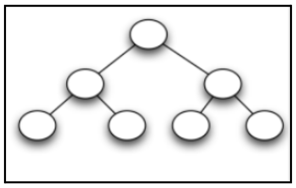
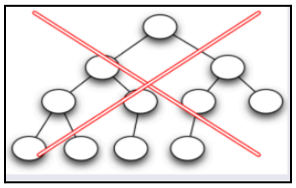
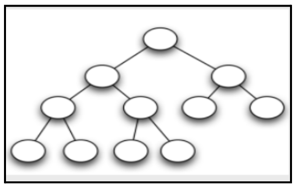
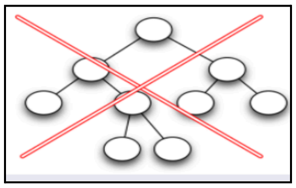
Heap

Heap Data Structure:

Recap:

- In a Binary Tree, a parent can have a maximum of 2 nodes and a minimum of 0 nodes.
- In a Complete Binary Tree, all the levels are filled entirely with nodes, except the lowest level of the tree where the nodes are added in a left-to-right manner not skipping any position.

A Heap is an Abstract Data Type (ADT) for storing values. Its underlying data structure is an array.

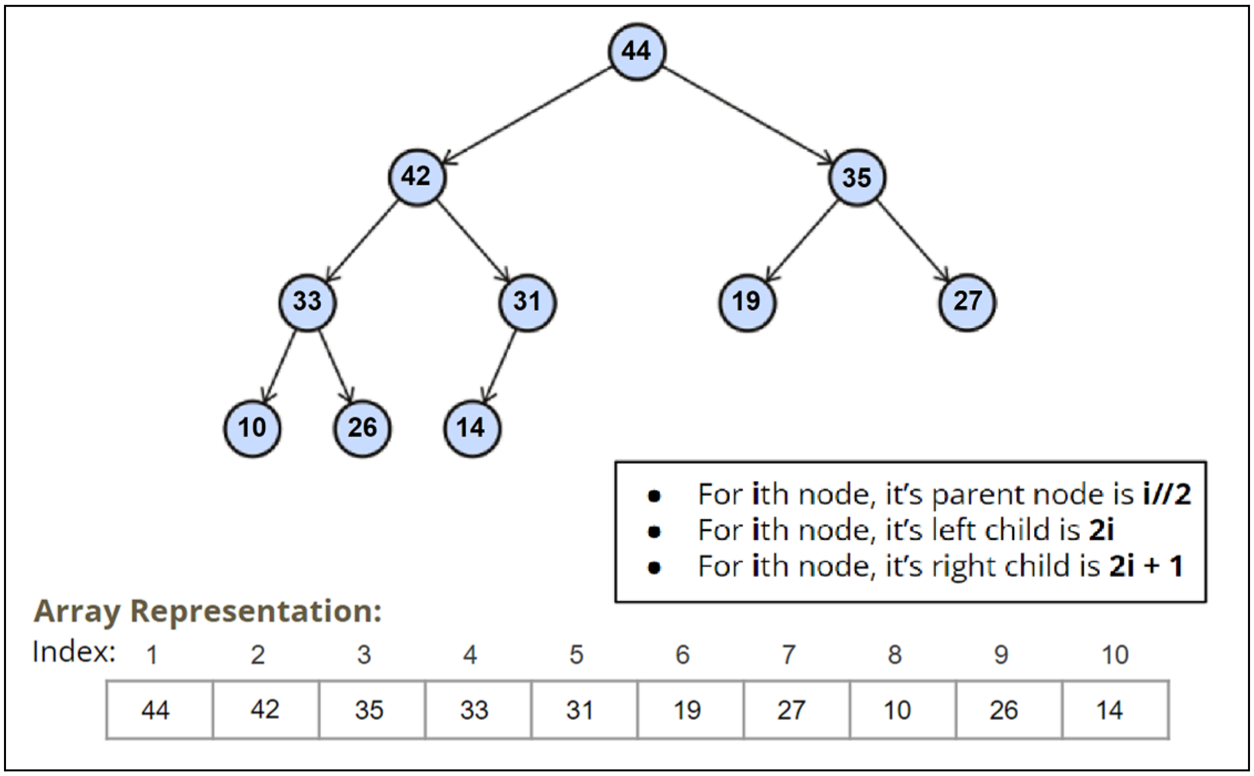
A Heap must be a complete binary tree and satisfy the heap property.

Heap property:

- The value of the parent must be greater than or equal to the values of the children (Max heap).
  or
- The value of the parent must be smaller than or equal to the values of the children. (Min heap).

There are two types of heaps. Max heap is mostly used (default). A heap can be either a max heap or a min heap but cannot be both simultaneously.

