Lock-free Binary Search Tree and Skip List

Name: Chenhao Li; Andrew ID: chenhao3

April 29, 2022

1 Summary

I implemented the lock-free binary search tree and skip list. I tested their correctness and evaluated their performance under different workload patterns and configurations. Finally, I come up with my observations about trade-offs in these two data structures.

The final presentation can be found at https://youtu.be/yllEXUBZE5U.

2 Background

Dictionary is an important abstract data type in real-world applications. The most basic dictionary needs to support `search`, `insert` and `remove` operations for given keys. It can also be extended to associate value(s) for a specific key.

The binary search tree (BST) and skip list are two efficient implementations for the dictionary when a partial order relationship is defined on the key set. BST associates each key with a node and organizes the nodes in a tree structure. In the original BST, the key in the parent node should be strictly greater than keys in the left subtree, and strictly smaller than keys in the right subtree. This allows Dictionary operations to only explore one branch based on the comparison between the key in the current node and the given key.

The asymptotic time complexity of tree operations is proportional to tree height. If the tree is properly balanced, it will be $O(\log(n))$, where n is the number of tree nodes. However, the tree can degenerate into a long chain, in which situation the operations will take $O(n)$ time. This situation is easy to achieve, such as continuously inserting incremental keys into the tree. There are many techniques to rebalance the tree, such as the Red-black Tree and AVL Tree. They usually require rotation operations involving multiple nodes.

Skip list, on the other hand, is a linear structure. It consists of multiple layers of ordered singly-linked lists. The number of layers in a node is determined by an exponential distribution based on a certain probability. Each node stores a set of `next` pointers equal to its number of layers. The bottom layer is an ordinary linked list, and higher layers are connected to skip multiple nodes.

Such a probabilistic data structure allows $O(\log(n))$ operations on average. In the worst case, it is possible that the operations also take $O(n)$ time. However, this situation is very rare, and it is not because of any specific operation pattern, but because of the bad result of the random number generator (really bad luck instead of malicious user input).

Serial algorithms for both the data structures are relatively simple, but are difficult to be directly extended to parallel versions, let alone lock-free versions. Because they involve complex pointer accesses and contention for common nodes (the root of the tree, the head node of the skip list). Many papers and applications suggest that skip list is easier to parallelize than BST because it does not involve non-local rebalance operations. This is the reason why I chose to implement these two data structures. I hope to compare their implementation difficulty, running speed, and memory overhead, so as to analyze whether the skip list is really a better choice for a concurrent dictionary.

3 Approach

BST and skip list both implement the following dictionary interface. The return value of `search` indicates whether the key is present in the dictionary. `insert` tries to insert the key. It returns true if the key originally doesn't exist and the insertion succeeds, and returns false otherwise. `remove` tries to remove the key. It returns true if the key originally exists and the removal succeeds, and returns false otherwise.

```
template<typename K>
class Dictionary {
  virtual bool search(K key);
  virtual bool insert(K key);
  virtual bool remove(K key);
};
```

Please note that this is only for illustration, and there is actually no such abstract class in my project. My project makes heavy use of C++ templates instead of traditional object-oriented programming.

The baseline implementation is an `std::set` coupled with a readers-writer lock (`pthread_rwlock_t`). There are nothing worth saying about it.

3.1 Binary Search Tree

3.1.1 Overview

My implementation is based on [NM14].

It does not strictly follow the original BST definition. It has two kinds of nodes: leaf nodes storing all keys and internal nodes storing parts of the keys for routing purposes. All leaf nodes have zero children, and all internal nodes have exactly two children. It is possible for a leaf node to have the same key as its parent, an internal node, but logically the tree only stores unique keys.

There is no rebalancing mechanism in this implementation. There are indeed some pieces of literature about a self-rebalancing concurrent BST, but they are too complex and beyond the scope of this project.

The algorithm coordinates multiple working threads by marking tree edges, i.e., the pointer from the parent node to its children. It uses the least significant two bits for marking. It is safe on all practical platforms because the dynamically allocated memory regions are at least 4-byte aligned. One bit is called `flag`, meaning both the parent and this child are going to be removed from the tree. The other bit is called `tag`, meaning the parent is going to be removed from the tree. Once an edge has been flagged or tagged, it cannot point to another node.

3.1.2 Seek

All the three operations in the public interface, `search`, `insert` and `remove`, rely on `seek`. It traverses the tree starting from the root node until reaching a leaf node. At each internal node, it either follows the left or the right child, depending on the comparison between the key in the node and the given key.

