Concurrent lock-free binary search tree implementations with range query support

Elias Papavasileiou

Thesis submitted in partial fulfillment of the requirements for the

Masters' of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisor: Associate Prof. *Panagiota Fatourou*

This work has been performed at the University of Crete, School of Sciences and Engineering, Computer Science Department.

The work has been supported by the Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS).

University of Crete
Computer Science Department

Concurrent lock-free binary search tree implementations with range query support

Thesis submitted by
Elias Papavasileiou
in partial fulfillment of the requirements for the
Masters' of Science degree in Computer Science

THESIS APPROVAL

Author: _____
Elias Papavasileiou

Committee approvals: _____
Panagiota Fatourou
Associate Professor, Thesis Supervisor

_____
Evangelos Markatos
Professor, Committee Member

_____
Kostas Magoutis
Associate Professor, Committee Member

Departmental approval: _____
Antonis Argyros
Professor, Director of Graduate Studies

Heraklion, July 2019

Concurrent lock-free binary search tree implementations with range query support

Abstract

In this thesis, we study concurrent binary search tree implementations that support range queries. Specifically, we present BPNB-BST[1], the first algorithm that supports wait-free range-queries in addition to lock-free Insert, Delete and Find, and has comparable performance to other state-of-the-art algorithms. Moreover, previous implementations provide the weaker progress guarantees of lock-freedom or obstruction freedom for range queries, whereas BPNB-BST guarantees wait-freedom. The distinction between lock-freedom and wait-freedom is important for time consuming operations such as range queries, because without strong progress guarantees such operations may starve.

BPNB-BST is linearizable, uses single-word compare-and-swap operations, and tolerates any number of crash failures. Additionally, in BPNB-BST: (1) update operations work in an independent way interfering with one another only if they work on the same neighborhood of the tree, (2) the helping mechanism employed by the algorithm to guarantee its strong progress guarantees is lightweight, and (3) the algorithm works in a dynamic environment where threads may dynamically join or leave the system.

We have performed a detailed experimental analysis which shows that BPNB-BST scales best with range query size, compared to other state-of-the-art implementations. Our experimental analysis reveals the performance properties of BPNB-BST, as well as interesting trade-offs between the different algorithms. Our experiments have heavily driven the optimizations we applied to our algorithm to make it exhibit such a good performance.

[1] P. Fatourou, E. Papavasileiou and E. Ruppert. Persistent non-blocking binary search trees supporting wait-free range queries. *In the 31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '19). ACM, New York, NY, USA*, pages 275-286, https://doi.org/10.1145/3323165.3323197, 2019.

Ταυτόχρονα Προσπελάσιμες Υλοποιήσεις Δυαδικών Δένδρων Αναζήτησης που υποστηρίζουν Επερωτήσεις Εύρους Τιμών

Περίληψη

Σε αυτή την εργασία, μελετάμε ταυτόχρονα προσπελάσιμες υλοποιήσεις δυαδικών δένδρων αναζήτησης που υποστηρίζουν επερωτήσεις εύρους τιμών. Συγκεκριμένα, παρουσιάζουμε τον BPNB-BST[1], τον πρώτο αλγόριθμο που παρέχει επερωτήσεις εύρους τιμών εξασφαλίζοντας την ισχυρή ιδιότητα προόδου Ελευθερία Αναμονής (wait-freedom) και έχει συγκρίσιμη απόδοση με εκείνη άλλων αλγορίθμων αιχμής, οι οποίοι όμως παρέχουν ασθενέστερες εγγυήσεις προόδου. Συγκεκριμένα, προηγούμενες υλοποιήσεις εγγυώνται μόνο την ιδιότητα Ελευθερία Κλειδωμάτων (ή και ακόμη πιο ασθενείς ιδιότητες προόδου) κατά την απάντηση επερωτήσεων εύρους τιμών. Αντίθετα, ο BPNB-BST εγγυάται την ισχυρότερη ιδιότητα προόδου Ελευθερία Αναμονής. Η διάκριση μεταξύ των ιδιοτήτων αυτών είναι σημαντική σε περιβάλλοντα που υποστηρίζουν χρονοβόρες λειτουργίες, όπως οι επερωτήσεις εύρους τιμών, καθώς χωρίς ισχυρές εγγυήσεις προόδου, ο τερματισμός τέτοιων λειτουργιών μπορεί να καθυστερεί επ΄ άπειρον.

Ο BPNB-BST είναι σειριοποιήσιμος, χρησιμοποιεί εντολές compare-and-swap που παρέχονται από το υλικό και είναι ανθεκτικός σε οποιοδήποτε αριθμό από αποτυχίες. Επιπλέον, στον BPNB-BST: (1) οι λειτουργίες ενημέρωσης εκτελούνται ανεξάρτητα, αλληλεπιδρώντας μεταξύ τους μόνο αν προσβούν την ίδια γειτονιά του δένδρου, (2) ο μηχανισμός βοήθειας που χρησιμοποιείται από τον αλγόριθμο για να διασφαλίσει τις ισχυρές εγγυήσεις προόδου που εγγυάται είναι ελαφρύς και (3) ο αλγόριθμος λειτουργεί σε ένα δυναμικό περιβάλλον όπου τα νήματα μπορούν να εισέρχονται ή να εγκαταλείπουν το σύστημα ανά πάσα χρονική στιγμή.

Εχουμε πραγματοποιήσει μια λεπτομερή πειραματική ανάλυση που δείχνει ότι ο BPNB-BST έχει καλύτερη δυνατότητα κλιμάκωσης με το μέγεθος της επερώτησης εύρους τιμών, συγκριτικά με τους τρέχοντες αλγόριθμους αιχμής. Η πειραματική μας ανάλυση φέρνει στην επιφάνεια τις ιδιότητες (και τις σχεδιαστικές αποφάσεις) που παίζουν καθοριστικό ρόλο στην απόδοση του BPNB-BST, καθώς και ενδιαφέροντα trade-offs μεταξύ των διάφορων αλγορίθμων. Τα πειράματά μας οδήγησαν σε μεγάλο βαθμό τις βελτιστοποιήσεις που εφαρμόσαμε στον αλγόριθμο για να έχει τόσο καλή απόδοση.

[1] P. Fatourou, E. Papavasileiou and E. Ruppert. Persistent non-blocking binary search trees supporting wait-free range queries. *In the 31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '19). ACM, New York, NY, USA*, pages 275-286, https://doi.org/10.1145/3323165.3323197, 2019.

στους γονείς μου

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Multicore hardware architectures are now the norm among general and special purpose computing machines. By exploiting the processing power of many cores, concurrent data structures offer much higher performance than the corresponding sequential ones. In particular, *non-blocking* concurrent data structures are desirable primarily for their strong progress guarantees and fault-tolerant behavior. In a *lock-free* implementation, the system makes progress as a whole despite any thread delays or crashes. In a *wait-free* implementation, every operation makes progress as long as it takes steps. The *linearizability* of an implementation is a desired property as well, to ensure the consistency of supported operations. In a linearizable implementation, every concurrent execution it produces can be mapped to a sequential one, in which the responses of the operations are the same as in the concurrent execution; moreover, the sequential execution respects the real-time ordering of the operations.

The dawn of big data era has motivated the development of concurrent data structures that support extended functionality. Modern big data processing engines utilize huge in-memory key-value stores. Applications built on top of these engines demand fast updates and large scale analytics to be performed efficiently and concurrently. Usually, a global or a partial view of the available data is required, in order to perform calculations and advanced operations based on that view. This is usually achieved with iterators, which return a snapshot of the available data contained in a key-value store. Range queries are a generalization of iterators that can return a partial (or global) view of the data structure. A RangeQuery(x,y) returns all keys of the data structure in the range [x,y]. Besides the standard Insert, Delete and Lookup operations, concurrent data structures with support for efficient RangeQuery operations provide simultaneous collection and processing of data. This way they have the potential to support a broad range of applications, ranging from data streams, to databases and big data analytics.

However, the design and implementation of efficient non-blocking concurrent

data structures is one of the most challenging topics of the area. One source of this arising difficulty is the concurrent environment the implementations are made to run in. As many threads may simultaneously try to access and/or update the same shared variables, correctness of the implementation has to be ensured. Moreover, strong progress properties such as lock-freedom or even more, wait-freedom, usually come with a non-negligible performance cost.

The support of efficient advanced operations such as range queries, executed in a concurrent manner with updates, constitutes an additional challenge. Large scale range queries usually have to operate on a big portion of the data structure, interfering with update operations. This may cause a significant slowdown to the updates. Therefore, one must come up with a clever synchronization mechanism between threads to provide good performance for all supported operations.

1.2 Our Contribution

In this thesis, we study concurrent binary search tree implementations that support range queries. Specifically, we present BPNB-BST [7], the first algorithm that supports wait-free range-queries and Finds in addition to lock-free Insert and Delete operations, and has comparable performance to other state-of-the-art algorithms. In BPNB-BST, a lighter, lock-free version of Finds is provided as well. Moreover, previous implementations provide the weaker progress guarantees of lock-freedom or obstruction freedom for range queries, whereas BPNB-BST guarantees wait-freedom. The distinction between lock-freedom and wait-freedom is important for time consuming operations such as range queries, because without strong progress guarantees such operations may starve (i.e., they may never terminate, as their progress might be hindered by other operations).

BPNB-BST builds upon NB-BST, and inherits some interesting properties from it: It is linearizable, uses single-word compare-and-swap operations, and tolerates any number of crash failures. Additionally, in BPNB-BST: (1) update operations work in an independent way interfering with one another only if they work on the same neighborhood of the tree, (2) the helping mechanism employed by the algorithm to guarantee its strong progress guarantees is lightweight, and (3) the algorithm works in a dynamic environment where threads may dynamically join or leave the system. Persistence in the case of BPNB-BST means that deleted nodes, instead of being physically removed from the tree, are marked and retained in the data structure. This way, a range query might visit deleted nodes if this is needed to complete its work. In addition, there is a global shared integer variable, called *Counter*, which is used to assign version numbers to nodes. In this way, efficient and concurrent support of range queries is achieved.

We have experimentally compared BPNB-BST to other state-of-the-art implementations with weaker progress guarantees, and showed that BPNB-BST scales best with range query size. Our experiments have heavily driven the optimizations we applied to our algorithm to make it exhibit such a good performance.

In particular, the application of batching in the leaves of the tree results in range query operations that are 10x faster than the non-batched version (which is called PNB-BST). With this experimental analysis, we also reveal the trade-offs that exist among different design decisions, exploring the scenarios where each algorithm performs better or worse, and providing explanations for the evaluation results.

1.3 Related Work

The Non-Blocking Binary Search Tree (NB-BST) [4] is the concurrent data structure on top of which BPNB-BST was built, to provide range queries. NB-BST was the first complete, non-blocking, linearizable binary search tree implementation that uses only single-word compare-and-swap (CAS) operations. NB-BST is a leaf-oriented binary search tree, which means that keys are stored only in the leaf nodes. This makes Deletes much easier to implement in a concurrent environment, because only a small neighborhood of the leaf to be deleted has to be modified. Therefore, threads that operate in different parts of the tree do not interfere with each other. Threads that operate on the same part of the tree synchronize by using flags, helping other threads before they make progress on their own. This way the non-blocking property of NB-BST is achieved.

The Lock Free Contention Adapting Search Tree (LFCA tree) [10] is a non-blocking, linearizable key-value store with lock-free range query support. Like NB-BST, it is leaf-oriented, but each leaf (called a *base node*), instead of storing a single key, stores a pointer to an immutable treap [11] and each leaf of a treap contains a sorted batch of keys. Moreover, LFCA tree splits and joins base nodes taking into account the encountered contention on them and the performance of range queries, a procedure that is called adaptation. Our experimental analysis shows that LFCA tree is outperformed by PNB-BST and BPNB-BST for medium and large range query sizes. This is because of the heavy helping mechanism that LFCA tree employs: updates are forced to help range queries to complete, and thus, the bigger the range query size, the more expensive helping becomes. This results to a significant performance penalty for LFCA tree. As for the performance of Insert, Delete and Find operations, we show that PNB-BST and BPNB-BST have similar performance with LFCA tree despite the extra cost for supporting wait-free range queries.

KIWI [14] is a non-blocking, linearizable key-value store designed with the purpose of supporting range queries concurrently with updates. Keys are stored in partially sorted arrays, and every array is stored inside a *chunk* data structure. These chunks are connected into a sorted linked list, and there is an index data structure that facilitates quick access to a particular chunk. The index is updated in a lazy manner as this is only necessary for efficiency but not for correctness. KIWI exhibits some common algorithmic decisions with PNB-BST, such as the global version counter and persistence, which are implemented in a different way in each algorithm. Nevertheless, there are a number of significant differences as

well. In KIWI, synchronization of updates and range queries is achieved by using an array of version numbers which has size n, where n is the number of processes in the system. However, this makes KIWI work for a known, constant number of processes, whereas PNB-BST and BPNB-BST work even when this number is unknown and changing over time. KIWI also employs a maintenance procedure called *rebalancing*. During rebalancing, KIWI might join underutilized chunks together, split overfull chunks, and perform cleanup operations on each chunk it processes. Updates are forced to help the rebalance procedure to complete, before they can make progress, in order for lock-freedom to be ensured. Our experiments show that this turns out to be expensive, as KIWI exhibits a performance degradation in most cases. Furthermore, because keys are only partially sorted in each array, there is often an additional delay for an operation to locate the key (or keys, in case of range query) it seeks. We should also note that, while KIWI is a custom made data structure for supporting range queries, PNB-BST showcases a technique to support wait-free range queries on top of a BST, which is a standard, widely-used data structure. We believe that this technique is likely to be more generally applicable to other tree-like data structures as well. Finally, there are claims that Find operations and range queries of KIWI are wait-free. However, this depends on having wait-free Find operations on the index data structure used by KIWI, and most papers that provide non-blocking implementations of index data structures do not provide wait-free Finds.

Brown and Avni [15] presented a non-blocking, linearizable k-ary search tree that supports obstruction-free range queries, using single-word CAS operations. In the k-ary tree, each leaf node stores up to k-1 keys, and each internal node has k-1 keys and k children. All nodes are immutable, so updates create new leaves to insert or delete keys. A range query consists of two phases: the collect phase and the validation phase. During the collect phase, a traversal of the tree is performed that saves pointers to all leaves containing keys in the requested range. The validation phase performs a second traversal and checks whether the collected pointers remain the same. A tag bit is used inside each leaf, so the validation phase can be performed optimistically to avoid traversing the tree a second time. If the validation succeeds, this means that the collected leaves were all in the tree at some point (i.e., they constitute a snapshot), and the range query is performed in the collected leaves. Otherwise, the range query restarts from the first phase. For this reason, only obstruction freedom is guaranteed for range queries, because continuous updates to the leaves in the requested range might cause a range query to restart forever. To ensure lock-freedom for updates, a helping scheme is used akin to that of NB-BST.

