ChatGPT

Lock-Free Skip Lists: A Technical Deep Dive

Overview of Skip List Data Structure and Concurrency Challenges

A **skip list** is an ordered data structure that maintains multiple levels of linked lists to achieve fast search, insertion, and deletion in expected O(log n) time [1] [2]. The bottom level is an ordinary sorted linked list containing all elements, and each higher level is a "express lane" that skips over more elements (each element appears in higher levels with some probability, often 50%) [3]. This layered structure allows queries to skip through the list quickly by traversing top levels and dropping down when overshooting, achieving logarithmic search on average. **Figure 1** illustrates a simple skip list with 4 levels (level 0 is bottom): higher layers have sparser "express" links, while level 0 links every node in sorted order.

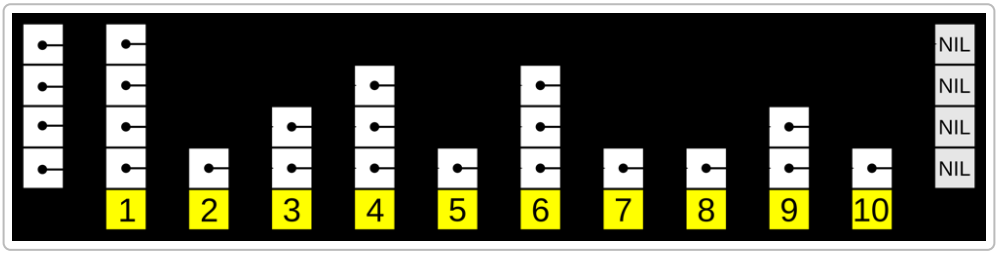
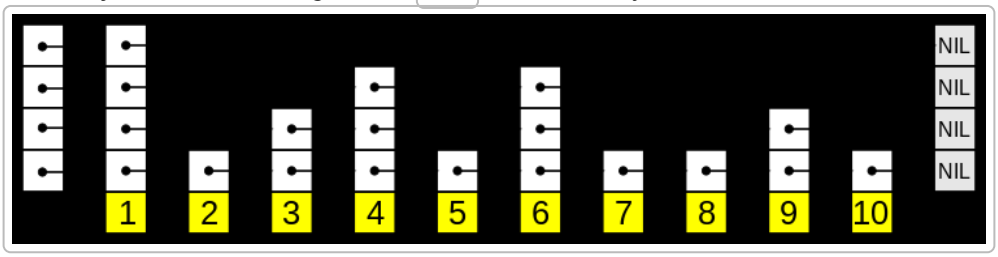


Figure 1: A sample skip list with nodes 1–10. Each column is a tower of a node present in one or more levels. Top levels skip over many nodes, accelerating search. `NIL` *denotes end-of-list sentinel nodes*



.

In a single-threaded setting, skip list operations (search, insert, delete) can be implemented by traversing and updating these linked lists easily. However, in a **concurrent** multi-threaded environment, skip lists pose significant challenges. Multiple updates may happen in parallel on different parts or even the same part of the skip list. An insertion or deletion spans multiple levels of pointers, so **partial updates** can temporarily violate the structural invariants if viewed at the wrong moment. For example, one thread might insert a new node with key 12 while another thread is concurrently deleting the node with key 9. It is possible (and valid in a correct lock-free algorithm) for some higher-level pointers to bypass node 9 before it is fully removed, causing node 5 and node 9 *both* to point to node 12 at an intermediate stage [4]. In that state, node 9 "still exists" in the bottom-level list but is **partially logically deleted** from some upper levels [4]. This means that on level 1, the pointer from 5 skips directly to 12 (skipping over 9), even though 9 is still in the bottom list. Such interleavings must be carefully handled so that **each level's sorted order remains valid** and

operations like search don't return incorrect results [4]. The data structure must remain **consistent** (no broken pointers, no lost nodes) even in the presence of concurrent modifications.

Concurrency challenges arise because an update in a skip list is not a single pointer change but a sequence of changes across multiple levels. Without proper synchronization, one thread could see a skip list in the middle of an update by another thread – for instance, a search might follow a pointer into a node that is in the process of being deleted. The naive approach would be to use locks to prevent such conflicts (e.g. locking the whole structure or locking individual nodes/levels). But locks introduce blocking: if a thread is preempted while holding a lock, other threads might block indefinitely. Instead, a **lock-free** skip list uses atomic operations so that **threads cooperate without mutual exclusion**. This requires designing the algorithms such that any thread can complete the necessary pointer updates (or help another thread complete them) using atomic primitives like *Compare-And-Swap* (CAS). The goal is to ensure **linearizability** (each operation appears to take effect atomically at some moment) and **lock-free progress** (some operation always makes progress, even if others are stalled) [5].

Key Lock-Free Programming Concepts for Skip Lists

Lock-free algorithms avoid holding locks; instead, they rely on atomic read-modify-write primitives. The most commonly used primitive is **Compare-And-Swap (CAS)**, which atomically compares the contents of a memory location to an expected value and, only if it matches, swaps in a new value. In C/C++ this is typically exposed via `std::atomic.compare_exchange` and in Java via `AtomicReference.compareAndSet`. Using CAS on pointer references allows threads to insert or remove nodes **optimistically**: a thread will read a pointer, prepare a new node or update, then attempt a CAS to swing the pointer to a new node. If some other thread changed that pointer in the interim, the CAS will fail, and the operation can retry. This way, threads *coordinate* by succeeding or failing CAS operations, rather than explicit locking. For example, the base linked-list of a skip list can use Harris's lock-free list algorithm, which marks a node as deleted and then uses CAS to unlink it [6] [7]. In a skip list, we apply such atomic pointer techniques to multiple levels.