It returns a record of four nodes: `leaf`, `parent`, `ancestor` and `successor`. `leaf` is the last node on the access path. `parent` is the second-to-last node on the access path. `(ancestor, successor)` is the last untagged edge encountered on the access path before visiting `parent` (`successor` itself can be `parent`). Several sentinel nodes are introduced at initialization to guarantee all these nodes are well defined on empty trees.

According to the definition of `tag`, all nodes on the access path starting from `successor` to `parent` (not inclusive) are in the process of being removed.

3.1.3 Search

`search` simply calls `seek` and returns whether the key in `leaf` is equal to the given key.

3.1.4 Insert

`insert` calls `seek` and returns false if the key in `leaf` is equal to the given key, which means that the key already exists and `insert` fails.

Otherwise, as illustrated in Figure 1, it creates two nodes, the internal node `newInternal` and the leaf node `newLeaf`. The key in `newInternal` is set to the maximum of the inserted key and the key in `leaf`. `newLeaf` is set to be the left or right child of `newInternal` based on key comparison.

It then tries to insert these nodes into the tree. This is done by replacing the old edge `(parent , leaf)` with the new edge `(parent, newInternal)`, with a CAS instruction on the original child field. In the illustration figure, this is replacing `(X, Y)` with `(X, U)`. The `expected` parameter in the CAS instruction requires that `(parent, leaf)` is not marked because, according to definition, a marked edge can no longer point to other nodes.

If the CAS instruction succeeds, the whole `insert` succeeds and returns true. Otherwise, it checks whether the edge still points to `leaf` and is marked. If so, the failure is because of a concurrent `remove` trying to remove `parent` from the tree. It then performs helping by executing the last two steps in `remove` (described later) and repeats the steps starting from `seek`. If not, the failure is because the same key gets inserted by a concurrent `insert`. So it returns false to indicate failure.

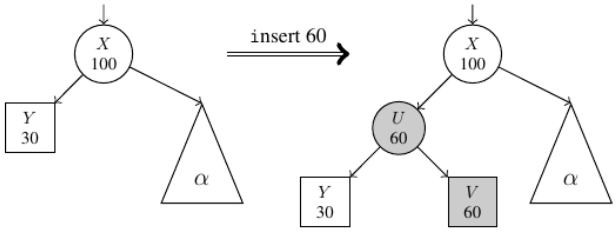


Figure 1: `insert` illustration from [NM14].

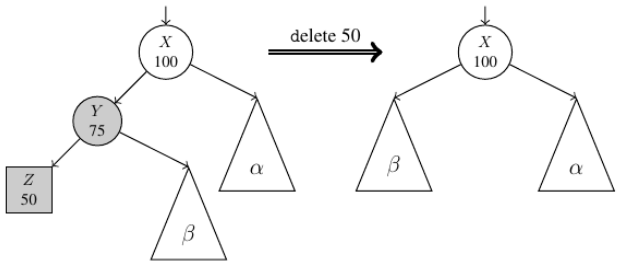


Figure 2: `remove` illustration from [NM14].

3.1.5 Remove

The conflict-free situation is illustrated in Figure 2. It removes `parent` and `leaf` from the tree, and makes the appropriate child field of the grandparent node `X` point to the sibling subtree β. However, because of the conflict between concurrent `remove`s, it is not always possible to operate on the last three nodes in the access path, and this is where `ancestor` and `successor` returned by `seek` works

In reality, `remove` has two modes: injection and cleanup. It starts with the injection mode, and calls `seek` and returns false if the key in `leaf` is not equal to the given key. It then tries to set the `flag` bit of the edge `(parent, leaf)` with CAS to indicate that both of them will be removed. If CAS succeeds, it records the `leaf` to be removed and enters the cleanup mode. Since then the whole `remove` is guaranteed to succeed (returns true) eventually. Otherwise, it uses the same helping logic as in `insert` and repeats injection.

In the cleanup mode, it tries to physically remove `parent` and `leaf`. The first trial uses the result in the injection mode, and the subsequent trials calls `seek` every time. If the following `seek`s returns a different `leaf` from the recorded one, it means that the key is removed by a concurrent `insert`

or `remove` via helping, so `remove` directly returns true. Please note that although the node is not physically removed by it, the `flag` bit is set by it, so the work is still credited to it.