Apart from the previously described data structures that have native support for range queries, there are several other works that can implement range queries by using snapshots [16] [17] [18] [21], or software transactional memory [19], [20]. However, experimental work in [10] and [14] shows that the performance of those algorithms is inferior compared to the performance of KIWI and LFCA tree (the

	Find	Insert	Delete	RangeQuery
NB-BST	lock-free	lock-free	lock-free	-
PNB-BST	lock-free (or **wait-free**)	lock-free	lock-free	**wait-free**
BPNB-BST	lock-free (or **wait-free**)	lock-free	lock-free	**wait-free**
LFCA tree	lock-free	lock-free	lock-free	lock-free
KIWI	wait-free[2]	lock-free	lock-free	wait-free[2]
k-ary tree	lock-free	lock-free	lock-free	obstruction-free

Table 1.1: Progress properties of different algorithms.

same is true for the performance of the k-ary tree; its performance is inferior compared to KIWI and LFCA tree). Moreover, global snapshots are inefficient in the case of small range queries, because a large snapshot is constructed in order to return just a few nodes. As for software transactional memory, besides the baseline overhead introduced by the STM implementation, large range queries are more likely to be inefficient because of the increasing overhead when the number of shared-memory words covered by a transaction is big. Most notably, none of these works support wait-free range queries.

Finally, there are other implementations that provide wait-free range queries on non tree-like data structures [22] [23] [24] [25] or in different settings [26].

Table 1.1 summarizes the progress guarantees for each operation of the different algorithms discussed in section 1.3.

1.4 Roadmap

Chapter 2 serves as a brief introduction to some of the most commonly used concurrency terms, providing to the reader the necessary background to understand the algorithms presented in later chapters. The topics discussed are the following:

- Abstract Data Types, Tree Data Structures, Binary Trees, Binary Search Trees, Leaf-Oriented Binary Search Trees.
- Shared Memory Systems, Shared Variables, Configurations, Executions.
- Blocking, Non-Blocking, Lock-Free, and Wait-Free implementations.
- Operations, Deadlock, Livelock, and the ABA problem.
- Progress and Correctness properties of Concurrent Algorithms, Linearizability.

Chapters 3 to 6 present the Non-Blocking Binary Search Tree (NB-BST), the Persistent Non-Blocking Binary Search Tree (PNB-BST), the Lock Free Contention Adapting Search Tree (LFCA tree) and the Batched Persistent Non-Blocking Binary Search Tree (BPNB-BST).

[2]Given the assumption that there exists a wait-free tree-based set implementation.

Finally, chapter 7 presents our experimental analysis, as well as some lessons learned, concluding the thesis.

Chapter 2

Model

This section provides to the reader the necessary background to understand the algorithms presented in later chapters.

2.1 Tree Data Structures

Abstract Data Types An *abstract data type* (or ADT) is a mathematical model for a certain class of data structures. It can define the possible values the data can take, the possible operations that can be performed on the data and also other features the data structure can exhibit. For example, the *dictionary* (or *set*) ADT declares that the data structure 1) holds unique keys (meaning that duplicate keys are not allowed) from a totally ordered universe, and 2) implements the Find, Insert and Delete operations.

Binary Trees A *Binary Tree* is a connected, directed acyclic graph where each node has 1) at most two children and 2) exactly one parent, except from a unique node, the *root* node, which has no parent. Nodes that have no children are called *leaf* nodes. Nodes that have one or two children are called *internal* nodes. In a *full* binary tree, every internal node has exactly two children.

Binary Search Trees A *Binary Search Tree* (or BST) is a binary tree with the following property: For every internal node nd that holds a key k, the left subtree of nd contains nodes with keys smaller than k and the right subtree of nd contains nodes with keys greater than or equal to k. A BST can implement the dictionary ADT. From now on, Insert and Delete operations will be collectively called *update* operations (or *updates*).

Leaf-Oriented Binary Search Trees A BST which implements the dictionary ADT is called *leaf-oriented* (or *external*) when all the keys that belong to the set represented by the tree are stored in the leaves of the tree. Internal nodes contain

keys that do not necessarily belong to the set and are solely used for routing, i.e. to direct a Find operation towards the correct leaf. The primary motivation for leaf-oriented trees is that the Delete operation is much simpler to implement, because it processes only a small number of nodes near the leaf to be deleted. In contrast, Delete in a classic BST needs to modify two parts of the tree, namely the node that contains the key to be deleted and its in-order successor or predecessor, that may be far away from each other, making the operation more difficult to implement in concurrent environments.

2.2 Shared Memory Systems

System We consider an *asynchronous* shared memory system where n threads t_1, t_2, ... t_n are executed concurrently (in arbitrary speeds). Threads are modeled as state machines and communicate with each other through shared *variables*. Each variable has a *type* which specifies the values that can take, the operations that can be executed on it, the value to be returned by each operation and the new value of the variable resulting from each operation. A *Read/Write* variable RW supports the operations:

- Read: Returns the current value of RW, leaving RW unchanged
- Write(RW,v): Writes v into RW and returns ack

Configurations In a shared memory system with n threads and m variables, a *configuration* C is a vector:

$$C = (s_1, ..., s_n, v_1, ..., v_m)$$

where s_i is the state of thread t_i, $\forall i : 1 \leq i \leq n$, and v_j is the value of the shared variable RW_j, $\forall j : 1 \leq j \leq m$. A configuration describes the concurrent system at some specific point in time. An *event* is a computational step by any thread. Every computational step by a thread t_i consists of the following that happen atomically:

- t_i chooses a shared variable RW_j and an operation op,
- t_i performs op on RW_j,
- t_i's state changes according to its previous state as well as the value returned by op.

An *execution fragment* is a sequence of the form $C_k, e_{k+1}, C_{k+1}, e_{k+2}, ...$, where each C is a configuration and each e is an event. After the application of e_{k+1} to C_k, the only changes that happen are to the state of the thread t_i that executes the computational step of e_{k+1} and the value of the shared variable RW_j that e_{k+1} is applied to. C_0 is an initial configuration where every thread has an initial

state and every shared variable has an initial value. An *execution* is an execution fragment that starts from an initial configuration C_0.

The concurrent implementations presented in this thesis implement tree data structures in a concurrent environment. To understand the behavior of concurrent implementations, their liveness (or *progress*) and safety (or *correctness*) properties are studied. An algorithm satisfies a safety property if the property holds in all finite prefixes of every execution that produces. Intuitively, it states that no unintended action has happened thus far. A liveness property is a condition that must hold at least a certain (and sometimes infinite) number of times during any execution of the algorithm. Intuitively, it states that some intended action eventually happens one or more times. In the rest of this thesis, we focus on concurrent implementations of tree-like data structures and on their executions.

Progress It is very common that different *operations* of an implementation can have different progress guarantees. An operation is *blocking*, if some thread executing an instance of the operation may have to wait for another thread to take steps before it can continue its execution. In blocking (or *lock-based*) implementations, locks are used for synchronization between threads.

In a *non-blocking* algorithm, no thread t_1 ever waits another thread t_2 to take steps before t_1 can proceed with its own execution. So, in a non-blocking algorithm, no thread uses locks. Non-blocking algorithms use *flags* (which are usually boolean, integer or enum variables) and/or atomic synchronization primitives provided by the hardware. An implementation is non-blocking if all the operations it supports are non-blocking.

The Compare and Swap primitive The *compare-and-swap* (or *CAS*) primitive is one of the most commonly used atomic synchronization primitives provided by the hardware. It is used to update the value of a shared variable in a safe manner. To illustrate its usage, consider a shared variable X and two values *expectedValue* and *newValue*. Then, CAS(X, expectedValue, newValue) will set X to newValue only if X is equal to expectedValue. Effectively, this atomic operation is theoretically equivalent to the following code:

```
1  ATOMIC boolean CAS(Shared Variable X, Value expectedValue, Value newValue) {
2    if (X == expectedValue) {
3      X = newValue
4      return TRUE
5    }
6    return FALSE
7  }
```

In case the value of X is updated by the CAS operation (i.e. line #3 is executed), we say that the CAS operation is *successful* and will return TRUE. In case the value of X is not updated by the CAS operation (i.e. line #3 is not executed),

we say that the CAS operation is *unsuccessful* and will return FALSE. Depending on the implementation of the CAS operation, instead of TRUE or FALSE, the old value of X (i.e. the value it had before the update) will be returned. The user then has to check whether the returned value matches the expectedValue, to determine whether the CAS operation was successful.

The ABA problem Consider a shared variable X whose initial value is A. A very common use of the CAS operation performed by a thread t is the following:

```
1   expectedValue = X
    ... (other actions)
2  CAS(X, expectedValue, newValue)
```

Thread t stores the last value `A` it saw in X in a local variable called *expectedValue* (line #1). Then, later on, it uses that value as the expectedValue of a CAS operation on X (line #2). Usually, to ensure correctness of the algorithm it is assumed that if the CAS operation succeeds (thus X has the value `A` at the time the CAS operation is executed), the value of X has not changed between the execution of line #1 and line #2. However, there is the case that after the execution of line #1 and before the execution of line #2 by thread t, other threads might change X to some value `B` and then change it back to its initial value `A`. Thread t will not be able to detect that change, and the execution of its CAS primitive will be successful (line #2). This is called the *ABA problem*, and when it happens, it usually breaks the correctness of the algorithm. In the context of concurrent dynamic data structures, the ABA problem may appear, for instance when an allocated object is deallocated and later reallocated and used again. In such settings, a straightforward solution to the ABA problem is the avoidance of memory address reuse, unless a garbage collection scheme is applied to allow address reuse in a safe way.

An implementation is *lock-free*, if in every infinite execution it produces, infinitely many operations are executed. This means that the system is making progress as a whole. However, specific operations may starve (i.e. they may never complete) because other operations which might run faster always make progress, disallowing these operations to make progress as well. Note that an implementation that does not use locks is not necessarily lock-free: For example, a non-blocking algorithm may produce an execution that contains a *livelock*, where each thread executing an operation takes turns and makes some steps, but no thread eventually completes the operation it is executing.

An operation op of an implementation `I` is wait-free if in each infinite execution that `I` produces, the execution of every instance of op terminates within a finite number of steps. This effectively means that a thread executing this operation makes progress as long as it takes steps. An implementation is wait-free when all the operations it implements are wait-free. Wait-freedom is a stronger property

than lock-freedom since it guarantees progress for each individual thread, whereas lock-freedom guarantees progress only at the system level.

Non-blocking algorithms guarantee progress even in the presence of thread failures, i.e. when one or more threads crash unexpectedly in the middle of their execution. Thus, non-blocking algorithms are more fault-tolerant than blocking ones. In blocking algorithms, the system might be heavily delayed or blocked if the thread that is holding a lock crashes before releasing the lock. Also, care is required to avoid *deadlocks*, where thread A may try to acquire lockB while holding lockA and, at the same time, thread B may try to acquire lockA while holding lockB. Thus, threadA and threadB may wait one another forever.

Correctness The most common correctness condition when designing and analyzing concurrent data structures is *linearizability*. An execution α is *linearizable* if for every completed operation in α (and for some of the uncompleted operations), a linearization point can be assigned within the execution interval of the operation, so that in every execution of the algorithm, the response of each operation is the same as the response of the corresponding operation in the sequential execution defined by the linearization points.

Chapter 3

The NB-BST Algorithm

This chapter provides a description of the Non-Blocking Binary Search Tree (NB-BST), a non-blocking, linearizable binary search tree implementation using single-word compare-and-swap (CAS) operations [4]. The full proof of these properties and the correctness of the algorithm can be found in [5]. The implementation used for the experimental analysis of chapter 7 is available at [6].

3.1 Introduction

A BST implements the dictionary abstract data type and thus it holds a set of unique keys, which may have associated values (key-value pairs). It supports three types of operations. Given a key k:

- Insert(k): Adds k to the tree and returns TRUE unless k already exists in the tree - in that case, it does not cause any change to the tree structure and returns FALSE.
- Delete(k): Removes k from the tree and returns TRUE unless k does not exist in the tree - in that case, it does not cause any change to the tree structure and returns FALSE.
- Find(k): Returns a pointer to the node that contains k provided that k is already in the tree, otherwise it returns NULL.

The NB-BST consists of two types of nodes: Leaf nodes (or *leaves*) which are nodes that have no children, and Internal nodes (or *internals*) which are nodes that have two children. Every internal node holds pointers only to its children and there are no parent pointers. The tree is *full*, meaning that every internal node has exactly two children. The tree is *external* (or *leaf-oriented*), meaning that all the keys are stored in the leaves of the tree (each leaf contains a key or key-value pair) and internal nodes are used just for routing: they direct operations to the correct leaf. This routing is achieved efficiently because of the BST property: All

keys stored in the left subtree of an internal node are smaller than the node's key and all keys stored in the right subtree are greater than or equal to its key.

The Find() operation performs a search by traversing the appropriate tree path starting from the root node downwards, until reaching a leaf.

The Insert() operation works as follows: First, it searches for the leaf that may contain the key to be inserted. If the key is there, it does nothing and returns FALSE. If the key is not there, it replaces that leaf, switching the pointer of the leaf's parent that points to this leaf, to point to a new subtree of three nodes: one internal and two leaves that are children of this internal. One of the subtree's leaves holds the key that is to be inserted, and the other holds the key of the replaced leaf (because it has to remain in the tree). The smaller of those two keys is placed in the left leaf, and the other one is placed in the right leaf as well as in the internal node, so that the BST property is preserved. Note that those two keys cannot be equal because the key that is to be inserted was not found in the tree.

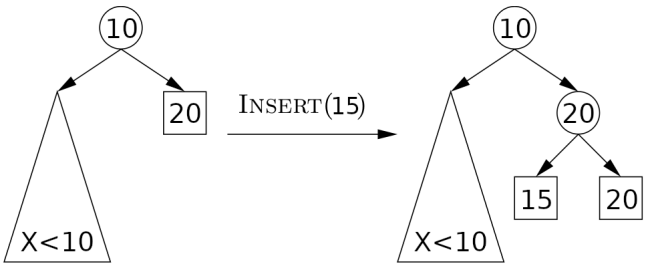


Figure 3.1: Insert() operation.

The Delete() operation works accordingly: First, it searches for the leaf that may contain the key to be deleted. If the key is not there, it does nothing and returns FALSE. If the key is found there, then this leaf has to be removed along with its parent, because there cannot be an internal with only one child in the tree. On the other hand, its sibling has to remain in the tree. These are achieved by switching the child pointer of its grandparent that points to its parent, to point to its sibling.

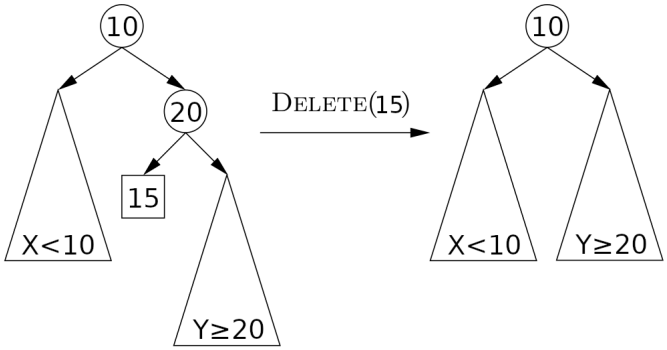


Figure 3.2: Delete() operation.

