🧾 Java Collections Cheat Sheet for DSA

| **🧩 Problem Type** | **✅ Best Collection** | **⚙️ Why It Works** |
| --- | --- | --- |
| **Stack-based problems** | Stack or Deque | LIFO behavior for parsing, recursion simulation |
| **Queue-based problems** | Queue or LinkedList | FIFO behavior for BFS, scheduling |
| **Priority problems** | PriorityQueue | Min/Max heap for greedy algorithms |
| **Frequency counting** | HashMap | Fast lookup and counting |
| **Unique elements** | HashSet | O(1) lookup for duplicates |
| **Sorted elements** | TreeSet or TreeMap | Maintains sorted order |
| **Sliding window** | Deque or TreeMap | Efficient window updates |
| **Graph adjacency list** | HashMap> | Easy to build and traverse |
| **Top K elements** | PriorityQueue | Efficient extraction of top elements |
| **Two-pointer problems** | ArrayList | Index-based access |
| **Backtracking state** | LinkedList | Easy to add/remove from ends |
| **Union-Find / DSU** | Custom array or HashMap | Requires custom logic |
| **Trie / Prefix Tree** | Custom class with HashMap | Needs node-based structure |

🧠 Bonus Tips

* Use Deque over Stack for better performance and flexibility.
* Prefer ArrayList for random access, LinkedList for frequent insertions/removals.
* TreeMap and TreeSet are great for problems needing **ordered traversal** or **range queries**.

🧾 Linked List Cheat Sheet (Java DSA)

🔗 Types of Linked Lists

| **Type** | **Description** | **Java Support** |
| --- | --- | --- |
| **Singly Linked List** | Each node points to the next | Custom implementation |
| **Doubly Linked List** | Each node points to both next and previous | LinkedList (Java Collections) |
| **Circular Linked List** | Last node points to head | Custom implementation |

🧰 Common Operations

| **Operation** | **Description** | **Time Complexity** |
| --- | --- | --- |
| Insert at head | Add node at beginning | O(1) |
| Insert at tail | Add node at end | O(n) (unless tail is tracked) |
| Delete node | Remove specific node | O(n) |
| Search | Find a value | O(n) |
| Reverse list | Flip node links | O(n) |
| Detect cycle | Use Floyd’s algorithm | O(n) |
| Merge two lists | Combine sorted lists | O(n) |
| Find middle | Use slow/fast pointers | O(n) |

Sample Node Class (Singly)

class ListNode {

int val;

ListNode next;

ListNode(int val) {

this.val = val;

}

}

🔄 Reverse a Linked List

public ListNode reverseList(ListNode head) {

ListNode prev = null;

while (head != null) {

ListNode next = head.next;

head.next = prev;

prev = head;

head = next;

}

return prev;

}

🔍 Detect Cycle (Floyd’s Algorithm)

public boolean hasCycle(ListNode head) {

ListNode slow = head, fast = head;

while (fast != null && fast.next != null) {

slow = slow.next;

fast = fast.next.next;

if (slow == fast) return true;

}

return false;

}

🧠 Patterns to Know

* **Two-pointer technique**: For middle, cycle detection, palindrome check.
* **Dummy node**: Simplifies edge cases in insertion/deletion.
* **Recursive reversal**: Elegant but stack-heavy.
* **In-place merge**: For sorted lists without extra space.

⚠️ Java LinkedList vs Custom

| **Feature** | **LinkedList** | **Custom Node Class** |
| --- | --- | --- |
| Built-in methods | Yes (add, remove, etc.) | No |
| Control over nodes | Limited | Full |
| Interview use | Rarely preferred | Often expected |