Achieving correctness with CAS requires careful attention to **memory consistency**. On modern CPUs and languages, atomic operations like CAS come with guarantees about memory order (for instance, in C++ `std::atomic` defaults to sequential consistency, and Java's `volatile` reads/writes have similar effects). These guarantees ensure that when one thread updates a pointer with CAS, other threads will see the change and also see any prior writes that initialized the new node's contents. For instance, when inserting a new node, a thread must initialize the node's forward pointers **before** publishing the node into the list with CAS. A sequentially consistent CAS on the list pointers suffices to ensure that any thread that sees the new node in the list can also see the node's internal state (fully initialized pointers, key, etc.). In practice, the CAS itself or accompanying memory fences provide the needed ordering. High-performance libraries sometimes relax ordering for certain operations (e.g. using acq_rel fences or even relaxed atomics on less critical levels) as an optimization [8], but the general principle is to preserve correctness by proper memory barriers.

Hazard pointers and memory reclamation: A major challenge in lock-free structures is managing memory without garbage collection. In languages like C or C++, when a node is removed from the skip list, we cannot immediately free its memory because other threads might still hold a reference (e.g. a thread that was in the middle of reading that node before another thread removed it). Lock-based designs avoid this by not freeing until no thread can hold the lock on the node, but in lock-free design we need other means. **Hazard pointers** are one technique for safe memory reclamation: they allow threads to announce which nodes they are currently accessing, thereby "protecting" those nodes from being freed [9]. When a thread

deletes a node, it defers reclamation (e.g. adds it to a retire list) until it can ensure no other thread still has a hazard pointer referencing it. Only then is the memory actually freed [9]. This guarantees that a pointer value cannot be reused for a new node while another thread might still use the old node – preventing the classic **ABA problem**. (The ABA problem occurs when a location is changed from value A to B and back to A; a CAS might falsely succeed because the value is A again, even though the state changed in between. Hazard pointers or tagged pointers solve this by ensuring a node's address isn't reused as A again until it's safe [10].)

> *Aside:* In garbage-collected environments (e.g. Java), memory reclamation is handled by the GC, simplifying this aspect. Indeed, some lock-free designs take advantage of GC – for example, the Java `ConcurrentSkipListMap` uses short-lived **marker nodes** to mark deletions, which are quickly garbage collected [10]. In C++, one might use hazard pointers, reference counting, or epoch-based reclamation (like RCU or epoch GC) to defer node deletion until no concurrent access is possible. In fact, robust libraries (e.g. **CDS** or Concurrency Kit) let you plug in a memory reclamation scheme; for instance, CDS's skip list allows integration with RCU or hazard pointer GC [11].

Atomic marking and the ABA problem: Some lock-free algorithms represent a "logical deletion" by marking a bit in a pointer (using something like an `AtomicMarkedReference` or using the least significant bit of an aligned pointer). However, using marked pointers on a multi-level node can be tricky and expensive (you'd need to manage a mark on each level's pointer). The Java skip list implementation chose a different approach: on deletion it **splices in a marker node** as the next node, rather than flipping a bit [12]. This marker is a standalone dummy node that indicates "the previous node is deleted." In effect, this acts like a "boxed" marked pointer – but using an actual node means standard CAS can be used on next pointers without extra tag bits [13]. The advantage is that reading a pointer is simple (no need to mask a bit) and marking a node is just a normal CAS to insert the marker. The drawback is that it relies on GC to clean up marker nodes and requires a bit more logic. (This technique **would not work well without GC** because marker nodes would accumulate or need manual reclamation [10], which hazard pointers could handle but with some complexity.)

In summary, a lock-free skip list relies on:

- **CAS operations** on pointers to insert and remove nodes atomically.
- **Memory ordering** guarantees to ensure all threads see a consistent view (especially that a node's contents are visible once the node is in the list).
- **Logical deletion flags or markers** to mark nodes as removed before physically unlinking them, preventing other threads from "missing" a deletion in progress.
- **Safe memory reclamation** (hazard pointers, etc.) to avoid use-after-free and ABA issues in non-GC environments.
- **Helping mechanisms** so that any thread can assist in completing an ongoing operation (this is key to lock-free progress).

Next, we delve into how these concepts are applied in designing lock-free skip list operations: **search**, **insert**, and **delete**.

Lock-Free Skip List Operations in Detail

Search (Containment Check)

The search operation (often exposed as `contains(key)` or finding a mapping by key) in a skip list traverses from the top layer down to layer 0, much like in a sequential skip list. In a lock-free design, searches are typically **wait-free** (or effectively wait-free) because they do not modify the structure – they only read pointers, which may change under them but can be retried or validated if needed.

