Java Collections Framework - Complete Guide for DSA

Collection Hierarchy Overview

```
Collection (Interface)
├── List (Interface)
│   ├── ArrayList
│   └── LinkedList
├── Set (Interface)
│   ├── HashSet
│   └── TreeSet
└── Queue (Interface)
    └── LinkedList

Map (Interface - separate hierarchy)
├── HashMap
└── TreeMap
```

---

1. ArrayList

Internal Structure

- **Underlying**: Dynamic array (Object[] array)
- **Default Capacity**: 10
- **Growth**: Increases by 50% when full (new size = old size + old size/2)

Key Characteristics

```java
import java.util.*;

ArrayList<Integer> list = new ArrayList<>();
// Creates with default capacity 10

ArrayList<Integer> list2 = new ArrayList<>(20);
// Creates with initial capacity 20

ArrayList<Integer> list3 = new ArrayList<>(Arrays.asList(1, 2, 3));
// Creates from existing collection
```

Time Complexity

Operation	Time Complexity	Notes
Access (get)	O(1)	Direct array access
Insert at end	O(1) amortized	O(n) when resize needed
Insert at index	O(n)	Requires shifting elements
Remove from end	O(1)	No shifting required
Remove from index	O(n)	Requires shifting elements
Search	O(n)	Linear search required
Contains	O(n)	Linear search

When to Use ArrayList

- Random access to elements needed
- More reads than writes
- Memory efficiency important
- Need indexed access

---

2. LinkedList

Internal Structure

- **Underlying**: Doubly linked list
- **Node Structure**: Each node has data, next, and previous pointers
- **No Capacity**: Grows as needed

Key Characteristics

```java
LinkedList<Integer> list = new LinkedList<>();

// Can be used as List, Deque, or Queue
List<Integer> asList = new LinkedList<>();
Deque<Integer> asDeque = new LinkedList<>();
Queue<Integer> asQueue = new LinkedList<>();
```

Time Complexity

Operation	Time Complexity	Notes
Access (get)	O(n)	Must traverse from head/tail
Insert at beginning	O(1)	Direct pointer manipulation

Operation	Time Complexity	Notes
Insert at end	O(1)	Direct pointer manipulation
Insert at index	O(n)	Must traverse to position
Remove from beginning	O(1)	Direct pointer manipulation
Remove from end	O(1)	Direct pointer manipulation
Remove from index	O(n)	Must traverse to position
Search	O(n)	Linear search required

When to Use LinkedList

- Frequent insertions/deletions at beginning/end
- Don't need random access
- Implementing stacks, queues, or deques
- Size varies significantly

---

3. HashMap

Internal Structure

- **Underlying**: Array of buckets (Node[] table)
- **Default Capacity**: 16
- **Load Factor**: 0.75 (resize when 75% full)
- **Collision Handling**: Chaining (Java 8+: Trees for large chains)

Key Characteristics

```java
HashMap<String, Integer> map = new HashMap<>();
HashMap<String, Integer> map2 = new HashMap<>(32); // Initial capacity
HashMap<String, Integer> map3 = new HashMap<>(32, 0.8f); // Capacity + load factor
```

Hash Function & Collisions

```java
// Simplified hash process:
// 1. key.hashCode() -> int hash
// 2. hash ^ (hash >>> 16) -> reduce collisions
// 3. (n-1) & hash -> bucket index (where n = table length)

// Java 8+ improvement: When bucket has >8 elements, converts to TreeNode
```

Time Complexity

Operation	Average	Worst Case	Notes
Get	O(1)	O(n)	Worst case: all keys hash to same bucket
Put	O(1)	O(n)	Worst case during resize or collision
Remove	O(1)	O(n)	Same as get
Contains Key	O(1)	O(n)	Same as get
Contains Value	O(n)	O(n)	Must check all entries

When to Use HashMap

- Need key-value mapping
- Fast lookups required
- Order doesn't matter
- Keys have good hash distribution

---

4. HashSet

Internal Structure

- **Underlying**: HashMap (values are dummy PRESENT object)
- **Same properties as HashMap**: Load factor, capacity, collision handling

Key Characteristics

```java
HashSet<Integer> set = new HashSet<>();
HashSet<Integer> set2 = new HashSet<>(Arrays.asList(1, 2, 3, 4));

// Internally uses HashMap
// set.add(element) -> map.put(element, PRESENT)
// set.contains(element) -> map.containsKey(element)
```

Time Complexity

Operation	Average	Worst Case
Add	O(1)	O(n)
Remove	O(1)	O(n)
Contains	O(1)	O(n)
Size	O(1)	O(1)

Operation	Average	Worst Case
Iterator	O(capacity)	O(capacity)

When to Use HashSet

- Need unique elements
- Fast membership testing
- Order doesn't matter
- Duplicate removal

