Heap Sort Algorithm

Heap Sort is a popular and efficient sorting algorithm in computer programming. Learning how to write the heap sort algorithm requires knowledge of two types of data structures - arrays and trees.

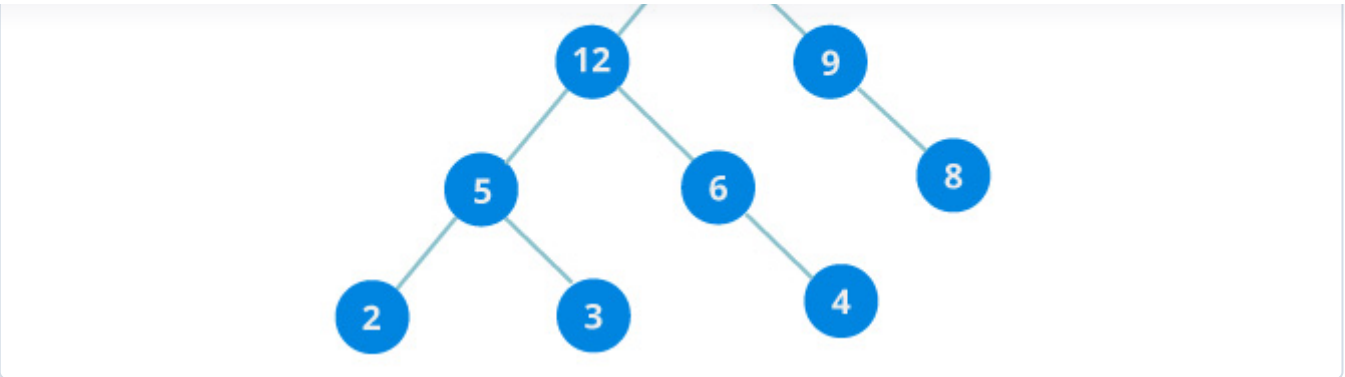
The initial set of numbers that we want to sort is stored in an array e.g. `[10, 3, 76, 34, 23, 32]` and after sorting, we get a sorted array `[3,10,23,32,34,76]`

Heap sort works by visualizing the elements of the array as a special kind of complete binary tree called heap.

What is a complete Binary Tree?

Binary Tree

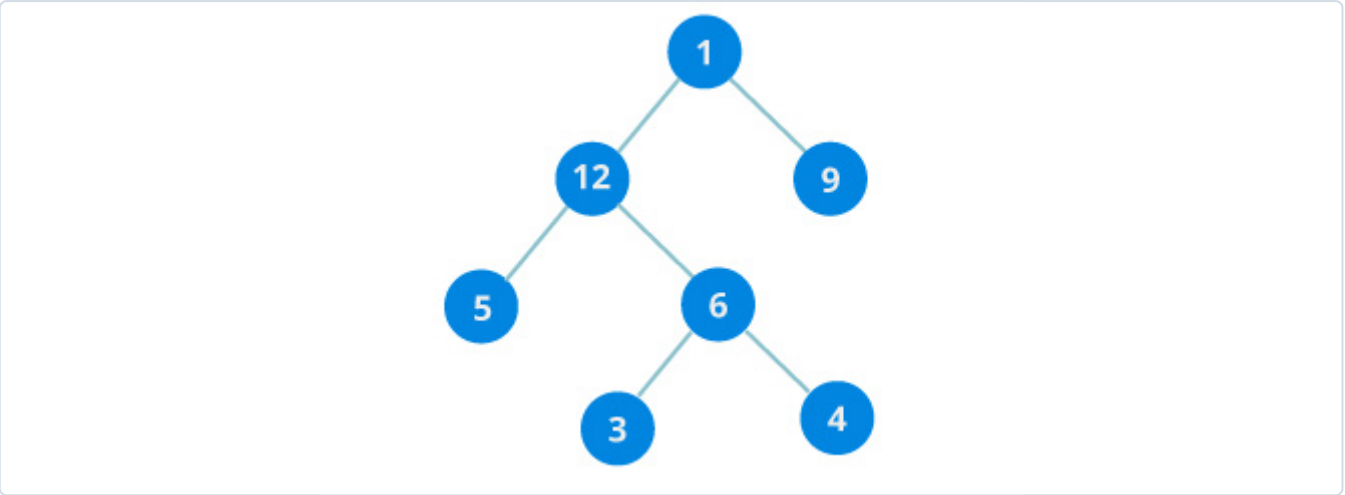
A binary tree is a tree data structure in which each parent node can have at most two children



In the above image, each element has at most two children.

Full Binary Tree

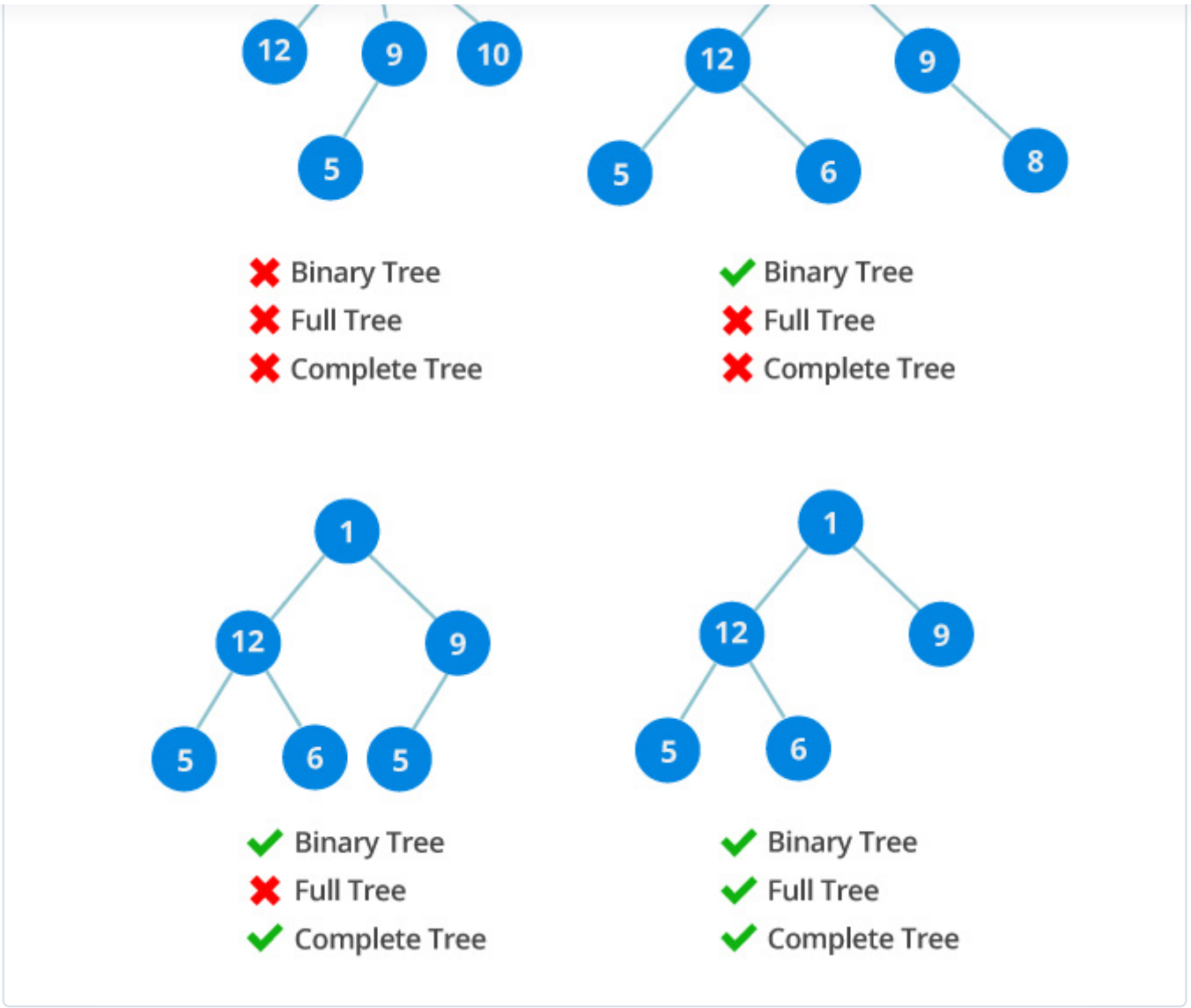
A full Binary tree is a special type of binary tree in which every parent node has either two or no children.



