Algorithms and Data Structures for Beginners

24 / 35

23 - Heap Properties

Mark Lesson Complete View Code Prev Next

Heap Properties

A heap is a specialized, tree-based data structure, which is a complete binary tree. It implements an abstract data type called the Priority Queue, but sometimes 'Heap' and 'Priority Queue' are used interchangeably.

We already learned that queues operate with a first-in-first-out basis but with a priority queue, the values are removed based on a given priority. The element with the highest priority is removed first.

Two types of heaps

1. Min Heap
2. Max Heap

Min heaps have the smallest value at the root node and when deleting, the smallest value has the highest priority.

Max heaps have the largest value at the root node and when deleting, the largest value has the highest priority.

In this chapter, we will be focusing on min heaps, but the implementation is exactly the same for max heap, except you would prioritize the maximum value instead of the minimum.

Heap Properties

For a binary tree to qualify as a heap, it must satisfy the following properties:

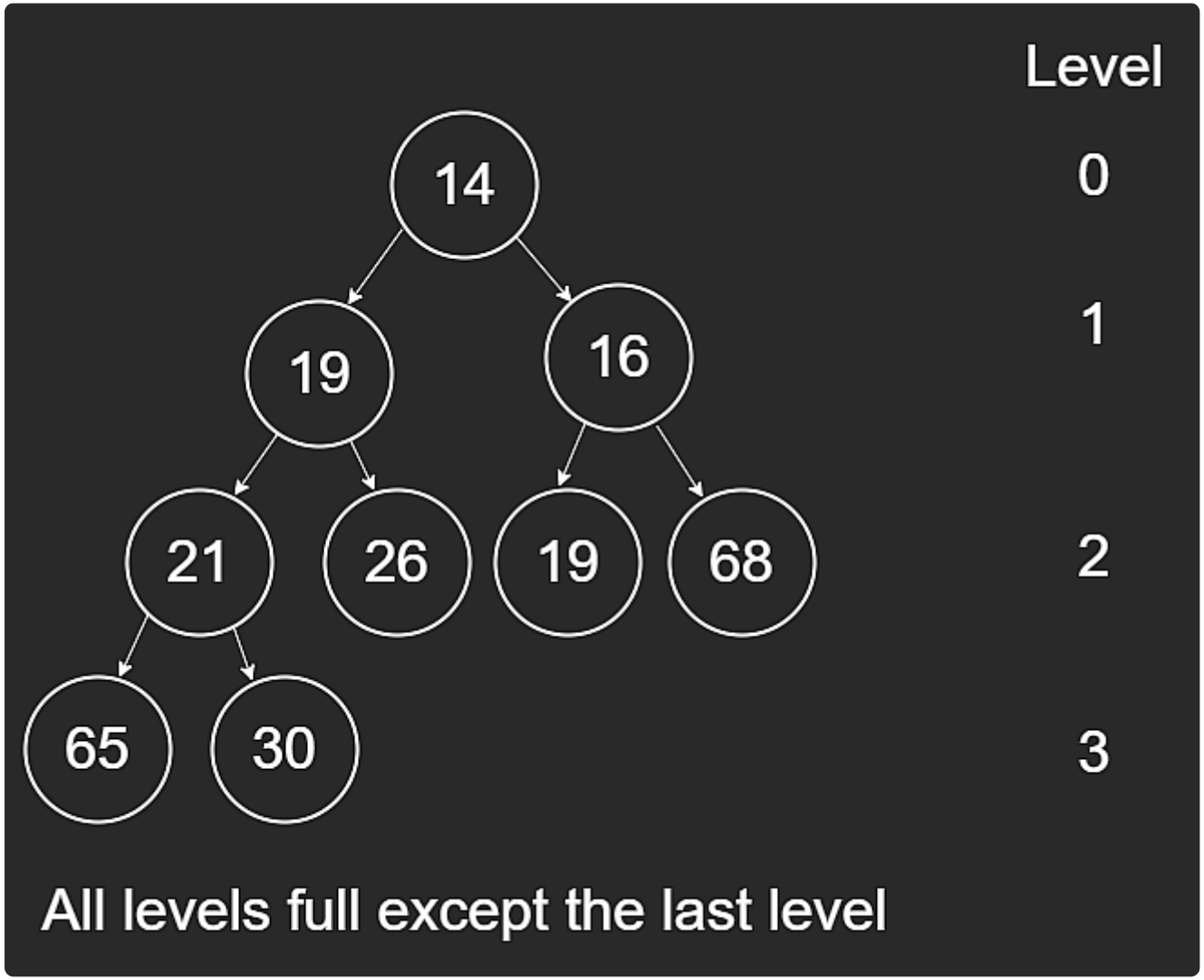
1. Structure Property A binary heap is a binary tree that is a **complete binary tree**, where every single level of the tree is filled completely, except the lowest level nodes, which are filled contiguously from left to right.

