Big-O
Time Complexities

Typical average-case complexities with **good hash functions** and **balanced trees**.

`Java Collections` `Python Built-ins` `JS Standard Objects` `Performance`

Advanced Programming tutorial by Dr. Nabajyoti Medhi, PhD, CSE

What We'll Cover

Analysis Scope

We focus on **average-case** time complexities for standard implementations. We assume **good hash functions** (minimizing collisions) and **balanced trees** (like Red-Black trees) where applicable.

Data Structures

Comparison covers four main categories across languages:

- Lists / Sequences
- Sets & Uniqueness
- Maps / Dictionaries
- Queues & Deques

Notation Key:

```
n = container size
m = size of other collection
k = slice size / subset
```

01 Java Collections
ArrayList, LinkedList, HashMap, TreeMap, Sets

02 Python Built-ins
CPython implementations: list, dict, set, tuple

03 JavaScript Structures
V8/Engine Typical: Array, Object, Map, Set

04 Cross-Language Summary
Key takeaways and performance patterns

```
// Java Collections Framework // Complexity Analysis import java.util.*; public class ComplexityDemo { // List, Set, Map, Queue performance... }
```

Java Collections Time Complexities

`O(1)` Constant · `O(log n)` Logarithmic · `O(n)` Linear

Typical average-case performance. *Amortized time.

List Interface

STRUCTURE	ACCESS	SEARCH	INS/REM	NOTES
`ArrayList`	O(1)	O(n)	O(n)	Append is O(1)*. Slow insert/remove at middle (array shift).
`LinkedList`	O(n)	O(n)	O(1)	O(1) insert/remove only if node reference is known (ends).

Set Interface

STRUCTURE	ADD	CONTAINS	REMOVE	NOTES
`HashSet`	O(1)	O(1)	O(1)	Unordered. Worst O(n) on heavy collisions.
`LinkedHashSet`	O(1)	O(1)	O(1)	Preserves insertion order. Slightly more memory.
`TreeSet`	O(log n)	O(log n)	O(log n)	Sorted order. Red-Black tree backed.

Map Interface

STRUCTURE	GET/PUT	CONTAINS	ITERATE	NOTES
`HashMap`	O(1)	O(1)	O(n)	No order guarantee. Key must implement hashCode/equals correctly.
`LinkedHashMap`	O(1)	O(1)	O(n)	Insertion or access order. Good for LRU caches.
`TreeMap`	O(log n)	O(log n)	O(n)	Sorted by key. Lower throughput than HashMap.

Queue / Deque

STRUCTURE	ADD FIRST/LAST	POLL FIRST/LAST	PEEK	NOTES
`ArrayDeque`	O(1)	O(1)	O(1)	Circular buffer. Faster than Stack. No nulls.
`LinkedList`	O(1)	O(1)	O(1)	More memory overhead per node than ArrayDeque.
`PriorityQueue`	O(log n)	O(log n)	O(1)	Heap structure. Accesses min/max element.

Python Built-ins Time Complexities

O(1) Constant | O(k) k=slice size | O(n) Linear

Values for CPython (Standard Implementation). Dictionary/Set are Hash Table based.

List (Dynamic Array)

OPERATION	AVERAGE TIME	NOTES
`Index [i]`	O(1)	Fast random access.
`append()`	O(1)	Amortized. Expansion is O(n).
`pop()`	O(1)	Remove from end.
`pop(0) / insert(0)`	O(n)	Shifts all elements.
[illegible]	[illegible]	Linear search.

Dict (Hash Map)

OPERATION	AVERAGE TIME	NOTES
`Get / Set Item`	O(1)	Very fast avg case.
`Delete Item`	O(1)	del d[k]
`k in d`	O(1)	Key membership.
`Iteration`	O(n)	Over keys/values.
[illegible]	[illegible]	Depends on dict size.

Set (Hash Set)

OPERATION	AVERAGE TIME	NOTES
`Add / Remove`	O(1)	No duplicates allowed.
`x in s`	O(1)	Fast membership check.
`Union (s|t)`	O(m+n)	Depends on size of both.
`Difference (s-t)`	O(n)	Depends on size of s.

Tuple & Deque

OPERATION	AVERAGE TIME	NOTES
`Tuple Index`	O(1)	Same as list, but immutable.
`Tuple Iteration`	O(n)	Linear scan.
`Deque append`	O(1)	Both ends (left/right).
`Deque pop(0)`	O(1)	Much faster than list pop(0).

JS JavaScript Time Complexities

`O(1)` Constant `O(log n)` Logarithmic `O(n)` Linear

Typical performance in modern engines (V8, SpiderMonkey).

Array

OPERATION	TIME	NOTES
`Index Access [i]`	O(1)	Direct memory access.
`push() / pop()`	O(1)	End of array. Amortized.
`shift() / unshift()`	O(n)	Front of array (re-indexing required).
`includes() / find()`	O(n)	Linear search.

Object (as Map)

OPERATION	TIME	NOTES
`Property Access`	O(1)	Average case. Hash-map backed.
`Insert / Delete`	O(1)	Can degrade if used as sparse array.
`key in obj`	O(1)	Prototype chain check included.
`Object.keys()`	O(n)	Creates new array of keys.

Map

OPERATION	TIME	NOTES
`get() / set()`	O(1)	Optimized hash table. Any key type.
`delete()`	O(1)	Constant time removal.
`has(key)`	O(1)	No prototype conflicts.
`Iteration`	O(n)	Guaranteed insertion order.

Set

OPERATION	TIME	NOTES
`add()`	O(1)	Stores unique values only.
`delete()`	O(1)	Removes specific value.
`has(value)`	O(1)	Much faster than Array.includes().
`Iteration`	O(n)	Guaranteed insertion order.