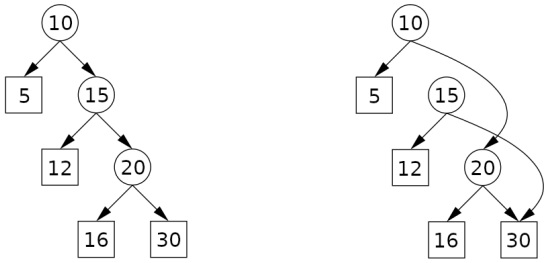
Remark that in BSTs that are not leaf-oriented, a Delete() of an internal node with his children may have to operate atomically in two places. Therefore, leaf-oriented BSTs have a significant advantage: Delete() operations are much simpler, because their updates to the tree take place in a small neighborhood around the leaf to be deleted.

3.2 Algorithmic Overview

In a concurrent setting, whenever an Insert() (or Delete()) operation changes a pointer of the tree in order to insert (or delete) nodes, this change has to be performed atomically for correctness to be ensured. This is achieved by utilizing the compare-and-swap (CAS) primitive. Insert() performs a CAS operation to switch the child pointer of the parent to point to the new subtree's Internal node - this is called an *ichild* CAS. Correspondingly, Delete() performs a CAS operation to switch the child pointer of the grandparent to point to the sibling - this is called a *dchild* CAS.

Because every Insert() and Delete() operates in a small neighborhood around the appropriate leaf, concurrent updates at distant parts of the tree do not conflict. However, the following examples illustrate some of the difficulties that arise regarding correctness when updates working on the same part of the tree are executed concurrently.

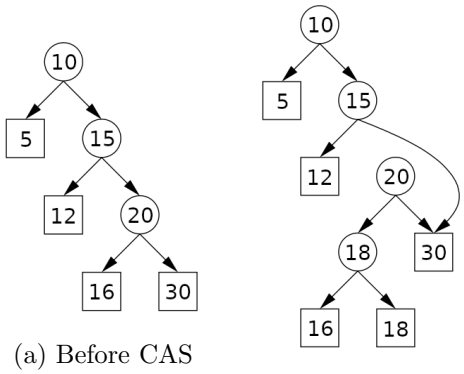
Example 1 | Delete(12) and Delete(16): In this example, two Delete() operations perform their CAS consecutively. However, in the end, the key 16 has not been deleted from the tree since a traversal from the root will visit it.



(a) Before CAS (b) After CAS

Figure 3.3: Conflicting Delete() operations.

Example 2 | Delete(16) and Insert(18): In this example, an Insert() and a Delete() perform their CAS consecutively. However, in the end, the key 18 has not been inserted in the tree since a traversal from the root will not visit it.



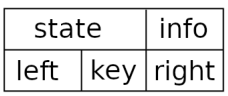
(a) Before CAS

(b) After CAS

Figure 3.4: Conflicting Insert() and Delete() operations.

Correctness could be preserved with the use of fine-grain locking schemes. Nevertheless, to make the implementation non-blocking, the ideas of *flags* and *help* are used.

Flags A flag is an indication that the procedure of updating the tree pointers begins. Every internal node contains an enum variable called *State*, which can take the values CLEAN, IFLAG, DFLAG and MARK (these values are collectively called *Flags*). Initially, all internal nodes are in the CLEAN State. Every internal node also contains a pointer to an *Info* struct. As its name implies, the Info struct contains all the necessary information for a process to complete the operation that set the flag.

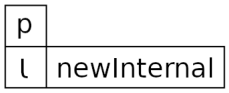


Internal node

The Info pointer and State variable are stored in the same CAS word (the *Update* variable), so they can be changed atomically with one CAS operation.

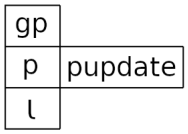
Leaf node

Before an Insert() operation performs an ichild CAS to replace a leaf, it uses CAS to change the State of the leaf's parent to IFLAG and the Info pointer to a new *IInfo* struct, where it has written all the necessary information for any process to complete the work of that Insert() operation. This is called an *iflag* CAS.



IInfo struct

Correspondingly, before a Delete() operation performs a dchild CAS to remove a leaf (and its parent), it uses CAS to change the State of its grandparent to DFLAG and the Info pointer to a new *DInfo* struct, where it has written all the necessary information for any process to complete the work of that Delete() operation. This is called a *dflag* CAS.



DInfo struct

Every internal node that is deleted from the tree is first marked. Once a node is marked, it remains marked forever, and it is ensured that its child pointers cannot change. The Delete() operation uses another CAS to change the State of the parent of the leaf to MARK, indicating that this node will be removed from the tree, and the Info pointer to the same DInfo struct with its grandparent's. This is called a *mark* CAS.

Help Every Insert() or Delete() operation, before executing the iflag or dflag CAS, checks whether the node is in the CLEAN State. If it is not, it begins to perform the work that the flag indicates, following the Info pointer and using the information written in the Info struct. This is called *helping*. Helping is utilized to ensure progress of the system, in case the operation that set the flag dies or is suspended for a long time. Then, it retries to do its own work.

To minimize the performance penalty that is caused by helping, a conservative helping strategy is used, where every operation helps only when its own progress is prevented. Thus, helping is performed by Insert() and Delete() operations, but not by Find(), which ignores flags and marks and does not help.

Because every Insert() or Delete() starts by reading the State variable of a node to decide whether it has to help, flags function as cooperative locks: They are owned by operations, instead of processes. If an Insert() performs successfully the iflag CAS, it is ensured that this operation will be finished either by that process or cooperatively by other processes who help. Correspondingly, if a Delete() performs successfully the mark CAS, it is ensured that this operation will be finished either by that process or cooperatively.

After an ichild or dchild CAS, an *iunflag* or *dunflag* CAS is used, to restore the parent's or grandparent's State to CLEAN. If the mark CAS fails, a dunflag CAS is used (which is called a *backtrack* CAS) to restore the grandparent's State to CLEAN.

Practically, a flagged node indicates that one of its child pointers will change, and a marked node indicates that this node will be (or is already) removed from the tree, and his child pointers will never change. By utilizing this flag and help scheme, both problems of figures 3.3 and 3.4 can be prevented. The operations' CAS cycle is depicted in Figure 3.5.

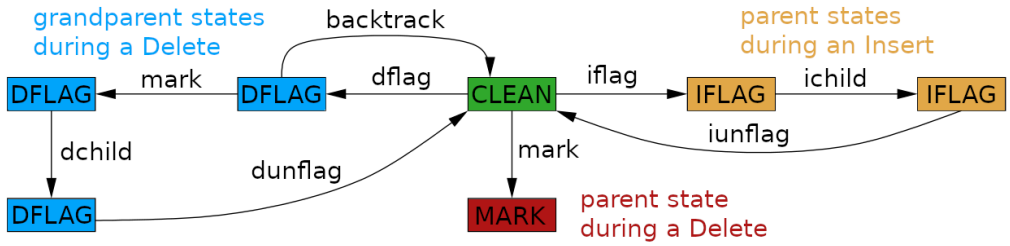


Figure 3.5: The effects of successful CAS operations.

3.3 Data Structures & Initialization

To avoid handling special cases with less than three nodes, the initial state of the tree consists of a root node and two leaf nodes, where all of them hold dummy keys that cannot be inserted or deleted from the tree. It is assumed that $\infty_2 > \infty_1$ and $\infty_1 > X$, where X can be any key that will be inserted in the tree:

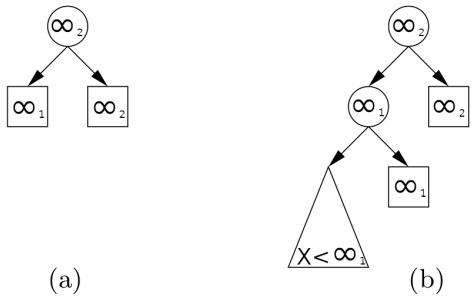


Figure 3.6: Trees showing leaves with dummy keys when the dictionary is (a) empty and (b) non-empty.

Thus, the tree will always contain at least three nodes. The Root pointer is a shared pointer that never changes. The key field of a node is initialized when the

node is created, and is never changed thereafter. The Info structs are immutable as well. The following figure contains the structs that are used by the algorithm:

```
1   type Update {
2       {CLEAN, DFLAG, IFLAG, MARK} state
3       Info *info
4   }
5   type Internal {
6       Key ∪ {∞1, ∞2} key
7       Update update
8       Node *left, *right
9   }
10  type Leaf {
11      Key ∪ {∞1, ∞2} key
12  }
13  type IInfo {
14      Internal *p, *newInternal
15      Leaf *l
16  }
17  type DInfo {
18      Internal *gp, *p
19      Leaf *l
20      Update pupdate
21  }
    ▷ Initialization:
22  shared Internal *Root := pointer to new Internal node
        with key field ∞2, update field ⟨Clean, ⊥⟩, and
        pointers to new Leaf nodes with keys ∞1 and
        ∞2 as left and right fields, respectively.
```

Figure 3.7: Type definitions and initialization.

3.4 Pseudocode

The pseudocode of the algorithm is presented in the next two pages. Figures 3.8 and 3.9 provide the pseudocode for NB-BST as it appears in [4]. A detailed explanation of the functions, as well as an example, follows.

```
23 SEARCH(Key k) : ⟨Internal*, Internal*, Leaf*, Update, Update⟩ {
24     Internal *gp, *p
25     Node *l := Root
26     Update gpupdate, pupdate
27     while l points to an internal node {
28         gp := p
29         p := l
30         gpupdate := pupdate
31         pupdate := p → update
32         if k < l → key then l := p → left else l := p → right
33     }
34     return ⟨gp, p, l, pupdate, gpupdate⟩
35 }
36 FIND(Key k) : Leaf* {
37     Leaf *l
38     ⟨−, −, l, −, −⟩ := SEARCH(k)
39     if l → key = k then return l
40     else return ⊥
41 }
42 INSERT(Key k) : boolean {
43     Internal *p, *newInternal
44     Leaf *l, *newSibling
45     Leaf *new := pointer to a new Leaf node whose key field is k
46     Update pupdate, result
47     IInfo *op
48     while TRUE {
49         ⟨−, p, l, pupdate, −⟩ := SEARCH(k)
50         if l → key = k then return FALSE                          ▷ Cannot insert duplicate key
51         if pupdate.state ≠ CLEAN then HELP(pupdate)                ▷ Help the other operation
52         else {
53             newSibling := pointer to a new Leaf whose key is l → key
54             newInternal := pointer to a new Internal node with key field max(k, l → key),
                   update field ⟨CLEAN, ⊥⟩, and with two child fields equal to new and newSibling
                   (the one with the smaller key is the left child)
55             op := pointer to a new IInfo record containing ⟨p, l, newInternal⟩
56             result := CAS(p → update, pupdate, ⟨IFLAG, op⟩)        ▷ iflag CAS
57             if result = pupdate then {                             ▷ The iflag CAS was successful
58                 HELPINSERT(op)                                     ▷ Finish the insertion
59                 return TRUE
60             }
61             else HELP(result)          ▷ The iflag CAS failed; help the operation that caused failure
62         }
63     }
64 }
65 HELPINSERT(IInfo *op) {
       ▷ Precondition: op points to an IInfo record (i.e., it is not ⊥)
66     CAS-CHILD(op → p, op → l, op → newInternal)                    ▷ ichild CAS
67     CAS(op → p → update, ⟨IFLAG, op⟩, ⟨CLEAN, op⟩)                 ▷ iunflag CAS
68 }
```

Figure 3.8: Pseudocode for Find, Insert and helper routines.

```
69  DELETE(Key k) : boolean {
70      Internal *gp, *p
71      Leaf *l
72      Update pupdate, gpupdate, result
73      DInfo *op
74      while TRUE {
75          ⟨gp, p, l, pupdate, gpupdate⟩ := SEARCH(k)
76          if l → key ≠ k then return FALSE                          ▷ Key k is not in the tree
77          if gpupdate.state ≠ CLEAN then HELP(gpupdate)
78          else if pupdate.state ≠ CLEAN then HELP(pupdate)
79          else {                                                    ▷ Try to flag gp
80              op := pointer to a new DInfo record containing ⟨gp, p, l, pupdate⟩
81              result := CAS(gp → update, gpupdate, ⟨DFLAG, op⟩)     ▷ dflag CAS
82              if result = gpupdate then {                           ▷ CAS successful
83                  if HELPDELETE(op) then return TRUE                ▷ Either finish deletion or unflag
84              }
85              else HELP(result)          ▷ The dflag CAS failed; help the operation that caused the failure
86          }
87      }
88  }
89  HELPDELETE(DInfo *op) : boolean {
        ▷ Precondition: op points to a DInfo record (i.e., it is not ⊥)
90      Update result                                                 ▷ Stores result of mark CAS
91      result := CAS(op → p → update, op → pupdate, ⟨MARK, op⟩)      ▷ mark CAS
92      if result = op → pupdate or result = ⟨MARK, op⟩ then {        ▷ op → p is successfully marked
93          HELPMARKED(op)                                            ▷ Complete the deletion
94          return TRUE                                               ▷ Tell DELETE routine it is done
95      }
96      else {                                                        ▷ The mark CAS failed
97          HELP(result)                                              ▷ Help operation that caused failure
98          CAS(op → gp → update, ⟨DFLAG, op⟩, ⟨CLEAN, op⟩)           ▷ backtrack CAS
99          return FALSE                                              ▷ Tell DELETE routine to try again
100     }
101 }
102 HELPMARKED(DInfo *op) {
        ▷ Precondition: op points to a DInfo record (i.e., it is not ⊥)
103     Node *other
        ▷ Set other to point to the sibling of the node to which op → l points
104     if op → p → right = op → l then other := op → p → left else other := op → p → right
        ▷ Splice the node to which op → p points out of the tree, replacing it by other
105     CAS-CHILD(op → gp, op → p, other)                             ▷ dchild CAS
106     CAS(op → gp → update, ⟨DFLAG, op⟩, ⟨CLEAN, op⟩)               ▷ dunflag CAS
107 }
108 HELP(Update u) {                                                  ▷ General-purpose helping routine
        ▷ Precondition: u has been stored in the update field of some internal node
109     if u.state = IFLAG then HELPINSERT(u.info)
110     else if u.state = MARK then HELPMARKED(u.info)
111     else if u.state = DFLAG then HELPDELETE(u.info)
112 }
113 CAS-CHILD(Internal *parent, Node *old, Node *new) {
114     if new → key < parent → key then
115         CAS(parent → left, old, new)
116     else
117         CAS(parent → right, old, new)
118 }
```

Figure 3.9: Pseudocode for Delete and helper routines.

3.5 Documentation

Search(k): Process p that executes Search(k) traverses a path of the tree from the root to a leaf, towards the key k. It behaves exactly as in a sequential implementation of a leaf-oriented BST, choosing which direction to go at each node by comparing the key stored there to k. It returns pointers to the leaf that the search terminates, its parent and grandparent. It also returns a copy of its parent's and grandparent's Update fields. These returned values are needed by Insert and Delete to perform helping if needed, perform CAS operations and create the Info struct.

Find(k): Wrapper of Search(k). Returns a pointer to the leaf that contains k if that is found, otherwise returns NULL.

Insert(k): Process p that executes Insert(k) tries to add key k to the tree, until it finds k in the tree or succeeds to insert it. It first calls Search(k) to get pointers to the leaf that the search terminates and its parent, and a copy of its parent's Update field (#49). If the leaf contains k, the operation terminates returning FALSE (#50). Otherwise, after constructing a new subtree of three nodes (#45, #53-54) and a new IInfo struct (#55), p attempts to perform an iflag CAS on parent (#56). If the parent's state is not clean (#51) or the iflag CAS is not successful (#61), p helps the operation that the parent is involved in and retries the insertion from scratch. If the iflag CAS is successful, p proceeds to finish Insert(k) by calling HelpInsert() (#58) and returns TRUE.