2. Order Property

The order property for a min-heap is that all of the descendents should be greater than their ancestor. In other words, if we have a tree rooted at `y`, every node in the right and the left sub-tree should be greater than or equal to `y`. This is a recursive property, similar to binary search trees.

In a max-heap, every node in the right and the left sub-tree is smaller than or equal to `y`.

The following visual shows a binary heap.



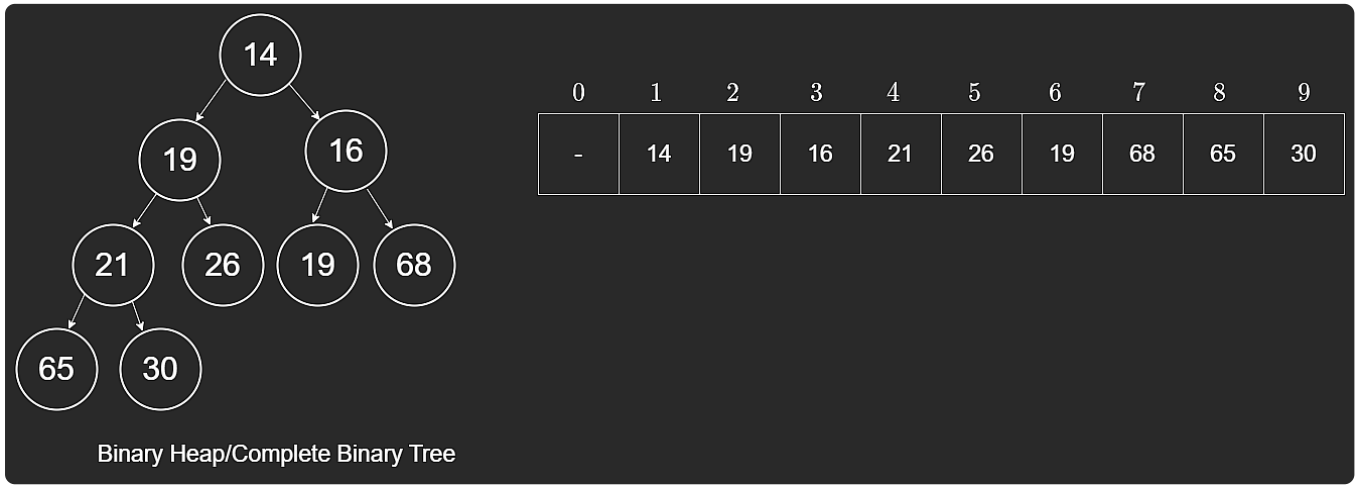
Implementation

Binary heaps are drawn using a tree data structure but under the hood, they are implemented using arrays. Let's show how we can do this by using the given binary

heap: `[14,19,16,21,26,19,68,65,30,null,null,null,null,null,null]`

We will take an array of size $n + 1$ where n is the number of nodes in our binary heap. This will make sense soon. We will visit our nodes in the same order as we visit nodes in breadth-first search - level by level, from left to right. We will insert these into our array in a contiguous fashion. However, we will start filling them from index `1` instead of `0`, for reasons we will discuss soon.

