Heaps

Definition 0.1

Heaps have the shape property of a binary tree (having 0-2 children), but heaps must also be *complete*.

This property is what makes heaps easy to implement with arrays

Min Heap

Definition 0.2

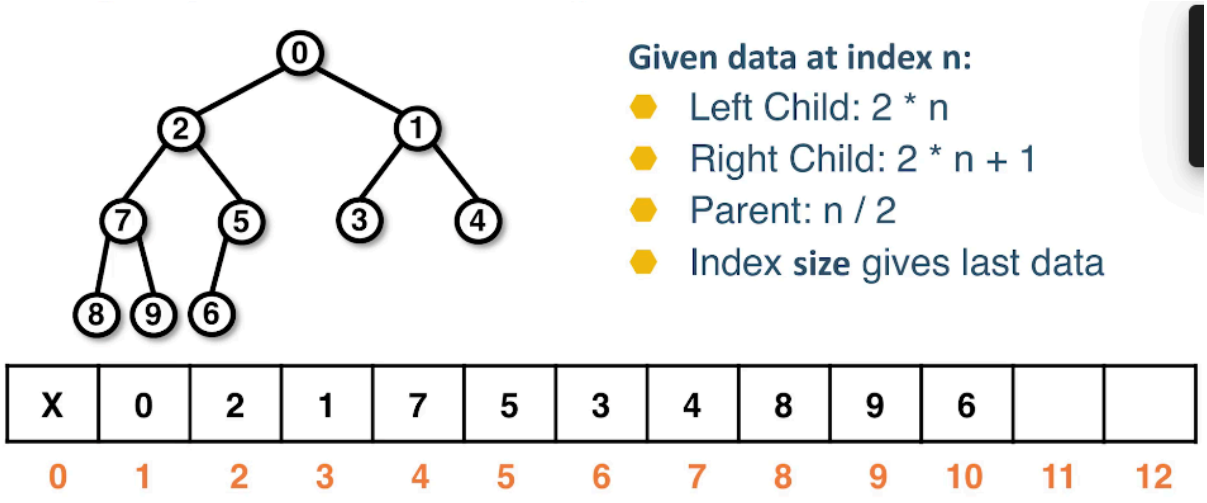
In **minheap**, the smallest data lives in the root and each child is greater than its parents value. There is no relationship between siblings

Heaps can be illustrated and *backed* as an array given their characteristics, displayed in level order.

Definition 0.3

Given data at index n:

- Left child: $2 * n$
- Right child: $2 * n + 1$
- Parent: $\frac{n}{2}$
  - Should be truncated if not an integer



Example 0.1

Heaps Use Cases

- Not designed for arbitrary searching, mostly designed for accessing root
  - They are no better than just searching an arraylist (O(n))
- Heaps are often used to back priority queues

Heap Operations

Add Algorithm

- Add to the next spot in the array to maintain completeness
  - this would be index[size]

> **Note 0.1**
>
> We do not use **index zero** for heaps

- Up heap starting from the new data to fix order property
  - Compare the data with the parent, and swap the data with the parent as necessary until we read the top or no swap is needed. *This differs for min heap and max heap.*

> **Theorem 0.1**
>
> Time complexity of adding a new element is **O(logn)**. While adding to end of array is O(1), the up-heap process is O(logn).

Remove Algorithm

- Move the last element of the heap and use it to replace the root (since we want to delete root)
- Down heap starting from the root to fix the order property. If two children, compare data with larger or smaller priority child, depending on if it is a min or max heap.

> **Theorem 0.2**
>
> Time complexity of removing an element is **O(logn)** due to the down-heap process.

Build Heap Operation

We use the **buildheap operation** for taking an unsorted, unordered data and turning it into a heap.

- First we put the data into an array, which will satisfy the *shape property*
- Then, in order to satisfy order property we look at the sub-heaps and down heap through the valid sub-heaps

We loop starting at `index size/2`, since this is the last element that has a child, and we go up to index 1 and repeatedly calling the down-heap method. For the rest of the array we don't need down heap if its already valid (?)

> **Theorem 0.3**
>
> There is more data in the bottom half of the tree than the top half, so as we go down each time the data doubles, meaning we have exponential grown in data as we go down.
>
> The down-heap cost is **O(1)** at the bottom of the tree but increases linearly as we go up the tree meaning most of the data at the top is **O(logn)**, so it balances out to be **O(n)** if you do the summation of the series.