Complete binary tree

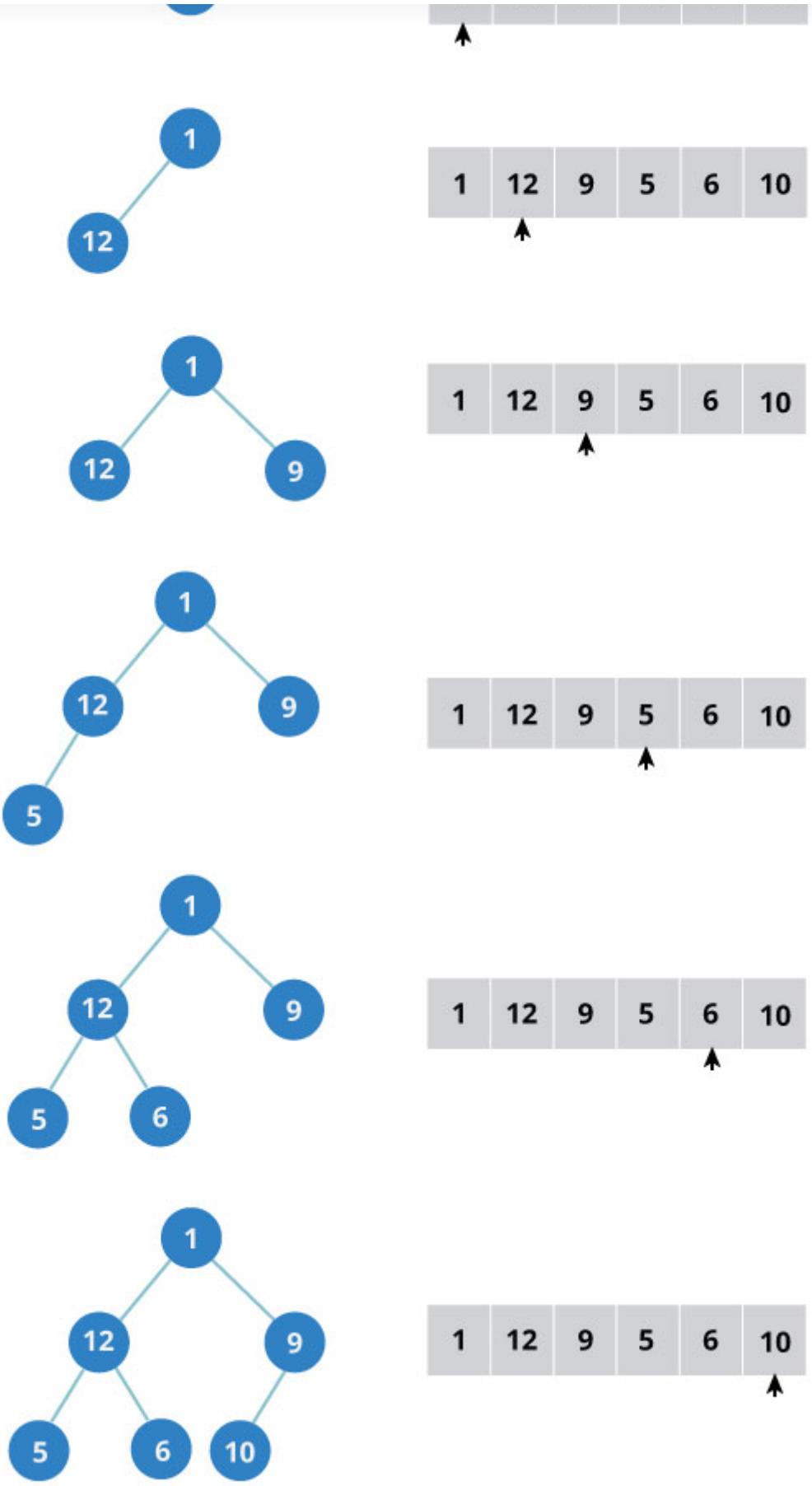
A complete binary tree is just like a full binary tree, but with two major differences

1. Every level must be completely filled
2. All the leaf elements must lean towards the left.
3. The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.



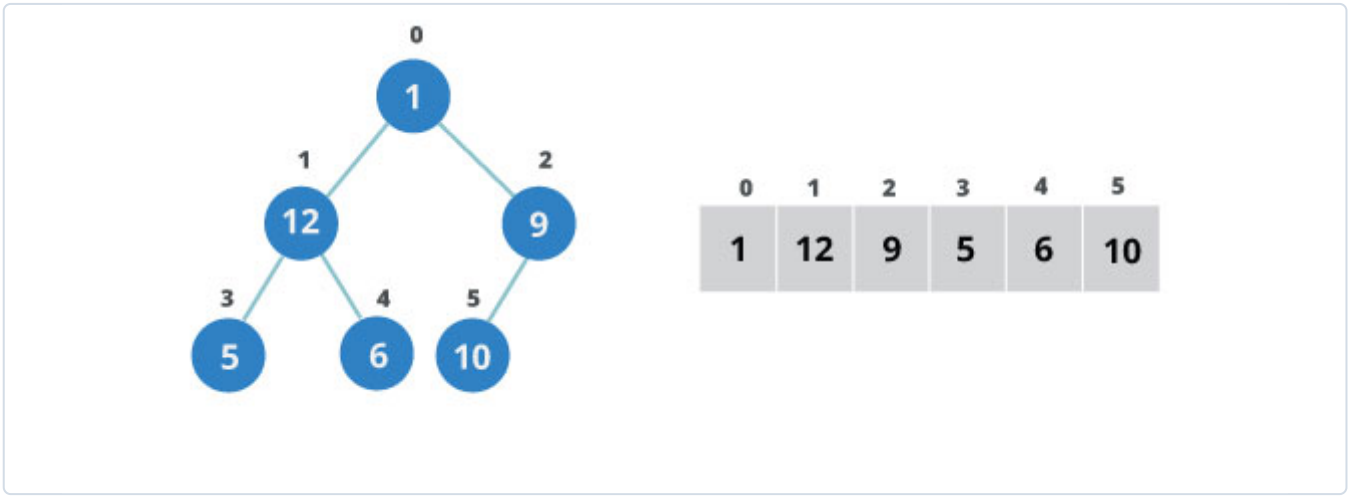
How to create a complete binary tree from an unsorted list (array)?