Delete(k): Process p that executes Delete(k) tries to remove key k from the tree, until it cannot find k in the tree anymore or succeeds to delete it. It first calls Search(k) to get pointers to the leaf that the search terminates, its parent and its grandparent, and a copy of its parent's and grandparent's Update fields (#75). If the leaf does not contain k, the operation terminates returning FALSE (#76). Otherwise, after constructing a new DInfo struct (#80), p attempts to perform a dflag CAS on grandparent (#81). If the parent's or grandparent's state is not clean (#77-78) or the dflag CAS is not successful (#85), p helps the operation that the parent or grandparent is involved in and retries the deletion from scratch. If the dflag CAS is successful, p proceeds to finish Delete(k) by calling HelpDelete() (#83) and returns TRUE only if HelpDelete() returns TRUE, otherwise retries the deletion from scratch.

HelpInsert(op): Process p that executes HelpInsert(op) performs the ichild CAS by calling CAS-Child() and the iunflag CAS, using information from the Info struct that op points to. Note that the iunflag CAS is performed regardless of whether the ichild CAS is successful. This is because if p's ichild CAS is not successful, then an ichild CAS using information from op must have been performed

successfully by some other process that helps p.

HelpDelete(*op*): Process p that executes HelpDelete(*op*) performs the mark CAS on parent using information from the Info struct that *op* points to (#91). If this CAS is successful or some helper of p performed this CAS successfully (#92), p proceeds to finish the deletion by calling HelpMarked() (#93) and returns TRUE (#94). Otherwise, p helps the conflicting operation to complete (#97), performs a backtrack CAS on grandparent (#98) and returns FALSE (#99).

HelpMarked(*op*): Process p that executes HelpMarked(*op*) performs the dchild CAS by calling CAS-Child() and the dunflag CAS, using information from the Info struct that *op* points to. As is the case for HelpInsert(), the dunflag CAS is performed regardless of whether the dchild CAS is successful. This is because if p's dchild CAS is not successful, then a dchild CAS using information from *op* must have been performed successfully by some other process that helps p.

Help(*u*): Calls either HelpInsert() or HelpDelete() or HelpMarked(), depending on the type of flag/mark in the *u* Update variable.

CAS-Child(*parent*, *old*, *new*): Performs a child CAS, switching the *parent*'s left or right pointer from *old* to *new*.

3.6 Example

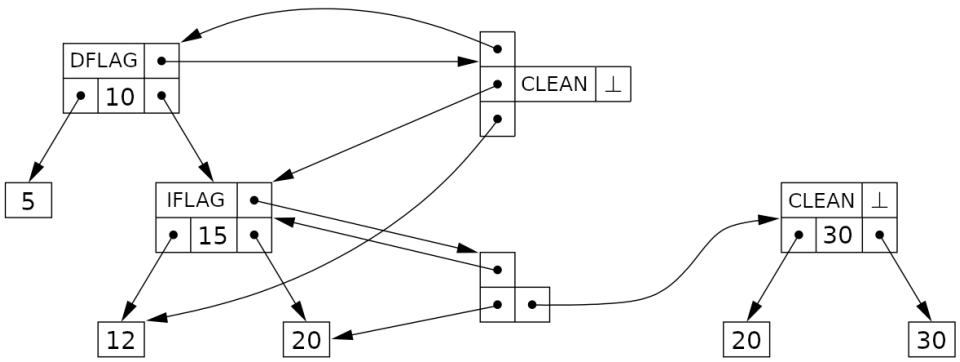


Figure 3.10: A Delete(12) and an Insert(30) are in progress.

In the above example, the tree contains the keys 5, 12 and 20. Delete(12) has flagged node 10, but before it marks node 15, Insert(30) manages to flag it. Delete's DInfo struct contains the Update field of node 15 that was returned by Search(12) (at the time that Search(12) read it, the State was CLEAN and the Info pointer

was NULL). This is different from the Update struct that node 15 currently has, therefore Delete's mark CAS will fail, leading to a backtrack CAS and a retry of the deletion. On the other hand, the insertion will succeed because the iflag CAS was performed successfully.

3.7 Garbage Collection

It is assumed for simplicity that the addresses of nodes and Info structs are not reused (each new node or Info struct gets a new memory location). Thus, whenever a node is flagged, the Info pointer of its Update variable gets a new value each time, different from any previous one. When the State of a node is restored to CLEAN, the current Info pointer is left deliberately inside. This prevents ABA problems on child pointers and Update variables.

3.8 Linearization Points

Find(), unsuccessful Insert() and unsuccessful Delete() are linearized at the time when the leaf they end up at is still in the tree. Successful Insert() or Delete() is linearized when it performs its child CAS (#115 or #117). The following table sums up the linearization points of each operation:

	Find	**Insert**	**Delete**
Successful	Latest point in Find's execution interval at which the leaf Search returns is still in the tree	child CAS	child CAS
Unsuccessful		Latest point in Insert's execution interval at which the leaf Search returns is still in the tree	Latest point in Delete's execution interval at which the leaf Search returns is still in the tree

Table 3.1: Linearization points of BST operations.

Chapter 4

The PNB-BST Algorithm

This chapter provides a description of the Persistent Non-Blocking Binary Search Tree (PNB-BST), a non-blocking, linearizable binary search tree implementation that supports wait-free range queries using single-word compare-and-swap (CAS) operations. An analysis of its properties can be found in [7]. The full proof of correctness of the algorithm can be found at [8]. Our implementation used for the experimental analysis of chapter 7 is available at [9].

4.1 Introduction

The PNB-BST algorithm builds on top of NB-BST algorithm presented in chapter 3 providing support for range query operations. Recall that in NB-BST, Insert() replaces a leaf with a new subtree of three nodes, and Delete() replaces the parent of a leaf with the leaf's sibling (Chapter 3, Fig. 3.1 and 3.2). In either case, after the operation finishes, the replaced nodes are not *reachable* from the root anymore, meaning that a Find() operation starting from the root, cannot access them. However, the PNB-BST data structure is a persistent version of NB-BST. Persistence is achieved by marking the replaced nodes and storing them in the tree under prev pointers, thus making them reachable from the root. The purpose of implementing persistence in this way is the efficient and concurrent support of range queries. In particular, PNB-BST supports four types of operations. Given the keys k, x and y:

- Insert(k): Adds k to the tree and returns TRUE unless k already exists in the tree - in that case, it does not cause any change to the tree and returns FALSE.

- Delete(k): Removes k from the tree and returns TRUE unless k does not exist in the tree - in that case, it does not cause any change to the tree and returns FALSE.

- Find(k): Returns a pointer to the node that contains k provided that k is in the tree, otherwise it returns NULL.

- RangeScan(x,y): Returns all keys existing in the tree that are greater than or equal to x and smaller than or equal to y.

In this implementation, Insert(), Delete() and Find() operations are lock-free whereas RangeScan() is wait-free. We note that Find() can also become wait free by executing a RangeScan(k,k). From now on, a process that executes an Insert() or a Delete() operation will be called an *updater*.

To guarantee the linearizability of operations, RangeScan() must be able to detect all the updates that are linearized before it, and to ignore all the updates that are linearized after it. To achieve this, PNB-BST is enhanced with the following mechanisms (that are discussed in more detail later on):

- A variable storing a sequence number which is incremented by RangeScan and read by the updaters. Each updater stores this sequence number in the new nodes that it will create and this is the sequence number of the node.

- A more advanced flagging and helping scheme than those of NB-BST.

- A validation scheme needed to synchronize updaters with processes executing range queries.

Moreover, operations are linearized differently in PNB-BST than in NB-BST.

As in NB-BST, PNB-BST maintains a full, external BST that consists of two types of nodes: Internal nodes and Leaf nodes. Every PNB-BST node has two fields in addition to the fields of an NB-BST node: A sequence number *seq*, and a pointer *prev* (Fig. 4.4). Unlike NB-BST, Leaf nodes have the same fields with Internal nodes. However, the child pointers exist only in Internal nodes.

The Find() operation is similar to that of NB-BST except that it performs helping when another operation operates on the parent or grandparent of the leaf that it arrives to.

As in NB-BST, the Insert() operation replaces a leaf with a new subtree of three nodes. Before it does so, it marks the leaf to be replaced as deleted. However, it additionally sets the prev pointer of the new subtree's internal node to point to that leaf (Fig. 4.1). This is necessary to achieve persistence.

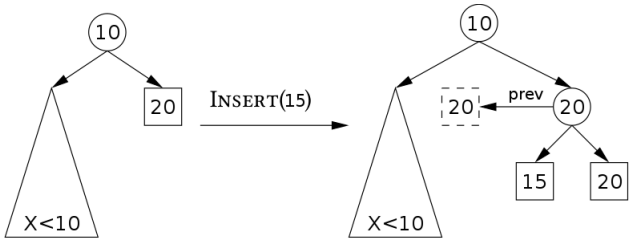


Figure 4.1: Insert() operation.

The Delete() operation replaces the parent of a leaf with a new copy of the leaf's sibling. Before it does so, it marks the nodes to be replaced (i.e. the old leaf, its sibling and their parent) as deleted. However, it additionally sets the prev pointer of the new sibling to point to that parent (Fig. 4.2). This is necessary to achieve persistence.

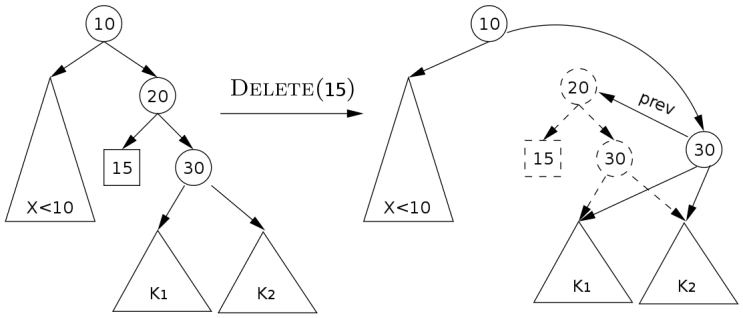


Figure 4.2: Delete() operation.

This way, a RangeScan() can visit older versions of nodes when this is needed to complete its work, as described in more detail later on.

4.2 Overview

4.2.1 Sequence numbers

PNB-BST utilizes a mechanism of sequence numbers to synchronize RangeScan() with the other operations. This is implemented as follows: There is a global shared integer variable called *Counter*, that is initialized to 0. Each RangeScan() starts by reading this Counter, and the value recorded is called the sequence number of the RangeScan() operation. RangeScan() atomically increments Counter immediately after. Note that there is no need to use FetchAndIncrement() to read Counter, i.e. many RangeScan operations may read the same Counter value. This is so because every increment of the Counter may cause some updaters to restart their operation.

Every other operation (updates and Find() operations) makes repeated attempts to perform its work by executing a loop. Each execution of the loop is called an *attempt*. An attempt starts by reading Counter. The sequence number of the operation is the value of the Counter read in the operation's last attempt.

This way the execution is logically divided into phases, each of them having a unique Counter value. Phase i is the period in which Counter has the value i. Each operation is then assigned (or *belongs*) to a phase, depending on its sequence number. Thus, an operation that has sequence number i, belongs to phase i.

During an attempt, each update operation writes its sequence number in the new nodes and Info struct it creates. Note that both Insert() and Delete() create new nodes, since due to persistence no node is ever physically deleted from the tree: Insert() creates a subtree of three nodes and Delete() creates a node that substitutes the sibling of the leaf to be deleted.

As later discussed in detail, PNB-BST assigns linearization points in a different way than NB-BST. More specifically, RangeScan() is linearized at the end of the phase it belongs to. Because many RangeScan() operations can have the same sequence number (thus belonging to the same phase), ties are broken arbitrarily. Find(), Insert() and Delete() are linearized at some point inside the phase they belong to. In particular, successful update operations are linearized at the time they perform their last successful flag CAS. More details will be provided later on.

To ensure linearizability, RangeScan() has to 1) ignore all updates that belong to later phases, and 2) get aware of all the latest updates that belong to an earlier or the same phase with its own.

Because of sequence numbers, RangeScan() is able to distinguish which nodes belong to later phases by comparing their sequence number to its own. Whenever it visits a node that belongs to a later phase, it ignores it by following the node's prev pointers, until reaching a node whose sequence number is smaller than or equal to its own. It is guaranteed that such a node always exists.

On the other hand, to guarantee that RangeScan() will see all updates that belong to an earlier or the same phase with its own, RangeScan() has to synchronize with pending update operations of these phases. As already stated, successful updates are linearized at the time they perform their last successful flag CAS. This brings the need for considering two cases in order to achieve synchronization between successful updates and RangeScan().

Consider an update that belongs to the same (or an earlier) phase with (than) a RangeScan().

1) **The update performs its flag CAS, then RangeScan() begins its execution and arrives to the flagged node**. Because the update has not yet updated the child pointers of any nodes in the tree, if RangeScan() ignores the flagged node and continues its work, it will miss the update and linearizability will be violated, because this update will be linearized before the RangeScan(). Thus, RangeScan() attempts to help the operation that

flagged the node before doing its own work. Find() also attempts to help, for the same reason.

2) **The update performs its flag CAS on a node that RangeScan() has already visited**. In that case, if the update continues its attempt, linearizability will be violated because RangeScan() will not traverse the same part of the tree again and thus will miss the update. To avoid this case, the update checks the current Counter value and continues its attempt only if that value is equal to its sequence number. In that case, it is guaranteed that no RangeScan() of the same sequence number has started traversing the tree - if it had, the Counter would have been incremented first. This is called the *handshaking* check, and if it fails, the update aborts its current attempt and retries.

Note that the sequence number of each update operation is stored in the Info struct that the operation creates. This value is used to perform the handshaking. This is necessary for the following reason:
Problematic scenario: Suppose that "seq" variable inside Info struct does not exist, so line #111 becomes "if Counter $\neq$ seq" and this seq variable is the local seq of the current operation. pA: **Insert(5)** (seqA=0) until #113. pB: **RangeScan(4,6)** (seqB=0, Counter=1) until #131. pC: **RangeScan(7,8)** (seqC=1, Counter=2) until #131. Finally, pD: **Insert(20)** (seqD=2) until end. pD has to help pA before it continues with its own Insert operation, so it executes pA's work successfully, writing its own seqD number to the newly inserted 5. Then, pB continues until end, missing pA's Insert(5) which was linearized at line #103. Thus, linearizability is violated.

4.2.2 Flagging, Helping and Validation scheme

Flagging In PNB-BST every node has an Update field, which contains an enum variable called *Type* (with possible values FLAG and MARK) and a pointer *info* to an Info struct, which contains an enum variable called *State* (with possible values $\perp$, TRY, COMMIT and ABORT) as shown in Fig. 4.4. The Type and State of a node reflects the situation it is currently involved in. This is summarized in fig. 4.3 .

