**✅ Where is Compare-And-Swap (CAS) Used in Java Collections?**

CAS is extensively used in **concurrent collections** from the java.util.concurrent package. These collections rely on **non-blocking algorithms** for better performance in **multi-threaded** environments.

**📌 Why Use CAS in Collections?**

1. **Thread Safety**: Ensures multiple threads can modify collections without locks.
2. **Non-Blocking**: Allows high-performance access without the cost of locking.
3. **Scalability**: Reduces contention and improves performance under heavy loads.

**✅ Collections Using CAS in Java**

| **Collection** | **CAS Usage** |
| --- | --- |
| ConcurrentHashMap | Atomic updates (inserting, updating, removing) |
| ConcurrentLinkedQueue | Lock-free insertion and deletion |
| ConcurrentLinkedDeque | Double-ended queue with lock-free operations |
| CopyOnWriteArrayList | Atomic updates with a new copy of the list |
| CopyOnWriteArraySet | Built on CopyOnWriteArrayList, uses CAS for writes |
| LinkedBlockingQueue | CAS for queue operations (head and tail pointers) |
| ArrayBlockingQueue | Uses CAS for concurrent producers and consumers |
| PriorityBlockingQueue | CAS for concurrent access to priority elements |
| ConcurrentSkipListMap | Lock-free navigation and updates |
| ConcurrentSkipListSet | Built on ConcurrentSkipListMap, uses CAS |

**📌 1. CAS in ConcurrentHashMap**

* **CAS ensures atomic put and remove operations.**
* Each **bucket** is updated using CAS to avoid locking the entire map.

**Example:**

import java.util.concurrent.ConcurrentHashMap;

public class ConcurrentHashMapCAS {

public static void main(String[] args) {

ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();

map.putIfAbsent("A", 1); // Uses CAS internally

map.computeIfPresent("A", (k, v) -> v + 1); // Atomic update

System.out.println(map); // {A=2}

}

}

**CAS Method (Source Code Example):**

final V putVal(K key, V value, boolean onlyIfAbsent) {

int hash = spread(key.hashCode());

if (casTabAt(tab, i, null, new Node<>(hash, key, value, null))) {

return null; // CAS inserts node atomically

}

}

**📌 2. CAS in ConcurrentLinkedQueue**

* **Lock-free queue** using **CAS** for **head** and **tail** updates.
* No locking during insertion or deletion.

**Example:**

import java.util.concurrent.ConcurrentLinkedQueue;

public class CASInConcurrentQueue {

public static void main(String[] args) {

ConcurrentLinkedQueue<Integer> queue = new ConcurrentLinkedQueue<>();

queue.offer(10); // CAS to add element

queue.poll(); // CAS to remove element

System.out.println(queue); // []

}

}

**CAS Usage in offer() (Source Code):**

if (tail.compareAndSet(currTail, newNode)) {

currTail.next = newNode; // CAS updates tail

}

**📌 3. CAS in CopyOnWriteArrayList**

* **Copy-on-Write** approach creates a **new array** on every update.
* Uses **AtomicReferenceArray** internally.

**Example:**

import java.util.concurrent.CopyOnWriteArrayList;

public class CASInCopyOnWrite {

public static void main(String[] args) {

CopyOnWriteArrayList<String> list = new CopyOnWriteArrayList<>();

list.add("Java"); // CAS ensures thread-safe insertion

list.addIfAbsent("Java"); // CAS prevents duplicate entries

System.out.println(list); // [Java]

}

}

**📌 4. CAS in ConcurrentSkipListMap**

* Implements a lock-free **sorted map** using **CAS**.
* Each node update uses CAS for atomicity.

**Example:**

import java.util.concurrent.ConcurrentSkipListMap;

public class CASInSkipList {

public static void main(String[] args) {

ConcurrentSkipListMap<Integer, String> map = new ConcurrentSkipListMap<>();

map.put(1, "A"); // CAS to add a node

map.put(2, "B");

System.out.println(map); // {1=A, 2=B}

}

}

**📌 5. CAS in ArrayBlockingQueue**

* Fixed-size queue using CAS for **enqueuing and dequeuing**.

**Example:**

import java.util.concurrent.ArrayBlockingQueue;

public class CASInArrayQueue {

public static void main(String[] args) throws InterruptedException {

ArrayBlockingQueue<Integer> queue = new ArrayBlockingQueue<>(5);

queue.put(1); // CAS to add element

System.out.println(queue.take()); // CAS to remove element

}

}

**✅ Why CAS Over Locks in Collections?**

1. **Performance**: CAS is faster in high-concurrency environments.
2. **Non-Blocking**: Threads do not block, avoiding deadlocks.
3. **Efficiency**: Less context switching compared to synchronized.

**📌 Common CAS-Based Collection Interview Questions**

1. **How does CAS work in ConcurrentHashMap?**
2. **What is the role of CAS in ConcurrentLinkedQueue?**
3. **Why does CopyOnWriteArrayList use CAS for updates?**
4. **What is the difference between CAS and locks in collections?**
5. **Explain how CAS prevents race conditions in concurrent collections.**
6. **How does AtomicReference enable CAS in ConcurrentSkipListMap?**
7. **What are the trade-offs of using CAS vs. locking in Java collections?**
8. **How does CAS improve the performance of concurrent queues?**

Would you like more examples or a deep dive into a specific collection? 😊