- Select first element of the list to be the root node. (First level - 1 element)
- Put the second element as a left child of the root node and the third element as a right child. (Second level - 2 elements)
- Put next two elements as children of left node of second level. Again, put the next two elements as children of right node of second level (3rd level - 4 elements).
- Keep repeating till you reach the last element.



Complete binary tree has an interesting property that we can use to find the children and parents of any node.

If the index of any element in the array is `i`, the element in the index `2i+1` will become the left child and element in `2i+2` index will become the right child. Also, the parent of any element at index `i` is given by the lower bound of `(i-1)/2`.



Let's test it out,

```
= element in 1 index
= 12


Right child of 1
= element in (2*0+2) index
= element in 2 index
= 9

Similarly,
Left child of 12 (index 1)
= element in (2*1+1) index
= element in 3 index
= 5

Right child of 12
= element in (2*1+2) index
= element in 4 index
= 6
```

Let us also confirm that the rules holds for finding parent of any node

```
Parent of 9 (position 2)
= (2-1)/2
= ½
= 0.5
~ 0 index
= 1

Parent of 12 (position 1)
= (1-1)/2
= 0 index
= 1
```

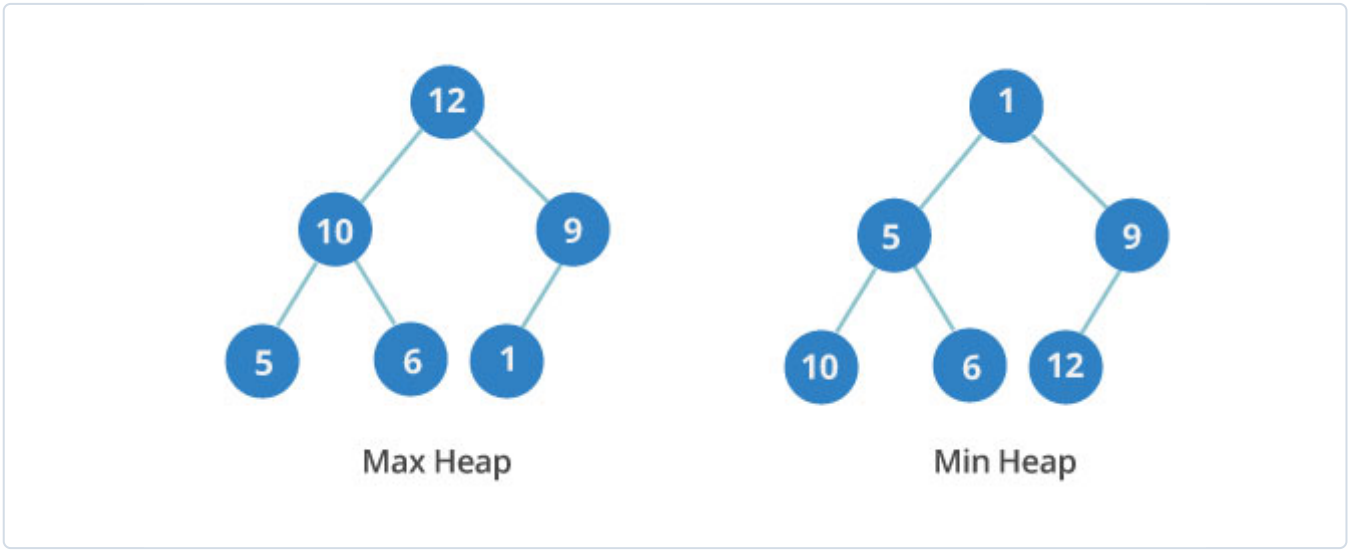
Understanding this mapping of array indexes to tree positions is critical to understanding how the Heap Data Structure works and how it is used to implement Heap Sort.

What is Heap Data Structure ?

Heap is a special tree-based data structure. A binary tree is said to follow a heap data structure if

children i.e. the largest element is at the root and both its children and smaller than the root and so on. Such a heap is called a max-heap. If instead all nodes are smaller than their children, it is called a min-heap

Following example diagram shows Max-Heap and Min-Heap.

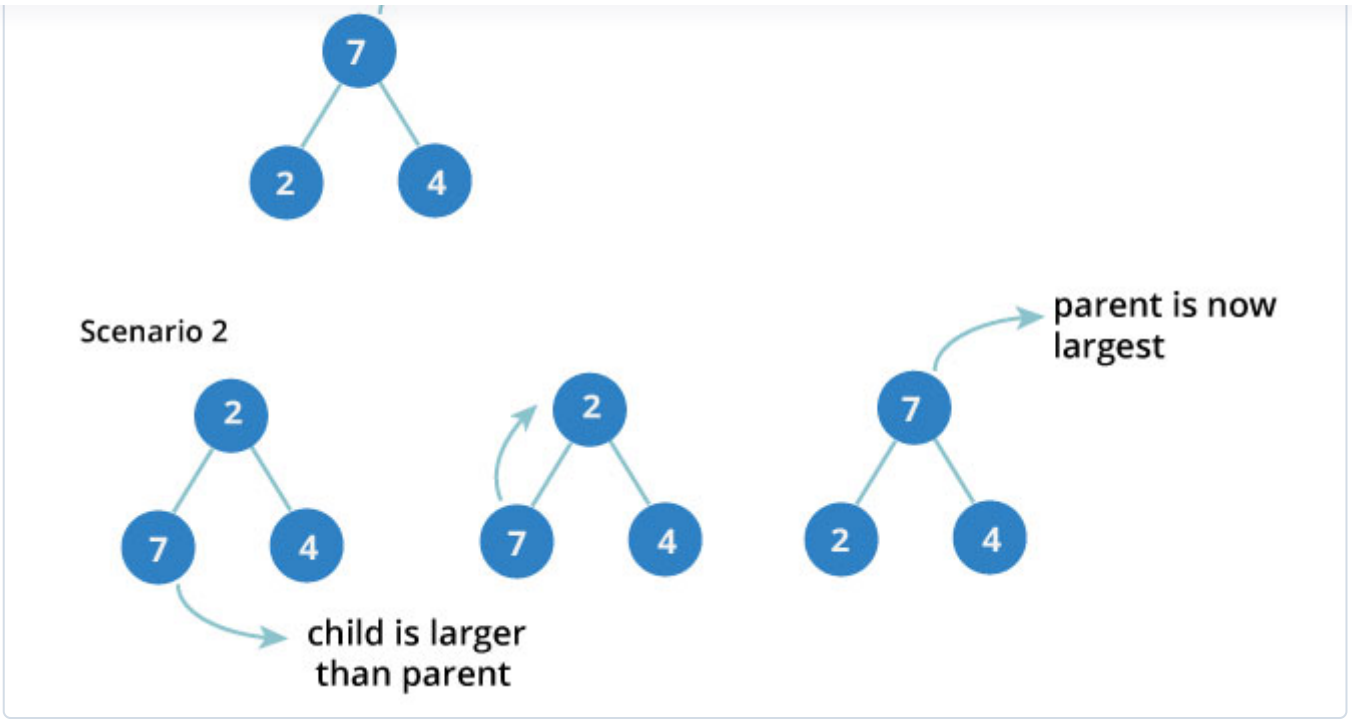


How to “heapify” a tree

Starting from a complete binary tree, we can modify it to become a Max-Heap by running a function called heapify on all the non-leaf elements of the heap.