State \ Type	FLAG	MARK
⊥	Intention to Insert or Delete a child node - **Before handshaking**	
TRY	Intention to Insert or Delete a child node - **After handshaking**	Intention to be deleted itself
COMMIT	Not deleted and not processed by any update operation	The node is deleted
ABORT	Not deleted and not processed by any update operation	Not deleted and not processed by any update operation

frozen state

Figure 4.3: Types and States that a node can be in.

All new nodes are initialized to (FLAG, ABORT). Note that (⊥, MARK) can never happen. A node is called *frozen* (blue-colored cells) when either it is deleted from the tree (COMMIT, MARK) or it is being processed by an operation that is still in progress (⊥ or TRY).

As in NB-BST, before an update operation attempts to change any child pointers of a node, it performs a flag CAS on this node, changing the node's Type and State to (FLAG, ⊥). A (FLAG, ⊥) indicates an upcoming change to the child pointers of the node, and makes the operation visible to the other processes for the first time, so they can help. The rest fields of the Info struct contain all the necessary information for any process to complete the work of the update operation.

Next, the handshaking check takes place and if it succeeds, the node's State is changed to TRY. After that, the nodes that will be deleted from the tree are marked one by one, with mark CAS operations. If all marks are successful and after the child CAS is performed (which is then guaranteed to succeed), the State of the Info struct pointed by the nodes' *info* pointers is changed from TRY to COMMIT, and the operation is completed successfully. Note that every marked node's info pointer is set to point to the same Info struct with the flagged node, so State can change simultaneously for all nodes involved in the operation. Upon failure of any mark CAS, the State of the nodes is changed from TRY to ABORT indicating that the operation is aborted (meaning that it will not take effect) and the update is retried by performing a new attempt.

Helping In PNB-BST, all types of operations (Find(), Insert(), Delete() and RangeScan()) perform helping when necessary. RangeScan() performs helping whenever it visits a node that is processed by an in-progress operation. After helping, RangeScan() continues its execution.

Updates and Find() perform helping whenever the parent or grandparent of the leaf they end up at is frozen. They also perform helping if any of the nodes they are going to process is being processed by another in-progress operation. After helping, they initiate a new attempt.

Validations In PNB-BST Find(), Insert() and Delete() perform validations. Roughly speaking, this is done to ensure that the nodes they are going to process have not been deleted from the tree and are not being processed by another in-progress operation.

4.3 Data Structures, Initialization and Pseudocode

In addition to the initialization that takes place in NB-BST, all fields of an Info struct, except from the State field, are immutable. Fig. 4.4 contains the structs that are used by the algorithm, and figures 4.5, 4.6 and 4.7 contain the pseudocode of the algorithm, followed by a detailed explanation of the functions. Figures 4.4, 4.5, 4.6 and 4.7 provide the pseudocode for PNB-BST as it appears in [7].

```
1   type Update {                ▷ stored in one CAS word
2       {Flag, Mark} type
3       Info *info
4   }
5   type Info {
6       {⊥, Try, Commit, Abort} state
7       Internal *nodes[]        ▷ nodes to be frozen
8       Update oldUpdate[]       ▷ old values for freeze CAS steps
9       Node *mark[]             ▷ nodes to be marked
10      Internal *par            ▷ node whose child will change
11      Node *oldChild           ▷ old value for child CAS
12      Node *newChild           ▷ new value for the child CAS
13      int seq                  ▷ sequence number
14  }
15  type Internal {
16      Key ∪ {∞1, ∞2} key
17      Update update
18      Node *left, *right
19      Node *prev
20      int seq
21  }
22  type Leaf {
23      Key ∪ {∞1, ∞2} key
24      Update update
25      Node *prev
26      int seq
27  }
28  ▷ Initialization:
29  shared counter Counter := 0
30  shared Info *Dummy := pointer to a new Info object whose state field is Abort, and whose other fields are ⊥
31  shared Internal *Root := pointer to new Internal node with key field ∞2, update field ⟨Flag, Dummy⟩,
        prev field ⊥, seq field 0, and its left and right fields pointing to new Leaf nodes whose prev fields
        are ⊥, seq fields are 0, and keys ∞1 and ∞2, respectively
```

Figure 4.4: Data structures and initialization.

```
32 SEARCH(Key k, int seq): ⟨Internal*, Internal*, Leaf*⟩ {
33     ▷ Precondition: seq ≥ 0
34     Internal *gp, *p
35     Node *l := Root
36     while l points to an internal node {
37         gp := p
38         p := l
39         l = READCHILD(p, k < p → key, seq)          ▷ Go to appropriate version-seq child of p
40     }
41     return ⟨gp, p, l⟩
42 }

43 READCHILD(Internal * p, Boolean left, int seq): Node* {
44     ▷ Precondition: p is non-⊥ and p → seq ≤ seq
45     if left then l := p → left else l := p → right
46     while (l → seq > seq) l := l → prev
47     return l;
48 }

49 VALIDATELINK(Internal *parent, Internal *child, Boolean left): ⟨Boolean, Update⟩ {
50     ▷ Preconditions: parent and child are non-⊥
51     Update up
52     up := parent → update
53     if FROZEN(up) then {
54         HELP(up.info)
55         return ⟨FALSE, ⊥⟩
56     }
57     if (left and child ≠ parent → left) or (¬left and child ≠ parent → right) then return ⟨FALSE, ⊥⟩
58     else return ⟨TRUE, up⟩
59 }

60 VALIDATELEAF(Internal *gp, Internal *p, Leaf *l, Key k) : ⟨Boolean, Update, Update⟩ {
61     ▷ Preconditions: p and l are non-⊥ and if p ≠ Root then gp is non-⊥
62     Update pupdate, gpupdate := ⊥
63     Boolean validated
64     ⟨validated, pupdate⟩ := VALIDATELINK(p, l, k < p → key)
65     if validated and p ≠ Root then ⟨validated, gpupdate⟩ := VALIDATELINK(gp, p, k < gp → key)
66     validated := validated and p → update = pupdate and (p = Root or gp → update = gpupdate)
67     return ⟨validated, gpupdate, pupdate⟩
68 }

69 FIND(Key k): Leaf* {
70     Internal * gp, p
71     Leaf *l
72     Boolean validated

73     while TRUE {
74         seq := Counter
75         ⟨−, p, l⟩ := SEARCH(k, seq)
76         ⟨validated, −, −⟩ := VALIDATELEAF(gp, p, l, k)
77         if validated then {
78             if l → key = k then return l
79             else return ⊥
80         }
81     }
82 }

83 CAS-CHILD(Internal *parent, Node *old, Node *new) {
       ▷ Precondition: parent points to an Internal node and new points to a Node (i.e., neither is ⊥) and new → prev = old
       ▷ This routine tries to change one of the child fields of the node that parent points to from old to new.
84     if new → key < parent → key then
85         CAS(parent → left, old, new)                ▷ child CAS
86     else
87         CAS(parent → right, old, new)               ▷ child CAS
88 }
```

Figure 4.5: Pseudocode for Find and helper routines.

```
89  FROZEN(Update up): Boolean {
90      return ((up.type = FLAG and up.info → state ∈ {⊥, TRY}) or
                (up.type = MARK and up.info → state ∈ {⊥, TRY, COMMIT}))
91  }

92  EXECUTE (Internal *nodes[], Update oldUpdate[], Internal *mark[], Internal *par,
                Node *oldChild, Node *newChild, int seq): Boolean {
93      ▷ Preconditions: (a) Elements of nodes are non-⊥, (b) mark is a subset of nodes, (c) par is an element of nodes,
94          ▷ (d) oldChild and newChild are distinct and non-⊥, (e) oldChild is an element of mark,
95          ▷ (f) newChild → prev = oldChild, and (g) if par = Root then newChild → key is infinite.
96      for i := 1 to length of oldUpdate {
97          if FROZEN(oldUpdate[i]) then {
98              if oldUpdate[i].info → state ∈ {⊥, TRY} then HELP(oldUpdate[i].info)
99              return FALSE
100         }
101     }
102     infp := pointer to a new Info record containing ⟨⊥, nodes, oldUpdate, mark, par, oldChild, newChild, seq⟩
103     if CAS(nodes[1] → update, oldUpdate[1], ⟨FLAG, infp⟩) then          ▷ flag CAS
104         return HELP(infp)
105     else return FALSE
106 }

107 HELP(Info *infp): boolean {
108     ▷ Precondition: infp is non-⊥ and does not point to the Dummy Info object
109     int i := 2
110     boolean continue

111     if Counter ≠ infp → seq then
112         CAS(infp → state, ⊥, ABORT)                                    ▷ abort CAS
113     else CAS(infp → state, ⊥, TRY)                                     ▷ try CAS
114     continue := (infp → state = TRY)
115     while continue and i ≤ length of infp → nodes do {
116         if infp → nodes[i] appears in infp → mark then
117             CAS(infp → nodes[i] → update, infp → oldUpdate[i], ⟨MARK, infp⟩) ▷ mark CAS
118         else CAS(infp → nodes[i] → update, infp → oldUpdate[i], ⟨FLAG, infp⟩) ▷ flag CAS
119         continue := (infp → nodes[i] → update.info = infp)
120         i := i + 1
121     }
122     if continue then {
123         CAS-CHILD(infp → par, infp → oldChild, infp → newChild)
124         infp → state := COMMIT
125     } else if infp → state = TRY then
126         infp → state := ABORT
127     return (infp → state = COMMIT)
128 }

129 RANGESCAN(int a, int b): Set {
130     seq := Counter
131     Inc(Counter)
132     return SCANHELPER(Root, seq, a, b)
133 }

134 SCANHELPER(Node *node, int seq, int a, int b): Set {
135     ▷ Precondition: node points to a node with node → seq ≤ seq
136     Info * infp

137     if node points to a leaf then return {node → key} ∩ [a, b]
138     else {
139         infp := node → update.info
140         if infp → state ∈ {⊥, TRY} then HELP(infp)
141         if a > node → key then return SCANHELPER(READCHILD(node, FALSE, seq), a, b)
142         else if b < node → key then return SCANHELPER(READCHILD(node, TRUE, seq), a, b)
143         else return SCANHELPER(READCHILD(node, FALSE, seq), a, b) ∪
144                 SCANHELPER(READCHILD(node, TRUE, seq), a, b)
145     }
146 }
```

Figure 4.6: Pseudocode for RangeScan and helper routines.

```
147 Insert(Key k): boolean {
148     Internal * gp, *p, *newInternal
149     Leaf *l, *newSibling
150     Leaf *new
151     Update pupdate
152     Info *infp
153     Boolean validated

154     while True {
155         seq := Counter
156         ⟨gp, p, l⟩ := Search(k, seq)
157         ⟨validated, −, pupdate⟩ := ValidateLeaf(gp, p, l, k)
158         if validated then {
159             if l → key = k then return False                    ▷ Key k is in the tree
160             else {
161                 new := pointer to a new Leaf node whose key field is k, its seq field is equal to seq, and its prev field is ⊥
162                 newSibling := pointer to a new Leaf whose key is l → key,
                        its prev field is equal to ⊥ and its seq field is equal to seq
163                 newInternal := pointer to a new Internal node with key field max(k, l → key),
                        update field ⟨Flag, Dummy⟩, its seq field equal to seq and its prev field equal to l,
                        and with two child fields equal to new and newSibling
                        (the one with the smaller key is the left child),
164                 if Execute([p, l], [pupdate, l → update], [l], p, l, newInternal, seq) then return True
165             }
166         }
167     }
168 }

169 Delete(Key k): boolean {
170     Internal *gp, *p
171     Leaf *l
172     Node *sibling, *newnode
173     Update pupdate, gpupdate, supdate
174     Info *infp
175     Boolean validated

176     while True {
177         seq := Counter
178         ⟨gp, p, l⟩ := Search(k, seq)
179         ⟨validated, gpupdate, pupdate⟩ := ValidateLeaf(gp, p, l, k)
180         if validated then {
181             if l → key ≠ k then return False                    ▷ Key k is not in the tree
182             sibling := ReadChild(p, l → key ≥ p → key, seq)
183             ⟨validated, −⟩ := ValidateLink(p, sibling, l → key ≥ p → key)
184             if validated then {
185                 newNode := pointer to a new copy of sibling with its seq field set to seq and its prev pointer set to p
186                 if sibling is Internal then {
187                     ⟨validated, supdate⟩ := ValidateLink(sibling, newNode → left, True)
188                     if validated then ⟨validated, −⟩ := ValidateLink(sibling, newNode → right, False)
189                 } else supdate = sibling → update
190                 if validated and Execute([gp, p, l, sibling], [gpupdate, pupdate, l → update, supdate],
                                             [p, l, sibling], gp, p, newNode, seq) then
191                     return True
192             }
193         }
194     }
195 }
```

Figure 4.7: Pseudocode for Insert and Delete.

4.4 Documentation

Search(*k*, *seq*): Process p that executes Search(k, seq) traverses a path of the tree from the root to a leaf consisting of nodes with version at most seq (#39) by following prev pointers in the nodes it traverses when necessary, towards the key k. It chooses which direction to go at each node by comparing the key stored there to k. It returns pointers to the leaf that the search terminates, its parent and grandparent.

Find(*k*): Wrapper of Search(k, seq). It gets a sequence number seq and calls Search(k, seq) (#74-75). After it validates the results of Search() (retries in case of validation failure), it returns a pointer to the leaf that contains k if that is found, otherwise it returns NULL (#76-81).

RangeScan(*x*,*y*): It returns all existing keys of the tree that are greater than or equal to x and smaller than or equal to y. It reads the Counter, increments it and calls ScanHelper() to perform the actual scanning.

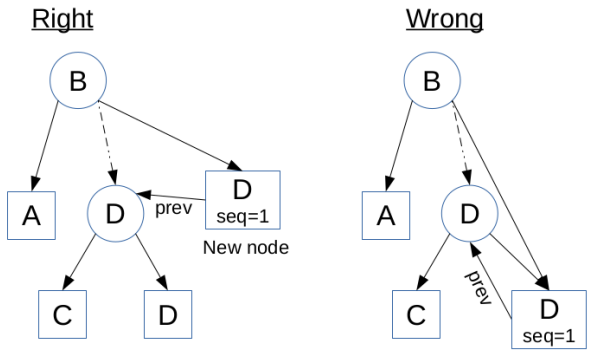
ScanHelper(*node*, *seq*, *x*, *y*): Starting from *node*, it recursively traverses the tree searching for leaves of version at most seq whose keys fall into range $[x,y]$, and performs helping when needed. At each node, if the node's key is smaller than x, the traversal continues on the right child of the node. If it is greater than y, the traversal continues only on the node's left child. Otherwise, both subtrees of the node are traversed. The proper version of the child node is discovered by calling ReadChild().

ReadChild(*p*, *left*, *seq*): Depending on the value of boolean parameter *left*, it returns the most recent version of the left or right child of p with sequence number at most seq.

Insert(*k*): Process p that executes Insert(k) tries to add key k to the tree, until it finds k in the tree or succeeds to insert it. It first gets a sequence number seq and calls Search(k, seq) (#155-156). After p validates the results of Search() by calling ValidateLeaf() (retries in case of validation failure), if the leaf contains k, the operation terminates returning FALSE (#159). Otherwise, after constructing a new subtree of three nodes (#161-163), p calls Execute() to execute the remaining actions.