A search algorithm will start at the head of the top level and move forward until it finds a node with a key not less than the target. If it overshoots (finds a key greater than target or reaches end), it drops down one level and continues, finally reaching level 0 (the bottom list). At level 0 it determines if the target key is present. In a concurrent setting, one must be careful to **ignore deleted nodes** during this traversal. A common approach is to treat a node as *logically absent* if it is marked as deleted (for example, a node whose `marked` flag is true or whose value field is null in the Java implementation [14]). The search will skip over such nodes. If a search finds the target key in an unmarked node at the bottom level, it can safely report the key as present; if it reaches where the key *should* be and either finds a different key or a `NULL` marker, it reports the key as not present.

To ensure wait-freedom, the search should not get stuck in an infinite loop even if the list is being modified. Lock-free skip lists typically achieve this. For instance, Java's `ConcurrentSkipListMap.containsKey()` performs a normal search, and if it encounters markers or nodes with null (deleted) values, it simply skips them and continues searching. It might have to retry a step if a pointer changed *while* being read (e.g., if a node we just read gets unlinked by another thread, some implementations detect that and restart the search at a higher level for safety [15]). However, even in the worst case, the search will keep making progress downwards and to the right, and will eventually either find the key or reach the end of the bottom list. Since it does not need to wait for other threads (it might help finish unlinking a node, but that still counts as progress), the *contains* operation can be considered wait-free – it will complete in a finite number of steps independent of other threads.

One subtlety is that if a search finds a node with the target key at a higher level, it cannot immediately return "found" because that node might be in the middle of being deleted. Most algorithms ensure that a node is only truly in the set if it is present at level 0 (the bottom list) and not marked. Thus, a search typically verifies the node at level 0. In other words, **the bottom-level list is the ground truth** – upper levels are just hints to speed up the search. The search algorithm may need to confirm that an apparent hit is still valid at the bottom level before returning success [16] [17]. If the bottom-level node is gone or marked, the search continues. This design guarantees that a search does not report a key that is in the middle of deletion.

Overall, the lock-free search is mostly straightforward: move through the levels, skip deleted nodes, and possibly assist in unlinking them (e.g., if we see a marked node, some implementations CAS it out on the fly). The important guarantee is that even if modifications keep happening, a search either finds the target or eventually traverses past it; it won't live-lock because each retry or adjustment still discards some portion of the list (or moves downward a level).

Insertion

Inserting a new element into a skip list involves choosing a random height for the new node and then linking the node into each level up to that height. In a concurrent lock-free setting, insertion must be done carefully to avoid conflicts with other insertions or deletions.

Find position: First, the inserting thread searches for the position of the key in the skip list. It typically maintains an array `pred[0..maxLevel]` of pointers to the predecessors at each level and `succ[0..maxLevel]` for the successors (the next nodes after where the new node will go). This is analogous to the "update" array in a sequential skip list. The search starts at the top and finds the place just before where the new key should be at each level, all the way down to level 0.

If the key is found to already exist (the `succ[0]` at level 0 has the same key), then the insertion can either fail or update the existing value (depending on design). Assuming we are implementing a set or map that does not allow duplicate keys, the operation might simply return (or update the value atomically if it's a map). Let's assume for this discussion that duplicates are not allowed, so we proceed only if the key is not already present.

Prepare new node: The thread picks a random level for the new node (using the skip list's random level generator). Say the new node's height is *h*. It allocates a new node with `node.height = h`. The node's forward pointer array (`node.forward[0..h-1]`) is initialized such that each `node.forward[i]` points to `succ[i]` (the next node as identified by our search) [18] [19] . In other words, we splice the new node in logically, by setting its pointers to the successors we found.

Insert at bottom level: Now we need to make the new node visible to other threads. The crucial step is inserting it at level 0, the bottom list, which is where other searches will definitely see it. We perform an atomic CAS on the `pred[0].forward` pointer: expected value should be `succ[0]` (the node that was originally after `pred[0]`), and new value is our `newNode`. If this CAS succeeds, we have inserted the new node at the bottom level. At that moment, the new node is *logically in the set* – any contains() that runs afterward will find it (unless it gets deleted). This CAS is the **linearization point** of the insertion; it "publishes" the new node in the shared structure.

- If the CAS at level 0 **fails**, it means something changed at level 0 since we did our search. Perhaps another thread inserted a node in between or deleted `pred[0]` or `succ[0]`. In this case, the insertion can't proceed because our positional information is stale. The typical response is to **retry**: redo the search to get an updated `pred` and `succ` (and possibly choose a new random level, or reuse the same – some algorithms redo from scratch, which is simpler). This retry loop will continue until the bottom-level CAS succeeds or we discover the key was inserted by someone else.

- If the CAS at level 0 **succeeds**, the new node is in the list at level 0. At this point, some implementations set a flag on the node (like `node.fullyLinked = false` initially) to indicate that the node is not yet present in higher levels [20] . This flag can be used to prevent other threads from "seeing" the node as fully present until all levels are linked. However, not all implementations need an explicit flag; it's also acceptable if a search doesn't find the node in upper levels yet – it will still find it at level 0. The new node is effectively in the set; the remaining work is to update the higher levels.