Since heapfiy uses recursion, it can be difficult to grasp. So let’s first think about how you would heapify a tree with just three elements.

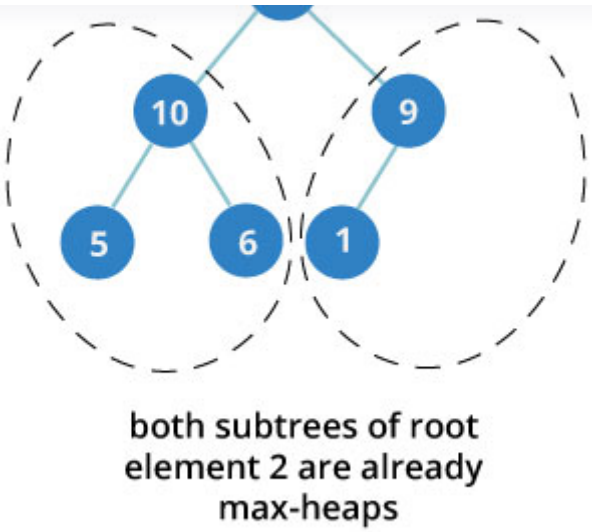
```
heapify(array)
    Root = array[0]
    Largest = largest( array[0] , array [2*0 + 1]. array[2*0+2])
    if(Root != Largest)
          Swap(Root, Largest)
```



The example above shows two scenarios - one in which the root is the largest element and we don't need to do anything. And another in which root had larger element as a child and we needed to swap to maintain max-heap property.

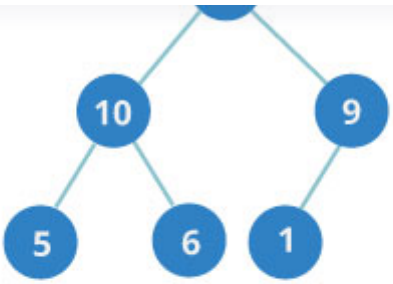
If you're worked with recursive algorithms before, you've probably identified that this must be the base case.

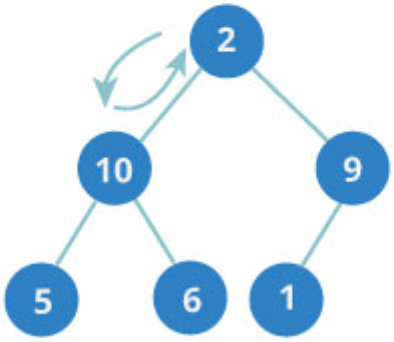
Now let's think of another scenario in which there are more than one levels.

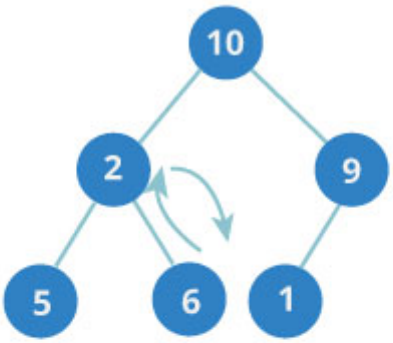


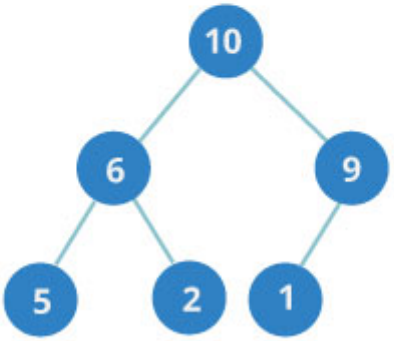
The top element isn't a max-heap but all the sub-trees are max-heaps.

To maintain the max-heap property for the entire tree, we will have to keep pushing 2 downwards until it reaches its correct position.









Thus, to maintain the max-heap property in a tree where both sub-trees are max-heaps, we need to run heapify on the root element repeatedly until it is larger than its children or it becomes a leaf node.

```
void heapify(int arr[], int n, int i)
{
   int largest = i;
   int l = 2*i + 1;
   int r = 2*i + 2;

   if (l < n && arr[l] > arr[largest])
     largest = l;

   if (right < n && arr[r] > arr[largest])
     largest = r;

   if (largest != i)
   {
     swap(arr[i], arr[largest]);

     // Recursively heapify the affected sub-tree
     heapify(arr, n, largest);
   }
}
```

This function works for both the base case and for a tree of any size. We can thus move the root element to the correct position to maintain the max-heap status for any tree size as long as the sub-trees are max-heaps.

Build max-heap

To build a max-heap from any tree, we can thus start heapifying each sub-tree from the bottom up and end up with a max-heap after the function is applied on all the elements including the root element.

In the case of complete tree, the first index of non-leaf node is given by `n/2 - 1`. All other nodes after that are leaf-nodes and thus don't need to be heapified.

