

WEEK-07 DAY-5

Friday - *Heaps*

- [WEEK-07 DAY-5](#)
 - [Friday - *Heaps*](#)
- [Graphs and Heaps](#)
- [Introduction to Heaps](#)
 - [Complete Trees](#)
 - [When to Use Heaps?](#)
 - [Binary Heap Implementation](#)
 - [Insert](#)
 - [Visualizing our heap as a tree of nodes:](#)
 - [Translating that into array operations:](#)
 - [DeleteMax](#)
 - [Visualizing our heap as a tree of nodes:](#)
 - [Translating that into array operations:](#)
 - [Time Complexity Analysis](#)
 - [Array Heapify Analysis](#)
 - [Space Complexity Analysis](#)

Graphs and Heaps

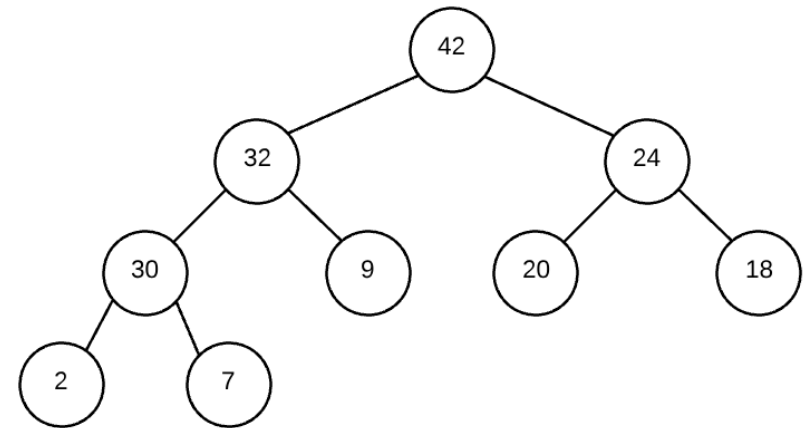
The objective of this lesson is for you to become comfortable with implementing common data structures. This is important because questions about data structures are incredibly likely to be interview questions for software engineers from junior to senior levels. Moreover, understanding how different data structures work will influence the libraries and frameworks that you choose when writing software.

When you are done, you will be able to:

1. Explain and implement a Heap.
2. Explain and implement a Graph.

Introduction to Heaps

Let's explore the **Heap** data structure! In particular, we'll explore **Binary Heaps**. A binary heap is a type of binary tree. However, a heap is not a binary *search* tree. A heap is a partially ordered data structure, whereas a BST has full order. In a heap, the root of the tree will be the maximum (max heap) or the minimum (min heap). Below is an example of a max heap:



Notice that the heap above does not follow search tree property where all values to the left of a node are less and all values to the right are greater or equal. Instead, the max heap invariant is:

- given any node, its children must be less than or equal to the node

This constraint makes heaps much more relaxed in structure compared to a search tree. There is no guaranteed order among "siblings" or "cousins" in a heap. The relationship only flows down the tree from parent to child. In other words, in a max heap, a node will be greater than all of its children, its grandchildren, its great-grandchildren, and so on. A consequence of this is the root being the absolute maximum of the entire tree. We'll be exploring max heaps together, but these arguments are symmetric for a min heap.

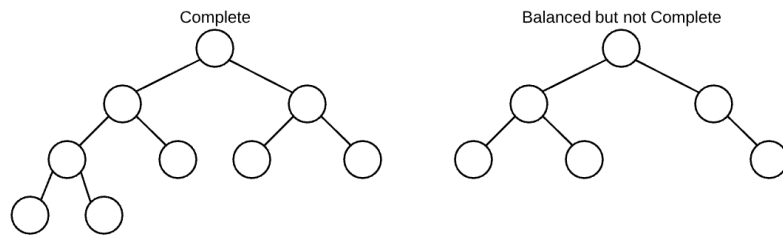
Complete Trees

We'll eventually implement a max heap together, but first we'll need to take a quick detour. Our design goal is to implement a data structure with efficient operations. Since a heap is a type of binary tree, recall the circumstances where we had a "best case" binary tree. We'll need to ensure our heap has minimal height, that is, it must be a balanced tree!

Our heap implementation will not only be balanced, but it will also be **complete**. To clarify, **every complete tree is also a balanced tree**, but not every balanced tree is also complete. Our definition of a complete tree is:

- a tree where all levels have the maximal number of nodes, except the bottom level
- AND the bottom level has all nodes filled as far left as possible

Here are few examples of the definition:



Notice that the tree is on the right fails the second point of our definition because there is a gap in the last level. Informally, you can think about a complete tree as packing its nodes as closely together as possible. This line of thinking will come into play when we code heaps later.

When to Use Heaps?

Heaps are the most useful when attacking problems that require you to "partially sort" data. This usually takes form in problems that have us calculate the largest or smallest n numbers of a collection. For example: What if you were asked to find the largest 5 numbers in an array in linear time, $O(n)$? The fastest sorting algorithms are $O(n \log n)$, so none of those algorithms will be good enough. However, we can use a heap to solve this problem in linear time.

We'll analyze this in depth when we implement a heap in the next section!

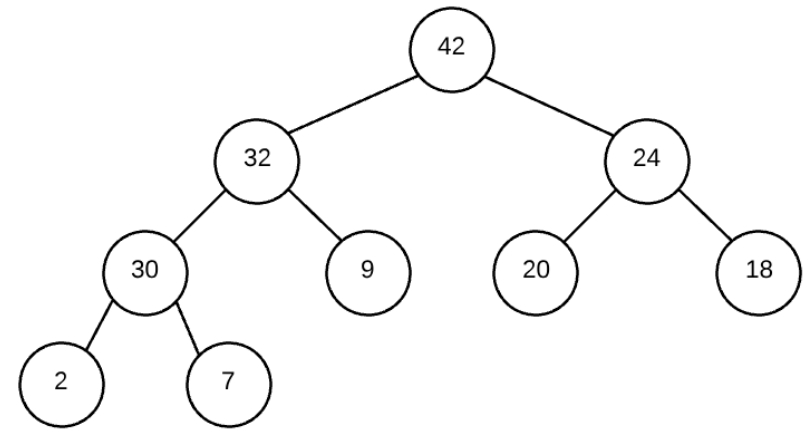
One of the most common uses of a binary heap is to implement a "priority queue". We learned before that a queue is a FIFO (First In, First Out) data structure. With a priority queue, items are removed from the queue based on a priority number. The priority number is used to place the items into the heap and pull them out in the correct priority order!

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 node 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 its parent is found at index $\text{Math.floor}(i / 2)$.

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 its parent while its 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 $O(n\log(n))$... 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.