

# Java Data Structure Comparisons

Here's a detailed comparison of **ArrayList vs LinkedList**, **HashSet vs TreeSet**, and **HashMap vs Hashtable**—frequently asked in interviews:

## ✓ ArrayList vs LinkedList

Feature	ArrayList	LinkedList
Underlying Structure	Dynamic array	Doubly linked list
Search (get/indexOf)	Fast ( $O(1)$ for <code>get(index)</code> )	Slow ( $O(n)$ )
Insertion at End	Fast (Amortized $O(1)$ )	Fast ( $O(1)$ )
Insertion at Middle	Slow ( $O(n)$ ) - needs array copy	Fast ( $O(1)$ if node is known)
Deletion	Slow ( $O(n)$ )	Fast ( $O(1)$ if node is known)
Memory Usage	Less (stores only data)	More (stores data + pointers)
Use Case	Best for random access	Best for frequent insertions/deletions

## ✓ HashSet vs TreeSet

Feature	HashSet	TreeSet
Underlying Structure	HashMap internally	Red-Black Tree (Self-balancing BST)
Order	No guaranteed order	Elements sorted in natural or custom order
Null Elements	Allows one <b>null</b>	Does <b>not</b> allow <b>null</b> (throws exception)
Performance (Add/Search/Delete)	$O(1)$ average time	$O(\log n)$ due to tree operations
Comparator Support	No	Yes ( <b>Comparable</b> or <b>Comparator</b> supported)
Use Case	Fast access, uniqueness only	Sorted data with uniqueness

## ✓ HashMap vs Hashtable

Feature	HashMap	Hashtable
Thread Safety	<b>Not</b> synchronized (not thread-safe)	Synchronized (thread-safe, legacy)
Performance	Faster (no locking)	Slower (synchronization overhead)
Null Keys/Values	Allows <b>one null key</b> , multiple null values	<b>Does NOT</b> allow null keys or null values
Introduced in	Java 1.2 (part of <b>collections framework</b> )	Java 1.0 (legacy class)
Fail-fast	Yes (throws <b>ConcurrentModificationException</b> on concurrent modification)	No (legacy behavior)

Feature	HashMap	Hashtable
Use Case	Single-threaded applications, or use <b>ConcurrentHashMap</b> for thread safety	Rarely used in new code

### Interview Tip:

- For **thread-safe** operations, prefer **ConcurrentHashMap** over **Hashtable**.
- For **sorting + uniqueness**, use **TreeSet**.
- For **insertion/deletion-heavy** lists, use **LinkedList**.

Would you like code examples for each comparison?

Great! Here's **Java code examples** for each comparison:

➔ **ArrayList vs LinkedList, HashSet vs TreeSet, and HashMap vs Hashtable.**

## ✓ 1. ArrayList vs LinkedList Example

```
java

import java.util.*;

public class ListComparison {
    public static void main(String[] args) {
        // ArrayList
        List<String> arrayList = new ArrayList<>();
        arrayList.add("Java");
        arrayList.add("Python");
        arrayList.add("C++");
        System.out.println("ArrayList: " + arrayList.get(1)); // Fast random access

        // LinkedList
        List<String> linkedList = new LinkedList<>();
        linkedList.add("Java");
        linkedList.add("Python");
        linkedList.add("C++");
        System.out.println("LinkedList: " + linkedList.get(1)); // Slower access
    }
}
```

## ✓ 2. HashSet vs TreeSet Example

```
java

import java.util.*;

public class SetComparison {
    public static void main(String[] args) {
        // HashSet - Unordered
        Set<String> hashSet = new HashSet<>();
        hashSet.add("Banana");
        hashSet.add("Apple");
        hashSet.add("Mango");
        System.out.println("HashSet: " + hashSet); // Order not guaranteed

        // TreeSet - Sorted
        Set<String> treeSet = new TreeSet<>();
        treeSet.add("Banana");
        treeSet.add("Apple");
        treeSet.add("Mango");
        System.out.println("TreeSet: " + treeSet); // Sorted order
    }
}
```