Heap data structure provides worst-case O(1) time access to the largest (max-heap) or smallest (min-heap) element and provides worst-case Θ(log n) time to extract the largest (max-heap) or smallest (min-heap) element.



- For **i**th node, it's parent node is **i//2**
- For **i**th node, it's left child is **2i**
- For **i**th node, it's right child is **2i + 1**

Array Representation:

Index:	1	2	3	4	5	6	7	8	9	10
	44	42	35	33	31	19	27	10	26	14

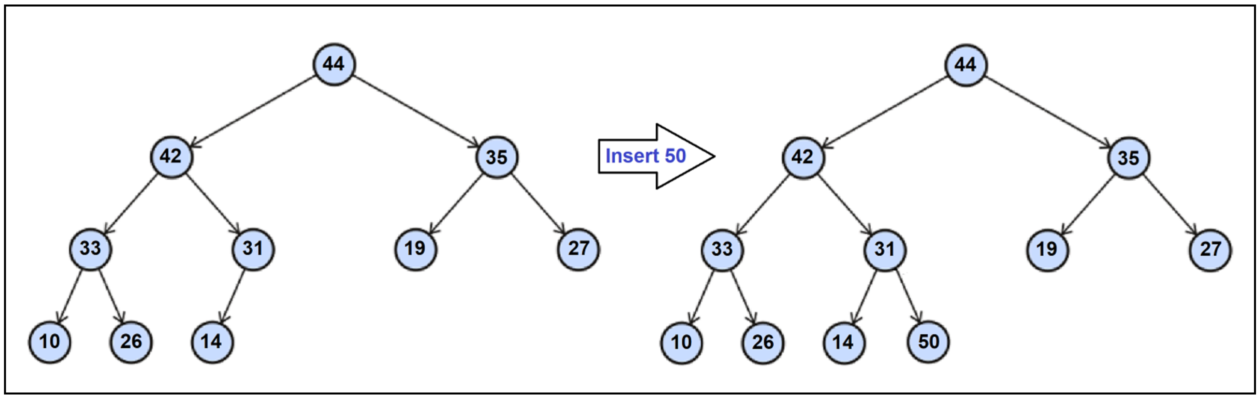
Note: Tree is used for efficient tracing. While programming, the data structure is a simple Array.

The benefit of using Array for Heap rather than Linked List is Arrays give you random access to its elements by indices. You can pick any element from the array by calling the corresponding index. Finding a parent and their children is trivial. Whereas, the Linked List is sequential. This means you need to keep visiting elements in the linked list unless you find the element you are looking for. Linked List does not allow random access as Array does. Also, each Linked List must have three (3) references to traverse the whole Tree (Parent, left, Right).

Heap Operations:

- Insert:

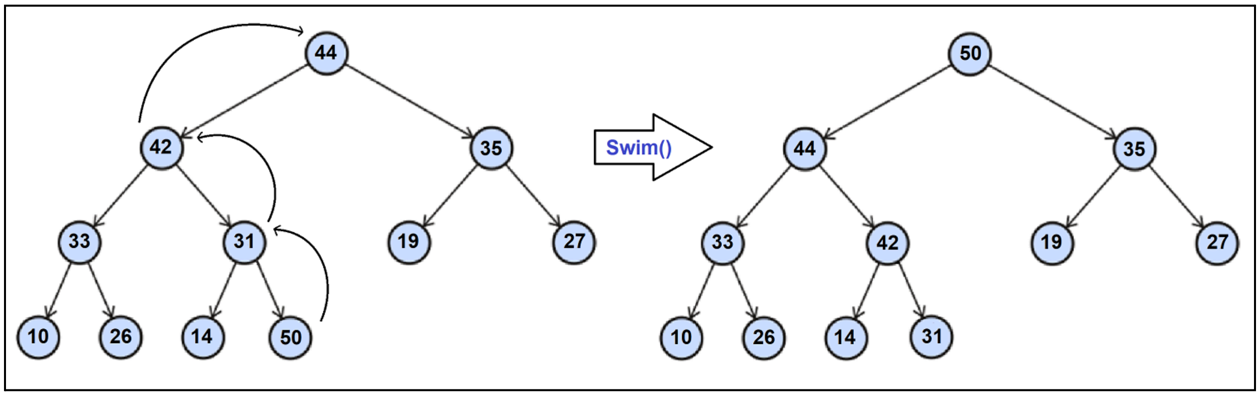
Inserts an element at the bottom of the Heap. Then we must make sure that the Heap property remains unchanged. When inserting an element in the Heap, we start from the left available position to the right.



Here, after inserting 50, the heap property is broken, Therefore, we need to put 50 in its correct position.

