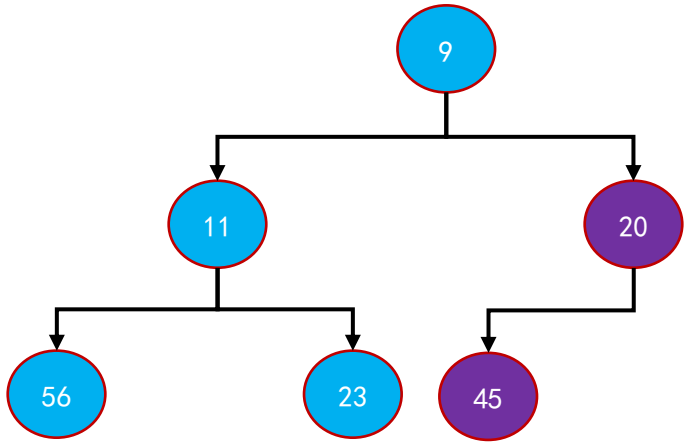
其他方法的实现

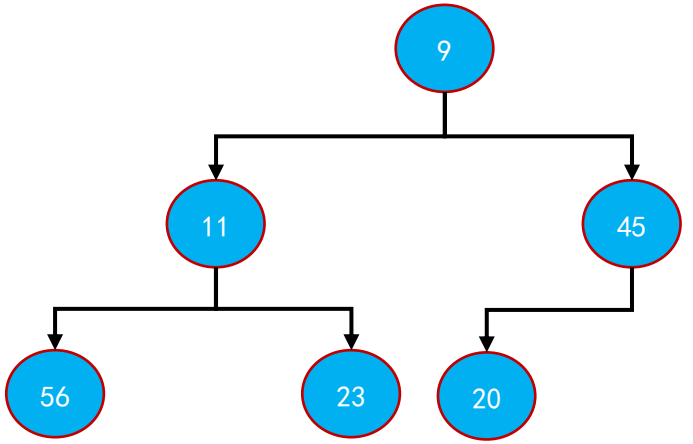
```
// 其他方法
peek(): T | null {
  return this.data[0] ?? null
}

isEmpty(): boolean {
  return this.data.length === 0
}

length(): number {
  return this.size
}
```

原地建堆

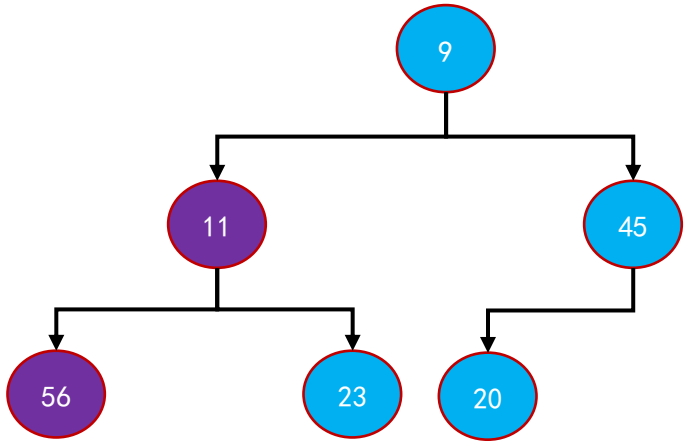
■ “原地建堆”（In-place heap construction.）是指建立堆的过程中，不使用额外的内存空间，直接在原有数组上进行操作。

