REFRESHER ON DATA STRUCTURES

Keerat Kaur Guliani
Machine Learning Operations, TUSK
UNIVERSITY OF TORONTO

AGENDA

Heaps

Hash Tables

Binary Search Tree

For every data structure:

Basics

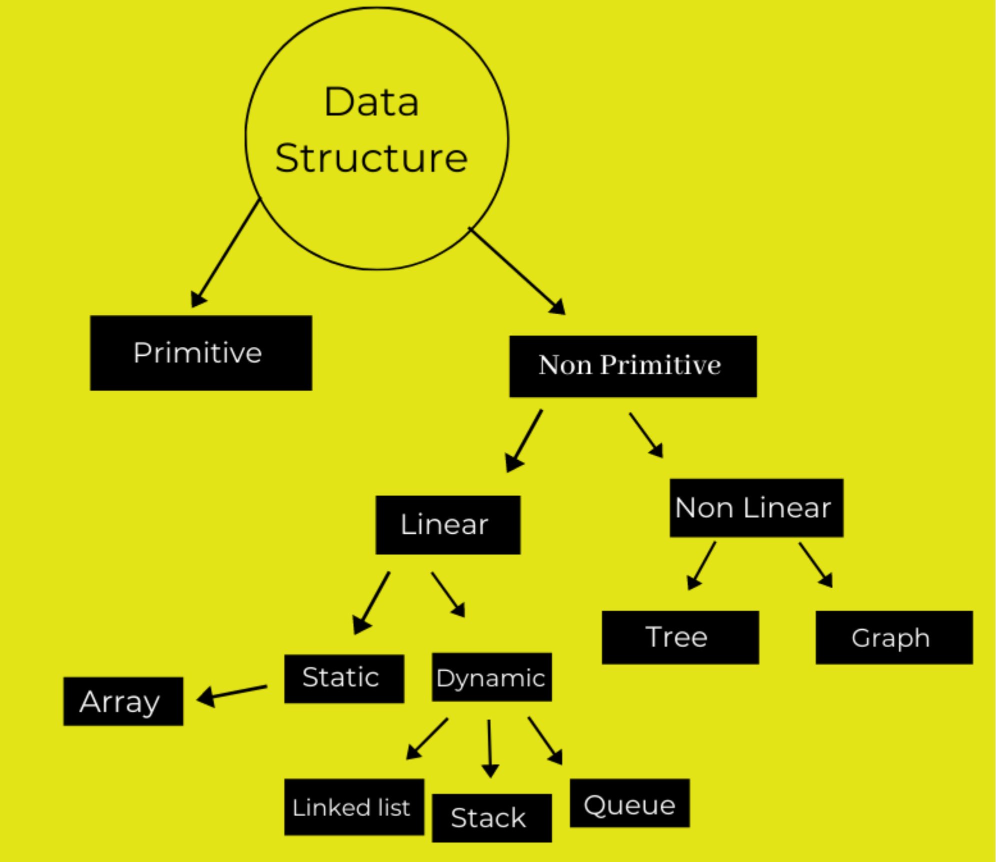
Application

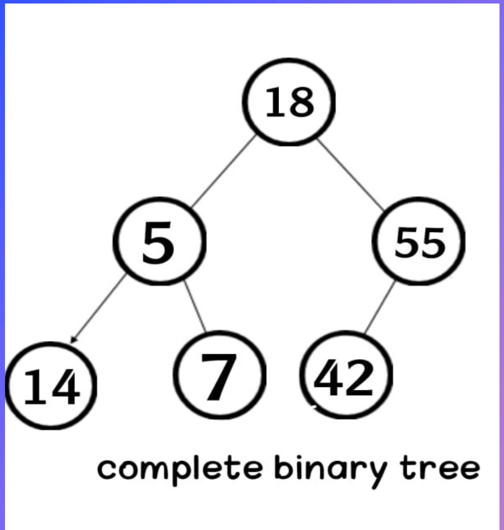
Operations