Otherwise, here come the last two steps. It sets the `tag` bit of the sibling edge of `parent` and tries to replace the old edge `(ancestor, successor)` with the new edge `(ancestor, sibling)` with CAS. The value of the `flag` bit in `(ancestor, sibling)` is copied to the new edge. This is a generalization of the conflict-free case, where `ancestor` is the grandparent node and `successor` is `paarent`. As described before, all nodes from `successor` to `parent` (not inclusive) are in the process of being removed, so the whole subtree can be safely removed. If the CAS succeeds, the whole `remove` succeeds and returns true. Otherwise, it repeats cleanup.

3.2 Skip List

3.2.1 Overview

My implementation is based on [Fra04].

The least significant one bit in each `next` pointer is used for marking. The `mark` bit is set before physically removing the node from the list to prevent concurrent operations from inserting directly after this node and losing the inserted node.

Additionally, a node has an extra `removed` field. It is adapted from the paper, which supports key-to-value mapping and uses the `NULL` value to indicate a node is logically removed before setting any `mark` bit. It is not really necessary in the key-only dictionary (just set the `mark` bit in the bottom-level `next` pointer), but I keep it to imitate the code structure in the paper and achieve better extensibility.

The node structure makes use of the technique of flexible array member and embed `next` pointers at the tail of the allocated memory region in order to reduce memory allocation. This implementation trick is very important for the overall performance. Since most key and value types are quite small (typically several bytes), the bookkeeping overhead of dynamic memory allocation can take a considerable proportion of the node structure. Therefore, reducing two memory allocations to one can significantly improve performance and reduce memory overhead.

The probability parameter for the skip list is set to $\frac{1}{4}$, i.e., a new node has the probability of $1 - \frac{1}{4}$ to have a depth of no more than 1, and $1 - (\frac{1}{4})^2$ to have a depth of no more than 2, and so on. The depth of the node can be obtained by calling the random number generator once: by counting the number of the trailing zeros of the random number in binary form.

3.2.2 Seek

Many aspects of the skip list are similar to BST. For example, `search`, `insert` and `remove` also relies on the common `seek`.

`seek` searches for a predecessor and a successor node at each level in the list. There is a pre-configured maximum number of levels. Sentinel nodes are connected at these levels, so the result is always well defined. They are adjacent nodes (in terms of this level) with keys respectively `<` and `>=` the given key.

In the search loop, it skips over marked nodes because they are no longer present logically. If there is a sequence of marked nodes between a level's `predecessor` and `successor`, they are physically removed by updating the `next` pointer in `predecessor` at this level to jump over them. If a node is removed in the bottom level, it is considered as completely removed from Skip List. If the starting node in a level is marked, the current result is stale and the whole `seek` retries from the beginning.

3.2.3 Search

`search` simply calls `seek` and returns whether the key in the bottom level `successor` is equal to the given key.

3.2.4 Insert

`insert` calls `seek`. If the key in the bottom level `successor` is equal to the given key, it returns false if the `removed` field is false, or does help by also executing the last two steps in `remove` and repeats the steps starting from `seek`.

Otherwise, it is ensured that the key originally doesn't exist. It creates a new node with a random number of levels with `next` pointers initialized to `successor`s in each level. The node is then inserted after the bottom-level `predecessor` with CAS. If it succeeds, `insert` is guaranteed to succeed eventually. Otherwise, the whole `insert` restarts. The node is then inserted into higher levels with CAS. If the CAS in one level fails, it only retries at this level.

3.2.5 Remove

`remove` calls `seek` and returns false if the key in the bottom level `successor` is not equal to the given key. Otherwise, it sets `removed` to true with CAS. If the previous value is false, the removal is successful.

Again, here come the last two steps. It marks all levels of `next` pointers in the bottom level `successor`. Finally, it performs another `seek` to the same key to ensure that the marked node is physically removed.

3.3 Garbage Collection

Garbage Collection (GC) can be non-trivial for lock-free concurrent data structures in a language without builtin automatic GC, because when a thread decides that a node can be freed, it may still be reachable in another thread, and accessing it can lead to memory corruption. Another difficulty is the ABA problem, where a memory region is freed and reclaimed by the underlying memory allocator, causing CAS instructions to make a wrong judgment that pointer fields have not changed.

My solution to this problem is to have a global blocking garbage collector. It uses an atomic integer to represent the current state. The least significant bit `pending` means whether there is a currently pending GC, and higher bits count the number of current active operations.

