Max Heap Introduction

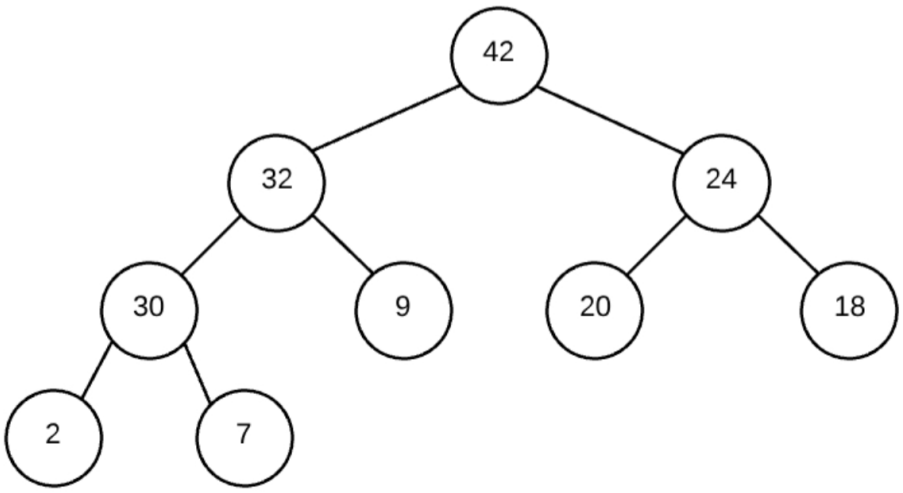
10 minutes

Binary Heap Implementation

Now that we are familiar with the structure of a heap, let's implement one! What may be surprising is that the usual way to implement a heap is by simply using an array. That is, we won't need to create a node class with pointers. Instead, each index of the array will represent a node, with the root being at index 1. We'll avoid using index 0 of the array so our math works out nicely. From this point, we'll use the following rules to interpret the array as a heap:

- index `i` represents a node in the heap
- the left child of node `i` can be found at index `2 * i`
- the right child of code `i` can be found at index `2 * i + 1`

In other words, the array `[null, 42, 32, 24, 30, 9, 20, 18, 2, 7]` represents the heap below. Take a moment to analyze how the array indices work out to represent left and right children.



Pretty clever math right? We can also describe the relationship from child to parent node. Say we are given a node at index `i` in the heap, then it's parent is found at index `Math.floor(i / 2)`.

It's useful to visualize heap algorithms using the classic image of nodes and edges, but we'll translate that into array index operations.

Insert

What's a heap if we can't add data into it? We'll need a `insert` method that will add a new value into the heap without voiding our heap property. In our `MaxHeap`, the property states that a node must be greater than its children.

Visualizing our heap as a tree of nodes:

1. We begin an insertion by adding the new node to the bottom leaf level of the heap, preferring to place the new node as far left in the level as possible. This ensures the tree remains complete.
2. Placing the new node there may momentarily break our heap property, so we need to restore it by moving the node up the tree into a legal position. Restoring the heap property is a matter of continually swapping the new node with it's parent while it's parent contains a smaller value. We refer to this process as `siftUp`

Translating that into array operations:

1. `push` the new value to the end of the array
2. continually swap that value toward the front of the array (following our child-parent index rules) until heap property is restored

DeleteMax

This is the "fetch" operation of a heap. Since we maintain heap property throughout, the root of the heap will always be the maximum value. We want to delete and return the root, whilst keeping the heap property.

Visualizing our heap as a tree of nodes:

1. We begin the deletion by saving a reference to the root value (the max) to return later. We then locate the right most node of the bottom level and copy it's value into the root of the tree. We easily delete the duplicate node at the leaf level. This ensures the tree remains complete.
2. Copying that value into the root may momentarily break our heap property, so we need to restore it by moving the node down the tree into a legal position. Restoring the heap property is a matter of continually swapping the node with the greater of it's two children. We refer to this process as `siftDown`.

Translating that into array operations:

1. The root is at index 1, so save it to return later. The right most node of the bottom level would just be the very last element of the array. Copy the last element into index 1, and pop off the last element (since it now appears at the root).
2. Continually swap the new root toward the back of the array (following our parent-child index rules) until heap property is restored. A node can have two children, so we should always prefer to swap with the greater child.

Time Complexity Analysis

- insert: `O(log(n))`
- deleteMax: `O(log(n))`

Recall that our heap will be a complete/balanced tree. This means it's height is `log(n)` where `n` is the number of items. Both `insert` and `deleteMax` have a time complexity of `log(n)` because of `siftUp` and `siftDown` respectively. In worst case `insert`, we will have to `siftUp` a leaf all the way to the root of the tree. In the worst case `deleteMax`, we will have to `siftDown` the new root all the way down to the leaf level. In either case, we'll have to traverse the full height of the tree, `log(n)`.

Array Heapify Analysis

Now that we have established `O(log(n))` for a single insertion, let's analyze the time complexity for turning an array into a heap (we call this heapify, coming in the next project :)). The algorithm itself is simple, just perform an `insert` for every element. Since there are `n` elements and each insert requires `log(n)` time, our total complexity for heapify is `O(nlog(n))` ... Or is it? There is actually a tighter bound on heapify. The proof requires some math that you won't find valuable in your job search, but do understand that the true time complexity of heapify is amortized `O(n)`. Amortized refers to the fact that our analysis is about performance over many insertions.

Space Complexity Analysis

- `O(n)`, since we use a single array to store heap data.

Did you find this lesson helpful?

No   Yes

✓ Mark As Complete

Finished with this task? Click **Mark as Complete** to continue to the next page!