CONCURRENT SKIP LISTS

Skip Lists are dynamic, probabilistic-based search structures that are both simple and efficient to implement. This paper starts off with a brief introduction to skip lists and the basic search, insert and delete operations. I then describe 2 algorithms used to maintain concurrency amongst skip lists: The first one is a lock-based algorithm by Prof. Pugh and the second algorithm is a lock-free implementation by Herlihy and Shavit.

I also touch upon the Java ConcurrentSkipListMap Class which is a lock free implementation of Skip Lists by Prof. Doug Lea and briefly describe the Java Memory Model that controls how concurrent instructions are executed on the JVM. I round off by describing the Java ConcurrentModificationException Class that is triggered if any rules of concurrency are violated by the programmer.
Introduction To Skip Lists

Skip Lists were first created by Prof. William Pugh in 1989 so they are a fairly recent data structure. The underlying structure in a Skip List is a Sorted Linked List so many operations that can be performed on a List can be extended with some modifications to apply to skip lists. Simply stated, a skip list can be thought of as a collected of sorted linked lists, structured into levels. The lowermost level (Level 1) contains all the elements of the list. Each higher level contains about half the elements from the previous level in an ideal skip list, although in general, this rule is not followed strictly. Hence, unlike AVL or Red-Black Search Trees, Skip Lists do not have to be rebalanced to remain efficient.

Here is what a skip list looks like:

All elements are present in Level 1, with randomly selected elements present in level 2, level 3 and so on. Basic operations performed on a list are described here.

Search

Search starts at the top left corner of the topmost level. Each element is traversed. If we reach a node that is greater than the one being search we move one level lower and continue searching from there. This process continued until we reach Level 1. If the element is found the search is successful. If the element is not found even in the lower most level, the search fails.

For a search operation the bound on a skip list is randomized. However all searches are expected to give a $O(\log n)$ time with a probability of $(1 - 1/n ^ {\alpha})$ where $\alpha$ is the number of search operations performed on the skip list.

Insert

Insertions are for the most part, extensions of inserting into a sorted linked list. Using search routines, the predecessor and the successor of the new element to be inserted are found. Then a pointer is created from the new element to the successor and the pointer of the predecessor is changed to point to the new element. For a
skip list, while inserting a new element, the level is chosen at random (there is usually a cap on the max possible level in a skip list). The probability of being level 1 is 50%, level 2 is 25%, level 3 is 12.5% etc. The new node is then spliced into the list between the predecessor and successor nodes. Depending on the skip list implementation, the splicing is done either starting from the topmost level going to the lower most level (Level 1) or the other way round. This is discussed in more detail in the later sections (concurrent skip lists). Some implementations of skip lists allow modification of values if the new node to be inserted is found in the skip list while other implementations report that the insertion failed.

Delete

Just like in Insert, for a delete it helps to start with how a delete is performed on a sorted linked list and then enhance that concept to a skip list. For a simple delete, the pointer of the victim is set to null and the predecessor of the victim is set to point to the successor. This way there are no references to the victim and it is then deleted or garbage collected.

To extend this to a skip list, instead of deleting a single element, we splice the list. Generally, the direction in which the list is spliced for a delete is the opposite of insert (top most level to lowermost level or vice-versa). For each level of the element the next pointer is set to null and the predecessor is set to point to the successor on that level. Once we get into concurrency, we discuss the concept of a logical delete and a physical delete.
Concurrent Skip Lists

Various implementations have been proposed for maintaining skip lists in a multi-threaded environment. It is important that the concurrency and correctness is maintained during all operations performed on a skip lists during these implementation. In this paper, I describe 2 different implementation of skip lists, one uses locks while the other one is lock-free. In each of these cases I explain how concurrency is maintained. There is also a LazySkipList implementation which is lock-based and was discussed in class. I will not go into details here but you can refer to the paper in reference #4 for more details.

A demonstration for insert and delete routines for each of the implementations described here is in the powerpoint presentation I used for this project.

Lock-based Implementation[2]

As the name suggests this implementation uses locks to maintain concurrency. In keeping with the approach taken in this paper so far, I will start off with giving the implementation in a sorted linked list and extend that to a skip list. Code snippets for each operation are presented in detail in Prof. Pugh’s paper (Reference #2) which can be referred to for a more detailed explanation on this method.

There are 2 different search operations described in this implementation: A WeakSearch function and a StrongSearch function. The WeakSearch is similar to the regular search operation described above. StrongSearch uses locks to ensure that the forward pointer to the node being searched has not been changed by another thread during the search operation. A lock is obtained on that pointer during the search operation which is released right before the function returns.