- HeapIncreaseKey / Swim:

Let the new node be ‘n’ (in this case it is the node that contains 50). Check ‘n’ with its parent. If the parent is smaller (n > parent) than the node ‘n’, replace ‘n’ with the parent. Continue this process until n is in its correct position.



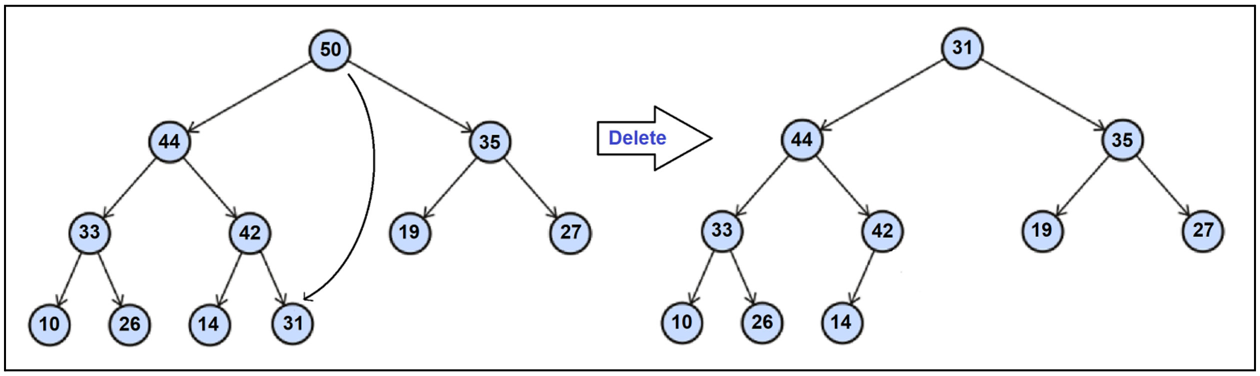
Best-case Time Complexity is O(1) when a key is inserted in the correct position at the first go.
Worst-case Time Complexity is when the newest node needs to climb up to the root
O(1) [insertion] + O(log n) [swim] = O(log n)

```
class MaxHeap:
    def __init__(self, capacity):
        self.capacity = capacity
        self.H = np.zeros(capacity + 1, dtype=int)
        self.size = 0
```

```
def insert(H, key):
        size = size + 1
        H[size] = key
        swim(H, size)

def swim(H, index):
        if index <= 1:
                return
        else:
                parent = index // 2
                if H[parent] < H[index]:  #Swap parent and child
                        H[index], H[parent] = H[parent], H[index]
                        swim(H, parent)
```

- Delete:

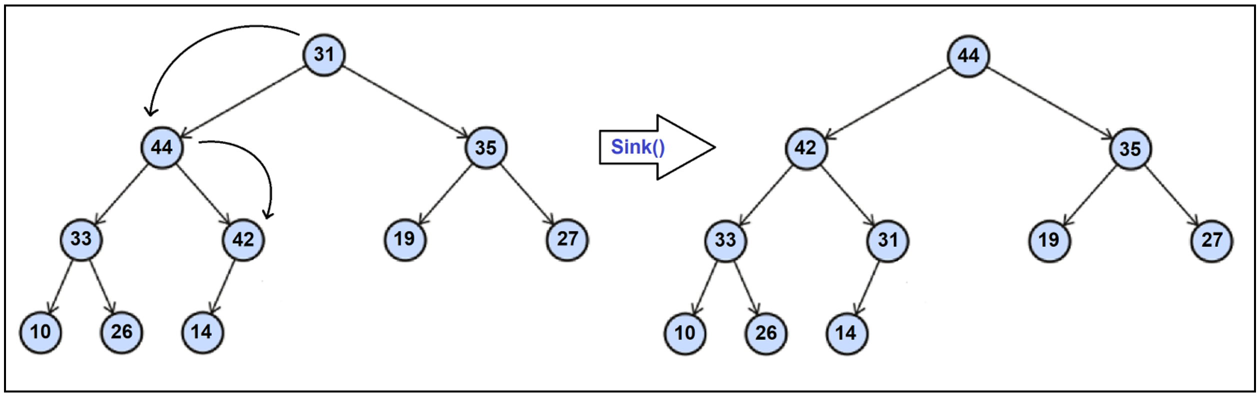
In heap, you cannot just randomly delete an item. Deletion is done by replacing the root with the last element. The Heap property will be broken as a small value will be at the top (root) of the Heap. Therefore we must put it in the right place.



Here, the root 50 will be replaced by the last element 31, and 31 will be removed. The heap property will be broken. Therefore, we need to put 31 in its correct position.

- MaxHeapify / Sink:

Let the replaced node be ‘n’ (in this case it is the node that contains 31 Check ‘n’ with its children. If the node ‘n’ is smaller (n < any child) than any child, replace ‘n’ with the largest child. Continue this process until n is in its correct position.



