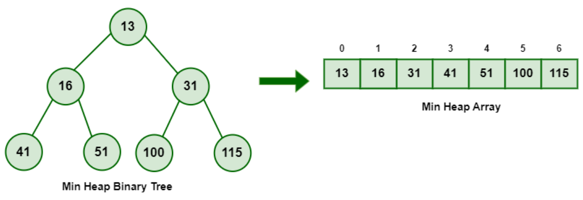
HEAP DATA STRUCTURES AND ALGORITHMS

Heap – Introduction

A Heap is a represented as Tree-Based Data Structure but actually the values are stored in Array. It has the following properties. It is a complete Complete Binary Tree. It either follows **Max Heap** or **Min Heap** property.

Max-Heap: The value of the root node must be the greatest among all its descendant nodes and the same thing must be done for its left and right sub-tree also.

Min-Heap: The value of the root node must be the smallest among all its descendant nodes and the same thing must be done for its left and right sub-tree also.



Priority Queues can be efficiently implemented using Binary Heap because it supports insert(), delete() and extractmax(), decreaseKey() operations in O(log N) time.

Properties of Heap

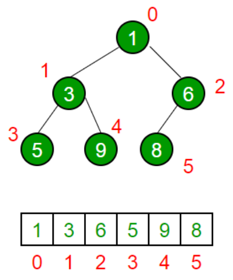
The minimum or maximum element is always at the root of the heap, allowing constant-time access. The relationship between a parent node at index **'i'** and its children is given by the formulas: left child at index **2i+1** and right child at index **2i+2** for 0-based indexing of node numbers.

As the tree is complete binary, all levels are filled except possibly the last level. And the last level is filled from left to right. When we insert an item, we insert it at the last available slot and then rearrange the nodes so that the heap property is maintained. When we remove an item, we swap root with the last node to make sure either the max or min item is removed. Then we rearrange the remaining nodes to ensure heap property (max or min).

How is Binary Heap Represented?

A Binary Heap is a **Complete Binary Tree**. A binary heap is typically represent ed as an array. The root element will be at arr[0]. The below table shows indices of other nodes for the ith node, i.e., arr[i]:

arr[(i-1)/2]	Returns the **Parent Node**
arr[(2*i)+1]	Returns the **Left Child Node**
arr[(2*i)+2]	Returns the **Right Child Node**



The traversal method use to achieve Array representation is **Level Order Traversal**.

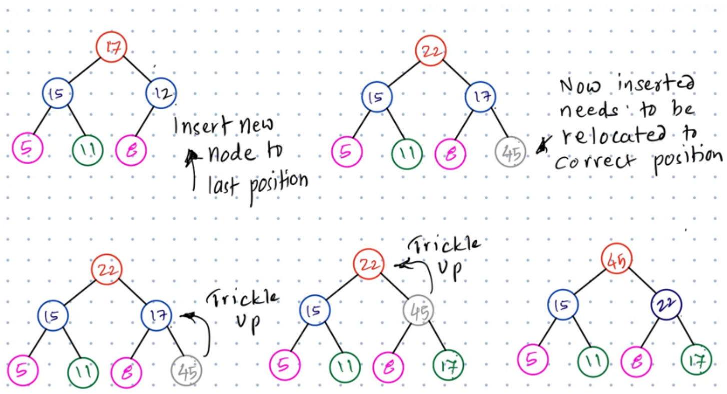
Operations on Heap

Insertion

Inserting a new key takes **O(log N)** time. We add a new key at the end of the tree. If the new key is greater than its parent, then we don't need to do anything.

Otherwise, we need to traverse up to fix the violated heap property. It uses **Up-Heap Algorithm** to insert a value in the Heap.

Trickle up is performed during addition of the node at the last position.

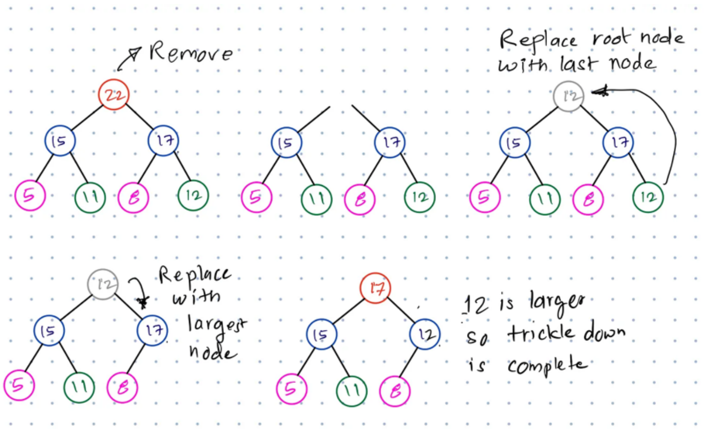

Insert new node to last position
Now inserted needs to be relocated to correct position
Trickle up
Trickle up

When a node needs to be added in the heap, it's always gets added at the last node position. If node position is root then trickle up operation is not required. Otherwise parent position is computed based on above formula. When position that is getting trickle up is larger than parent, then its swapped with parent node. Recursively trickle up is called for the parent position.

Deletion

Deleting a key also takes **O(log N)** time. We replace the key to be deleted with the minimum infinite by calling **decreaseKey()**. After decreaseKey(), the minus infinite value must r each root, so we call **extractMin()** to remove the key.

Trickle Down: place element to the correct position within the heap by swapping the nodes down. Trickle down is performed during removal of the root node from heap. When a node needs to be removed from heap, it's always the root node that gets removed.


Remove
Replace root node with last node
Replace with largest node
12 is larger so trickle down is complete

First left and right child positions are computed, then position that needs to be trickled down is compared to its left and right child. Largest child is swapped at parent position. Trickle down is recursively called at its largest child position.

Heapify

Conversion randomly ordered elements into heap. Heapify is efficient way to create heap rather than using trickle up or down process. Because trickle up or down needs to loop through all elements of the heap to ensure all elements are at correct position. Heapify loops backwards from last parent, computes child positions of the parent and if children exists then swaps parent with largest child. Then recursively calls heapify or trickle down at the largest position. So basically heapify and trickle down are similar except heapify gets called only at the parent nodes.