Linking higher levels: The thread now proceeds to insert the new node's pointer into level 1, level 2, ..., up to level *h-1*. Each of these involves a CAS on the corresponding `pred[i].forward`. We already had `pred` and `succ` for each level from the initial search. However, those might be out-of-date now because while we inserted at level 0, other operations could have happened at higher levels. A simple approach is to attempt the CAS for each level in order, and if any CAS fails, **re-find the predecessors** for that level (or restart the whole insertion, depending on strategy). A more optimistic approach (as used in some skip list algorithms) is to try to insert the index nodes and if one of them fails due to contention, simply *give up* on that level – meaning the new node might not appear in that level's linked list. This sounds dangerous, but it doesn't break correctness: the node is still in level 0 so it will be found by searches, and skip list properties are probabilistic anyway (failing to insert one index makes the structure slightly less balanced but not incorrect or permanently inefficient) [21]. In practice, though, many implementations will retry a few times to insert all levels, to maintain the probabilistic balance.

If using **separate index nodes** (like Java's implementation does internally), the process is conceptually similar: after inserting the base node at level 0, the algorithm creates index nodes that tower above the base node and tries to splice each index into the appropriate level's index list. If one of those insertions fails, it might retry or just abandon that level's index. Java's `ConcurrentSkipListMap` in fact performs the base-level insertion first, then in a *separate pass* adds index nodes for higher levels [22]. This two-phase approach simplifies failure handling: if an index insertion fails, it can decide to drop that index or retry independently without affecting the base list integrity.

Finish insertion: Once the new node has been linked in all intended levels, we can mark it as fully inserted (e.g. set `fullyLinked = true` if that flag is used) [20]. At this point, the node is present in the skip list at all relevant levels. If a concurrent search had found the node at level 0 before this, it might have noticed it wasn't fully linked and waited or retried – but typically we design searches to always verify at bottom level anyway. The `fullyLinked` flag is more relevant if we want to ensure that a concurrent *remove* doesn't remove a node until it's completely inserted (to avoid removing a half-inserted node). In designs without an explicit flag, the equivalent guarantee is achieved by ordering: you only make the node visible at level 0 after everything else is ready.

An important aspect is that the **linearizability** of insert is at the successful CAS on level 0. Even though additional steps happen later, they don't change the fact that the node is in the set from that moment. If an insertion fails midway through adding higher-level pointers, the node is still in the list (it just may not have all the shortcuts one would expect). Some algorithms might detect this situation and later "fix" the index levels (this could be considered a form of *helping* as well, where a search or another insertion notices a missing level and optionally inserts it – though such fixes are not common since it complicates things).

To summarize insertion in pseudo-code (simplified):

```
function insert(key, value):
    loop:
       preds[], succs[] = findPredecessors(key)  // search each level
       if succs[0].key == key:
           return false  // already present (duplicate)
       newNode = Node(key, value, height = randomLevel())
       // initialize forward pointers
```

```
    for i in 0 .. newNode.height-1:
        newNode.forward[i] = succs[i]
    // try to splice at level 0
    if !CAS(preds[0].forward, succs[0], newNode):
        continue loop  // retry if base level insertion fails
    // success at level 0, now link higher levels
    for i in 1 .. newNode.height-1:
        while true:
            if CAS(preds[i].forward, succs[i], newNode):
                break  // inserted at level i
            // if failed, update preds[i], succs[i] by re-scanning level i
            preds[i], succs[i] = findPredSuccOnLevel(i, key)
            if (some condition indicating node should not be at this level):
                break  // optionally skip adding index if too contended
    newNode.fullyLinked = true  // now fully inserted
    return true
```

This pseudo-code omits some details (like how to handle concurrent deletions affecting `preds` and `succs`), but it captures the general approach: use CAS to insert at bottom, then CAS for each upper level. Notably, if something goes wrong at an upper level, we might break out early – either way, the node is in the list after the bottom CAS.

Deletion

Deletion (removing a key) in a lock-free skip list is typically a two-phase affair: first **logically mark** the node as deleted, then **physically unlink** it from all levels. This mirrors the common approach in lock-free linked lists (Harris's algorithm) where you mark a node and then remove it, allowing other threads to see the mark and help with removal [6] [23]. The challenge is that in a skip list, a node appears in multiple lists (levels), so marking must ensure the node is ignored in all those lists, and unlinking must remove all those pointers.

Find the node: The delete operation will first search for the target key, similar to insert's search. It identifies the predecessor and successor at each level, and specifically the node to delete (`targetNode = succs[0]` at level 0, if that node's key == target). If the key isn't found, the deletion is done (return false or not found).

If the node is found, the node may be in the midst of being inserted or deleted by another thread. If we use a `fullyLinked` flag, we should only proceed if `targetNode.fullyLinked == true` (meaning the node was completely inserted) and `targetNode.marked == false` (not already being deleted). If the node isn't fully linked yet, we might wait or retry because it would be improper to remove a half-inserted node (some implementations simply don't allow this scenario: they ensure a node is fully linked immediately at insert).

Logical deletion (marking): We now logically delete the node. The aim is to *mark* the node in a way that other threads will treat it as gone. There are a few strategies:

- **Mark flag:** The node might have a `marked` boolean field (as used in many algorithms) that we set to `true` using an atomic operation. If we do this, any thread performing a search or insertion that encounters the node should check this flag and skip the node if it's true. Setting this flag is usually done with an atomic CAS (from false to true) on the node's field. If we find it's already true, someone else beat us to the punch (the node is already being deleted).