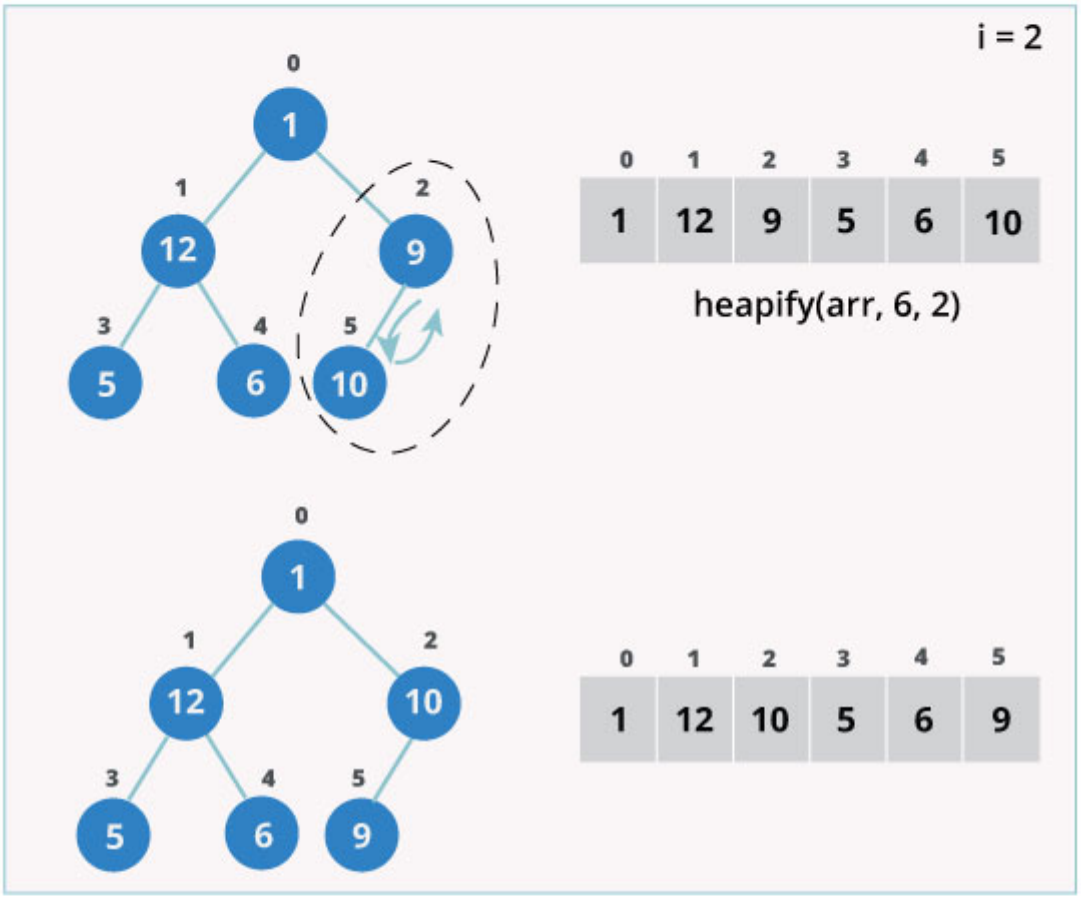
So, we can build a maximum heap as

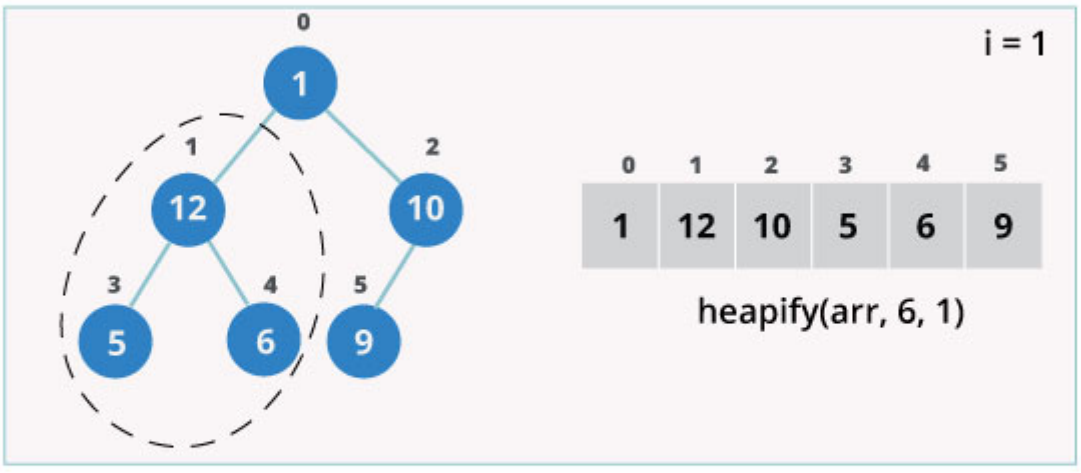
```
  // Build heap (rearrange array)
   for (int i = n / 2 - 1; i >= 0; i--)
     heapify(arr, n, i);
```