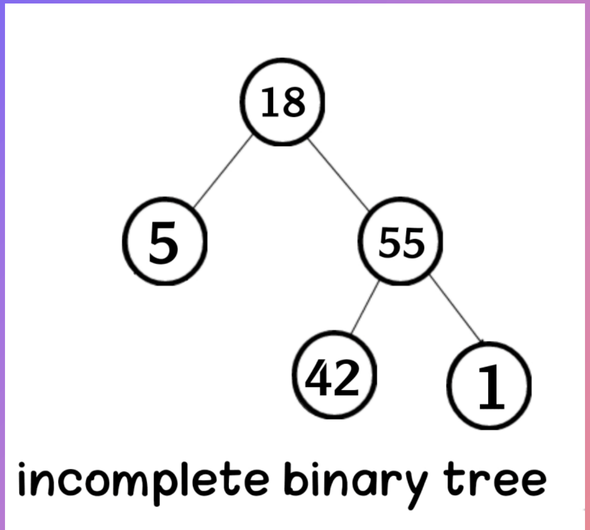
Code Walkthrough

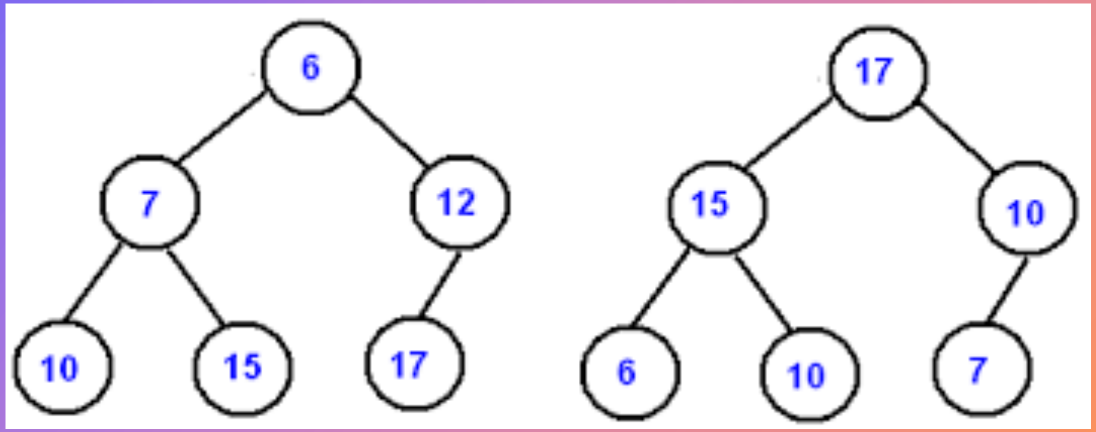
What are data structures and why do we need them?

- Format that stores data values and the relationship between them.
- Used for organizing data in memory.
- Provides efficiency, reusability and abstraction.