- **Value nulling + marker node:** In Java's skip list, instead of a separate boolean flag, they use the value field as a marker (setting the node's value to `null` signals logical deletion) [14]. They then proceed to CAS a special *marker node* into the `next` pointer. We'll elaborate on that marker in a moment. The key point is that a search will treat a node with `value == null` as logically deleted [14].

- **Top-down marking of forwards:** Another approach (used in some research algorithms [24] [25]) is to mark the pointers in the node's forward array. For example, you might mark each forward pointer at higher levels by setting a low bit or using an atomic marked reference to indicate "this level pointer is deleted". One would start from the top level of the node's tower and mark each level's pointer, down to level 1. Only after all higher-level pointers are marked do we mark level 0. The idea is that until level 0 is marked, the node is still considered part of the set (because searches that drop to level 0 might still find it). Once level 0 is marked (or the value is null, etc.), the node is logically deleted. Marking top-down ensures that nobody will find this node via upper levels once we start the process (they'll see a marked pointer and either help finish deletion or drop down).

In Java's algorithm, the logical deletion is done by **CASing the node's value to null** (step 1 below) [26]. This is effectively the logical removal – after that, any map lookup will treat the key as absent. Other threads can still be traversing the node's next pointers, but they will notice the null value and know this node is removed.

Physical removal (unlinking): After marking, the node needs to be *unlinked* from each level's linked list so that it no longer appears at all. This usually involves CAS operations on the predecessor's forward pointers. There is a nuance: we must consider that multiple threads could try to delete the same node concurrently, and also that some other thread might already be helping with the unlinking once it sees the node marked. The unlinking phase is where **helping** is common: if thread A marks node X and is about to unlink it, but gets delayed, thread B (doing a traversal) might see that X is marked and perform the unlink CAS itself. That's fine – the goal is simply that eventually the node is removed from all lists.

A possible unlinking sequence: we have the `pred` array from our search (predecessors at each level). We can iterate from level 0 up to `targetNode.height-1` (or possibly top-down; the order doesn't actually matter as long as all levels get done and we ensure certain ordering for linearizability — usually level 0 removal might be done last for linearization). For each level i, do `CAS(pred[i].forward[i], targetNode, targetNode.forward[i])`. This means: "if pred at this level still points to the target node, swing it to skip over the target to target's successor at that level." Some of these CASes may fail, if another thread already changed that pointer (either via another insert or another deletion that affected that level). If a CAS fails, we simply skip it — either the level is already handled or we'll detect on retry that it's done. In

many algorithms, once the node is marked, they might attempt the unlink at level 0 first or last depending on convention, but since the node is marked, even if it's physically present in some level, it is logically gone.

The Java implementation's deletion has three main steps which are worth noting (they correspond to the general approach above) [27] [28] :

Deletion Step	Atomic Action & Effect
1. Mark node logically	CAS the node's value field from non-null to `null` [26] . After this, no thread will consider this node as containing a valid mapping (logical deletion achieved). The node is still linked in the lists, but effectively "invisible" to queries [26] .
2. Prevent further linkages	CAS the node's `next` pointer to point to a new **marker node** [29] . This marker is a dummy node that signals "the previous node is deleted." Once this CAS succeeds, no other insert can attach new nodes after the target node, because any CAS to target's `next` will fail (it's no longer pointing to the original successor but to the marker) [29] . This step is like marking the pointer itself, preventing late-arriving insertions at upper levels from seeing an intact pointer.
3. Unlink from list	CAS the predecessor's forward pointer to skip over the target node (and the marker) to directly point to the target's successor [28] . This physically removes the node from the list at that level. After this, no new traversal will encounter the node. The node (and marker) will eventually be garbage-collected [28] . If this CAS fails, it means some other thread already helped by unlinking the node, which is fine.

Once step 1 is done, the node is logically gone. Step 2 and 3 may be done by the deleting thread or by *helping* threads. The design ensures system-wide progress: if the deleter gets stalled after step 1, any other thread encountering the half-deleted node will see the null value and can proceed to step 2 or step 3 on behalf of the deleter [23] . This way, deletion is lock-free – it doesn't matter if one thread pauses, others ensure the work completes and the node gets removed. In Harris's original linked list algorithm and its variants, this kind of helping (especially the physical unlinking in step 3) is crucial so that the data structure doesn't accumulate garbage nodes or leave searches stuck following deleted nodes.

It's worth noting that some algorithms do the marking in a specific order to maintain correctness. For instance, one method (Fomitchev & Ruppert 2004) marks a "flag bit" on the predecessor's pointer to indicate a deletion is in progress before actually doing it, to avoid long search chains through marked nodes [30] [31] . But modern simpler designs often rely on just marking the node itself and using helping to clean up.

After the node is unlinked from level 0, it is completely inaccessible to searches. At this point, if we're in a non-GC environment, we would retire the node's memory (e.g., put it in a hazard pointer retire list). Hazard pointers would have been protecting it during any reads; once no hazard pointers point to it and it's unlinked, it can be safely freed. In our example, Java's GC takes care of actual memory reclamation (the unused node and marker will be collected once no references exist).