Delete(*k*): Process p that executes Delete(k) tries to remove k from the tree, until it cannot find k in the tree anymore or succeeds to delete it. It first gets a sequence number seq and calls Search(k, seq) (#177-178). After p validates the results of Search() (retries in case of validation failure), if the leaf does not contain k, the operation terminates returning FALSE (#181). Otherwise, after constructing

a new node that will substitute sibling, and after validating its child pointers in case sibling is an internal node, p calls Execute() to execute the remaining actions. Recall that Delete() replaces the parent of a leaf with a new copy of the leaf's sibling. This is necessary for the following reason:



Problematic scenario: Suppose that Delete does not make a new copy of sibling. pA: **Find(E)** (seqA=0) until #74. pB: **RangeScan(X,Y)** until #131. pC: **Delete(C)** (seqC=1) until the end. pA: continues forever, cycling between the two nodes with key D because ReadChild will return the internal node with key D each time that is called by Search, thus Search will never exit the while loop in lines #36-40.

Execute(*infp*): Process p that executes Execute(*infp*) checks if any of the old Update values of the nodes involved in the operation are frozen (by calling frozen()), and in that case it performs helping and returns FALSE (#96-99). Otherwise, p constructs an info struct for the operation that called it (#102) and performs a flag CAS on the first of the nodes involved (#103). If the flag CAS is performed successfully, p continues the execution of the operation by calling Help(). Otherwise, it returns FALSE.

Help(*infp*): Process p that executes Help(*infp*) performs the handshaking (#111-113), the mark CAS operations (#115-120) and the child CAS by calling CAS-Child() (#123), using information from the Info struct that *infp* points to. If the handshaking or any of the mark CAS operations fails, p aborts the operation and returns FALSE, otherwise it commits the operation (#124) and returns TRUE. Note that if a child CAS using information from *infp* is executed, even if it fails, a helper of the operation has successfully executed a similar child CAS using information from *infp*.
To understand why the lines #125-126 are needed, consider the following execution:
Problematic scenario: Suppose that lines #125-126 do not exist. pA: **Delete(X)** until #102. pB: **Insert(5)** until #103. Suppose that pB flags the sibling of X (which is an Internal node). pA: continues, reaches line #117, tries to Mark sibling during the 3rd iteration of the loop and fails because this node is already flagged

by pB. Continuing its execution, pA returns False and retries to Delete(X), forever. As a result, the non-blocking property is violated.

ValidateLeaf(*gp*, *p*, *l*, *k*): Process p that executes ValidateLeaf(*gp*, *p*, *l*, *k*) calls two times ValidateLink(), to validate the links between grandparent-parent (#64) and parent-leaf (#65). After that, p validates the update fields of grandparent and parent (#66) and returns them together with the boolean result of the validation (#67). The returned values are stored in the Info struct created by Insert() and Delete() operations which then also use them to perform CAS operations.

ValidateLink(*parent*, *child*, *left*): Process p that executes ValidateLink(*parent*, *child*, *left*) checks if *parent* is frozen (#53) and in that case p helps complete the operation that the parent is involved in (#54) and returns FALSE (#55). If *parent* is not frozen, p validates the link between *parent* and left or right *child* (depending on *left* boolean parameter) and returns TRUE together with the *parent*'s update field, otherwise if the validation of the link fails it returns FALSE (#57-58).

Frozen(*up*): Returns TRUE if *up* is in frozen state, otherwise returns FALSE.

CAS-Child(*parent*, *old*, *new*): Performs a child CAS, switching the *parent*'s left or right pointer from *old* to *new*.

4.5 Linearization points

Find(), unsuccessful Insert() and unsuccessful Delete() are linearized at the time when the leaf they end up at is successfully validated (#66 with validated variable having the value TRUE). A successful Insert() or Delete() is linearized when it performs the flag CAS (#103). RangeScan() is linearized at the end of its phase, with ties broken arbitrarily. The following table sums up the linearization points of the operations in PNB-BST:

	Find	**Insert**	**Delete**	**RangeScan**
Successful	Terminal leaf is	flag CAS	flag CAS	End of
Unsuccessful	successfully validated	same with Find	same with Find	its phase

Table 4.1: Linearization points of PNB-BST operations.

Chapter 5

The LFCA Algorithm

This chapter provides a description of the Lock-Free Contention Adapting (LFCA) Search Tree, a lock-free, linearizable key-value store with range query support. To boost performance, LFCA performs modifications of the tree taking into consideration the encountered contention and the performance of range queries, a procedure that is called *adaptation*. The full proof of the properties and the correctness of the algorithm can be found in [10]. The implementation used for the experimental analysis of chapter 7 is available at [13].

5.1 Introduction

The Lock-Free Contention Adapting search tree supports four types of operations. Given the keys k, x and y:

- Insert(k): Adds k to the tree and returns TRUE unless k already exists in the tree - in that case, it does not cause any change to the tree and returns FALSE.
- Remove(k): Deletes k from the tree and returns TRUE unless k does not exist in the tree - in that case, it does not cause any change to the tree and returns FALSE.
- Lookup(k): Returns TRUE if k is in the tree, otherwise it returns FALSE.
- Query(x,y): Returns all keys of the tree that are greater than or equal to x and smaller than or equal to y.

In this implementation, all operations are non-blocking.

LFCA tree is a partially external tree that consists of three types of nodes: *route* nodes, *base* nodes and *leaf containers* (Fig. 5.1). These nodes are organized in layers. The upper layer contains the route nodes that form a binary search tree. The middle layer contains the base nodes, which are the leaves of this binary search

tree. The bottom layer consists of leaf containers, in which all keys of the LFCA tree are stored. Each leaf container is an immutable treap [11]. Every base node contains a pointer that points to the root of a treap. Roughly speaking, a treap is a binary search tree, with the following additional property: After an insertion of a leaf, a random number of rotations are performed in the path of the new leaf to keep the tree balanced over time.

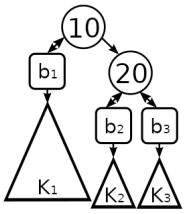
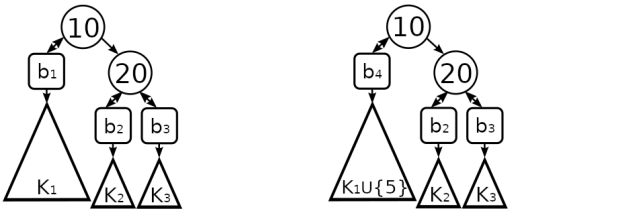


Figure 5.1: The LFCA tree. Route nodes are depicted as round boxes, base nodes as squared boxes and leaf containers as triangles.

Every operation starts by performing a search on route nodes to find the base node that contains a pointer to the root of the immutable treap it is going to operate on.

The Lookup() operation, after it finds the appropriate base node, it performs a search in the treap pointed to by the base node's treap pointer.

The Insert() operation, after it finds the appropriate base node, performs a CAS to attempt to replace it with a new base node which contains a pointer to the root of a treap that holds all keys from the previous treap plus the new key. The Remove() operation works in a similar manner, except that the new treap holds all keys from the previous treap except from the key to be deleted. Although immutable, treaps support the creation of a new version of the data structure in log(n) time, where n is the number of items in the treap. This is achieved by copying the path from the treap root to the appropriate treap leaf. Therefore, updates are efficient.



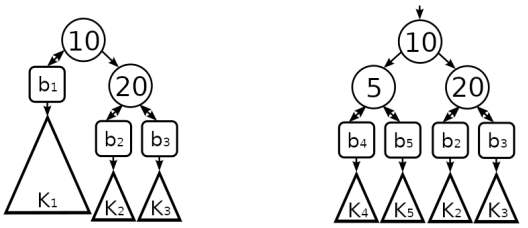
(a) Initial state (b) After an Insert(5)

Figure 5.2: The effect of an Insert() operation.

Every base node contains a statistics value (i.e. an integer) that indicates the

degree of contention on it. Each operation, after performing its work, it checks this statistics value to decide whether a modification of the tree structure (or *adaptation*) needs to take place. There are two kinds of adaptations, *split* and *join*, that correspond to high and low contention respectively.

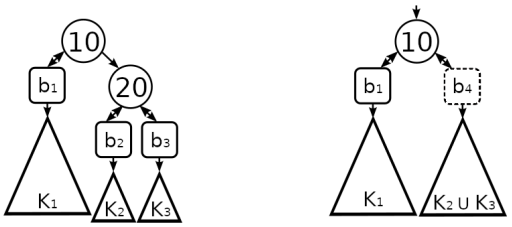
Split replaces a base node with a subtree of three nodes consisting of an internal node and two new base nodes, effectively dividing the set represented by the initial base node's leaf container in two sets of approximately equal size, to reduce contention.



(a) Initial state (b) After a split

Figure 5.3: The effect of a split operation.

Join replaces two adjacent base nodes with a new one, effectively merging two leaf containers in one. This is done by replacing the two base nodes and their parent with a new base node. In this way, range queries can work more efficiently in that part of the tree.



(a) Initial state (b) After a join

Figure 5.4: The effect of a join operation.

Splits and joins are also supported by treaps in log(n) time, where n is the number of items in the treap.

The Query() operation visits all base nodes whose leaf containers hold the keys that fall in the range of the query, joins these leaf containers into a new immutable treap and performs the range query on it.

5.2 Overview

LFCA tree differs from NB-BST by applying the following two ideas: 1) Using the treap data structure, keys are batched in the leaves of the treaps and 2) Splits and joins (which are performed in a way similar to Insert and Delete in NB-BST) are performed based on the contention encountered on the base nodes. The reason behind the first idea is that batching is beneficial in terms of performance when executing range queries. The reason behind the second idea is that fine-grained synchronization usually favors single-item operations such as Insert(), Remove() and Lookup(), whereas coarse-grained synchronization usually favors multi-item operations such as Query(). With the adaptation approach, synchronization in each part of the tree can dynamically become as coarse- or as fine-grained as needed, trading between the range query performance and the scalability of updates, to increase the overall performance.

This is implemented as follows: Each base node, besides a pointer to its treap, called *data*, and a pointer to its parent route node, called *parent*, also contains an integer variable, called *stat* (or *statistics* variable) which indicates the degree of contention happening on that base node. Whenever the stat variable goes above or below some predefined limits, a join or a split operation is executed. More specifically, stat is initialized to 0. Whenever it becomes greater than a constant variable which has the predefined value 1000, a split will be executed, and whenever it becomes smaller than -1000, a join will be executed. This way synchronization becomes more fine-grained in parts of the tree where contention has been high and more coarse-grained in parts where contention has been low or a range query often has to access many base nodes to complete its operation.

Replaceability To achieve synchronization between operations that need to work on the same set of base nodes, the concept of base node *replaceability* is used. Each base node has an enum variable called *type*. Initially, all base nodes are created by Insert() operations with type *normal*, indicating that they are replaceable. Whenever a Query() or a join operation intends to work on a specific set of base nodes, it first makes each one of them irreplaceable by replacing it with a new base node whose type is not normal. Only replaceable nodes can be replaced from the tree. Irreplaceability becomes the equivalent of locking, and progress is achieved by helping.

Heuristics Contention is measured based on success or failure of the CAS primitives that are executed by the algorithm. The function new_stat() returns the current value of the statistics variable.

Adaptation Adaptation to contention is realized with splits and joins.

Recall that a split (or *high_contention_adaptation*) replaces a base node with a new route node that has two new base nodes as children. Each new base node's

treap contains approximately half of the keys of the replaced base node's treap. Moreover, a join (or *low_contention_adaptation*) replaces two neighboring base nodes with a new base node whose treap contains the union of the keys of the previous treaps.

5.3 Initialization & Pseudocode

At the beginning, the root node is a base node of type normal_base with its treap and parent pointers set to NULL and its statistics variable set to 0.

The pseudocode of the algorithm is presented in the next three pages. Figures 5.5, 5.6 and 5.7 provide the pseudocode for LFCA tree as it appears in [12]. A detailed explanation of the functions follows.

```
// Constants
#define CONT_CONTRIB     250       // For adaptation
#define LOW_CONT_CONTRIB 1
#define RANGE_CONTRIB    100
#define HIGH_CONT        1000
#define LOW_CONT         -1000
#define NOT_FOUND        (node*)1  // Special pointers
#define NOT_SET          (treap*)1
#define PREPARING        (node*)0  // Used for join
#define DONE             (node*)1
#define ABORTED          (node*)2
enum contention_info { contended, uncontened, noinfo }
// Data Structures
struct route_node {
  int key;                          // Split key
  atomic node* left;                // < key
  atomic node* right;               // >= key
  atomic bool valid = true;         // Used for join
  atomic node* join_id = NULL;
}
struct normal_base {
  treap* data = NULL;   // Items in the set
  int stat = 0;         // Statistics variable
  node* parent = NULL;  // Parent node or NULL (root)
}
struct join_main with_fields_from normal_base {
  node* neigh1;   // First (not joined) neighbor base
  atomic node* neigh2 = PREPARING;  // Joined n
  node* gparent;  // Grandparent
  node* otherb;   // Other branch
}
struct join_neighbor with_fields_from normal_base {
  node* main_node   // The main node for the join
}
struct rs {         // Result storage for range queries
  atomic treap* result = NOT_SET;  // The result
  atomic bool more_than_one_base = false;
}
struct range_base with_fields_from normal_base {
  int lo; int hi;  // Lower and higher key of the range
  rs* storage;
}
enum node_type {
  route, normal, join_main, join_neighbor, range
}
struct node with_fields_from normal_base, range_base,
                             join_main, join_neighbor{
  node_type type;
}
struct lfcat{
  atomic node* root;
}
// Helper functions
bool try_replace(lfcat* m, node* b, node* new_b){
  if( b->parent == NULL )
    return CAS(&m->root, b, new_b);
  else if(aload(&b->parent->left) == b)
    return CAS(&b->parent->left, b, new_b);
  else if(aload(&b->parent->right) == b)
    return CAS(&b->parent->right, b, new_b);
  else return false;
}
bool is_replaceable(node* n) {
  return (n->type == normal ||
          (n->type == join_main &&
           aload(&n->neigh2) == ABORTED) ||
          (n->type == join_neighbor &&
           (aload(&n->main_node->neigh2) == ABORTED ||
            aload(&n->main_node->neigh2) == DONE)) ||
          (n->type == range &&
           aload(&n->storage->result) != NOT_SET));
}
```

```
// Helper functions
void help_if_needed(lfcatree* t, node* n){
  if(n->type == join_neighbor) n = n->main_node;
  if(n->type == join_main &&
     aload(&n->neigh2) == PREPARING){
    CAS(&n->neigh2, PREPARING, ABORTED);
  }else if(n->type == join_main &&
           aload(&n->neigh2) > ABORTED){
    complete_join(t, n);
  }else if(n->type == range &&
           aload(&n->storage->result) == NOT_SET){
    all_in_range(t, n->lo, n->hi, n->storage);
  }
}
int new_stat(node* n, contention_info info){
  int range_sub = 0;
  if(n->type == range &&
     aload(&n->storage->more_than_one_base))
    range_sub = RANGE_CONTRIB;
  if (info == contended && n->stat <= HIGH_CONT) {
    return n->stat + CONT_CONTRIB - range_sub;
  }else if(info == uncontened && n->stat >= LOW_CONT){
    return n->stat - LOW_CONT_CONTRIB - range_sub;
  }else return n->stat;
}
void adapt_if_needed(lfcatree* t, node* b){
  if(!is_replaceable(b)) return;
  else if(new_stat(b, noinfo) > HIGH_CONT)
    high_contention_adaptation(t, b);
  else if(new_stat(b, noinfo) < LOW_CONT)
    low_contention_adaptation(t, b);
}

bool do_update(lfcatree* m,
               treap*(*u)(treap*,int,bool*), int i){
  contention_info cont_info = uncontened;
  while(true){
    node* base = find_base_node(aload(&m->root), i);
    if(is_replaceable(base)){
      bool res;
      node* newb = new node{
        type = normal,
        parent = base->parent,
        data = u(base->data, i, &res),
        stat = new_stat(base, cont_info)
      }
      if(try_replace(m, base, newb)){
        adapt_if_needed(m, newb);
        return res;
      }
    }
    cont_info = contended;
    help_if_needed(m, base);
  }
}
// Public interface
bool insert(lfcat* m, int i){
  return do_update(m, treap_insert, i);
}
bool remove(lfcat* m, int i){
  return do_update(m, treap_remove, i);
}
bool lookup(lfcat* m, int i){
  node* base = find_base_node(aload(&m->root), i);
  return treap_lookup(base->data, i);
}
void query(lfcat* m, int lo, int hi,
           void (*trav)(int, void*), void* aux){
  treap* result = all_in_range(m, lo, hi, NULL);
  treap_query(result, lo, hi, trav, aux);
}
```