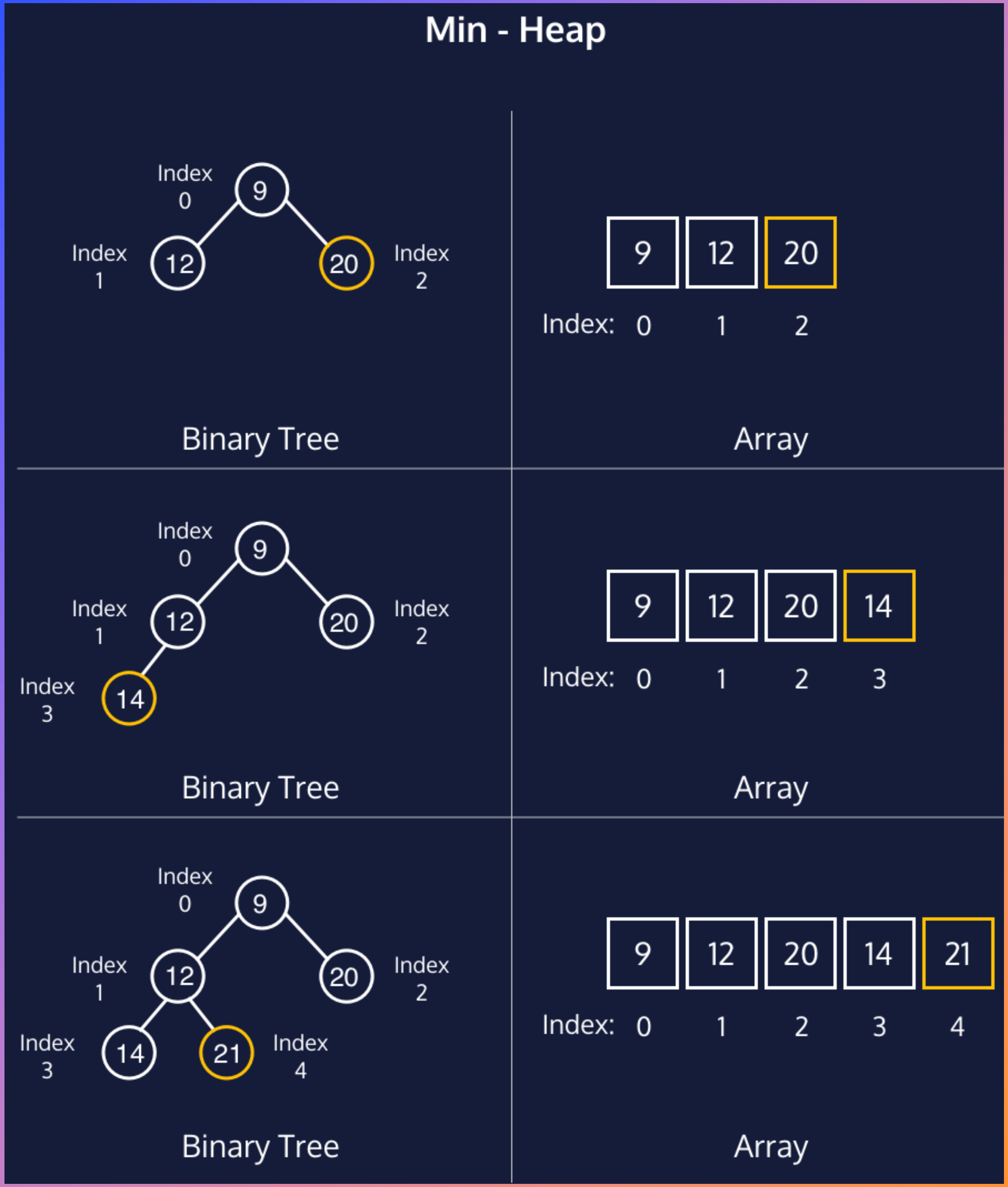
Identify which is which?

Heaps

A **heap** is a specialized tree-based data structure which is essentially a complete tree that satisfies the **heap property**:

Min heap: Parent ≤ Children

Max heap: Parent ≥ Children



Applications of Heaps

1. Priority Queue
2. Heap Sort
3. Getting The minimum value or the maximum value in a constant time

Operations with Heaps

Heapify

Process to rearrange the elements of the heap in order to maintain the heap property. It is done when a certain node causes an imbalance in the heap due to some operation on that node.

Complexity: ?

Find Max/Min

Simply returns the max/min element found at the root node of the heap.

Complexity: ?

Extract Min/Max

Returns and deletes the maximum or minimum element in max-heap and min-heap respectively.

Complexity: ?

Operations with Heaps

Insertion

The insertion in the heap follows the following steps:

- Insert the new element at the end of the heap
- Since the newly inserted element can distort the properties of the heap, perform heapify

Complexity: **?**

Deletion

The deletion operations follow the following step:

- Replace the element to be deleted by the last element in the heap.
- Delete the last item from the heap.
- Now, the last element is placed at some position in heap, it may not follow the property of the heap, so we need to perform heapify to correct that