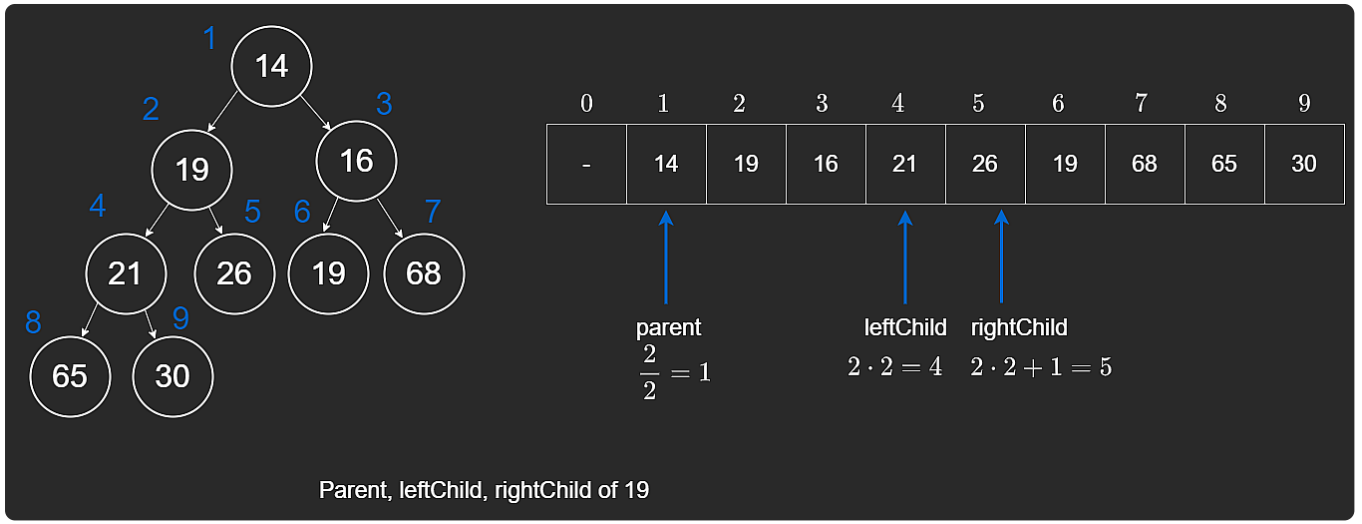
Once our array has been filled up, it would look like the following:



The reason why we start filling up our array from index `1` is because it helps us figure out the index at which a node's left child, right child, or the parent resides. Because binary heaps are complete binary trees, no space is required for pointers. Instead, a node's left child, right child and parent can be calculated using the following formulas, where i is the index of a given node.

`leftChild` $= 2 * i$ `rightChild` $= 2 * i + 1$ `parent` $= i / 2$

Now, suppose we wanted to find the above properties of node `19`. The following visual demonstrates how using the formulas helps us figure them out. The number within the circle at each node in the tree is the value stored at that node. The number above a node (in blue) is the corresponding index in the array. It is important to note that these formulas only work when the tree is a complete binary tree. We can also now appreciate why we start at index `1`. Suppose we wanted to find `14`'s left and right child and `14` was at `0`. Well, any number multiplied by a 0 is 0, and would tell us that the left child resides at the `0` th index, which is of course not the case.



Parent, leftChild, rightChild of 19

> *Whenever any operations are done on the heap, such as remove or add, we have to make sure that the min-heap properties are satisfied and the three*

> *aforementioned formulas are still valid. We will discuss this in the next chapter.*

Below is the code implementation of heap.

```
class Heap:
     heap = a list of integers, objects etc.
     constrtuctor():
        heap = intialize the list
        heap.add(0)
```

Closing Notes

Looking at max heap and min heap properties, if a problem asks us to find the minimum or the maximum, a heap is a viable option. We will see how efficient it is when it comes to removal and addition operations.

Github Privacy Terms