Figure 5.5: Data structures, helper routines and public interface

```
node* find_next_base_stack(stack* s) {
  node* base = pop(s);
  node* t = top(s);
  if(t == NULL) return NULL;
  if(aload(&t->left) == base)
    return leftmost_and_stack(aload(&t->right), s);
  int be_greater_than = t->key;
  while(t != NULL){
    if(aload(&t->valid) && t->key > be_greater_than)
      return leftmost_and_stack(aload(&t->right), s);
    else { pop(s); t = top(s); }
  }
  return NULL;
}
node* new_range_base(node* b, int lo, int hi, rs* s){
  return new node{... = b,    // Assign fields from b
                  lo = lo, hi = hi, storage = s}; }
treap*
all_in_range(lfcat* t, int lo, int hi, rs* help_s){
  stack* s = new_stack();
  stack* backup_s = new_stack();
  stack* done = new_stack();
  node* b;
  rs* my_s;
 find_first:b = find_base_stack(aload(&t->root),lo,s);
  if(help_s != NULL){
    if(b->type != range || help_s != b->storage){
      return aload(&help_s->result);
    }else my_s = help_s;
  }else if(is_replaceable(b)){
    my_s = new rs;
    node* n = new_range_base(b, lo, hi, my_s);
    if(!try_replace(t, b, n)) goto find_first;
    replace_top(s, n);
  }else if( b->type == range   && b->hi >= hi){
    return all_in_range(t, b->lo, b->hi, b->storage);
  }else{
    help_if_needed(t, b);
    goto find_first;
  }
  while(true){      // Find remaining base nodes
    push(done, b);
    copy_state_to(s, backup_s);
    if(!empty(b->data) && max(b->data) >= hi) break;
  find_next_base_node: b = find_next_base_stack(s);
    if(b == NULL) break;
    else if(aload(&my_s->result) != NOT_SET){
      return aload(&my_s->result);
    }else if(b->type == range && b->storage == my_s){
      continue;
    }else if(is_replaceable(b)){
      node* n = new_range_base(b, lo, hi, my_s);
      if(try_replace(t, b, n)) {
        replace_top(s, n); continue;
      } else {
        copy_state_to(backup_s, s);
        goto find_next_base_node;
      }
    }else{
      help_if_needed(t, b);
      copy_state_to(backup_s, s);
      goto find_next_base_node;
    }
  }
  treap* res = done->stack_array[0]->data;
  for(int i = 1; i < done->size; i++)
    res = treap_join(res, done->stack_array[i]->data);
  if(CAS(&my_s->result,NOT_SET,res) && done->size > 1)
    astore(&my_s->more_than_one_base, true);
  adapt_if_needed(t, done->array[r() % done->size]);
  return aload(&my_s->result);
}
```

```
node* secure_join_left(lfcatree* t, node* b){
  node* n0 = leftmost(aload(&b->parent->right));
  if(!is_replaceable(n0)) return NULL;
  node* m = new node{
    ...  = b,    // Assign fields from b
    type = join_main};
  if(!CAS(&b->parent->left, b, m)) return NULL;
  node* n1 = new node{
    ...       = n0,    // Assign fields from n0
    type      = join_neighbor,
    main_node = m};
  if(!try_replace(t, n0, n1)) goto fail0;
  if(!CAS(&m->parent->join_id, NULL, m))
    goto fail0;
  node* gparent = parent_of(t, m->parent);
  if(gparent == NOT_FOUND ||
     (gparent != NULL &&
      !CAS(&gparent->join_id,NULL,m))) goto fail1;
  m->gparent = gparent;
  m->otherb = aload(&m->parent->right);
  m->neigh1 = n1;
  node* joinedp = m->otherb==n1 ? gparent: n1->parent;
  if(CAS(&m->neigh2, PREPARING,
         new node{... = n1,    // Assign fields from n1
                  type = join_neighbor,
                  parent = joinedp,
                  main_node = m,
                  data = treap_join(m, n1)}))
    return m;
  if(gparent == NULL) goto fail1;
  astore(&gparent->join_id, NULL);
 fail1: astore(&m->parent->join_id, NULL);
 fail0: astore(&m->neigh2, ABORTED);
  return NULL;
}
void complete_join(lfcatree* t, node* m){
  node* n2 = aload(&m->neigh2);
  if(n2 == DONE) return;
  try_replace(t, m->neigh1, n2);
  astore(&m->parent->valid, false);
  node* replacement =
    m->otherb == m->neigh1 ? n2 : m->otherb;
  if (m->gparent == NULL){
    CAS(&t->root, m->parent, replacement);
  }else if(aload(&m->gparent->left) == m->parent){
    CAS(&m->gparent->left, m->parent, replacement);
    CAS(&m->gparent->join_id, m, NULL);
  }else if(aload(&m->gparent->right) == m->parent){
    ...      // Symmetric case
  }
  astore(&m->neigh2, DONE);
}
void low_contention_adaptation(lfcatree* t, node* b){
  if(b->parent == NULL) return;
  if(aload(&b->parent->left) == b){
    node* m = secure_join_left(t, b);
    if (m != NULL) complete_join(t, m);
  }else if (aload(&b->parent->right) == b){
     ...    // Symmetric case
  }
}
void high_contention_adaptation(lfcatree* m, node* b){
  if(less_than_two_items(b->data)) return;
  node* r = new node{
   type  = route,
   key   = split_key(b->data),
   left  = new node{type = normal, parent= r, stat= 0,
                    data = split_left(b->data)},
   right = ...,   // Symmetric case
   valid = true};
  try_replace(m, b, r);
}
```

Figure 5.6: Range query, Split and Join

```
288 node* find_base_node(node* n, int i) {
289   while(n->type == route){ // Traverse the tree of
290     if(i < n->key){        // route nodes
291       n = aload(&n->left);
292     }else{;
293       n = aload(&n->right);
294     }
295   }
296   return n;
297 }
298 node* find_base_stack(node* n, int i, stack* s) {
299   stack_reset(s);
300   while(n->type == route){
301     push(s, n);
302     if(i < n->key){
303       n = aload(&n->left);
304     } else {
305       n = aload(&n->right);
306     }
307   }
308   push(s, n);
309   return n;
310 }
311 node* leftmost_and_stack(node* n, stack* s){
312   while (n->type == route) {
313     push(s, n);
314     n = aload(&n->left);
315   }
316   push(s, n);
317   return n;
318 }
319 node* parent_of(lfcatree* t, node* n){
320   node* prev_node = NULL;
321   node* curr_node = aload(&t->root);
322   while(curr_node != n && curr_node->type == route){
323     prev_node = curr_node;
324     if(n->key < curr_node->key){
325       curr_node = aload(&curr_node->left);
326     } else {
327       curr_node = aload(&curr_node->right);
328     }
329   }
330   if(curr_node->type != route){
331     return NOT_FOUND;
332   }
333   return prev_node;
334 }
```

Figure 5.7: Auxiliary code for the LFCA tree.

5.4 Detailed description of routines

5.4.1 Update operations

Lookup(*m*, *i*): Process p that executes Lookup(m, i) takes as parameter a pointer m to the tree of route nodes and a key i. It first locates a base node whose treap may contain the key i by calling find_base_node() (#136) and then calls treap_lookup() to search for the key i in that treap. It returns TRUE if i is found in the tree, otherwise returns FALSE.

Insert(*m*, *i*): It calls do_update(m, *treap_insert*, i) to add key i to the tree m. It returns TRUE if i is successfully added to the tree, otherwise returns FALSE.

Remove(*m*, *i*): It calls do_update(m, *treap_remove*, i) to delete key i from the tree m. It returns TRUE if i is successfully deleted from the tree, otherwise returns FALSE.

do_update(*m*, *u*, *i*): Process p that executes do_update(m, u, i) carries out the actual work of an Insert() or a Remove() operation. It performs repeated attempts until it succeeds to replace a base node (#109). In each attempt, p first locates the base node whose treap may contain the key i by calling find_base_node() (#110). Then, p constructs a new base node which contains a pointer to the updated treap (#113-118) and tries to replace the old base node with the new one (#119). If the replacement succeeds, p performs an adaptation if that is needed (#120). It then returns TRUE if the update was successful, otherwise it returns FALSE (#121). If the base node is not replaceable (#111) or the replacement fails (#119), then contention is encountered, i.e. a conflicting operation tries to modify the same part of the tree. In that case, p may help the conflicting operation (#125) and starts a new attempt from scratch (#126).

find_base_node(*n*, *i*): Process p that executes find_base_node(n, i) executes a search on the tree of route nodes, by traversing a path starting from n towards the key i. It chooses which direction to go at each route node by comparing the key stored there to i. It returns a pointer to the base node that it ends up.

5.4.2 Range Queries

Query(*m*, *lo*, *hi*, *trav*, *aux*): Process p that executes Query(m, lo, hi, *trav*, *aux*) returns all keys of the tree that fall between lo and hi. It first creates a new immutable treap that contains a superset of keys in the range [lo, hi] by calling all_in_range() (#141) and then calls treap_query() to traverse this treap and get the keys in the range [lo, hi] (#142).

all_in_range(*t*, *lo*, *hi*, *help_s*): Process p that executes all_in_range(*t*, *lo*, *hi*, *help_s*) performs most of the actual work of a Query() operation. Specifically, p locates the first base node which is reached when searching for key lo (#163-#183). Moreover, it locks (by making irreplaceable) all base nodes that may contain keys in the range [*lo*, *hi*] (#184-207). Finally, it joins all of the treaps that these base nodes point to to get a new treap, making the base nodes replaceable again, and returns that treap (#208-214).

We now provide a more detailed technical description of all_in_range. Initially, p locates a first base node whose treap may contain the key *lo* and saves this base node with the path that consists of route nodes and ends up on it in a stack *s* (#168). Process p later uses this stack to traverse the rest of the base nodes in ascending order. If all_in_range() is called with a non-NULL result storage argument *help_s*, this means that p is trying to help some other ongoing Query() operation to complete its work (#169). If the other Query() is already finished (#170) the resulting treap is returned (#171). If not, the helping continues (#172). On the other hand, a NULL *help_s* argument means that p is starting a new range query from scratch. If the first base node is replaceable, p tries to replace it with a new base node of type range_base (#173-176) and if p fails to do so it restarts from scratch, otherwise it updates the stack (#177) and continues.
However, if the first base node is not replaceable p helps the conflicting operation and retries (#180-182), except when the conflicting operation is another Query() whose *hi* parameter is greater than or equal to the *hi* parameter of its own Query() operation (#178) - in that case, p helps that Query() operation and, instead of retrying, p borrows its treap result (#179) because it is guaranteed that this treap result contains all the keys that fall in range [*lo*, *hi*].
Process p continues by locating and trying to replace the remaining base nodes in a while loop (#184-207). It maintains an additional stack *done* where it saves all base nodes whose treaps will be joined to form the treap result. Initially, p pushes the first node to the stack *done* (#185) and saves the stack *s* to a backup stack called *backup_s* (#186). Then, p determines if the search should stop by comparing the maximum key of the current node's treap container with *hi* (#187). Next, p locates the next base node and stops the search if this node does not exist (#188-189). If the node exists and the resulting treap has already formed, the resulting treap is being returned (#190-191). If the node has already been replaced, p proceeds to search for the next node (#192-193). If the node is replaceable, p tries to replace it with a new base node which contains *lo*, *hi* and the active result storage *my_s*. If that replacement is performed successfully, p updates the stack *s* and proceeds to search for the next node (#194-197). On the other hand, if the replacement fails, p restores the stack to the previous state and searches for this node again (#198-200). If the node is not replaceable, p helps the conflicting operation, restores the stack to the previous state and searches for this node again (#202-205).
Finally, p joins all treaps from the replaced base nodes to one resulting treap

(#208-210) and tries to set the treap pointer of the resulting storage to point to the resulting treap (#211). Note that even if this CAS fails, it means that some other process that is helping p performs this Query() on behalf of p and executes this CAS at the same time. The boolean variable *more_than_one_base* is set if p collects the treaps of more than one base nodes (#211-212), an adaptation of a randomly selected base node is performed if needed (#213) and the resulting treap is returned (#214).

find_base_stack(n, i, s): Process p that executes find_base_stack(n, i, s) traverses a path of the tree that consists of route nodes starting from the root of the tree of route nodes n, downwards, until reaching a base node whose treap may contain the key i. It returns that base node and mutates the stack s to contain all nodes on the search path from n to that base node.

find_next_base_stack(s): Process p that executes find_next_base_stack(s) returns the next base node in ascending key order, modifying the stack s to contain the search path to that base node. Recall that s contains all route nodes in the path to the appropriate base node. This stack is used for performing the successor traversal, to execute range queries without using recursion.
Process p pops the topmost node *base* from the stack s which is the node the range query has just processed, as well as its parent t. It then finds the next base node of *base* in the in-order traversal. To do so, there are two cases to consider:

i) This node is the leftmost node in t's right subtree.

ii) This node resides in the right subtree of the ancestor of t with higher key that is first met when popping nodes from s. It is found by calling leftmost_and_stack.

leftmost_and_stack(n, s): Process p that executes leftmost_and_stack(n, s) traverses a path of the tree that consists of route nodes starting from n downwards, following the left child pointers, until reaching a base node. It returns that base node and mutates the stack s to contain all nodes on the search path from n to that base node.

5.4.3 Splits & Joins

high_contention_adaptation(m, b): Process p that executes high_contention_adaptation(m, b) performs a split operation, dividing the treap of base node b in two approximately equal parts. If the treap of b contains two or more items (#278), b is substituted by a new route node (#286) which has two new base nodes as children, each of them containing a treap that holds approximately half of the keys of b's treap.

Join is performed in two phases: In the first phase (*secure_join_left*) the two base nodes that will be joined are marked (Fig. 5.8(b),(c)) as well as the parent and grandparent of the one that is the join_main (Fig. 5.8(d)). Also, the new base node with the resulting treap is created (Fig. 5.8(e)).