Complexity: **?**

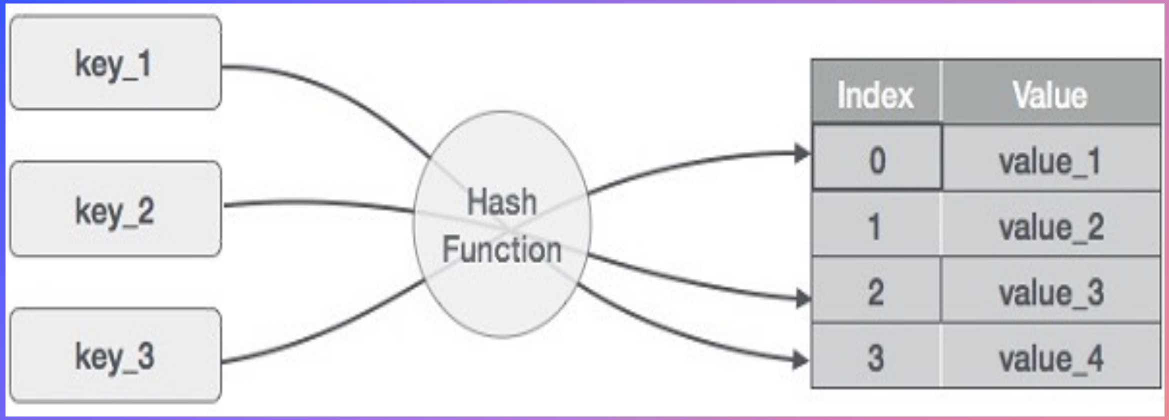
Complexities

OPERATION	TIME COMPLEXITY		SPACE COMPLEXITY
Insertion	Best Case:	O(1)	O(1)
	Worst Case:	O(logN)	
	Average Case:	O(logN)	
Deletion	Best Case:	O(1)	O(1)
	Worst Case:	O(logN)	
	Average Case:	O(logN)	
Searching	Best Case:	O(1)	O(1)
	Worst Case:	O(N)	
	Average Case:	O(N)	
Max Value	In MaxHeap:	O(1)	O(1)
	In MinHeap:	O(N)	
Min Value	In MinHeap:	O(1)	O(1)
	In MaxHeap:	O(N)	
Sorting	All Cases:	O(NlogN)	O(1)
Creating a Heap	By Inserting all elements:	O(NlogN)	O(N)
	Using Heapify	O(N)	O(1)

IMPLEMENTATION

https://colab.research.google.com/drive/1nlxGhyvakdG4ZvEJP3ZvUMufQHQlgn4b?usp=sharing

Hash Table



Sr.No.	Key	Hash	Array Index
1	1	1 % 20 = 1	1
2	2	2 % 20 = 2	2
3	42	42 % 20 = 2	2
4	4	4 % 20 = 4	4
5	12	12 % 20 = 12	12
6	14	14 % 20 = 14	14
7	17	17 % 20 = 17	17
8	13	13 % 20 = 13	13
9	37	37 % 20 = 17	17

- Stores data in an associative manner
- Storage medium: Array; Each data value has its own unique index value. Hash technique is used to generate this index value
- Major +: If we know the index of the desired data, search/insert become very fast irrespective of the size of the data.
- In Python: Dictionary data type represents implementation of hash tables.

Collision Resolution

Linear Probing

			×							
key	0	1	2	3	4	5	6	7	8	9
value		1231	5982				4536	4637		8359
			$\boxed{\uparrow}$							
			2392							

				✔						
key	0	1	2	3	4	5	6	7	8	9
value		1231	5982				4536	4637		8359
				$\boxed{\uparrow}$						
				2392						

Chaining

key	0	1	2	3	4	5	6	7	8	9
value		1231	5982				4536	4637		8359
			↓							
			2392							

key	0	1	2	3	4	5	6	7	8	9
value		1231	5982				4536	4637		8359
			↓				↓			
			2392				8986			
			↓							
			8762							

Applications of Hash Tables

- Password verification
- Plagiarism / Pattern Matching
- Compilers

Operations with Hash Tables

Searching

The insertion in the hash table follows the following steps:

- Compute the hash code of the key passed and locate the element using that hash code as index in the array.
- Use linear probing to get the element ahead if the element is not found at the computed hash code.

Complexity: **?**

Insertion/Deletion

The deletion operations follow the following step:

- Compute the hash code of the key passed and locate the element using that hash code as index in the array.
- (Insertion) Use linear probing for empty location, if an element is found at the computed hash code.
- (Deletion) Use linear probing to get the element ahead if an element is not found at the computed hash code. When found, store a dummy item there to keep the performance of the hash table intact.

