Yusrat Sadia Nailat

Heap - Theory

An ADT defines a set of operations and their behavior without specifying the underlying implementation details. For example, a Priority Queue ADT defines operations like `insert`, `deleteMin` (or `deleteMax`), and `peekMin` (or `peekMax`), but it doesn't dictate how these operations are carried out or how the data is stored.

Heap is an abstract data structure (ADT) because when we deal with heap, we do not specific the implementation method of heap.

Two types of Heap

1. Max Heap: the value of the root/parent is greater than the children nodes
2. Min Heap: the value of the root/heap is smaller than the the children nodes

The array representing a heap can be the following. The elements are placed on array **Level-wise**

0	1	2	3	4	5	6
111	112	113	114	115	116	117

We visualise heap using trees due to the simplicity of representation.

The left child is: $2i + 1$
The right child is: $2i + 2$

What is the Benefit of using ARRAY for Heap rather than Linked List?

Arrays have random access to each element by its index. On the other hand, Linked List is sequential, we need to traverse the linked list to find a particular element. Due to random access to its elements, array is used to implement heap. The time complexity for searching a particular element in array is $\mathcal{O}(1)$, on the other hand, for a Linked List with N number of elements, the worst time complexity is $\mathcal{O}(N)$

Yusrat Sadia Nailat

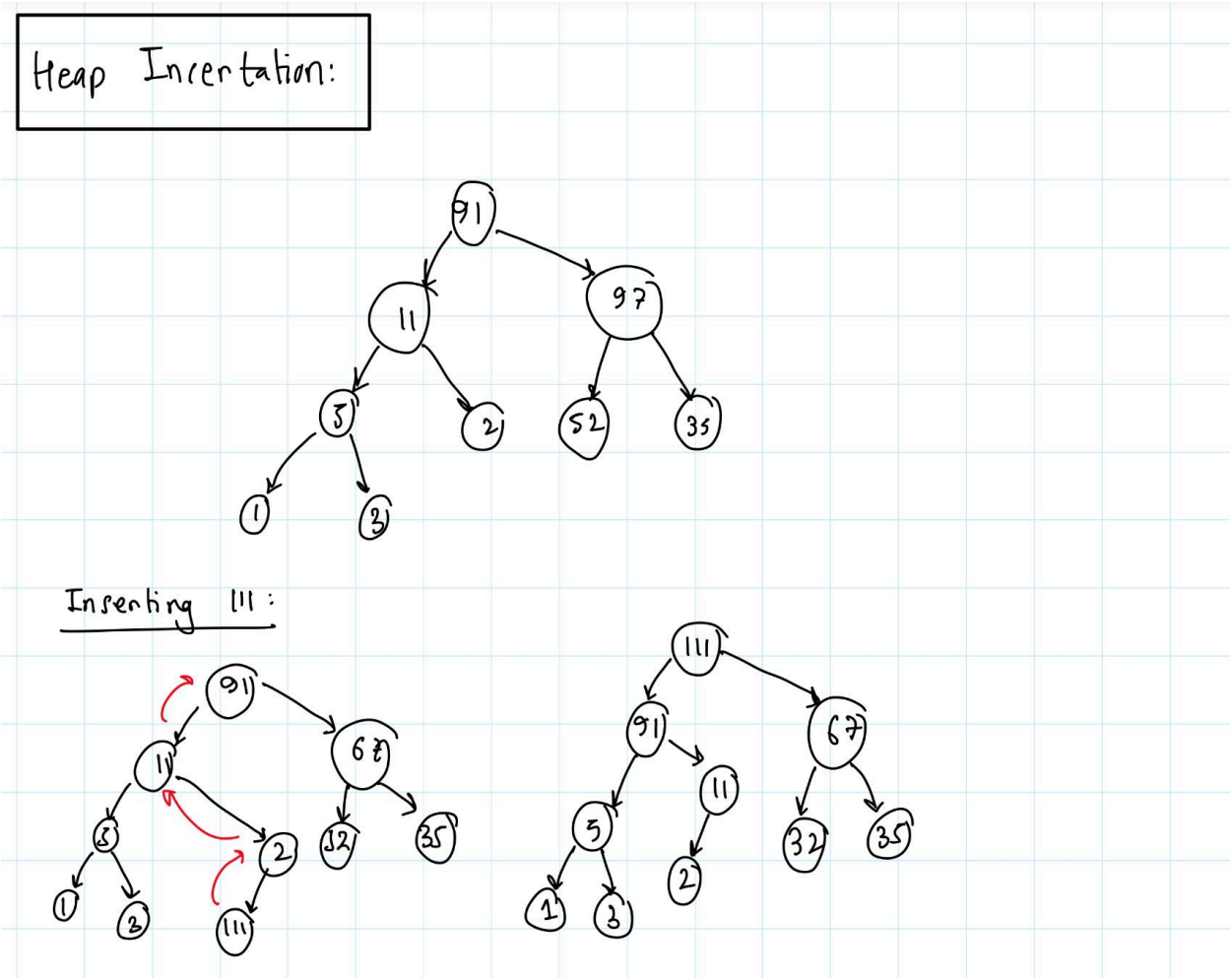
Insert on Heap

- The element is initially inserted into the **left most available position.** We start from the left available position to the right.
- The time complexity of insertion is $\mathcal{O}(1)$
- However, the heap structure may be broken. So we need to swim the element up to the root (which is the worst case). Since the height of a Binary Tree is $log(N)$, where N is the number of nodes in the BT.

The worst time complexity for insertion in Heap is

$$\mathcal{O}(1) + \mathcal{O}(logN) = \mathcal{O}(logN)$$

Simulation:



