Courses Practice Roadmap Pro

Algorithms and Data Structures for Beginners

25 / 35

25 - Heapify

14:34

Mark Lesson Complete

View Code

Prev Next

Suggested Problems

Status	Star	Problem	Difficulty	Video Solution	Code
		Last Stone Weight	Easy		C++
		K Closest Points to Origin	Medium		C++
		Kth Largest Element In An Array	Medium		C++

Heapify

Recall that to build a binary search tree, the time complexity is $O(n\ log\ n)$. We could build our heap the same way and those operations would also run in $O(log\ n)$ time. Heapify tells us there is a better way of doing it. It allows us to perform this operation in $O(n)$ time.

Concept

The idea behind using heapify to build a heap is to satisfy the structure and the order property. We need to make sure that our binary heap is a complete binary tree and that every node's value is at most its parent's value.

Because the leaf nodes can't violate the min-heap properties, there is no need to perform `heapify()` on them.

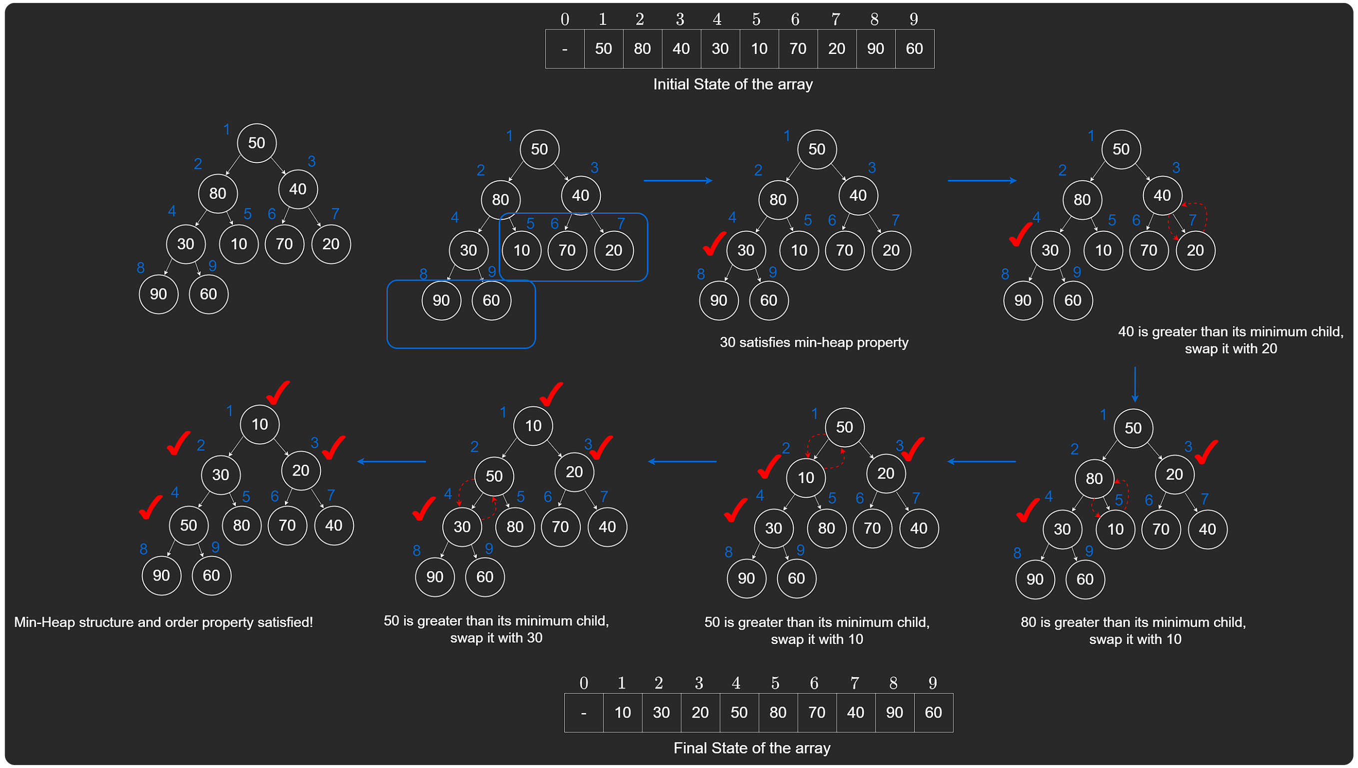
Since we are skipping all of the leaf nodes, we only need to start at `heap.length // 2`. Then, we need to percolate down the exact same way we did in the previous chapter in the `pop()` method. We will not be going over the code in detail as majority of it is the same as the `pop()` method.

```
fn heapify(arr):
    // 0-th position is moved to the end
    arr.append(arr[0])

    heap = arr
    cur = (heap.length - 1) // 2
    while cur > 0:

        i = cur
        while 2 * i < heap.length:
            if (2 * i + 1 < heap.length and
            heap[2 * i + 1] < heap[2 * i] and
            heap[i] > heap[2 * i + 1]):
                # Swap right child
```

```
                tmp = heap[i]
                heap[i] = heap[2 * i + 1]
                heap[2 * i + 1] = tmp
                i = 2 * i + 1
            elif heap[i] > heap[2 * i]:
                // Swap left child
                tmp = heap[i]
                heap[i] = heap[2 * i]
                heap[2 * i] = tmp
                i = 2 * i
            else:
                break
        cur -= 1
```

Starting from the first non-leaf node, we will percolate down, the exact same way we did in the `pop()` function. After each iteration, we are going to decrement the index by 1 so we can perform `heapify()` on the next node, all the way until index 1.

The visual below demonstrates `heapify()` being performed on all nodes starting from index 4. The nodes in the blue rectangles are leaf nodes.



Time Complexity

Given that there are n nodes in a binary tree, there are roughly n / 2 leaf nodes. Using this information, we can figure out how many levels each node has to percolate down and the amount of work `heapify()` performs at each level.

We don't perform heapify at the 3rd / last level. The nodes on the 2nd level need to percolate down one level, and the nodes on the 1st level are percolating down two levels, with the root node having to percolate down all the levels. So while the number of nodes is halving each time, the number of levels needed to be percolated increases. There is a very neat mathematical summation that is formed when we add all the work together, which simplifies to $O(n)$, but we will not be covering that. It is highly unlikely that you will be asked to prove the time complexity of `heapify()` , so it is enough to know that it is in $O(n)$

If you are interested in learning the proof behind why there are roughly $n\ /\ 2$ leaf nodes, these 5 slides from **Virginia Tech** are valuable.

Closing Notes

BST problems are common but when it comes to using a data structure as a utility, they are much more common with a heap. If you continously need to find the maximum or the minimum value in a problem, using a min or a max-heap are great options.

> *Sometimes, the problem asks to find the "Top K" elements with some criteria. These questions are built to be solved with heaps.*

Contact: neetcodebusiness@gmail.com

Github Privacy Terms