When a memory region can be freed, the active operation pushes the pointer into a private list to the thread. At the end of every operation, if the number of freed pointers reaches a preconfigured threshold, it sets the `pending` bit with CAS. Only one thread can set the bit successfully. From now until the end of GC, there will be no new active operations. At the begining of every operation, if the `pending` bit is set, the thread waits until it is cleared. The thread that has successfully set the `pending` bit then waits for the number of current working operations to drop to zero and really performs GC by freeing all pointers in all threads. This can achieve higher throughput because all threads need to stop and wait for GC anyway, so their garbage is collected at one time.

There are some notable details in my implementation. Instead of using `thread_local` variables, I use an array with a length equal to the number of the threads, and each thread access a distinct element. This is because having one thread iterating over `thread_local` variables in all threads is very difficult to implement. Conceptually this is not possible because it breaks the definition of `thread_local`. Practically this is indeed possible (see folly/ThreadLocal.h), but still too complex and beyond the scope of the project. This requires that all operations to take an extra parameter of the thread id. Plus, my implementation assumes that the key type only has a trivial destructor, so that the GC module can simply free the memory without invoking the destructor.

4 Results

All experiments were performed on PSC Bridges-2 Regular Memory machines. They have the following technical specifications (cited from homework instructions).

- CPU: 2x AMD EPYC 7742 (2.25-3.40 GHz, 2x64 cores per node)
- RAM: 256 GB
- Cache: 256 MB L3 cache, 8 memory channels

The software version is `gcc/10.2.0`. I used OpenMP as the multithreading primitive and set the environment variable `OMP_PLACES=cores`, which can pin a thread to a core and avoid much scheduling overhead.

4.1 Correctness Test

As illustrated in the lecture, concurrent inserts on a linked list can corrupt the linked list structure, causing nodes to be lost. Similar errors also exist in both BST and skip lists, and since pointer structures are more complex, they are only worse if not carefully protected.

I designed test suits to verify the correctness of my implementation. The following operation patterns are included. There is no need to test searches in concurrent operations specially, because it can essentially be seen as a sub-routine of inserts and removes.

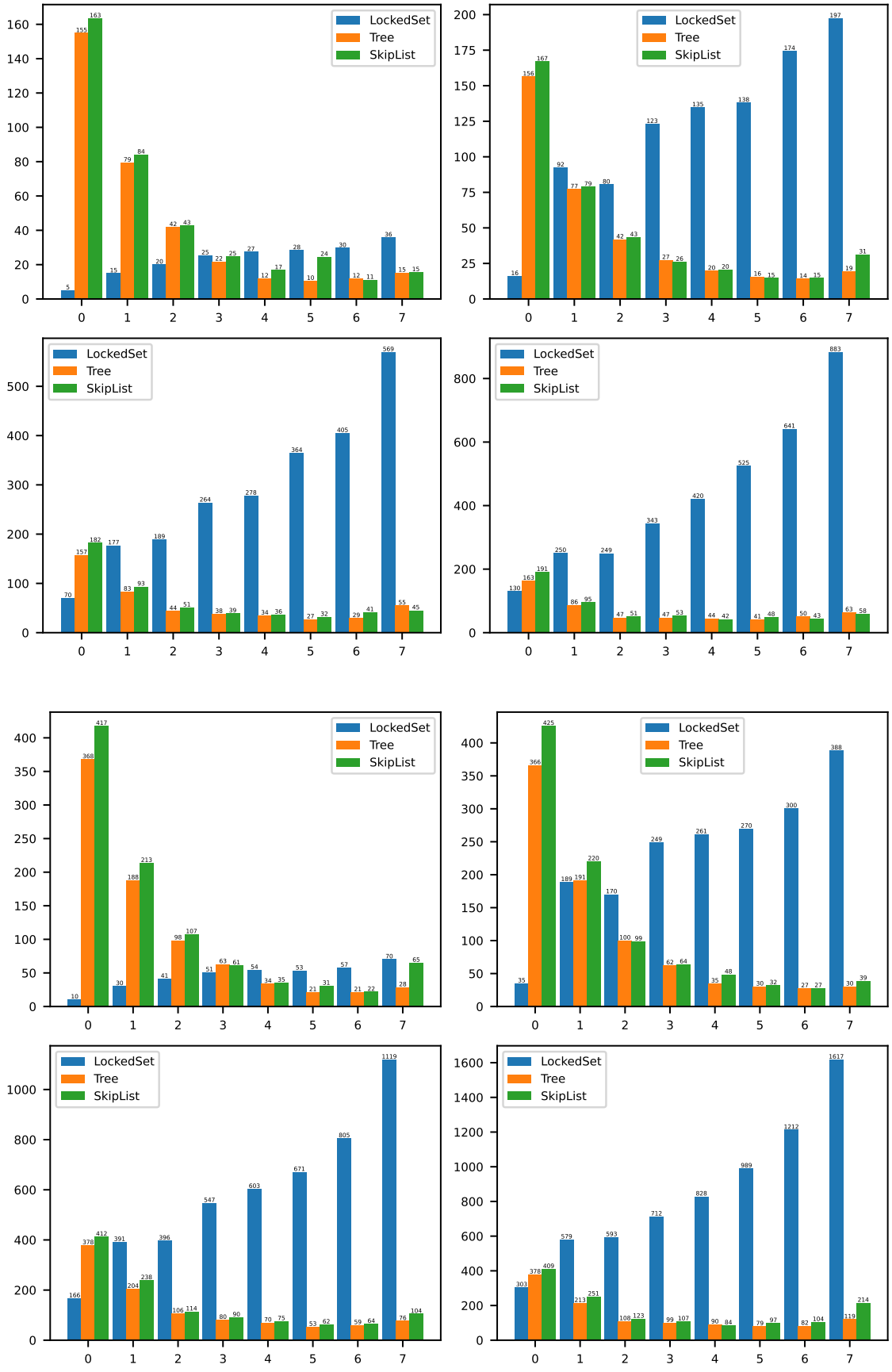
- Concurrent inserts of random and disjoint keys.
- Concurrent removes of random and disjoint keys.
- A mixture of concurrent inserts and removes of random and disjoint keys.
- Concurrent inserts and removes of the same key.