Concurrency considerations: Multiple threads may attempt to delete the same node concurrently. In our example, suppose two threads both want to remove key 50. They both find the node. One will succeed in CASing the value to null (step 1); the other will fail its CAS (value already null). At that point, the second

thread knows someone else is deleting the node, and it can either help with the next steps or simply consider the node deleted and return (since logically the deletion happened). Either way, they won't conflict in a harmful way – the CAS ensures only one marks the value. Similarly, with marker insertion (step 2), only one thread will do the actual insertion; any others will find that `next` is already a marker. And unlinking (step 3) can be done by whichever thread sees an opportunity. This cooperative interaction is how lock-free algorithms maintain correctness without explicit locks: **CAS serves as a handshake** between threads to divide work.

Helping Mechanisms and ABA Handling

Throughout these operations, we've seen the idea of **helping** multiple times. Helping means if a thread notices some structure that indicates another thread's operation is incomplete (like a marked node not yet unlinked, or a new node in the list not yet fully indexed), it will take steps to finish that operation instead of waiting or ignoring it. This ensures **lock-free progress**: one thread's pause doesn't prevent overall progress, because others will finish its work if needed [23] . For example, if thread A is in the middle of deleting node X (marked it null but hasn't swung the predecessor's pointer yet) and thread B comes along, B will see X is marked and can CAS the predecessor to skip X. Thread A upon resuming might find it's already done. This way, **no thread is ever blocked** waiting for another; at least one operation will complete. The contains/search operations often implicitly help by skipping or unlinking marked nodes, and insertions might help by e.g. adjusting a neighbor's index if it notices something out of place (though insert help is less common than delete help).

The **ABA problem** was addressed in part by hazard pointers in memory management, but it also surfaces in pointer manipulation. In our skip list algorithms, we often rely on the fact that a pointer's content changes (like from pointing to a real node to pointing to a marker node) so that any CAS by another thread expecting the old node will fail. For instance, when a node is marked logically by nulling its value and then its `next` is changed to a marker, this ensures that any thread that *thought* it might append something after the node will fail because the `next` pointer is no longer what it was. Thus the deletion sequence inherently provides **ABA safety** for the pointer: the node's `next` can never go back to its original successor once a marker is in place [32] [29] . Additionally, hazard pointers prevent a scenario where a node's memory is freed and reused as a new node at the same address while another thread still holds a reference. As a result, although lock-free algorithms must be designed with ABA in mind, techniques like versioned pointers or hazard pointers and careful ordering of updates mitigate ABA issues in practice.

Node Structure, Pointers, and Atomic Operations

A skip list node in a lock-free implementation typically contains: - The **key** (and value, for maps). - An array of **forward pointers** (often called `forward` or `next` pointers) for each level the node is present on. - Flags or fields for concurrency: e.g., an atomic boolean `marked` (to indicate logical deletion) and possibly `fullyLinked` (to indicate the node has been completely inserted at all levels) [20] [33] . In Java's implementation, they avoided an explicit `marked` flag and instead used `value == null` as the marker [14] .

These forward pointers are all managed as atomic references. In C++ one might declare them as `std::atomic<Node*> forward[k]`. In Java, `volatile` fields or `AtomicReference` ensure that updates are visible and CASable. The **head** of the skip list is a dummy node of maximum level (often with

key -∞) that serves as the start of each list, and sometimes there's a tail dummy (key +∞ or null) at the end of each list as a sentinel.

There are two popular design choices for node structure: 1. **Single node with multiple forwards:** Each node has a fixed-size array for forward pointers (size = maxLevel, though only first `node.height` of them are used). This is memory-efficient (one allocation per node) and simple. CAS operations need to operate on each `forward[i]` when updating links. 2. **Tower of index nodes:** The base level node (with the key/value) is accompanied by separate index nodes for each higher level. Each index node contains a pointer to the base node or the next index down, plus a `right` pointer to the next index at the same level. This is how the Java `ConcurrentSkipListMap` is implemented internally. The benefit is some separation of concerns (the base list can be managed with one algorithm and the index levels with a simpler algorithm that can afford to occasionally fail without retry [21]). The drawback is more allocations (one per level for each node). In the Java code, these index nodes are created on the fly during insertion and removed during deletion.

Regardless of approach, **all pointer updates use CAS**. For example: - In single-node design, to remove a node at level i, do `CAS(pred.forward[i], targetNode, targetNode.forward[i])`. - In index-node design, to remove an index node, CAS its predecessor's `right` pointer to skip it.

The **atomic operations** must also ensure proper memory visibility. Usually, the high-level language's atomics handle this (e.g., a successful CAS has release semantics for the writer and acquire for the reader in many implementations, which is enough to ensure nobody sees a partially initialized node). Libraries like libcds allow tuning the memory model: one can use a relaxed memory order for non-critical levels if desired [8], but the default is to play safe.

Memory management details: When using hazard pointers or epochs, a node that has been unlinked is retired. The actual memory free happens after a grace period. In some skip list implementations, an extra complexity arises: when a node is deleted, it might still have incoming pointers from higher-level indices or even backward pointers used for recovery (as in some research algorithms [34]). The blog by Fraser on a C skip list with hazard pointers mentions that they introduced reference counting on nodes in addition to hazard pointers because of back-pointer "garbage collection roots" – essentially, a deleted node had a pointer to its predecessor, which could keep the predecessor alive longer [34]. They handled it by bumping a refcount on those predecessors to ensure safe reclamation [35]. The takeaway is that one must be very cautious that **no thread can follow a pointer to a freed node**. Techniques like hazard pointers, reference counts (used sparingly), or epoch-based schemes are non-trivial but necessary in languages without automatic GC.