Insert

For the insert routine, the predecessor and successor nodes are first determined using the search function. A new node is then created that points to the successor node. A lock is then assigned to the “next” pointer of the predecessor node (which currently points to the successor of the new node). This prevents any other thread from modifying the pointer. The predecessor node is now made to point to the new node and the lock is then released, thus completing the insertion.
In a skip list, the max level of the new node is randomly selected. The node is first spliced into the existing list by first finding all the predecessors and successors at each level and then assigning the pointers of the new node at each level to the respective successors. Here is where concurrency really begins. Starting the the lowermost level, a lock is assigned to the “next” pointer of the predecessor node of the new node. This pointer is then changed to point to the new node. Once the new node is inserted at the lowermost level, it is part of the list and is picked up by any subsequent search operations. This implementation works on the principal that the level of the node only affects the efficiency of the skip list operations but not the correction. Hence having the node in the lowermost level is enough to make the skip list correct. Once this is done, the same operation of assigning a lock to the next pointer of the predecessor, changing it to point to the new node and releasing the lock, is performed until the maximum level of the new node is reached. This completes the insertion routine.

**Delete**

The delete operation in this implementation operates on the idea of pointer reversal. If a node N is to be deleted, the “next” pointer of N's predecessor is set to point to N's successor. Once that is done, N's next pointer is set to point back to its predecessor. This ensures that any thread that is currently working on node N will still have the correct result and not be left pointer-less.

The node to be deleted (the victim) is first found using the search routine. A lock is then assigned to that node. After this 2 more locks are assigned: one to the next pointer of the victim and the other to the next pointer of the victim's predecessor. The predecessor pointer is then changed the point to the victim's successor node while the victim’s pointer is changed to point to its predecessor. The 2 locks are then released. Instead of immediately deleting the victim, it is added to the garbage collection queue and deleted once all references to it are removed (across all threads).

To extend this to a skip list, this process starts at the top most level. We already have the max level for each node stored so while acquiring a lock on the node, we only apply the lock at the top most level of that node. If the search routine initiated by another thread finds that the top most level is locked, it returns false. Then starting from the top most level the 2 pointers are locked, modified and the locks released as described above, all the way till the last level. After that the lock on the top most level of that node is released and the node is queued for garbage collection, which completes the delete operation.
Lock-free Implementation\textsuperscript{[3]}

This implementation is adapted from "The Art of Multiprocessor Programming" by Maurice Herlihy and Nir Shavit. It uses the atomic compareAndSet operation to implement a lock-free version of the skip list.

The search operation here does slightly more besides searching. We look into that after the insert and delete. Let's look at the insert and delete assuming a regular search operation.

\textbf{Insert}

Let us do this directly for the skip list instead of expanding from a sorted list as the first part is quite similar to what was done in the locked-based implementation. The predecessor and successor nodes are found for each level using the search routine for the new node that needs to be inserted. The new node is then created with its pointers pointing to its successor nodes in each level. (The max level, as before, is chosen randomly.) Now, starting at the lowermost level, the predecessor nodes' pointers are changed using the atomic compareAndSet operation to point to the new node. The algorithm compares that the predecessor is the same as that was returned by the search function as well as that it is pointing to the same successor as returned by the search. If either of this has changed, compareAndSet returns false and the search is done again for that level returning a new set of predecessors and successors. CompareAndSet is tried again until successful. This can potentially result in an infinite loop if new elements are constantly being added or removed but the chances of that happening in practice are almost nil. Once all predecessors point to the current node, the insert is complete.

\textbf{Delete}

The concept of a logical delete is used in this implementation. This means that the node is not really deleted but is "marked" to be deleted. Some other routine completes the delete operation at a later point. First the node to be deleted is found using the search operation. Starting from the topmost level, for each level the next pointer of the victim is "marked" to be deleted but is not deleted. If a search is performed for this node and the search routine finds that any of its pointers are marked, it returns false.
After the logical delete is performed, the search() method is called. This is where the actual delete happens. The search routine uses compareAndSet to change the pointer from the victim's predecessor to point to the victim's successor. Like it does for the insert, if the compareAndSet fails for any level, search is executed again to find the updated predecessor or successor for that level. Once all pointers have been changed from the topmost to the lowermost level, the delete operation is complete.
Java ConcurrentSkipListMap Implementation\textsuperscript{[5]}

The ConcurrentSkipListMap class in Java was designed by Prof. Doug Lea as part of the Java version 1.5 release. It uses the java.io.Serializable framework for a lock-free implementation of skip lists. This class is the concurrent version of the Java SortedMap class (which is thread-safe but doesn't allow concurrent operations). Java also provides the ConcurrentSkipListSet class (not discussed here) which is the concurrent version of the SortedSet class.