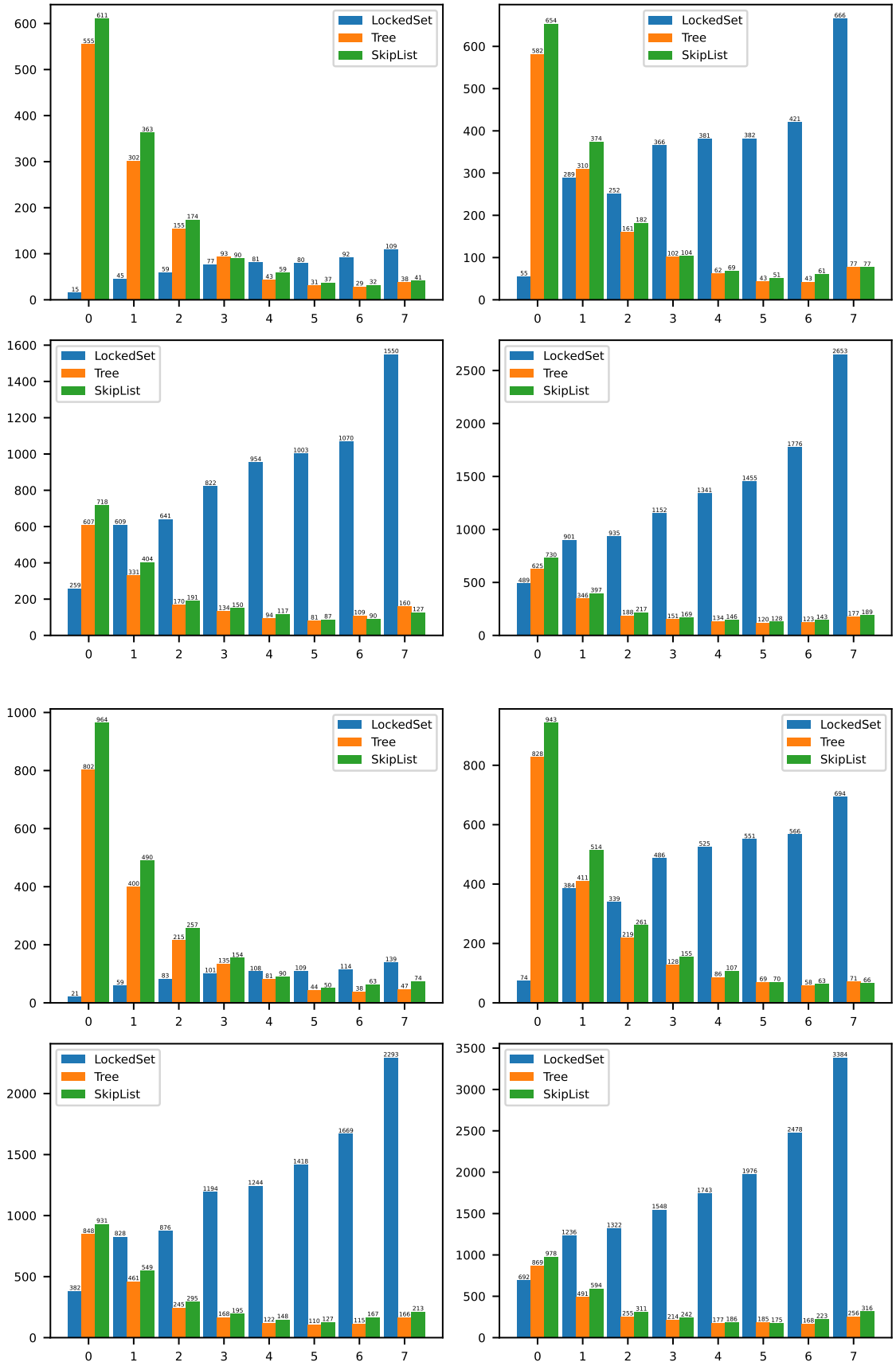
In each case, the results of the BST and skip list must be the same as the baseline. I also used Valgrind to check memory corruption and memory leak, and I was able to pass all the tests. Please note that I cannot use Valgrind to check race conditions because it does not understand low-level synchronization primitives (atomic operations), but only understands Posix primitives.