ABA avoidance: In addition to hazard pointers, some implementations use **tagged pointers** – for example, augmenting a pointer with a version counter that increments on each update. This way, even if a pointer value is reused for a different node, the counter differentiates it. However, this doubles the CAS width if using double-word CAS, which can be expensive. Most skip list algorithms instead rely on not reusing node addresses until safe (hazard pointers) or use the marker-node trick (so that a pointer never goes back to its old value – it goes to marker and then to something else). The Concurrency Kit documentation notes that CAS-based algorithms are prone to ABA and typically require ABA prevention via counters or safe memory reclamation (SMR) [36] [37] – which aligns with what we have discussed: hazard pointers (one form of SMR) solve this in practice.

Real-World Implementations and Examples

Lock-free skip lists have been studied and implemented in both academia and industry. A well-known implementation is Java's `ConcurrentSkipListMap`, which is a fully concurrent sorted map introduced in Java 6. It is **lock-free** and uses skip list internally (as opposed to `TreeMap` which is single-threaded red-black tree) [38]. In `ConcurrentSkipListMap`, all updates are done via CAS on `volatile` fields, and there are **no locks** for its main operations [38]. The design was influenced by the algorithm of Herlihy et al. (2007–2010) which provided a lock-free skip list with a wait-free contains operation (the Java version's `containsKey` is wait-free). The JDK implementation (largely designed by Doug Lea) uses the techniques we described: logical deletion by nulling the value, insertion of marker nodes, and out-of-band index management [12] [27]. Its internal commentary explicitly references the lock-free list algorithm by Harris and Michael as a foundation [39]. The use of marker nodes instead of mark-bits was a deliberate performance choice: *"Rather than using mark-bits to mark list deletions… splice in another node… using otherwise impossible field values"* [12]. This avoids the overhead of atomic marked references on each pointer and takes advantage of the Java GC to clean up markers quickly [40] [10]. As a result, traversal in the Java skip list only needs to check one node ahead for a marker rather than checking a mark bit on every pointer dereference [41], which is a nice optimization.

On the C/C++ side, there are libraries such as **LibCDS (Concurrent Data Structures)** which include a lock-free skip list set/map. The CDS skip list implementation is based on the algorithm in *"The Art of Multiprocessor Programming"* (by Herlihy & Shavit, 2008) [11], which presents a lock-free skip list design in Chapter 14.4. That design is similar to what we've described, including using hazard pointers for memory management. In the CDS library, you can parameterize the skip list with different garbage collectors (like reference counting, epoch-based reclamation, or RCU) [11]. The skip list nodes in CDS are single objects with an array of atomics for forwards, and deletion uses a marked-bit technique (they use tagged pointers or a bit-stealing trick since they integrate with their general-purpose hazard pointer framework). The library ensures operations are linearizable and it provides statistics counters to observe things like how often insert CAS retries occur, etc.

Research-wise, the first lock-free skip list is often credited to Sundell & Tsigas (circa 2003) who described a lock-free skip list in a technical report [42]. Their approach also built on a lock-free list and included some complexities like allowing only one deletion at a time on the bottom list to simplify things (as referenced in later analyses [43]). Fomitchev & Ruppert (PODC 2004) introduced a lock-free skip list that added **backlink pointers** to help traversals recover from encountering deleted nodes [30]. They also used **flag bits** on next pointers to indicate a deletion in progress, so that a traversal doesn't have to follow long chains of backlinks one by one [30] [44]. These innovations improved performance by preventing a search from getting "stuck" in a long sequence of marked nodes. Their skip list achieved lock-free property and they provided a proof of linearizability and an amortized complexity analysis.

More recent research has continued to build on skip lists. For instance, **Herlihy, Shavit, Luchangco, and Lev (2007)** came up with an *optimistic* skip list algorithm that actually uses fine-grained locking (not lock-free but very scalable) – demonstrating the spectrum of techniques (sometimes a combination of optimistic locks and lock-free reads can perform well). In 2010, Herlihy et al. also had a lock-free skip list algorithm (with wait-free contains) which has influenced practical implementations (possibly the basis of a patent [45]). **Jiffy (Kobus et al. 2021)** is a lock-free skip list variant that supports *batch updates and snapshots* for use in databases [46], showing that even decades later, skip lists are an active area for advanced features.

Use cases: Concurrent skip lists are used in systems where a sorted data structure with concurrent access is needed. One example is as an **in-memory index** for key-value stores. Skip lists provide sorted order iteration and fast seeks, which is why they were used in MemSQL and LevelDB's MemTable (though Google's LevelDB memtable is single-threaded, Cassandra and others have used concurrent skip lists for similar purposes). The Java `ConcurrentSkipListMap` is often used in concurrent algorithms that need ordered sets or maps, like in implementing priority queues, ordered work queues, or maintaining subscriptions in publish-subscribe systems where locks would be too slow. A lock-free skip list can also serve as the building block for a **concurrent priority queue** (by always taking the smallest element, which is at the head of the skip list). In fact, researchers have built priority queues on skip lists that scale well by avoiding the bottleneck of a heap's lock [47] [48].