---

5. TreeMap

Internal Structure

- **Underlying**: Red-Black Tree (self-balancing BST)
- **Ordering**: Natural ordering or custom Comparator
- **Guarantees**: Sorted order, balanced tree

Key Characteristics

```java
TreeMap<String, Integer> map = new TreeMap<>(); // Natural ordering
TreeMap<String, Integer> map2 = new TreeMap<>(Collections.reverseOrder());
TreeMap<String, Integer> map3 = new TreeMap<>((a, b) -> a.length() - b.length());
```

Time Complexity

Operation	Time Complexity	Notes
Get	O(log n)	Tree traversal
Put	O(log n)	Tree insertion + rebalancing
Remove	O(log n)	Tree deletion + rebalancing
First/Last Key	O(log n)	Tree traversal
Floor/Ceiling	O(log n)	Tree navigation
SubMap operations	O(log n)	Tree navigation

Special Methods

```java
```

```
TreeMap<Integer, String> map = new TreeMap<>();
map.put(1, "One"); map.put(3, "Three"); map.put(5, "Five");

map.firstKey();        // 1
map.lastKey();         // 5
map.lowerKey(3);       // 1 (largest key < 3)
map.floorKey(4);       // 3 (largest key <= 4)
map.ceilingKey(4);     // 5 (smallest key >= 4)
map.higherKey(3);      // 5 (smallest key > 3)
map.subMap(2, 6);      // {3=Three, 5=Five}
```

When to Use TreeMap

- Need sorted key-value mapping
- Range queries required
- Floor/ceiling operations needed
- Ordered iteration important

---

6. TreeSet

Internal Structure

- **Underlying**: TreeMap (values are dummy PRESENT object)
- **Same properties as TreeMap**: Red-Black Tree, sorted order

Key Characteristics

```java
TreeSet<Integer> set = new TreeSet<>();
TreeSet<String> set2 = new TreeSet<>(Collections.reverseOrder());
TreeSet<Integer> set3 = new TreeSet<>(Arrays.asList(3, 1, 4, 1, 5)); // {1, 3, 4, 5}
```

Time Complexity

Operation	Time Complexity
Add	O(log n)
Remove	O(log n)
Contains	O(log n)
First/Last	O(log n)
Floor/Ceiling	O(log n)
SubSet operations	O(log n)

Special Methods

```java
TreeSet<Integer> set = new TreeSet<>(Arrays.asList(1, 3, 5, 7, 9));

set.first();          // 1
set.last();           // 9
set.lower(5);         // 3
set.floor(6);         // 5
set.ceiling(6);       // 7
set.higher(5);        // 7
set.subSet(3, 8);     // {3, 5, 7}
set.headSet(5);       // {1, 3}
set.tailSet(5);       // {5, 7, 9}
```

When to Use TreeSet

- Need unique elements in sorted order
- Range operations required
- Floor/ceiling operations needed
- Ordered iteration important

Memory Considerations

ArrayList vs LinkedList

- **ArrayList**: More memory efficient (no node overhead)
- **LinkedList**: Extra memory for node pointers (24 bytes per node on 64-bit JVM)

HashMap vs TreeMap

- **HashMap**: Lower memory overhead, faster access
- **TreeMap**: Higher memory overhead (node pointers + color bit), slower but ordered

HashSet vs TreeSet

- **HashSet**: Same as HashMap considerations
- **TreeSet**: Same as TreeMap considerations

Choosing the Right Collection

For Lists:

- **ArrayList**: Default choice, random access needed
- **LinkedList**: Frequent insertions/deletions at ends, queue/deque operations

For Sets:

- **HashSet**: Default choice, fastest operations
- **TreeSet**: Need sorted unique elements, range operations

For Maps:

- **HashMap**: Default choice, fastest key-value operations
- **TreeMap**: Need sorted keys, range queries, navigation methods

---

Common DSA Patterns

1. Frequency Counting

```java
Map<Character, Integer> freq = new HashMap<>();
for (char c : str.toCharArray()) {
    freq.put(c, freq.getOrDefault(c, 0) + 1);
}
```

2. Two Sum Pattern

```java
Map<Integer, Integer> map = new HashMap<>();
for (int i = 0; i < nums.length; i++) {
    int complement = target - nums[i];
    if (map.containsKey(complement)) {
        return new int[]{map.get(complement), i};
    }
    map.put(nums[i], i);
}
```

3. Sliding Window with Set

```java
```

```java
Set<Character> window = new HashSet<>();
int left = 0;
for (int right = 0; right < s.length(); right++) {
    while (window.contains(s.charAt(right))) {
        window.remove(s.charAt(left++));
    }
    window.add(s.charAt(right));
    // Process window
}
```

4. Range Queries with TreeSet

```java
TreeSet<Integer> set = new TreeSet<>();
// Find elements in range [low, high]
NavigableSet<Integer> range = set.subSet(low, true, high, true);
```

This foundation is crucial for DSA success. Practice implementing basic operations and understand when to use each collection!