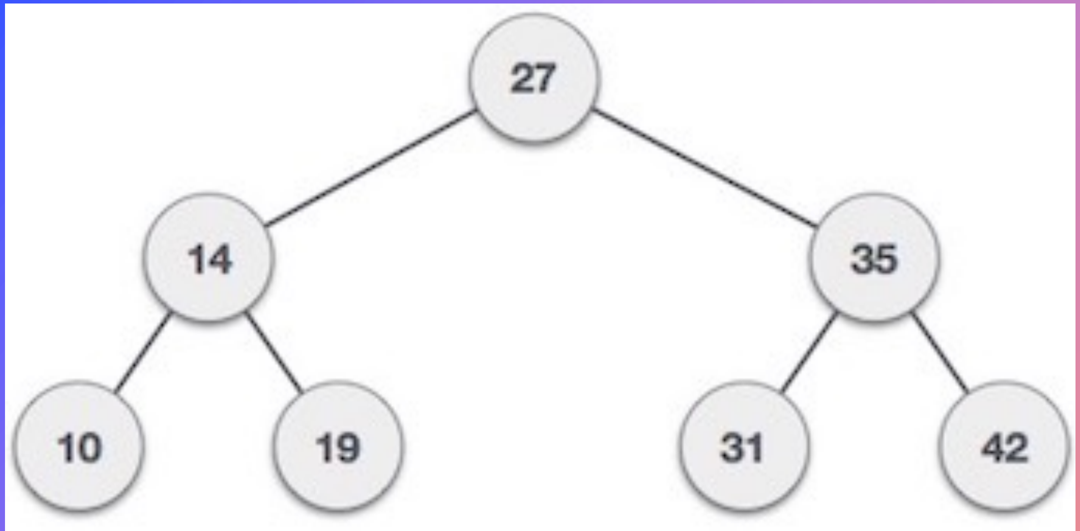
Complexity: **?**

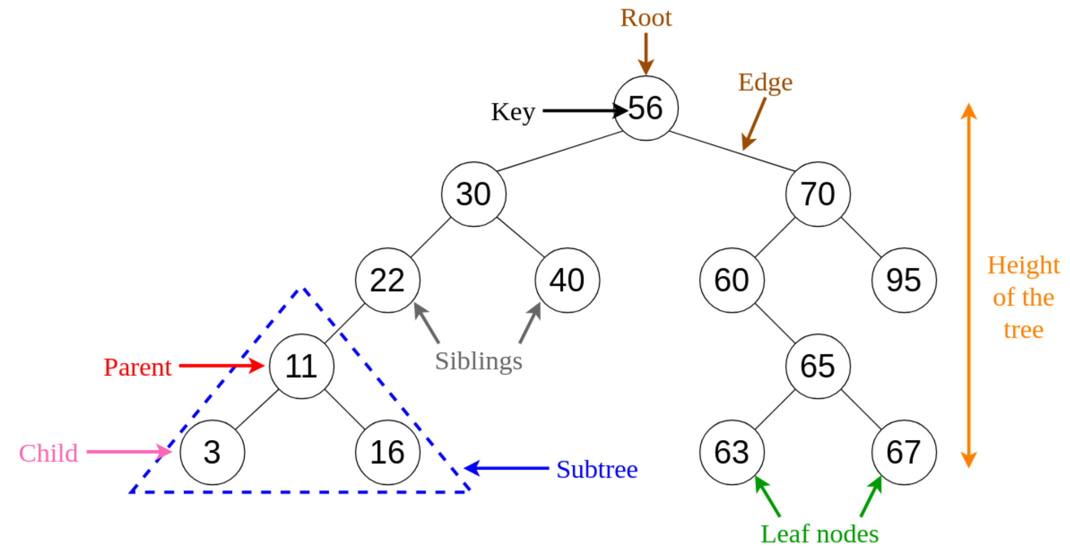
Complexities

ACTIVITY	BEST CASE COMPLEXITY	AVERAGE CASE COMPLEXITY	WORST CASE COMPLEXITY
Searching	O(1)	O(1)	O(n)
Insertion	O(1)	O(1)	O(n)
Deletion	O(1)	O(1)	O(n)
Space Complexity	O(n)	O(n)	O(n)

IMPLEMENTATION

https://drive.google.com/file/d/17ENICJW81scufGd6sSb0nVhN6X2_tZNC/view?usp=sharing





Binary Search Tree

Tree:

- Each tree has a root node at the top (also known as Parent Node) containing some value (can be any datatype).
- The root node has zero or more child nodes.
- Each child node has zero or more child nodes, and so on. This creates a subtree in the tree. Every node has its own subtree made up of its children and their children, etc. This means that every node on its own can be a tree.

