

Data Structures				
	Operation	Time - Average Case	Time - Worst Case	
Array	Search*	$O(N)$	$O(N)$	If we are using a resizable array and it is currently full, inserting at the end can have $O(N)$: Duplicating array's size and copying N elements to it.
	Insert at first position	$O(N)$	$O(N)$	
	Insert at the end	$O(1)$	$O(1)$	
	Remove first position	$O(N)$	$O(N)$	
	Remove at the end	$O(1)$	$O(1)$	
	Update**	$O(1)$	$O(1)$	
Linked List	Search*	$O(N)$	$O(N)$	
	Insert at first position	$O(1)$	$O(1)$	
	Insert at the end	$O(N)$	$O(N)$	
	Remove first position	$O(1)$	$O(1)$	
	Remove at the end	$O(N)$	$O(N)$	
	Update**	$O(N)$	$O(N)$	
Hash Table	Search*	$O(1)$	$O(N)$	Performance depends on the choice of hash function and hash size
	Insert	$O(1)$	$O(N)$	
	Remove	$O(1)$	$O(N)$	
	Update**	$O(1)$	$O(N)$	
Unbalanced Binary Search Tree	Search*	$O(\log(N))$	$O(N)$	Could also use notation $O(h)$ for Average Case, with h being the tree's height ($=\log(N)$)
	Insert	$O(\log(N))$	$O(N)$	
	Remove	$O(\log(N))$	$O(N)$	
	Update**	$O(\log(N))$	$O(N)$	
Balanced Binary Search Tree	Search*	$O(\log(N))$	$O(\log(N))$	Rotations in a AVL tree are $O(1)$, so they don't impact overall complexity
	Insert	$O(\log(N))$	$O(\log(N))$	
	Remove	$O(\log(N))$	$O(\log(N))$	
	Update**	$O(\log(N))$	$O(\log(N))$	
Min/Max Heap	Search*	$O(N)$	$O(N)$	Inserting is just putting it in the rightmost position, except if it is the min/max value (worst case)
	Insert	$O(1)$	$O(\log(N))$	
	Remove	$O(\log(N))$	$O(\log(N))$	
	Peak	$O(1)$	$O(1)$	

Stack***	Push	O(1)	O(1)	
	Pop	O(1)	O(1)	
	Peak	O(1)	O(1)	
	Is Empty	O(1)	O(1)	
Queue****	Add	O(1)	O(1)	
	Remove	O(1)	O(1)	
	Peek	O(1)	O(1)	
	Is Empty	O(1)	O(1)	

*Search = Suppose we are searching for an element which value is V, we want to find its position. *int search(V)*

**Update = Suppose we want to update an element which is in position P. *void update(P)*

***Stack = Implemented as an array (doesn't shrink or grow well because you have to provide a max size) or linked list

****Queue = Implemented as a linked list and saving two pointers (front and rear). Implementing as an array would provide worse time complexity.

Algorithms			
	Time - Average Case	Time - Worst Case	
Breadth-First Search	O(V+E)	O(V+E)	V = Nb of Vertices E = Nb of Edges
Depth-First Search	O(V+E)	O(V+E)	
Binary Search	O(log(N))	O(log(N))	
Quick Sort	O(N*log(N))	O(N ²)	
Merge Sort	O(N*log(N))	O(N*log(N))	

Python Functions				
	Operation	Time - Average Case	Time - Worst Case	
List	Get	$O(1)$	$O(1)$	
	Append	$O(1)$	$O(N)$	
	Insert	$O(N)$	$O(N)$	
	Del	$O(N)$	$O(N)$	
	Find	$O(N)$	$O(N)$	
	Min/Max	$O(N)$	$O(N)$	
Dict	Insert	$O(1)$	$O(N)$	Hash table with linear probing
	Get	$O(1)$	$O(N)$	
	Pop	$O(1)$	$O(N)$	
Set	Checking if item is in set	$O(1)$	$O(N)$	Set implements a hash table
	Add	$O(1)$	$O(N)$	
	Union	$O(\text{len}(s1) + \text{len}(s2))$	$O(\text{len}(s1) + \text{len}(s2))$	
	Intersection	$O(\min(\text{len}(s1), \text{len}(s2)))$	$O(\min(\text{len}(s1), \text{len}(s2)))$	
Join	-	$O(N)$	$O(N)$	
Split	-	$O(N)$	$O(N)$	
Strip	-	$O(N)$	$O(N)$	
Sort	-	$O(N \cdot \log(N))$	$O(N \cdot \log(N))$	Implements Timsort, hybrid of Mergesort and Insertionsort