4.2 Performance Test

The following four figures demonstrates the runtime comparison of the baseline concurrent dictionary (`LockedSet`), the lock-free binary search tree (`Tree`), and the lock-free skip list (`SkipList`) under 250000, 500000, 750000, 1000000 random dictionary operations. The initial dictionary in all the cases contains 1000000 keys. Each test was repeated 10 times to take the average runtime.

In the four sub-figures of each of the four figures, the operation ratios of modifying the dictionary (`insert` plus `remove`) are 0.01, 0.1, 0.5, 0.9 respectively. These settings cover situations from read-dominant to write-dominant. The y-axis is the runtime in milliseconds. The x-axis is the logarithm of the number of threads ($\log_2(T)$), where 7 stands for 128 threads.


LockedSet
Tree
SkipList



The four figures are basically identical, while the sub-figures are more informative. It can be

seen from them that `LockedSet` has certain performance advantages when there are very few write operations. This is because the contention of locks is low, and the constant factor of various operations of the Red-black tree provided by the Standard Template Library is smaller than that of the ordinary binary tree and the skip list.

However, as soon as the ratio of writes increases slightly, `LockedSet` immediately exhibits very severe conflicts, and its performance drops greatly. On the other hand, `Tree` and `SkipList` can benefit from multithreading well, but more write operations will also limit their acceleration. In the read-dominant situation, the speedup of 128 threads ranges from 10 to 20. In the write-dominant situation, the speedup of 128 threads ranges from 2 to 4. Overall, `Tree` is slightly faster than `SkipList`. Considering the average of the normalized speed gaps in all data points (this may not be a rigorous statistical way, but sufficient for an intuitive comparison), `Tree` is faster than `SkipList` by 15.88%.

In order to find the limit in speedup, I performed similar experiments (1000000 random operations, 0.9 modification ratio) on GHC machines and recorded the number of cache references and cache misses with Perf. The results of `Tree` and `SkipList` are as follows.

Threads	1	2	4	8
Time (ms)	719.557	414.247	258.966	191.725
Cache References	612306393	578932164	559343367	554757191
Cache Misses	1865276599	1920327071	2068170487	2407194213
Miss Rate	0.328	0.301	0.27	0.23

