



Courses

Practice

Roadmap

Pro



# Algorithms and Data Structures for Beginners

24 / 35

## 24 - Push and Pop

17:43

☐

Mark Lesson Complete



View Code

Prev

Next

## 27 Hash Implementation

## Suggested Problems

Status	Star	Problem ↕	Difficulty ↕	Video Solution	Code
<input type="checkbox"/>	★	<b>Kth Largest Element In a Stream</b>	Easy		

## Heaps Push and Pop

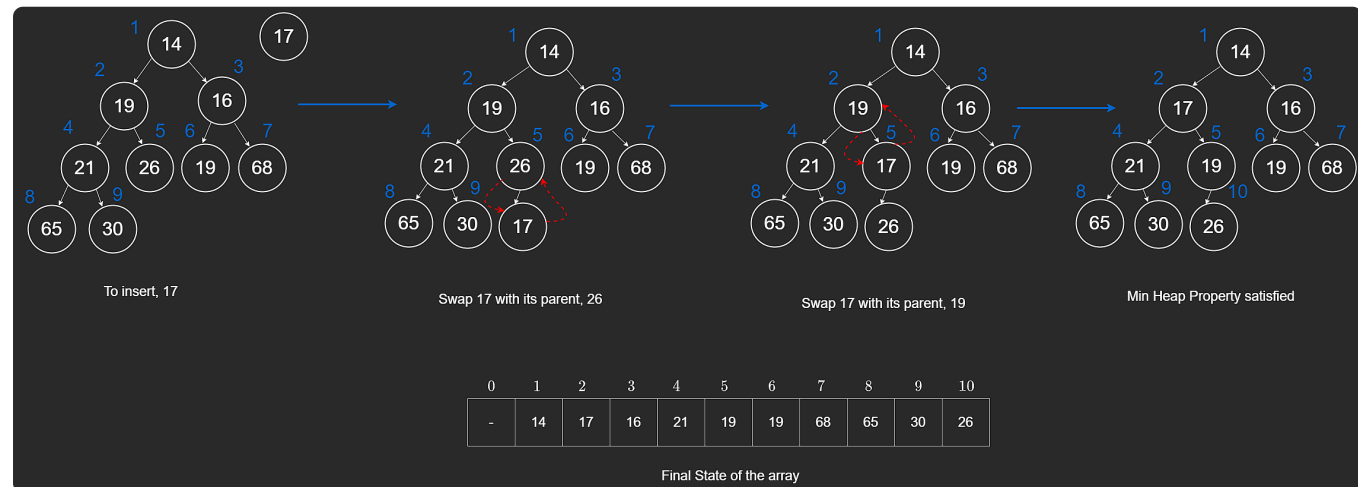
### Push

Taking the same binary heap from before:

`[14,19,16,21,26,19,68,65,30,null,null,null,null,null]` , let's say we wish to push `17` . We need to make sure we push `17` such that we maintain our structure and order property.

Since a binary heap is a complete binary tree, and we are required to fill nodes in a contiguous fashion, pushing `17` should happen at the 10th index. However, this might violate the min heap property, which means we will have to percolate `17` up the tree until we find its correct position.

In this case, because **17** is greater than its parent, **26**, it needs to percolate up until it is no longer greater than its parent. So, we swap **17** with **26** and now **17**'s parent is **19**, which again violates the min-heap property. We perform another swap and now **17** is greater than its parent, which is **14**. **17** is also smaller than all of its descendants because **19** was already smaller than all of its descendants. The resulting min-heap would look like the following.



```
fn push(val):
    heap.add(val)
    i = heap.length - 1

    // As long as we are at a valid index and the current node
    // is smaller than its parent, swap it with its parent and
    // calculate the index of the new parent
    while i > 1 and heap[i] < heap[i // 2]:
        tmp = heap[i]
```

```
heap[i] = heap[i // 2]
heap[i // 2] = tmp
i = i // 2
```

*The "/" indicates taking the floor of the resulting answer so we can round down, e.g. 5.5 becomes 5 since indices are whole numbers.*

Since we know the tree will always be balanced, the time complexity of the push operation is  $O(\log n)$ .