Additionally, in a BST:

- Each node has a maximum of up to two children.
- left_subtree (keys) < node (key) ≤ right_subtree (keys)

Applications of BSTs

- **Dictionary and spell checker applications:** BSTs can be used to store dictionaries, where searching for a word is reduced to searching a BST.
- **Database indexing:** BSTs can be used as an index to quickly search for specific values in large databases.
- **Priority Queues**: BSTs can be used to implement priority queues, where elements with the highest or lowest priority can be efficiently retrieved.
- **Graph algorithms:** BSTs can be used to efficiently implement graph algorithms like Dijkstra's shortest path algorithm.

Traversal in BSTs

Pre-Order

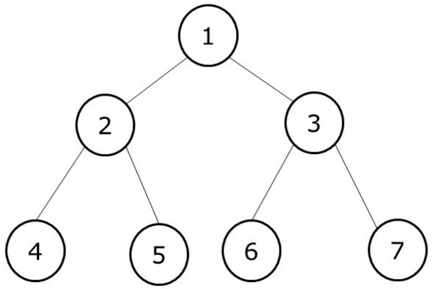
The root node is visited last, hence the name. First, we traverse the left subtree, then the right subtree and finally the root node.

Post-Order

The root node is visited first, then the left subtree and finally the right subtree.

In-Order

The left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.



Preorder	[1,2,4,5,3,6,7]
Inorder	[4,2,5,1,6,3,7]
Postorder	[4,5,2,6,7,3,1]

Operations with BSTs

Searching

- Whenever an element is to be searched, start searching from the root node.
- Then if the data is less than the key value, search for the element in the left subtree.
- Otherwise, search for the element in the right subtree.
- Follow the same algorithm for each node.

Complexity: ?

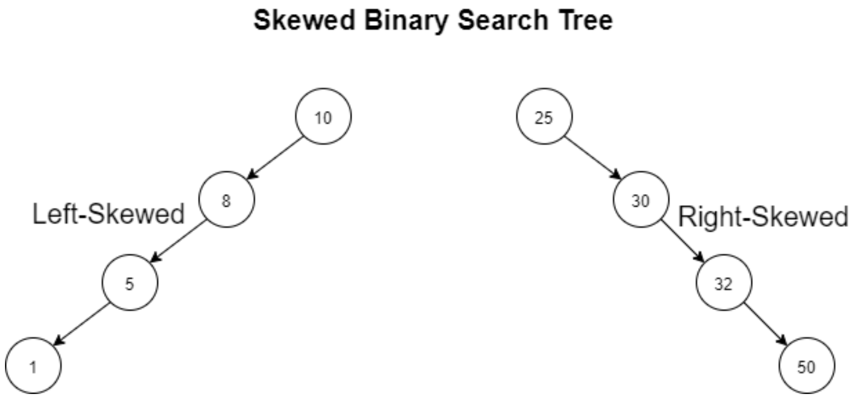
Insertion/Deletion

- Start searching from the root node.
- Then if the data is less than the key value, search for the (empty if inserting) location in the left subtree and insert/delete the data.
- Otherwise, search for the (empty if inserting) location in the right subtree and insert/delete the data.

Complexity: ?

Complexities

OPERATION	WORST CASE	AVERAGE CASE	BEST CASE	SPACE
Search	O(N)	O(logN)	O(1)	O(N)
Insert	O(N)	O(logN)	O(1)	O(N)
Delete	O(N)	O(logN)	O(N)	O(N)



IMPLEMENTATION

https://colab.research.google.com/drive/1RC3ybT-gLdlTq79ucm__VJqi126iU5Ls?usp=sharing

QUESTIONS?