

Chapter 7

Time & Space Complexity

Not just Big-O — understand WHY each operation has its complexity

Stack — Complexity Table

Case	Time	Space	Reason
push(x)	O(1)	O(1)	Direct array index / linked list head insert
pop()	O(1)	O(1)	Direct array index / linked list head removal
peek()	O(1)	O(1)	Read arr[top] or head.data — no traversal
isEmpty())	O(1)	O(1)	Check top == -1 or head == null
Search(x)	O(n)	O(1)	Must traverse entire stack — no random access
Space (n items)	—	O(n)	n elements stored in array or n nodes

Queue — Complexity Table

Case	Time	Space	Reason
enqueue (x)	O(1)	O(1)	Insert at rear — direct index (circular) or tail insert
dequeue)	O(1)	O(1)	Remove from front — direct index or head removal

peek()	O(1)	O(1)	Read arr[front] or head.data
isEmpty())	O(1)	O(1)	Check size == 0
Search(x)	O(n)	O(1)	Must scan all elements from front to rear
Space (n items)	—	O(n)	n elements stored

■ Special Cases — Amortized Analysis

Queue using Two Stacks (LeetCode 232): • enqueue: always $O(1)$ — just push to Stack1 • dequeue: $O(1)$ amortized — pour Stack1 into Stack2 only when Stack2 is empty WHY amortized? Each element is moved from S1 to S2 exactly once across its lifetime. Total cost for n operations = $O(n)$. Per operation average = $O(1)$. One expensive dequeue pays for many cheap ones.

■ Recursion + Stack Complexity

Every recursive function implicitly uses the call stack. Space complexity of recursion = depth of recursion tree.

Pattern	Recursion Depth	Stack Space
Linear recursion (factorial)	n	O(n)
Binary recursion (merge sort)	log n	O(log n)
Tree DFS traversal	height h	O(h)
Iterative with explicit stack	controlled	O(n) worst