A unique feature of this implementation is that it uses both the nodes' key and value to check and set deleted nodes. To do this, it takes advantage of the fact that if a search fails, the function returns NULL. When a deleted operation is performed, it sets the value of that node to NULL for performing a logical delete. If another thread searches for the node and it is found, it still returns NULL indicating the node is absent from the list. Details of the insert and delete operations are discussed below.

**Insert**

The developer can pass a flag to the insert method indicating whether to return false if insert is performed on an existing value or the update the value of that key. The insert operation is performed in a similar fashion to other inserts for the most part. The predecessor and successor nodes are found at each level. Starting from the top most level, the algorithm checks if the previous node still exists or has been deleted by another thread. If its marked or the value is null, then the current previous node is found. Once a stable previous node is found, an atomic operation is used to update its pointer to point to the new node. This is done for each level until level 1 and then the node insertion is complete.

**Delete**

The delete operation starts from level 1 and progresses to the higher level. It differs from the other implementations discussed here in the following way: The node to be deleted is first found using the search function. A "marker" node is then inserted as the next node after the victim and the victim points to that. This performs a logical delete so if the search comes across the victim, it checks to see if its points to the marker node and if so, the search returns false. This also has another advantage in maintaining concurrency in that no new nodes will be appended to the victim. Then an atomic operation changes the pointer of the victims predecessor to point to its successor (now the marker's successor). After this the marker and the victim are removed. Once this is done for the max level of the node, delete is completed.
Java Memory Model [7]

The Java Memory Model (JMM) specifies the minimum guarantee that the JVM must make about when writes to variables become visible to other threads. This is important because the memory models differ from processor to processor so unless the order of execution of commands is not specified, it can be somewhat randomized in case of inter-thread interaction. JMM standardizes these rules across various platforms for the Java Programming Language.

The JMM is specified in terms of actions which include reads and writes to variables, locks and unlocks of monitors, and starting and joining with threads. It defines a partial-ordering called happens-before for each action within the program. Partial ordering means that the relation is asymmetric, reflexive, and transitive but for any two sets x and y, it need not be that they are partially-ordered with respect to each other; i.e. they elements with x and y may be partially ordered but x and y are independent of each other.

Some rules for happens-before are:

- Each action in a thread happens-before every other action in the thread that comes later in the program order (Program order rule).

- An unlock on a monitor lock happens-before every subsequent unlock on that same monitor lock (monitor-lock rule).

- A write to a volatile variable field happens-before every subsequent read of that same field (Volatile variable rule).

- A call to Thread.start on a thread happens-before every action started by that Thread (Thread start rule).

- Any action in a thread happens-before any other thread detects that thread has been terminated (Thread termination rule).

- A thread calling interrupt on another thread happens-before the interrupted thread detects the interrupt (Interruption rule).

- The end of a constructor for an object happens-before the start of the finalizer for that object (Finalizer rule).

- If A happens-before B, and B happens-before C, then A happens-before C (Transitivity).
Java ConcurrentModificationException Class

As mentioned above, the ConcurrentSkipListMap and the ConcurrentSkipListSet classes in Java are concurrent extensions of the SortedMap and SortedSet classes. Even though the later classes are not concurrent, they still guarantee a thread safe operation. This also applied to other data structures like the HashTable, ArrayList etc which are all considered thread-safe. For ensuring thread safety, Java prevents concurrent modification of these data structure unless explicit locks are provided. In the absence of locks, Java detects the concurrent operations that tries to modify data in one of these data structures and throws the ConcurrentModificationException. The most common example of this is while using an Iterator on one of these data structures. If an attempt is made to add or remove an element while iterating on the data structure, the ConcurrentModificationException is raised. This is done by keeping a count of how many elements were originally in the list and if the number of elements changes, the exception is generated.\[6]\n
It is interesting how the term thread-safety is different from the term concurrency and if a data structure or a model is thread safe, it merely implies that it prevents the corruption of data by concurrent modification but does not necessarily allow threads to alter the data concurrently. Using classes like ConcurrentSkipListMap or ConcurrentHashMap guarantees concurrent modification. So these classes do not use this exception despite not using locks.
References