## ✓ 3. HashMap vs Hashtable Example

```
java

import java.util.*;

public class MapComparison {
    public static void main(String[] args) {
        // HashMap - allows null keys and values
        Map<String, String> hashMap = new HashMap<>();
    }
}
```

```

hashMap.put(null, "NullKey"); // One null key allowed
hashMap.put("One", "Java");
hashMap.put("Two", null); // Null value allowed
System.out.println("HashMap: " + hashMap);

// Hashtable - does NOT allow null keys or values
Map<String, String> hashtable = new Hashtable<>();
hashtable.put("One", "Java");
hashtable.put("Two", "Python");
// hashtable.put(null, "Value"); // ✗ Throws NullPointerException
// hashtable.put("Key", null); // ✗ Throws NullPointerException
System.out.println("Hashtable: " + hashtable);
    }
}



```

## ✓ Summary:

- **ArrayList.get()** is faster than **LinkedList.get()** but slower for insert/delete in middle.
- **HashSet** is unordered; **TreeSet** is sorted.
- **HashMap** is non-thread-safe but flexible; **Hashtable** is thread-safe but outdated.

Would you like performance benchmarking code or interview Q&A around these?

Perfect! Here's a continuation that includes:

1.  **Performance Benchmarking Code** for `ArrayList` vs `LinkedList`
2.  **Interview Q&A** for each pair:
  - `ArrayList` vs `LinkedList`
  - `HashSet` vs `TreeSet`
  - `HashMap` vs `Hashtable`

## 1. Performance Benchmark: `ArrayList` vs `LinkedList`

This test compares insertion time at the **start** of the list.

```
java

import java.util.*;

public class ListPerformanceTest {
    public static void main(String[] args) {
        int size = 100_000;

        // ArrayList Test
        List<Integer> arrayList = new ArrayList<>();
        long startTime = System.currentTimeMillis();
        for (int i = 0; i < size; i++) {
            arrayList.add(0, i); // insert at beginning
        }
        long endTime = System.currentTimeMillis();
        System.out.println("ArrayList insert at head: " + (endTime - startTime) + " ms");

        // LinkedList Test
        List<Integer> linkedList = new LinkedList<>();
        startTime = System.currentTimeMillis();
        for (int i = 0; i < size; i++) {
            linkedList.add(0, i); // insert at beginning
        }
        endTime = System.currentTimeMillis();
        System.out.println("LinkedList insert at head: " + (endTime - startTime) + " ms");
    }
}
```

## 2. Interview Q&A

### ◆ `ArrayList` vs `LinkedList`

**Q: Which is faster for random access?**

➔ `ArrayList` –  $O(1)$  time for `get(index)`

**Q: When is `LinkedList` better than `ArrayList`?**

➔ When you need frequent insertion or deletion at **beginning or middle** of the list.

**Q: Which one consumes more memory?**

➔ `LinkedList` (stores node + pointer to previous/next).

### ◆ `HashSet` vs `TreeSet`

**Q: When would you prefer TreeSet over HashSet?**

→ When you need **sorted order** (natural or custom comparator).

**Q: Is HashSet faster than TreeSet?**

→ Yes, HashSet offers average  **$O(1)$**  performance; TreeSet is  **$O(\log n)$** .

**Q: Can TreeSet store null values?**

→ No, it throws `NullPointerException` because sorting `null` is not allowed.

---

#### ◆ HashMap vs Hashtable

**Q: What's the main difference between HashMap and Hashtable?**

→ HashMap is **not thread-safe**, Hashtable is **synchronized**.

**Q: Which one is preferred in multi-threaded apps?**

→ Use `ConcurrentHashMap` (modern alternative to `Hashtable`).

**Q: Does HashMap allow null keys/values?**

→ Yes – one null key and multiple null values.

---

Would you like a performance test for `HashSet` vs `TreeSet` and `HashMap` vs `Hashtable` as well?