Big – O

From Lists:

- Linked List: insert– **O(1)**; remove – **O(n)**.
- Array List: insert – **O(n)**; remove – **O(n)**.
- All other operations have time complexity **O(1)**.
- Double linked list: operations insert – **O(1)**.
- remove – **O(1)**.

Operation	Array List	Linked List	Double-Linked List
Empty	O(1)	O(1)	O(1)
Last	O(1)	O(1)	O(1)
Full	O(1)	O(1)	O(1)
FindFirst	O(1)	O(1)	O(1)
FindNext	O(1)	O(1)	O(1)
FindPrevious	-	-	O(1)
Retrieve	O(1)	O(1)	O(1)
Update	O(1)	O(1)	O(1)
Insert	O(n)	O(1)	O(1)
Remove	O(n)	O(n)	O(1)

From Queue:

- Linked List: Enqueue is O(n), Serve is O(1).
- Array Implementation: Enqueue is O(1), Serve is O(1).
- Heap: Enqueue is O(log n), Serve is O(log n).

• Operation	• Queue (LL)	• Queue (CA)	• Priority Queue (LL)	• Priority Queue (CA)
• **Full**	• O(1)	• O(1)	• O(1)	• O(1)
• **Length**	• O(1)	• O(1)	• O(1)	• O(1)
• **Enqueue**	• O(1)	• O(1)	• O(n)	• O(n)
• **Serve**	• O(1)	• O(1)	• O(1)	• O(1)

From Stack:

- All operations are O(1) (worst and best)

From Binary Trees:

- FindKey: find an element of a particular key value in a binary tree.

- In binary tree this operation is O(n).

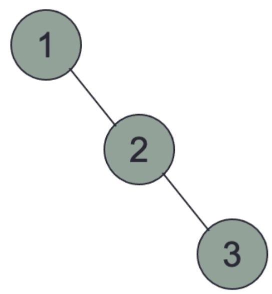
Methods not mentioned here are O(1).

method	Best case	Worst case
traverse	O(n)	O(n)
preorder	O(n)	O(n)
In order	O(n)	O(n)
postorder	O(n)	O(n)
FindParent	O(1)	O(n)
find	O(1)	O(n)
deleteSubtree	O(1)	O(n)

From Binary Search Trees:

- **In a Binary Search Tree (BST) findkey operation can be performed very efficiently: $O(\log_2 n)$.**

- Consider a situation when data elements are inserted in a BST in sorted order: 1, 2, 3, …



- BST becomes a degenerate tree.
- Search operation **FindKey** takes **O(n)**, which is as inefficient as in a list.

In remove, insert, and findKey:
Best: O(1)
Average: O(logn)
Worst: O(n)

From AVL Trees:

- insert and delete elements so that its height is guaranteed to be O(logn).

- Important operation *Findkey*() can be implemented in O(logn) time.

From Hash:

An important operation Findkey() has a time complexity:
O(n) in Lists,
O(n) in Binary Trees,
O(log n) in BSTs,
O(log n) in AVL trees.
Can Findkey() be implemented with a time complexity better than O(log n)?
With Hash Tables it is possible to implement Findkey() with O(1) time complexity.

From Heaps:

- Since a heap has height $O(\log n)$, upheap runs in $O(\log n)$ time
- Consider a priority queue with n items implemented by means of a heap
- the space used is $O(n)$
- methods enqueue and serve take $O(\log n)$ time
- methods length, full take time $O(1)$ time
- Using a heap-based priority queue, we can sort a sequence of n elements in $O(n \log n)$ time

	Running time
siftUp (upHeap)	O(log n)
siftDown (downHeap)	O(log n)
enqueue in heap priority queue	O(log n)
serve() in heap priority queue	O(log n)
Bottom-up construction of a heap	O(n)
Heap sort	O(n log n)

From Graphs:

- Adjacency Matrix representation is very simple, but the space requirement is $O(n^2)$ if the number of vertices is n.

- In an Adjacency list the space requirement is O(e + n) where e is the number of edges and n is the number of vertices.