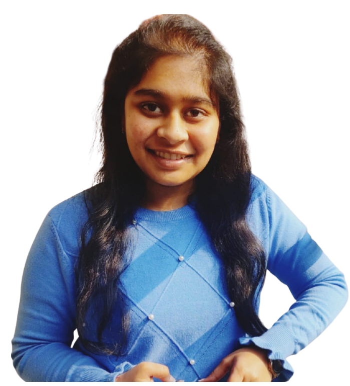
Geeks Man Algorithms Lesson 7

Trilok Kaushik

Founder of Geeksman

Aditya Aggarwal

Aditya Gupta



Suhani Mittal

Team Coordinators

Sorting (Part= III)

Need of Heap *??*

The way we provide input to the algorithm makes a lot of difference i.e if we provide unsorted array searching time would be O(n) and if we provide sorted array time complexity would be O(logn).

Suppose input data is in array form :-

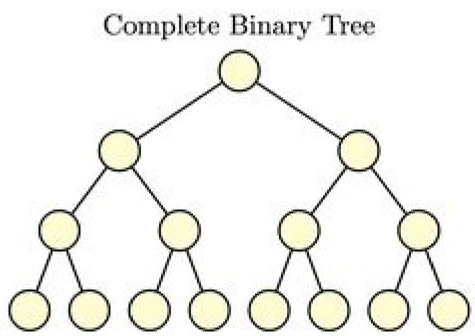
	Insertion	Search	Find min/max.	Delete
Unsorted Array	**O(1)**	**O(n)**	**O(n)**	**O(n)**
Sorted Array	**O(n)** [O(logn+n)]	**O(logn)**	**O(1)**	**O(n)**

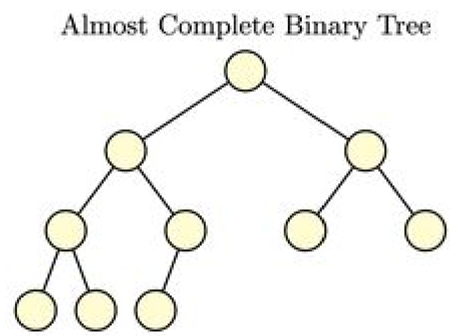
Suppose data is in other Data structure :-

	Insertion	*Search*	*Find Min/Max.*	*Delete*
LinkedList	**O(1)=unsort O(n)=sort**	**O(n)**	**O(n)**	**O(1+n) =O(n)**
Heap (Tree)	**O(logn)**	**O(n/logn)**	**O(1)**	**O(logn)**

Depending what type of operation we want we need Data structure which gives the optimal result of that operation.

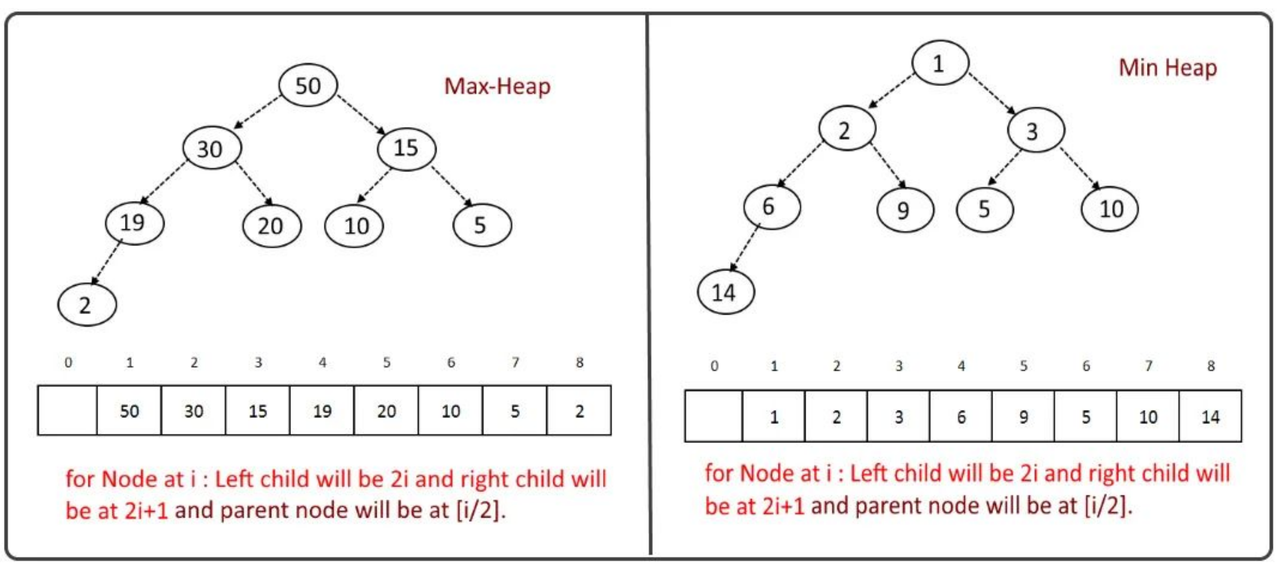
Heap :- It is an almost complete Binary Tree (Tree having nodes <=2 is called Binary Tree) .