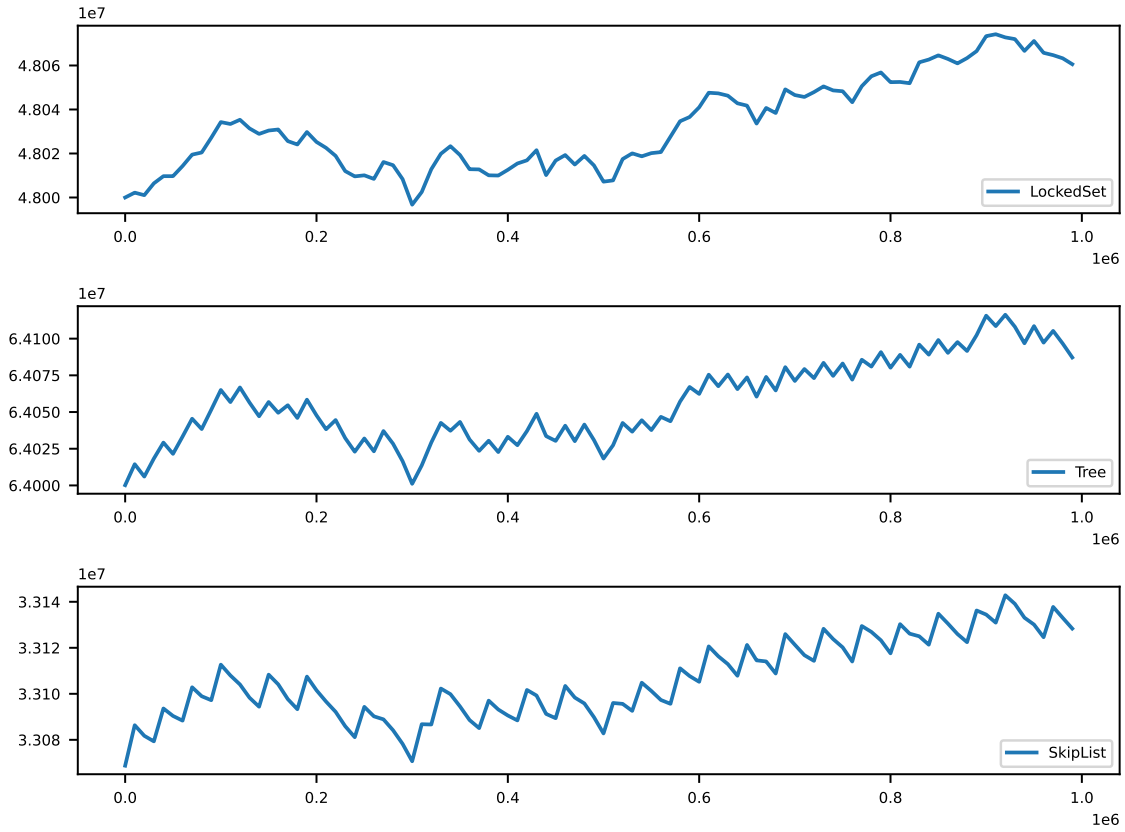
Threads	1	2	4	8
Time (ms)	870.833	495.473	285.871	194.84
Cache References	633432493	650952906	648709198	651209800
Cache Misses	2659824918	2880215604	3145323042	3402454339
Miss Rate	0.238	0.226	0.206	0.191

As can be seen from both the tables, the number of cache misses decreases as the number of threads increases, which may be because each core processes fewer data and the hit rate of the core-private cache increases. However, the number of cache references increases significantly. This may be due to increased retries from failed CAS operations. It makes the overall workload not conserved under different threads, so the ideal speedup cannot be achieved in the case of high thread counts.

4.2.1 Memory Usage

I collected program's memory usage by intercepting `malloc` and `free`, and overloading `operator new` and `operator delete`. Please note that my data collection depends on the implementation details of `malloc` (it needs to read the chunk header), and takes `malloc` overhead into consideration. I think this reflects the actual memory consumption and is more reasonable than just counting the memory requested by the application.

The following figure shows the memory usage of performing 1000000 operations consisting of random inserts and removes, each taking 50%, on data structures with 1000000 initial nodes. The y-axis is the memory usage in bytes. The x-axis is the number of performed operations. Operations are performed sequentially because parallel execution is inconvenient for collecting statistics, and there is no essential difference in memory usage between the two cases.



The most straightforward conclusion is that `Tree > LockedSet > SkipList` in terms of memory usage. In `Tree`, each node takes 32 bytes considering `malloc` overhead. It can be proved that if nodes in a binary tree have either zero children or two children, then the number of leaf nodes is the number of internal nodes plus one. Since the leaf nodes stores all keys, the total number of nodes is roughly twice as many as the number of keys. In `LockedSet`, each node Red-black tree takes 48 bytes. In `SkipList`, about $\frac{15}{16}$ nodes take 32 bytes each.