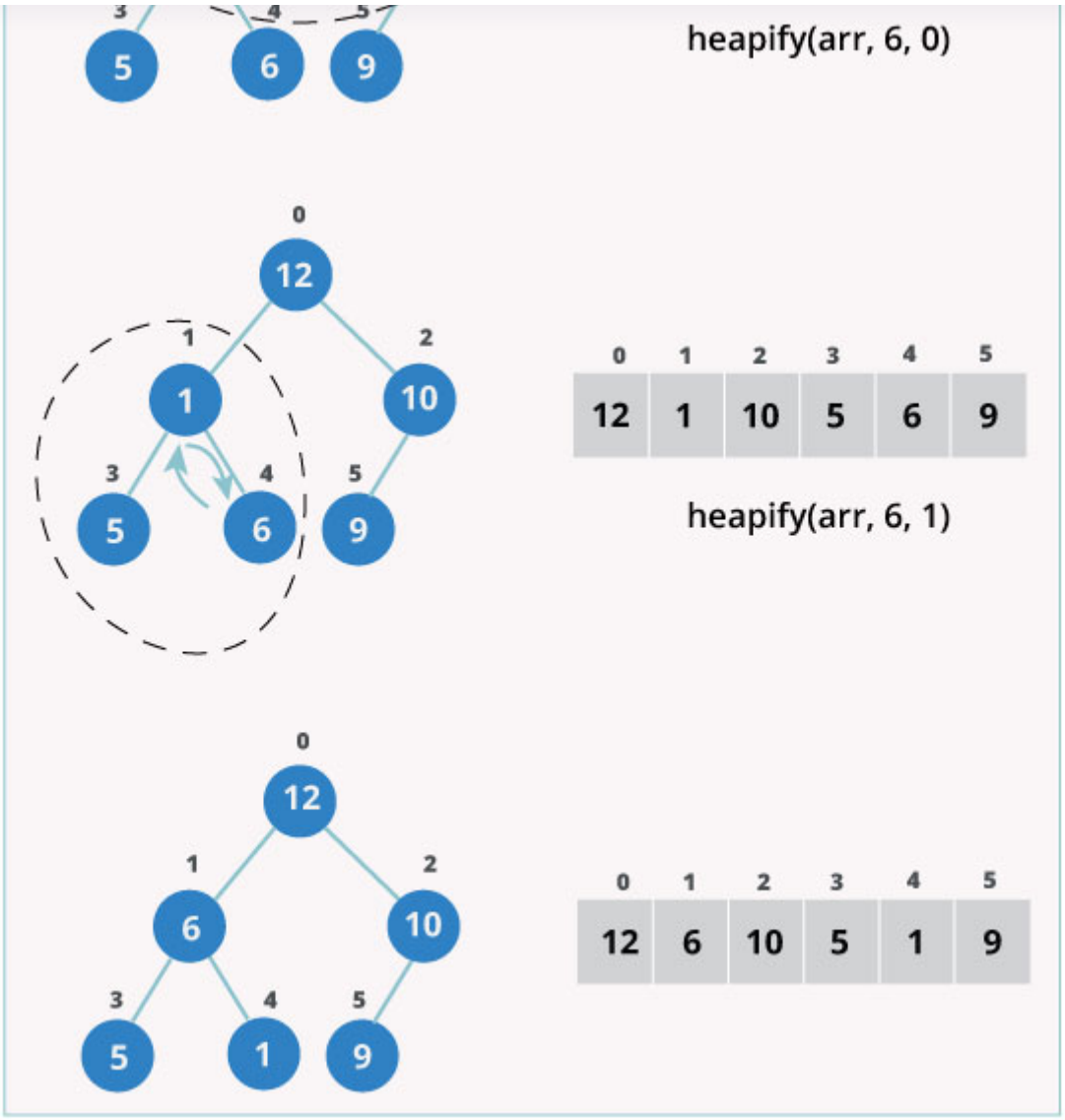
1	12	9	5	6	10

n 6

i = 6/2-1
= 2 -> 0







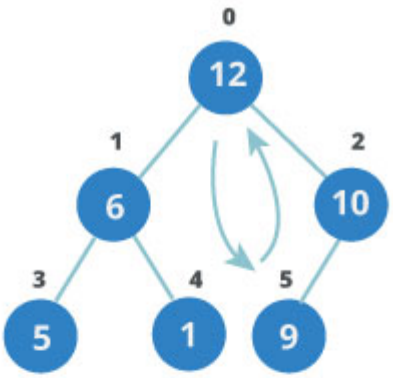
As show in the above diagram, we start by heapifying the lowest smallest trees and gradually move up until we reach the root element.

If you've understood everything till here, congratulations, you are on your way to mastering the Heap sort.

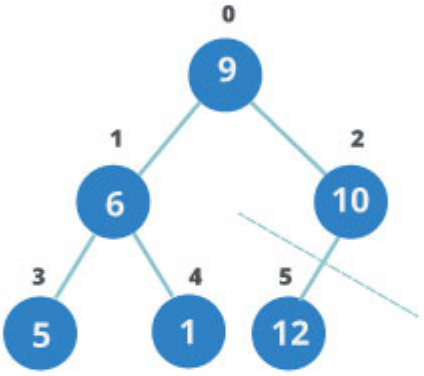
Procedures to follow for Heapsort

1. Since the tree satisfies Max-Heap property, then the largest item is stored at the root node.
2. Remove the root element and put at the end of the array (nth position) Put the last item of the tree (heap) at the vacant place.

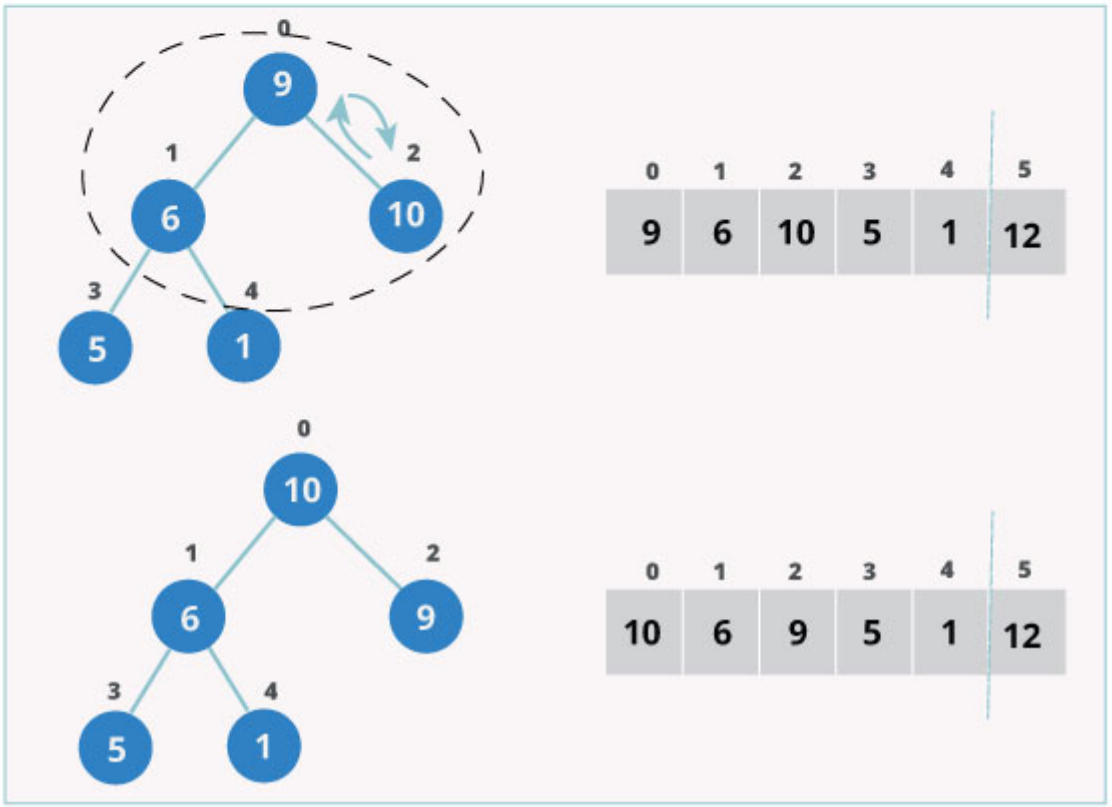
4. The process is repeated until all the items of the list is sorted.