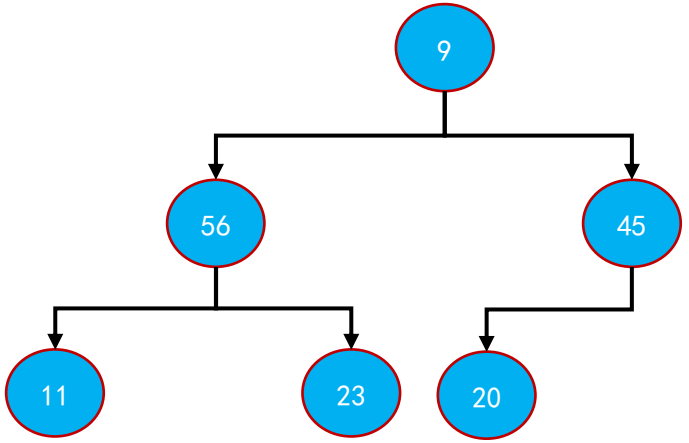




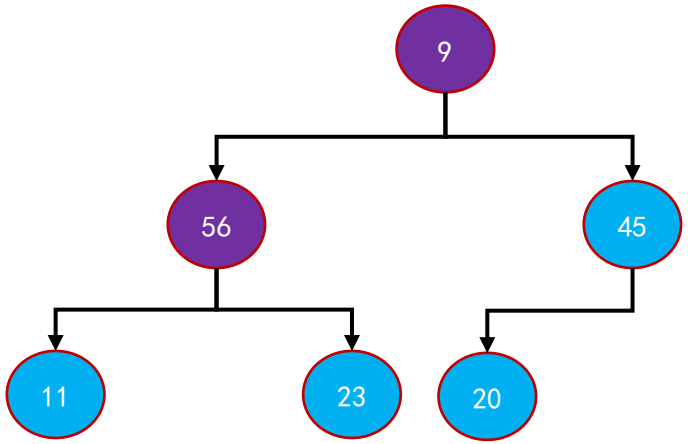


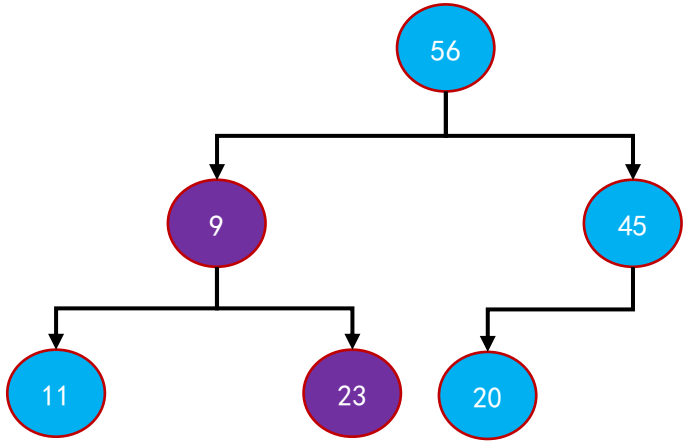

```
[9, 11, 20, 56, 23, 45]
```

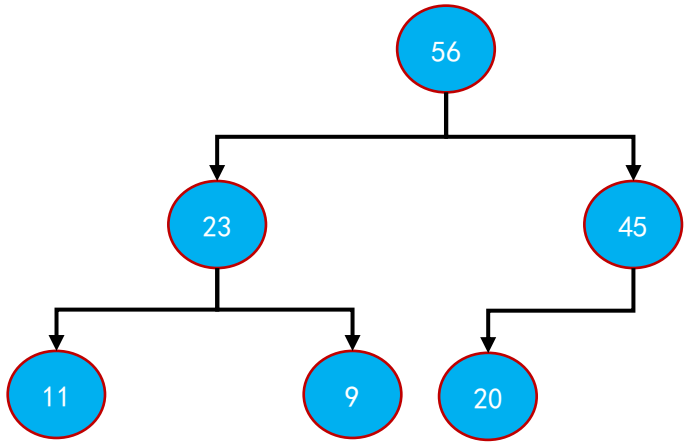




原地建堆








```
build_heap(arr: T[]) {
  this.data = arr
  this.size = arr.length
  let start = Math.floor((this.size - 1) / 2)
  for (let i = start; i >= 0; i--) {
    this.heapify_down(i)
  }
}
```

```
const arr = [9, 11, 20, 56, 23, 45]
const heap = new Heap<number>()
heap.build_heap(arr)
console.log(arr)
```

```
[ 56, 23, 45, 11, 9, 20 ]
```

这种原地建堆的方式，我们称之为自下而上的下滤。
也可以使用自上而下的上滤，但是效率较低，作为课下自行研究。