2 Types of Heap :-

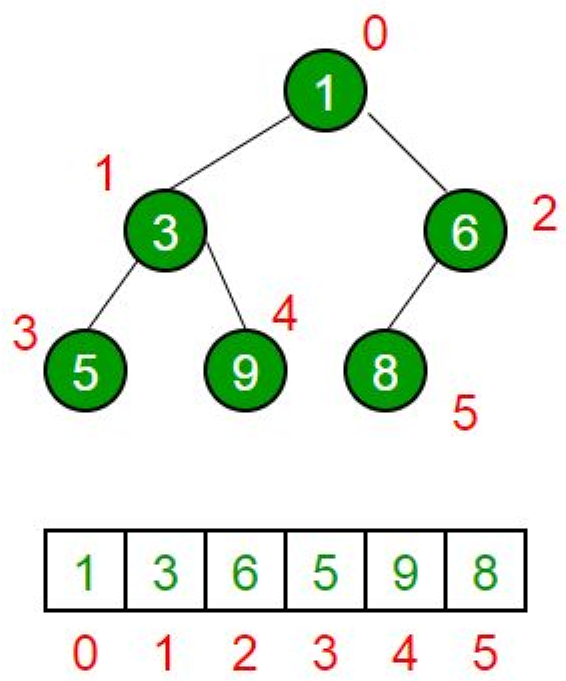
1. **Max Heap** = An almost complete binary tree in which root element is Maximum.
2. **Min Heap** = An almost complete Binary tree in which root Element is Minimum.



Implementation of Heap :-

- ➔ We will not construct heaps through linked lists to form trees as it will require lots of space in memory.

- ➔ We will store the elements in an array and interpret them in the form of a tree.



7. ***Heap Sort :-***

1. Building Max_heap :-

Code :

```
void max_heapify(int arr[] , int i) {
  int l, r;
  l = 2*i;
  r = 2*i+1;
   int largest =i;
 if ( l < arr.size && arr[l] > arr[largest] )
```

```
        largest=l;
if ( r < arr.size && arr[r] > arr[largest] )
        largest=r;
 if(largest != i ){
      swap(&arr[i],&arr[largest])
      max_heapify(arr,largest)
    }
 }
```

Time COmplexity :

    Total comparisons = 2 = O(1)
    Moving down in a tree = O(logn) [from root to leaf]

Therefore , total time by Max_heap = O(logn)

Space Complexity :

    Extra cells = O(1)
    Stack Space = no. of levels in a tree = O(logn)

Total Space Complexity = O(logn)

- ## Building max_heap over entire array

```
void build_maxheap(int arr[],int n)  {
      for(int k = n/2; k >= 1; k--)  {
       max_heapify(a,k,n);
  }
}
```

Time for building heap = O(n)

Therefore constructing heap over entire array = Sum of heapify at all heights from 0 to logn of a heap tree .

2. Extracting / Deleting Max Element :-

Code :

```
void deleteRoot(int arr[], int& n)
{
    // Get the last element
    int lastElement = arr[n - 1];

    // Replace root with first element
    arr[n-1] = arr[0];
    arr[0] = lastElement;

    // Decrease size of heap by 1
    n = n - 1;

    // heapify the root node
    max_heap(arr, n, 0);
}
```

Time Complexity of deletion = Time of building heap = O(logn)

3. Inserting an element :-

Code :

```
void insertNode(int arr[], int& n, int Key)
{
    // Increase the size of Heap by 1
    n = n + 1;

    // Insert the element at end of Heap
    arr[n - 1] = Key;

    // Heapify the new node following a
    // Bottom-up approach
```

```
    heapify(arr, n, n - 1);
}
```

Similarly , Time for inserting element = time for heapify = O(logn)

4. Heap Sort :-

Code :

```
void heapSort(int arr[], int n)
{
    // Build heap (rearrange array)
    build_max_heap(arr);
    // One by one extract an element from heap
    for (int i = n - 1; i >= 0; i--) {
        // Move current root to end
        swap(arr[0], arr[i]);
        arr.size--;
          // call max heapify on the reduced heap
          heapify(arr, i, 0);
    }
}
```

Time Complexity = O(nlogn)