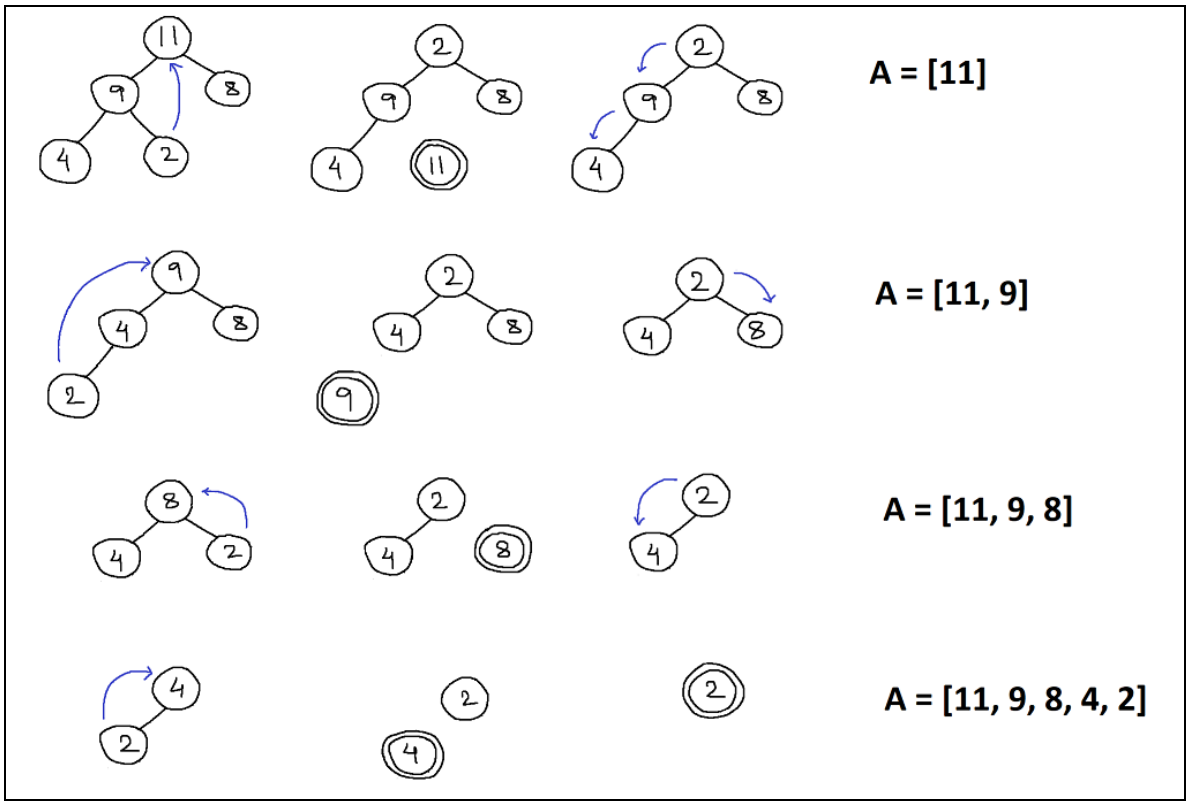
Deleted element will always be the maximum element available in max-heap.Time Complexity is O(1) [deletion] + O(log n) [sink] = O(log n)

```
def delete(H):
        if size == 0:
                return
        else:
                max_value = H[1]
                H[size], H[1] = H[1], H[size]  #Swap root and last node
                size -= 1  #Delete last node
                maxHeapify(H, 1)

def maxHeapify(H, index):
        if size == 0:
                return
        else:
                left = 2 * index
                right = 2 * index + 1
                largest = index

                if left <= size and H[left] > H[largest]:
                        largest = left
                if right <= size and H[right] > H[largest]:
                        largest = right
                if largest != index:
                        H[index], H[largest] = H[largest], H[index]
                        maxHeapify(H, largest)
```

- Heap Sort:

Delete + Sink all the nodes of the heap and store them in an array. The array will return a sorted array in descending order. Reversing the array will give a sorted array in ascending order.

Simulation:



Delete + Sink takes O(log n) and for ‘n’ nodes, Heap Sort will take O(n log n).

```
def HeapSort(H, size):
    for i in range(1, size):
        delete(H)  #save the max value in an array
```

- Build Max Heap:

You are given an arbitrary array and asked to build it into a heap. This can be done using 2 ways

1. Using swim: This takes O(n log n)

```
def BuildHeap(H, size):
        for i in range(1, size):
                swim(H, i)
```

2. Using sink: This takes O(n)

```
def BuildHeap(H, size):
        for i in range(size // 2, 0, -1):
                maxHeapify(H, i)
```