11. **What is the internal working of a Stack?**
    - A Stack uses a dynamic array or linked list to manage elements, with the LIFO (Last In, First Out) principle, where elements are added and removed from the same end (top of the stack).
12. **How does a Queue work internally?**
    - A Queue follows the FIFO (First In, First Out) principle. It can be implemented using arrays, linked lists, or priority queues depending on the requirements.

These questions probe your understanding of the core data structures in Java and how they handle operations like insertions, deletions, and searching internally.

now explain in detail the internal working

Here's a detailed explanation of the internal workings of some of the most common Java collections:

---

1. HashMap

- **Structure**:
  - Internally uses an array of `` `Node<K, V>` `` where each node contains:
    - `` `hash` `` (hash value of the key)
    - `` `key` ``
    - `` `value` ``
    - `` `next` `` (reference to the next node in case of collision)
- **How it works**:
  1. The hash code of the key is computed and then modded with the array size to determine the index.
  2. If the index is empty, the key-value pair is stored there.
  3. If a collision occurs (two keys map to the same index), a linked list or a red-black tree (if the bucket size exceeds a threshold, usually 8) is used.
  4. On retrieval, the key's hash code is recalculated to find the bucket, and the bucket's linked list or tree is searched.
- **Resizing**:
  - When the load factor (default 0.75) is exceeded, the array is doubled, and all existing keys are rehashed and redistributed.

---

2. TreeMap

- **Structure**:
  - Uses a **Red-Black Tree** internally.
  - Ensures sorted order based on the natural ordering of keys or a provided comparator.
- **How it works**:
  - On insertion, the key is compared using `` `compareTo` `` or `` `Comparator` `` and placed in the correct position, maintaining the binary search tree property.
  - Red-Black Tree ensures that the tree remains balanced with O(log n) insertion, deletion, and lookup times.

3. HashSet

- **Structure**:
  - Internally uses a `HashMap` where:
    - The keys in the map are the elements of the `HashSet`.
    - The value is a dummy constant object (`PRESENT`).
- **How it works**:
  - When adding an element, the `add()` method calls `put()` on the underlying `HashMap`.
  - Uniqueness is maintained by checking the `hashCode` and `equals` methods.

---

4. LinkedList

- **Structure**:
  - Consists of **nodes** where each node contains:
    - `data`: The actual value.
    - `next`: A reference to the next node.
    - `prev`: (In `DoublyLinkedList`) A reference to the previous node.
- **How it works**:
  - Elements are added or removed by adjusting the `next` and `prev` pointers.
  - Random access is slow (O(n)) because traversal from the head or tail is required to find an element.

---

5. ArrayList

- **Structure**:
  - Uses a dynamic array.
- **How it works**:
  - When an element is added and the array is full, a new array with 1.5 times the capacity is created, and all elements are copied to the new array.
  - Random access is fast (O(1)), but insertions and deletions in the middle involve shifting elements, making them O(n).

---

6. LinkedHashMap

- **Structure**:
  - Extends `HashMap` and maintains a **doubly linked list**.
- **How it works**:
  - The doubly linked list keeps track of the insertion order or access order (if `accessOrder` is true in the constructor).
  - Iteration over the map entries preserves this order.

---

7. PriorityQueue

- **Structure**:
  - Implemented using a **binary heap**.
  - The root of the heap always contains the element with the highest priority (based on natural ordering or a custom comparator).
- **How it works**:

- For insertion, the element is added to the end of the array and "bubbled up" to maintain the heap property.
- For removal, the root is removed, and the last element is moved to the root and "bubbled down."

---

8. ConcurrentHashMap

- **Structure**:
  - Uses a segmented locking mechanism where the map is divided into segments, each handling its synchronization.
- **How it works**:
  - Allows multiple threads to read and write in different segments concurrently.
  - Uses a bucket array similar to `HashMap`, but each segment is independently synchronized.

---

9. TreeSet

- **Structure**:
  - Internally backed by a `TreeMap`.
- **How it works**:
  - Elements are stored as keys in the TreeMap with a dummy value.
  - Ensures uniqueness and maintains a sorted order of elements.

---

10. Stack

- **Structure**:
  - Extends `Vector` but uses a LIFO (Last In, First Out) principle.
- **How it works**:
  - The `push()` method appends elements to the end of the `Vector`.
  - The `pop()` method removes and returns the last element.

---

11. Queue

- **Structure**:
  - Different implementations like `LinkedList` or `PriorityQueue` depending on the use case.
- **How it works**:
  - `add()` or `offer()` to insert elements.
  - `poll()` or `remove()` to remove the head of the queue.

These internal workings are crucial for understanding the performance characteristics and choosing the right collection for a particular use case.