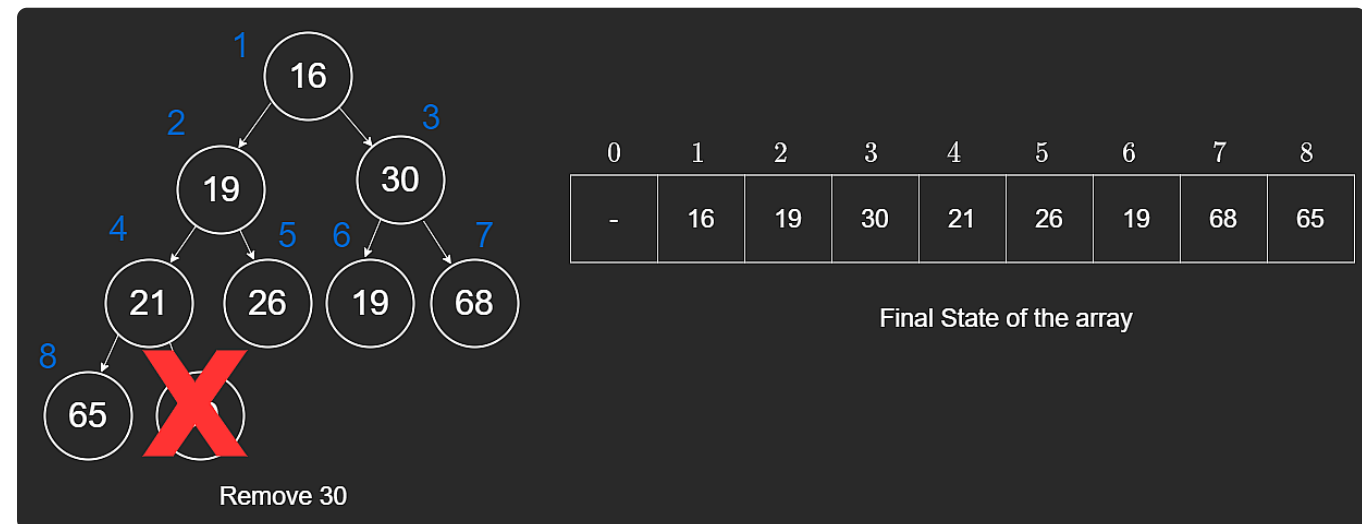
## Pop

### The obvious way

Popping from a heap is more complicated than the push operation. One way that you might have already thought about is pop the root node and replace it with `min(left_child, right_child)`. The issue here is that while the order property is intact, we have violated the structure property. Taking the tree from before, popping `14`, and swapping it with `16` - `min(left_child, right_child)` would require `19` to replace `16`. Now, level 2 has a missing node i.e `19` is missing a left child.

### The correct way

The correct way is to take the right-most node of the last level and swap it with the root node. We have now maintained the structure property. However, the order property is violated. To fix the order property, we have to make sure that **30** finds its place. To do so, we will run a loop and swap **30** with `min(left_child, right_child)`. We swap **30** with **16**, then **19** with **30**. The resulting tree will look like the following.



```
// "heap" is a global variable
fn pop():
    if heap.length == 1:
        return null
    if heap.length == 2:
        return heap.pop()

    res = heap[1]
```

```
// Move last value to root
heap[1] = heap.pop()
i = 1
// As long as we have at least a left child
while 2 * i < heap.length:
    // If we have a right child and right child is the minimum
    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
    // If we have a left child and left child is the minimum
    else if heap[i] > heap[2 * i]:
        // Swap left child
        tmp = heap[i]
        heap[i] = heap[2 * i]
        heap[2 * i] = tmp
        i = 2 * i
    // Otherwise the node already satisfies both the properties
    // and get out of the for loop
    else:
        break
return res
```

The pseudocode shown above might seem daunting at first so let's go over it. If our `heap` is empty, there is nothing to pop, hence the `return null`. Our heap also could have just one node, in which case, we will just pop that node and don't need to make any adjustments. If the above two statements have not executed, it must be the case that we have children, meaning we need to perform a swap.

We store our `14` into a variable called `res` so that we don't lose it. Then, we can pop from our heap, and replace `30` to be at the root node.

Our while loop runs as long as we have a left child and we determine this by making sure `2 * i` is not out of bounds. Then, there are three cases we concern ourselves with:

1. The node has no children
2. The node **only** has a left child
3. The node has two children

*When considering a binary heap, it is not possible to have only a right child because then it no longer is a complete binary tree and violates the structure property.*

Because we are guaranteed to have a left child in the while loop, we need to now check if the node also has a right child, which we check by `2 * i + 1`. We also

make sure that the current node is greater than its children because of the order property. We replace the node with the minimum of its two children.

If no right child exists and the current node's value is greater than its left child, we swap it with the left child.

If none of the above cases execute, then it must be the case that our node is in the proper position already, satisfying both the order and the structural property.

## Closing Notes

The time complexity of the operations discussed so far can be summarized by the following table.

Operation	Big-O Time
Get Min/Max	$O(1)$
Push	$O(\log n)$
Pop	$O(\log n)$





Copyright © 2023 NeetCode.io All rights reserved.  
Contact: [neetcodebusiness@gmail.com](mailto:neetcodebusiness@gmail.com)

[Github](#) [Privacy](#) [Terms](#)