0	1	2	3	4	5
12	6	10	5	1	9



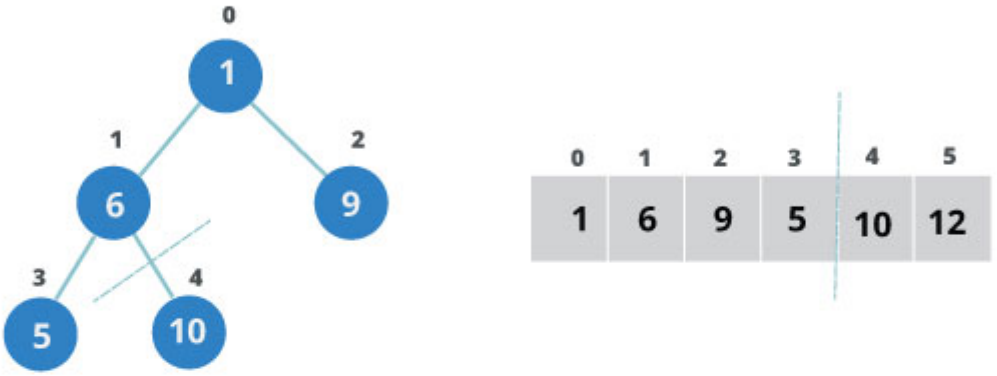
0	1	2	3	4	5
9	6	10	5	1	12

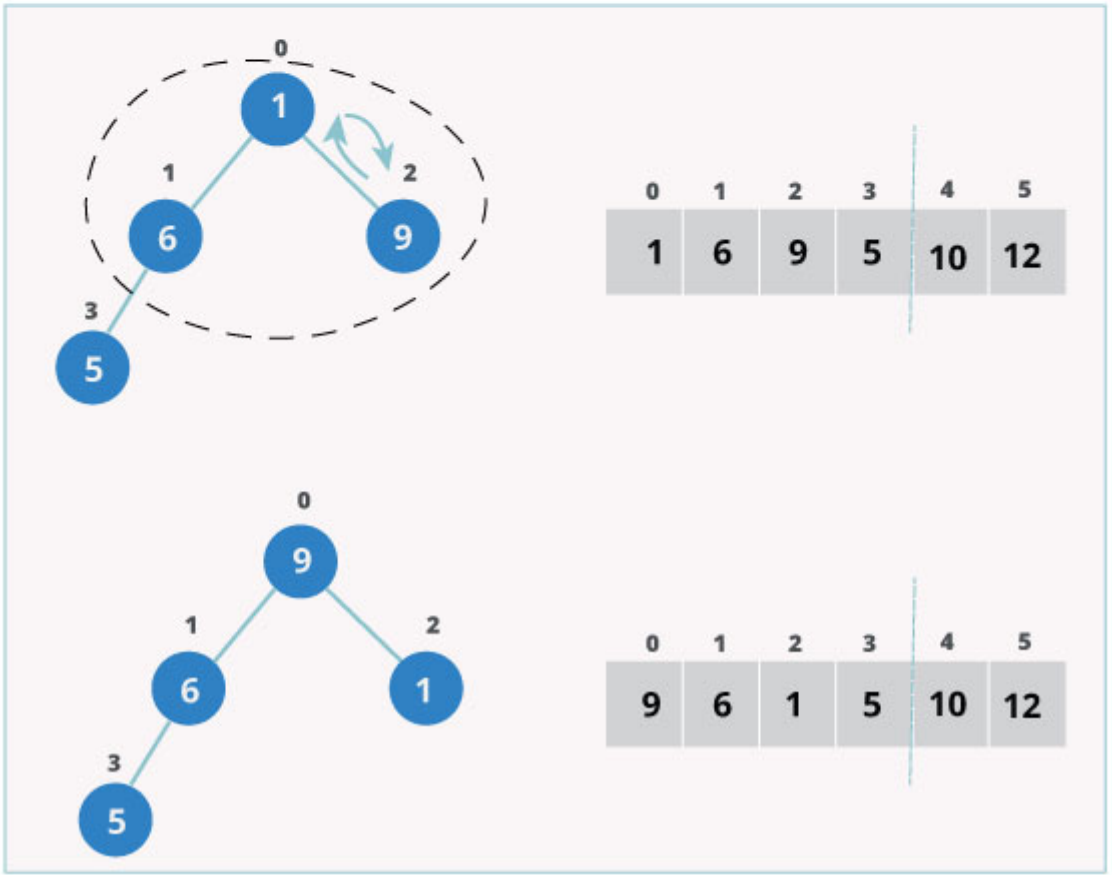


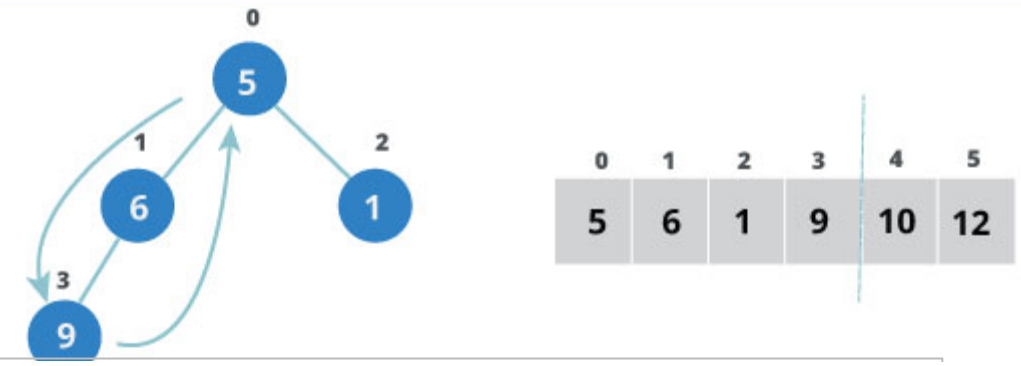
0	1	2	3	4	5
9	6	10	5	1	12

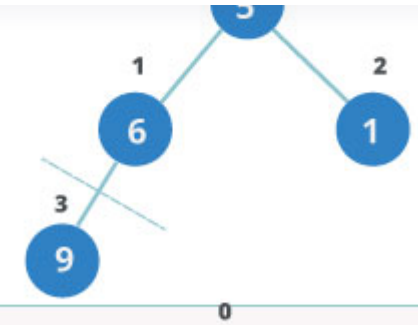
0	1	2	3	4	5
10	6	9	5	1	12

0

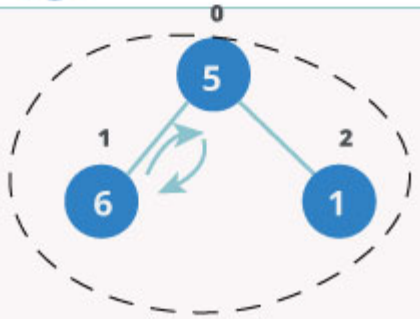




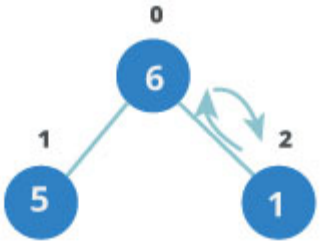




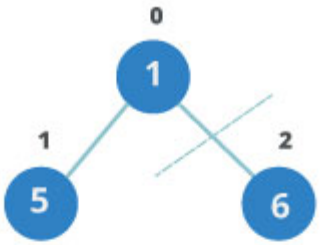
0	1	2	3	4	5
5	6	1	9	10	12



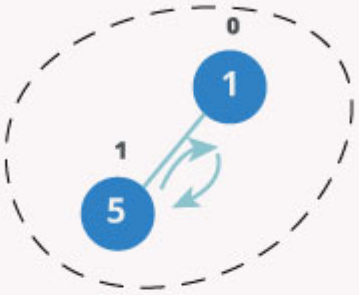
0	1	2	3	4	5
5	6	1	9	10	12



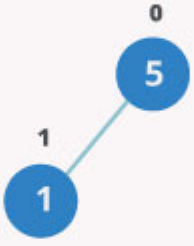
0	1	2	3	4	5
6	5	1	9	10	12



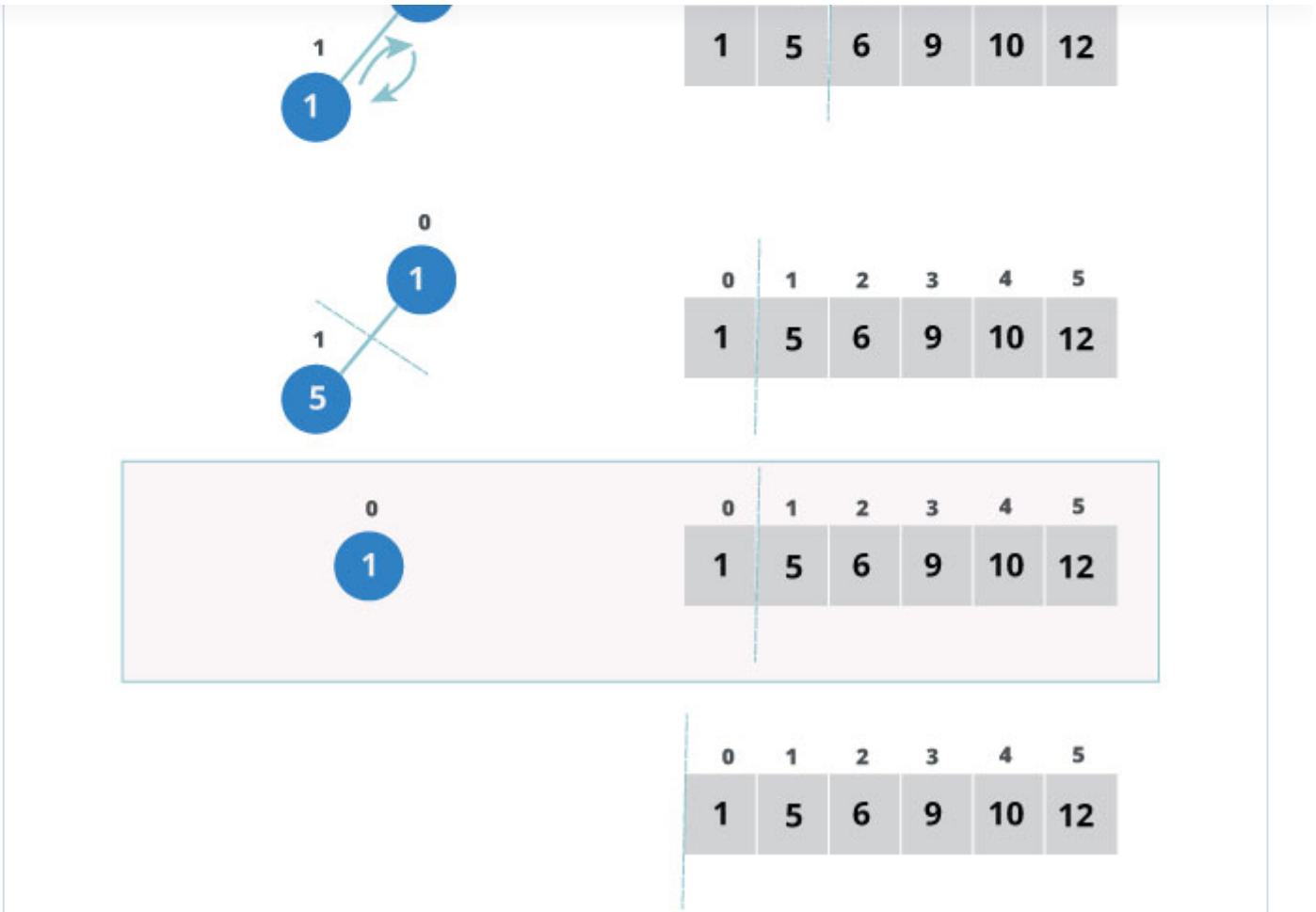
0	1	2	3	4	5
1	5	6	9	10	12



0	1	2	3	4	5
1	5	6	9	10	12



0	1	2	3	4	5
1	5	6	9	10	12



The code below shows the operation.

```
for (int i=n-1; i>=0; i--)
  {
    // Move current root to end
    swap(arr[0], arr[i]);

    // call max heapify on the reduced heap
    heapify(arr, i, 0);
  }
```

Performance

Heap Sort has `O(nlogn)` time complexities for all the cases (best case, average case and worst case).

As we have seen earlier, to fully heapify an element whose subtrees are already max-heaps, we need to keep comparing the element with its left and right children and pushing it downwards until it reaches a point where both its children are smaller than it.

In the worst case scenario, we will need to move an element from the root to the leaf node making a multiple of `log(n)` comparisons and swaps.

During the build_max_heap stage, we do that for `n/2` elements so the worst case complexity of the build_heap step is `n/2*log(n) ~ nlogn`.

During the sorting step, we exchange the root element with the last element and heapify the root element. For each element, this again takes `logn` worst time because we might have to bring the element all the way from the root to the leaf. Since we repeat this `n` times, the heap_sort step is also `nlogn`.

Also since the `build_max_heap` and `heap_sort` steps are executed one after another, the algorithmic complexity is not multiplied and it remains in the order of `nlogn`.

Also it performs sorting in `O(1)` space complexity. Comparing with Quick Sort, it has better worst case `( O(nlogn) )`. Quick Sort has complexity `O(n^2)` for worst case. But in other cases, Quick Sort is fast. Introsort is an alternative to heapsort that combines quicksort and heapsort to retain advantages of both: worst case speed of heapsort and average speed of quicksort.

Application of Heap Sort

