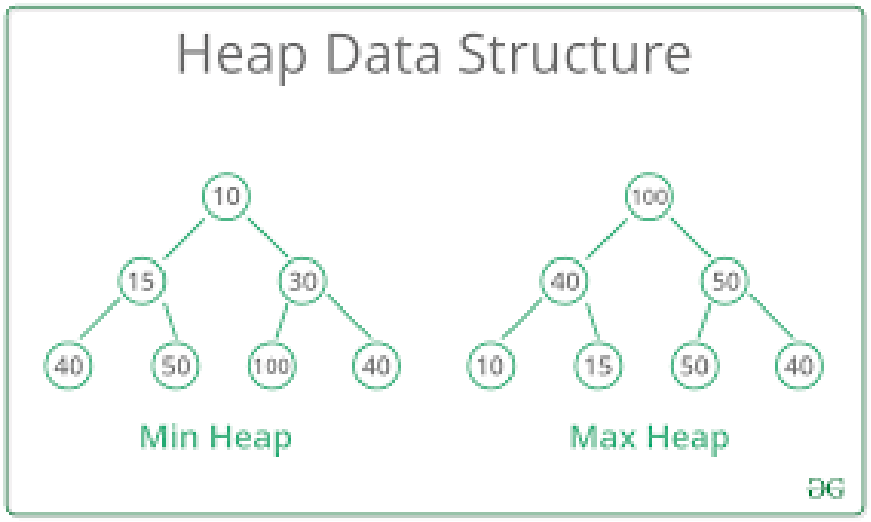
# Data Structure

# Lab #1

# Report





# Omar Mohamed Emam

# 44

## A Binary Heap is a Binary Tree with following properties.

1) It's a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible). This property of Binary Heap makes them suitable to be stored in an array.
2) A Binary Heap is either Min Heap or Max Heap. In a Min Binary Heap, the key at root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree. Max Binary Heap is similar to Min Heap

## INODE:

The class that contains the main methods for every element in the heap like get left get right, get parent, get value or set value
All methods is done through manipulating through the array of the type INODE and in collaboration with the class IHEAP

A Binary Heap is a Complete Binary Tree. A binary heap is typically represented as an array.

The root element will be at Arr[0].

| | |
|---|---|
| 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 |

## IHEAP:

Contains The Main Methods for the binary heap

### Build:

Simple Approach: Suppose, we need to build a Max-Heap from the above-given array elements. It can be clearly seen that the above complete binary tree formed does not follow the Heap property. So, the idea is to heapify the complete binary tree formed from the array in reverse level order following top-down approach.
That is first heapify, the last node in level order traversal of the tree, then heapify the second last node and so on.
Time Complexity: Heapify a single node takes O(Log N) time complexity where N is the total number of Nodes. Therefore, building the entire Heap will take N heapify operations and the total time complexity will be O(N*logN).

Optimized Approach: The above approach can be optimized by observing the fact that the leaf nodes need not to be *heapified* as they already follow the heap property. Also, the array representation of the complete binary tree contains the level order traversal of the tree.
So the idea is to find the position of the last non-leaf node and perform
the heapify operation of each non-leaf node in reverse level order.

## Insert:

Elements can be inserted to the heap following a similar approach as discussed above for deletion. The idea is to:

- First increase the heap size by 1, so that it can store the new element.
- Insert the new element at the end of the Heap.
- This newly inserted element may distort the properties of Heap for its parents. So, in order to keep the properties of Heap heapify this newly inserted element following a bottom-up approach.

# ISort:

Contains Implementation for:

## 1. Heap Sort :

1. Build a max heap from the input data.
2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.
3. Repeat above steps while size of heap is greater than 1.

Time Complexity: Time complexity of heapify is O(Log n). Time complexity of create And Build Heap is O(n) and overall time complexity of Heap Sort is O(n Log n).

## 2.Bubble Sort :

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.
Worst and Average Case Time Complexity: O(n*n). Worst case occurs when array is reverse sorted.
Best Case Time Complexity: O(n). Best case occurs when array is already sorted.
Auxiliary Space: O(1)

## 3.Merge Sort:

Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one

Time Complexity: Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.
$T(n) = 2T(n/2) + O(n)$

The above recurrence can be solved either using Recurrence Tree method or Master method. It falls in case II of Master Method and solution of the recurrence is O(nlogn)
.
Time complexity of Merge Sort is O(nlogn) in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and take linear time to merge two halves.
Auxiliary Space: O(n)

# Sample Run:

```
public static void main(String[] args)  {
    ArrayList test1 = new ArrayList<Integer>();
    test1.add(10);
    test1.add(5);
    test1.add(20);
    test1.add(7);
    test1.add(2);
    test1.add(-1);
    ArrayList test2 = new ArrayList<Integer>();
    ArrayList test3 = new ArrayList<Integer>();
    test3 = test2 = test1;
    Sort s = new Sort();
     long startTime = System.nanoTime();
    s.sortSlow(test1);
    long endTime = System.nanoTime();
    long duration1 = (endTime - startTime);
    for (int i = 0; i < test1.size(); i++) {
        System.out.println(test1.get(i));
    }
    System.out.println("Slow Sort time is " + duration1 + " ns");
    long startTime2 = System.nanoTime();
    s.sortFast(test2);
    long endTime2 = System.nanoTime();
    long duration2 = (endTime2 - startTime2);
    System.out.println("fast Sort time is "+duration2 + "  ns");
    long startTime3 = System.nanoTime();
    s.heapSort(test3);
    long endTime3 = System.nanoTime();
    long duration3 = (endTime3 - startTime3);
    System.out.println("Heap Sort time is "+ duration3+"  ns");
```

```
-1
2
5
7
10
20
Slow Sort time is 363438 ns
fast Sort time is 49350  ns
Heap Sort time is 306601203  ns
```