Yusrat Sadia Nailat

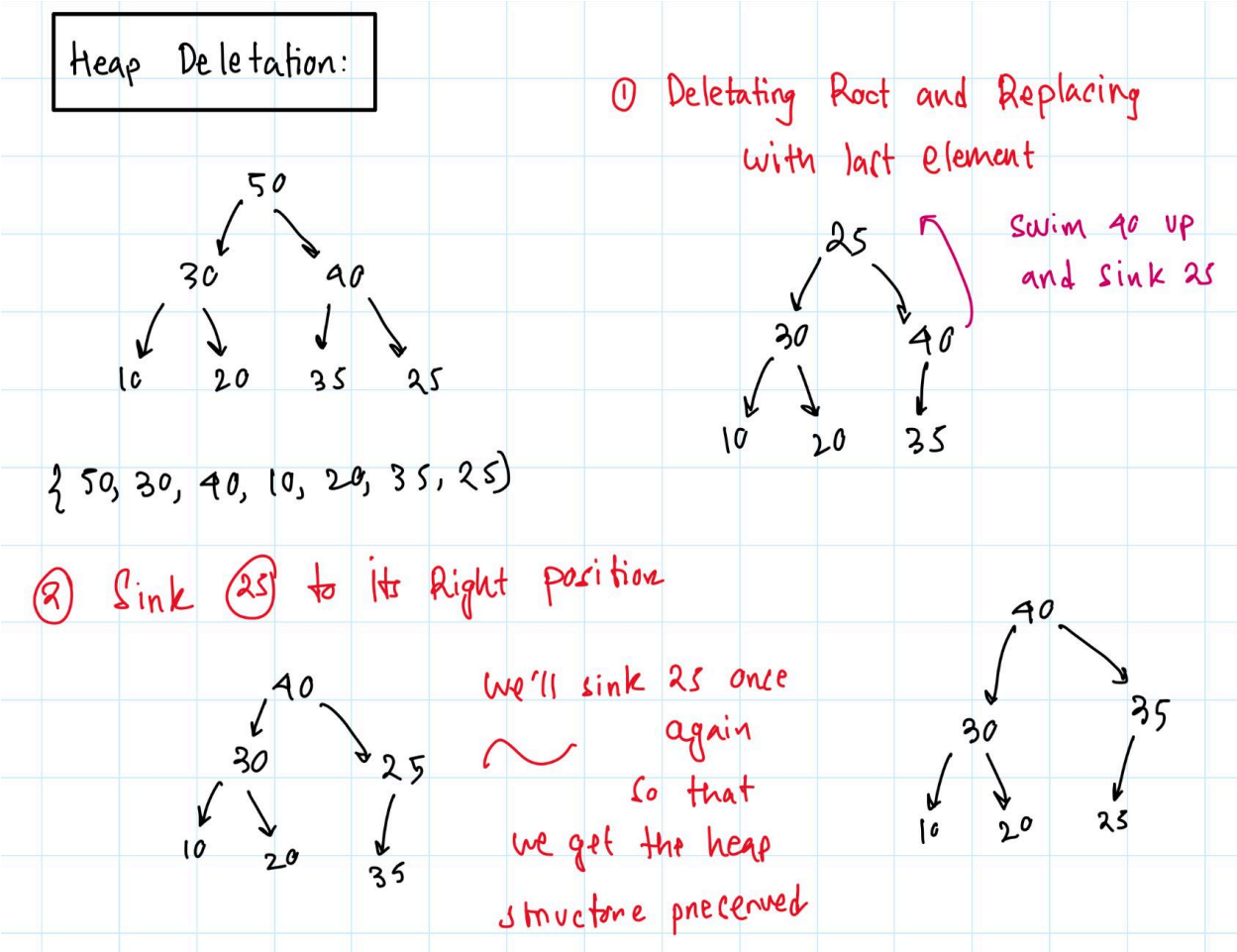
Delete on Heap

- For heap, we can only access the root element, so we delete the root element first
- We replace the root now with the **last element of the given array (heap)**
- **Deleting the root will always break the heap structure.**
- We sink the root element in the right position to maintain the heap structure.

- The time complexity of Deletion is $\mathcal{O}(1)$
- We need to sink the element down to the root (which is the worst case). Since the height of a Binary Tree is $log(N)$, where N is the number of nodes in the BT. The time complexity for sink is $\mathcal{O}(log(N))$.

The worst time complexity for deletion in Heap is

$$\mathcal{O}(1) + \mathcal{O}(logN) = \mathcal{O}(logN)$$



Yusrat Sadia Nailat

Heap Sort

- The time complexity for deleting each element is $\mathcal{O}(logN)$
- So, the time complexity for deleting N elements from a heap is $\mathcal{O}(NlogN)$

Theoretical Questions for Exam:

A heap is ALWAYS a COMPLETE and BALANCED binary tree. By convention, the heap is balanced and since the leaf nodes are filled from left to right and the internal nodes are all filled it is always a complete binary tree. The advantages of these properties are:

1. It guarantees logarithmic height for efficiency.
2. It allows array-based storage without gaps.
3. It makes heap operations simple and consistent.

What is maximum number of swaps needed to heapify a single value in a heap of height h ?

Answer: the worst case is that we are heapifying the root of the heap to the leaf node of the heap. Since the height of the heap is H. Then you will need at most H swaps heapify that value.

What is the maximum number of nodes in a Heap of height H ?

Answer: $2^{H+1} - 1$

Simulation:

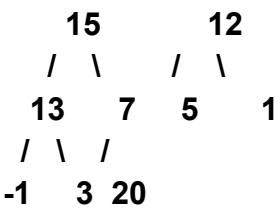
IV. Heap = [N, 19, 15, 12, 13, 7, 5, 1, -1, 3] This array indicates a maxHeap. Value 20 is inserted in the heap. What will be the index of the value 7 in the new heap?

a. 4, b. 5, c. 9, d. 10

Inserting 20 on the left most available position

19
**/ **

Yusrat Sadia Nailat



Heapifying, we get