Systems concerned with security and embedded system such as Linux Kernel uses Heap Sort because of the `O(n log n)` upper bound on Heapsort's running time and constant `O(1)` upper bound on its auxiliary storage.

Although Heap Sort has `O(n log n)` time complexity even for worst case, it doesn't have more applications (compared to other sorting algorithms like Quick Sort, Merge Sort). However, its underlying data structure, heap, can be

Priority Queues.

Heap Sort Implementation in different Programming Languages

C++ Implementation

```cpp
// C++ program for implementation of Heap Sort
#include <iostream>
using namespace std;

void heapify(int arr[], int n, int i)
{
   // Find largest among root, left child and right child
   int largest = i;
   int l = 2*i + 1;
   int r = 2*i + 2;

   if (l < n && arr[l] > arr[largest])
     largest = l;

   if (right < n && arr[r] > arr[largest])
     largest = r;

   // Swap and continue heapifying if root is not largest
   if (largest != i)
   {
     swap(arr[i], arr[largest]);
     heapify(arr, n, largest);
   }
}

// main function to do heap sort
void heapSort(int arr[], int n)
{
   // Build max heap
```

Java program for implementation of Heap Sort

```
{

    public void sort(int arr[])
    {
        int n = arr.length;

        // Build max heap
        for (int i = n / 2 - 1; i >= 0; i--) {
          heapify(arr, n, i);
        }


                // Heap sort
        for (int i=n-1; i>=0; i--)
        {
            int temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;

                // Heapify root element
            heapify(arr, i, 0);
        }
    }

    void heapify(int arr[], int n, int i)
    {
        // Find largest among root, left child and right child
```

Python program for implementation of heap sort (Python 3)

```
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2

    if l < n and arr[i] < arr[l]:
        largest = l

    if r < n and arr[largest] < arr[r]:
        largest = r

    # If root is not largest, swap with largest and continue heapifying
    if largest != i:
        arr[i],arr[largest] = arr[largest],arr[i]
        heapify(arr, n, largest)

def heapSort(arr):
    n = len(arr)

    # Build max heap
    for i in range(n, -1, -1):
        heapify(arr, n, i)


    for i in range(n-1, 0, -1):
        # swap
        arr[i], arr[0] = arr[0], arr[i]
```

Next Tutorial:
Bucket Sort (/dsa/bucket-sort)

Previous Tutorial:
Quicksort (/dsa/quick-sort)

Related Tutorials

Greedy Algorithm

(/dsa/greedy-algorithm)

DS & Algorithms

Types of Queue

(/dsa/types-of-queue)

DS & Algorithms

Deque

(/dsa/deque)

DS & Algorithms

Hashing

(/dsa/hashing)