In production, one must carefully choose memory reclamation methods. Hazard pointers are straightforward but can incur overhead under high thread counts due to scanning retire lists. Epoch-based reclamation is faster in the absence of long-lived operations but requires threads to cooperate (if a thread is slow to enter a new epoch, it can delay garbage). Some specialized concurrent skip list implementations (e.g., those in databases) have used epoch-based reclamation with quiescent state detection, since operations on the skip list are usually short.

Conclusion

Lock-free skip lists are a sophisticated fusion of a classic probabilistic data structure with non-blocking synchronization techniques. By using CAS and careful design, they allow multiple threads to perform searches and updates without ever locking the structure, yielding high scalability. We saw how skip list operations (search, insert, delete) are orchestrated with atomic steps: inserting a node involves a CAS at the bottom level and possibly several CAS for index levels, and deleting a node involves marking it (logical delete) and unlinking it (physical remove) with help from other threads. Key challenges such as ensuring consistency across levels, handling concurrent modifications, and reclaiming memory are met through strategies like marking nodes, using helper "marker" nodes or flags, hazard pointers for safe memory free, and helper threads completing each other's work.

The end result is a data structure that provides thread-safe sorted set/map operations with expected O(log n) efficiency and without locking pauses. Real-world implementations like Java's `ConcurrentSkipListMap` attest to the practicality of these algorithms, and they have become a standard tool in the concurrency toolkit. However, implementing a lock-free skip list correctly is a considerable engineering challenge – one must get every detail right (from atomic pointer semantics to memory reclamation) to avoid subtle bugs. Thankfully, the literature and existing libraries provide blueprints (e.g. Herlihy & Shavit's textbook chapter [11], or proven implementations) that a practitioner can follow. With those in hand, one can implement a lock-free skip list in C++ (using `<atomic>` and perhaps a hazard pointer library) or in other languages, gaining a data structure that offers **non-blocking performance** and robust thread-safe behavior for ordered data.

Sources:

- Herlihy, M., & Shavit, N. *The Art of Multiprocessor Programming*, Chap. 14.4: Lock-Free Concurrent Skiplist [11].
- JDK Source for `ConcurrentSkipListMap` (Java 8/11), concurrency comments [6] [27] [28].

- Fomitchev, M., & Ruppert, E. (2004). *Lock-free linked lists and skip lists*. PODC'04 [30] [12].
- Sundell, H., & Tsigas, P. (2003). *Fast and Lock-Free Concurrent Priority Queues* (tech report, introduced first lock-free skip list) [42].
- Hazard pointers for lock-free memory reclamation (M. Michael, 2004) [9].
- **Additional citations inline.**

---

[1] [2] [3] Skip list - Wikipedia
https://en.wikipedia.org/wiki/Skip_list

[4] [16] [17] [18] [19] [20] [24] [25] [33] [46] 15618_project_final
https://supertaunt.github.io/CMU_15618_project.github.io/15618_project_final.pdf

[5] [30] [31] [42] [44] paper28.dvi
https://www.eecs.yorku.ca/~eruppert/papers/lfll.pdf

[6] [7] [10] [12] [13] [14] [15] [21] [22] [23] [26] [27] [28] [29] [32] [39] [40] [41] java/util/concurrent/ConcurrentSkipListMap.java - platform/prebuilts/fullsdk/sources/android-29 - Git at Google
https://android.googlesource.com/platform/prebuilts/fullsdk/sources/android-29/+/refs/heads/androidx-wear-release/java/util/concurrent/ConcurrentSkipListMap.java

[8] [11] cds: cds::container::SkipListSet< cds::urcu::gc< RCU >, T, Traits > Class Template Reference
https://libcds.sourceforge.net/doc/cds-api/classcds_1_1container_1_1_skip_list_set_3_01cds_1_1urcu_1_1gc_3_01_r_c_u_01_4_00_01_t_00_01_traits_01_4.html

[9] [34] [35] Concurrent skiplist implementation
https://rsea.rs/skiplist/

[36] [37] Concurrency Kit - Towards accessible non-blocking technology for C
https://blog.linuxplumbersconf.org/2012/wp-content/uploads/2012/09/2012-lpc-scaling-concurrency-kit-albahra.pdf

[38] java - Why is there a ConcurrentSkipListMap, but no unsynchronized version? - Stack Overflow
https://stackoverflow.com/questions/26917345/why-is-there-a-concurrentskiplistmap-but-no-unsynchronized-version

[43] [PDF] Efficient & Lock-Free Modified Skip List in Concurrent Environment
http://ijcat.com/archives/volume4/issue3/ijcatr04031007.pdf

[45] [PDF] Maurice Peter Herlihy - Brown Computer Science
https://cs.brown.edu/~mph/HerlihyCV.pdf

[47] Fast and lock-free concurrent priority queues for multi-thread systems
https://www.sciencedirect.com/science/article/pii/S0743731504002333

[48] Fast and lock-free concurrent priority queues for multi-thread systems
https://www.researchgate.net/publication/4023231_Fast_and_lock-free_concurrent_priority_queues_for_multi-thread_systems