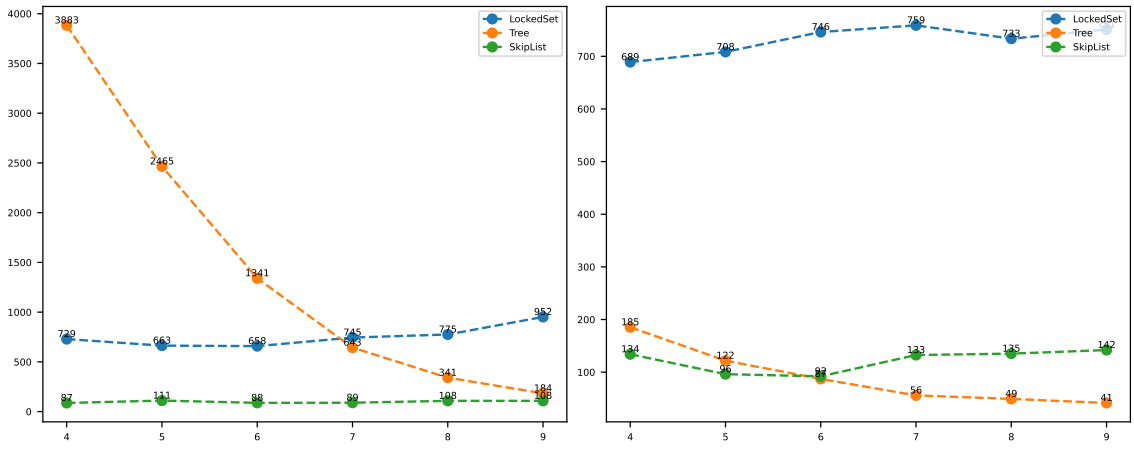
Another observation is that there is some jitter in the memory usage of `Tree` and `SkipList`. This is because the GC module needs a certain number of nodes to be released before actually freeing them. Decreasing the GC threshold can reduce jitter, but it will also increase the GC frequency, thus reducing performance. I have not carefully studied the quantitative relationship between them because it is usually acceptable to waste some memory for better performance (considering the order of magnitude of the y-axis, the jitter is actually very small).

4.3 Imbalance Test

As already mentioned in the background section, for BSTs that do not support rebalancing, the $O(\log(n))$ time complexity can be broken by malicious user input. One solution is to shuffle the data items before inserting them into the tree. However, in real-world applications, data often needs to be processed streamly, so reordering may not be possible. Therefore, testing the resistance of these data structures to malicious input also has practical significance.

This experiment inserts a sequence of partially sorted keys to the two data structures. Formally speaking, the degree of sortedness can be measured by the longest distance between reverse pairs `(a[i], a[j])` such that `i < j` and `a[i] > a[j]`. To make this distance of the input data reach D in the experiment setting, the input sequence is first sorted, and then each of its consecutive chunks with the length of D is randomly shuffled.

The following two figures show the runtime comparison of performing 1000000 insert operations on empty dictionaries. The left figure uses 16 threads, and the right figure uses 64 threads. The y-axis is the runtime in milliseconds. The x-axis is the logarithm of the longest distance between reverse pairs ($\log_2(D)$).



It can be seen that both `LockedSet` and `SkipList` are resistant to partially sorted data, and different D has no significant effect on insertion performance. But `Tree` is significantly affected by the degree of sortedness. Within a certain range of the left figure, the runtime is roughly inversely proportional to D, i.e., when D is reduced by half, the runtime is doubled. This effect reduces significantly in the right figure, where more threads are used. This may be because multi-threaded parallel inserts break the ordering of input data.

5 Conclusion

This project aims to implement the lock-free binary search tree and lock-free skip list and find interesting observations and insights in the trade-off in these two data structures through experiments. I believe I have successfully accomplished this goal. Experiments show that if parallelism is low and the main workload is reading, A simple STL data structure protected by a readers-writer lock gives the best performance. In scenarios where parallelism is really required, the lock-free binary search tree has slightly better performance while introducing many limitations, such as higher memory overhead and unstable performance in the face of different input patterns. It is only wise to use it in a system where the programmer has control over input data, and performance is a much higher priority than memory consumption. On the other hand, the lock-free skip list trades flexibilities for slightly lower performance, and is more useful in most systems. But it also has its own problem that the performance is somewhat dependent on the random number generator, so it may not be suitable for systems with strict requirements on worst-case performance. As for the implementation complexity, as a qualitative comparison, the amount of code of the two lock-free data structures is roughly the same, and the skip list is slightly less.

6 Work Distribution

I completed this project alone.

References

[Fra04] Keir Fraser. Practical lock-freedom. Technical report, University of Cambridge, Computer Laboratory, 2004.

[NM14] Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, page 317–328, New York, NY, USA, 2014. Association for Computing Machinery.