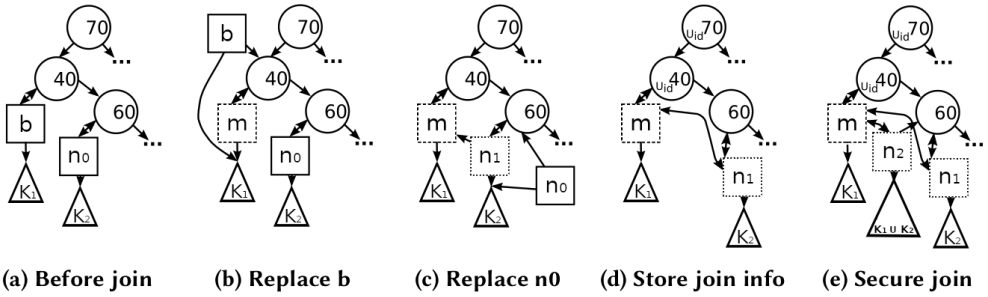


Figure 5.8: First phase of a join.

In the second phase (*complete_join*) the join_neighbor base node is replaced with the new base node that holds the resulting treap (Fig. 5.9(f)) while the join_main node and its parent are removed from the tree (Fig. 5.9(g),(h)).

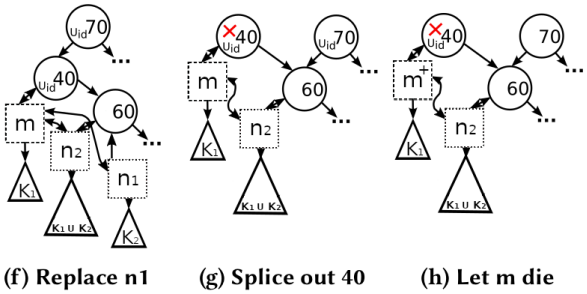


Figure 5.9: Second phase of a join.

A detailed description of the functions executing the join, follows.

low_contention_adaptation(*t*, *b*): Process p that executes low_contention_adaptation(*t*, *b*) performs a join operation, merging the treaps of base node *b* and a neighboring base node into a new treap. If there is no neighboring base node, p returns (#269). Otherwise, in case *b* is the left child of its parent, p calls secure_join_left() and if this returns a non-NULL node, p calls complete_join() to finish the remaining work (#270-272). The case of *b* being the right child of its parent is symmetrical.

secure_join_left(t, b): Process p that executes secure_join_left(t, b) prepares everything needed for a successful join operation. It starts by making the nodes that will be involved in the join irreplaceable. Initially, p locates the neighboring node of b, which is called n0 (#217). It then tries to replace b with a new base node m of type join_main (#219-221). If this succeeds, the node m is irreplaceable (#65-66) and the join operation becomes visible to the other processes, who can abort the join (#76-78) because of the default value PREPARING of m's neigh2 variable (#28) - m and its parent will eventually be removed from the tree if the join completes successfully. Next, p tries to replace n0 with a new base node n1 of type join_neighbor, with its main node set to m (#223-227). In case this replacement attempt fails, the operation is aborted and NULL is returned (#248-249) whereas in case of success, the node n1 is irreplaceable (#67-69) and other processes can still abort the join (#75, and then #76-78). Next, p attempts to set the join_id variable of m's parent and grandparent to m, effectively marking those two route nodes, to ensure that any other conflicting join operations cannot modify them (#228-233). In case of failure the join_id of grandparent is restored to NULL, the operation is aborted and NULL is returned (#247-249). The rest of the information necessary for the join to complete (m's grandparent, sibling and neighbor) is stored in m (#234-236). Next, p determines the parent of the base node that will host the resulting treap depending on whether m and n1 are siblings or not (#237), and tries to replace the PREPARING value of m's neigh2 variable with that base node, which is called n2 (#238-243). If this replacement attempt succeeds, m is returned (#244), and the join operation cannot be aborted anymore - it will be completed by this or other helping processes with a call to complete_join() (#79-81). Otherwise, if this replacement attempt fails, the join_id of parent and grandparent is restored to NULL, the operation is aborted and NULL is returned (#245-249).

complete_join(t, m): Process p that executes complete_join(t, m) executes the remaining steps of a join operation. This function cannot fail or backtrack - failure of any CAS operation means that it is executed simultaneously by another process which is helping p. Initially, p replaces the join_neighbor base node with the new base node that contains the resulting treap of the join (#252-254). It then makes m's parent invalid (#255). This is an indicator used by Query() operations to avoid visiting the treap of an invalid node (#152) because its treap has already been merged into the join_neighbor's treap (#243). Next, p determines the node that will replace m's parent (thus called replacement node) depending on whether m and n2 are siblings or not (#256-257). Finally, p splices m with its parent out of the tree, by switching the appropriate child pointer of its grandparent to point to the replacement node (#258-265) and sets m's neigh2 variable to DONE, indicating that the join completed successfully.

5.4.4 Auxilliary Functions

new_stat(*n*, *info*): Process p that executes new_stat(*n*, *info*) calculates and returns a new value for the statistics variable of base node *n*, taking into account the value of enum *info* that indicates contention. If *info* has the value noinfo, p returns the current statistics value. Otherwise, if *info* has the value contended and the current statistics value is below the upper limit indicated by the constant HIGH_CONT, the calculated value is increased by the constant CONT_CONTRIB and if *n* is a base node of type range it is decreased by the constant RANGE_CONTRIB. If *info* has the value uncontended and the current statistics value is above the lower limit indicated by the constant LOW_CONT, the calculated value is decreased by the constant LOW_CONTRIB and if *n* is a base node of type range it is decreased by the constant RANGE_CONTRIB.

try_replace(*m*, *b*, *new_b*): Process p that executes try_replace(*m*, *b*, *new_b*) performs a CAS operation to replace base node *b* with *new_b*. It returns TRUE if the CAS is successful, or FALSE if the CAS is unsuccessful or *b* is no longer in the tree.

adapt_if_needed(*t*, *b*): Process p that executes adapt_if_needed(*t*, *b*) returns immediately if the base node *b* is not replaceable (#99). Otherwise, if the statistics value of *b* exceeds the upper or is below the lower contention limit, it calls the appropriate adaptation function (#100-103).

is_replaceable(*n*): Returns TRUE if the node *n* is considered to be replaceable, otherwise returns FALSE. There are four cases for *n* to be replaceable:

1) It is of type *normal* (#64). This is the initial state of all base nodes that are not involved in any Query() or join operation.

2) It is of type *join_main* and its neigh2 field is ABORTED (#65-66). This means that the join operation in which it is involved is aborted, thus the node is replaceable again. Note that when its neigh2 field is DONE, it means that this node has been removed from the tree and thus it cannot be replaceable ever again.

3) It is of type *join_neighbor* and its corresponding join_main node's neigh2 field is either ABORTED or DONE (#67-69). This means that the join operation in which it is involved is either aborted or completed successfully, thus the node is replaceable again.

4) It is of type *range* and the result treap pointer of its storage field is set to the resulting treap (#70-71). This means that the Query() operation in which it is involved is completed successfully, thus the node is replaceable again.

help_if_needed(*t*, *n*): Process p that executes help_if_needed(t, n) performs helping, according to the type of operation the node n is involved in. There are three cases, depending on the type of node n:

1) It is of type *join_neighbor*. In that case, the main_node of the join operation is visited and the helping continues with the next case (#75).

2) It is of type *join_main*. In that case, if the neigh2 field of the node is still set to PREPARING, meaning that secure_join_left() has not finished yet, the join operation is aborted (#76-78). Otherwise, if neigh2 is set to the new treap that contains the result of the join, complete_join() is called to help the join operation finish (#79-81).

3) It is of type *range* and the result treap pointer of its storage field is not set to the resulting treap yet (#82-84). In that case, all_in_range() is called to help the Query() operation finish.

5.5 Linearization points

Lookup() is linearized either at the time when the base node it ends up on is still in the tree, or when it visits the parent of a base node which is being spliced out of the tree due to an ongoing join operation. An Insert() or Remove() is linearized when it performs the CAS which changes the appropriate child pointer of the base node's parent to the new base node (#119, which leads to either #56 or #58 or #60). Query() is linearized at the time the result pointer is changed from NOT_SET to the treap that contains the result of the query (#211), after all appropriate base nodes have been replaced. The following table sums up the linearization points of the operations in LFCA tree:

Lookup	**Insert**	**Remove**	**Query**
Base node is reachable	CAS on child pointer of base node's parent	CAS on child pointer of base node's parent	CAS on result treap pointer

Table 5.1: Linearization points of LFCA tree operations.

Chapter 6

The BPNB-BST Algorithm

This chapter provides a description of the Batched Persistent Non-Blocking Binary Search Tree (BPNB-BST), an optimized version of PNB-BST algorithm that utilizes key batching in the leaves. This algorithm appears in [7]. Our implementation used for the experimental analysis of chapter 7 is available at [9].

6.1 Introduction

Our experimental work, presented in chapter 7, shows that the performance of every operation of PNB-BST (and especially the performance of range query operations) can be greatly affected by batching keys in leaf nodes. LFCA tree and KIWI utilize key batching, i.e. keys are stored in arrays, and each array is stored inside a node. In PNB-BST however, every node contains only one key, so no batching takes place. We have developed BPNB-BST, an optimized version of PNB-BST that supports key batching in the leaves. Each leaf contains a sorted immutable array of keys, of size at most m, where m is a parameter called the *batching degree* and determines the maximum size of the array (i.e. the maximum number of keys a leaf node can contain). In BPNB-BST, m is fixed to a constant value.

6.2 Overview

In BPNB-BST the operations Insert(), Delete(), Find() and RangeScan() are performed differently than in PNB-BST. However, the sequence number mechanism, as well as the helping and validation schemes, are exactly the same as in PNB-BST. For that reason, only the functions that differ from PNB-BST are presented in the pseudocode section below.

A successful Insert(k) operation first calls Search() to get a leaf l and its parent p. If l is not full (it does not already contain m keys), Insert(k) creates a new leaf containing all keys stored in l, plus k. Then, it uses CAS to change the appropriate

child pointer of p to point to the new leaf (Fig. 6.1). If l is full, then it is replaced by a subtree of three nodes in a way similar to that in PNB-BST. However, the set of keys stored in l is split to two key sets of about the same size which are stored in the two leaves of the new subtree (Fig. 6.2).

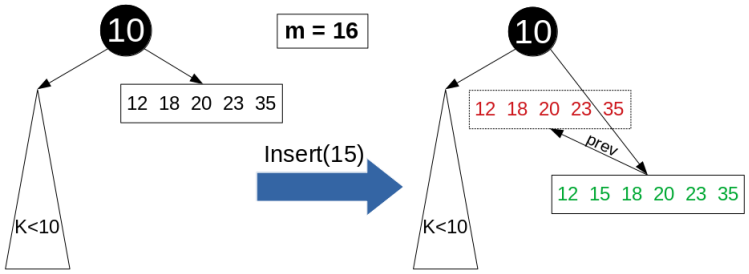


Figure 6.1: Insert() in a non-full leaf.

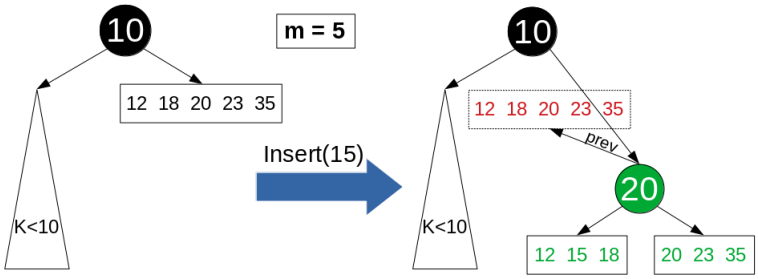


Figure 6.2: Insert() in a full leaf.

Similarly, a successful Delete(k) operation calls Search() to get a leaf l, its parent p and its grandparent gp. If k is not the only key stored in l, Delete(k) creates a new leaf containing all keys stored in l, except k. Then, it uses CAS to change the appropriate child pointer of p to point to the new leaf 6.3. Therefore, in this case, Delete(k) acts in a way similar to Insert(k) in a non-full leaf. Thus, it only flags the node p and marks only node l in this case. Otherwise, flagging and marking occur as in PNB-BST, and the appropriate child pointer of gp is updated to point to a copy of the sibling of l 6.4. Unsuccessful updates are performed as in PNB-BST.

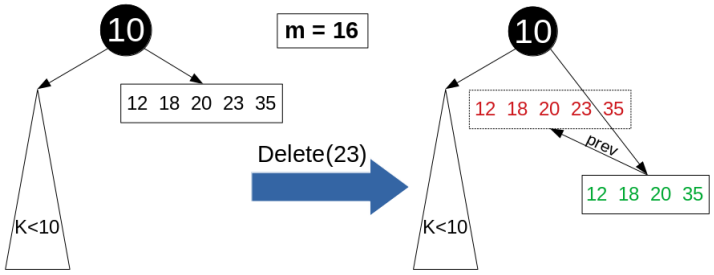


Figure 6.3: Delete() of a non-last key of a leaf.

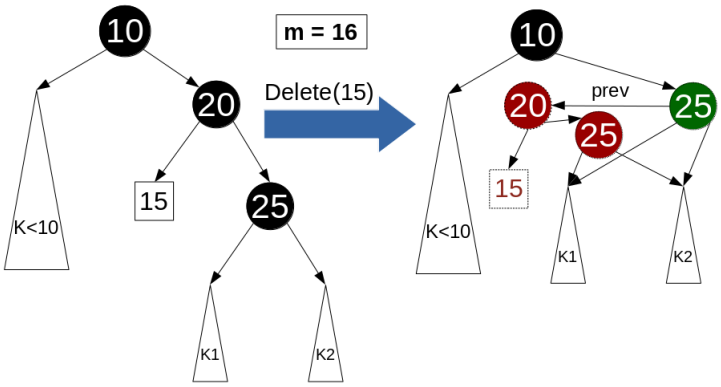


Figure 6.4: Delete() of the last key of a leaf.

Find(k) calls Search() to get a leaf l, and then performs a binary search on the array of keys stored in l. For the array of keys of each leaf it traverses, RangeScan(x,y) locates the smaller and the larger key of the array that falls in range [x,y]. Then, it returns these two keys and all keys between them. Helping is performed by both Find() and RangeScan() in a way similar as in PNB-BST.

6.3 Pseudocode

The pseudocode for BPNB-BST is presented below. New code appears in blue. Recall that the three nodes of the initial subtree in NB-BST contain the keys ∞_1 and ∞_2. The same is true for BPNB-BST except that, for the leaf nodes, instead of keys, arrays of size 1 are used with the only key being ∞_1 and ∞_2, respectively.

```
1 type Leaf {        ▷ The rest of the structs (chapter 4, Fig. 4.4) remain the same.
2     Key ∪ {∞1, ∞2} keys[]        ▷ Array of at most m elements
3     Update update
4     Node * prev
5     int seq
6 }
```

```
 7 Find(Key k): Leaf * {
 8     Internal * gp, p
 9     Leaf * l
10     boolean validated
11     while TRUE {
12         seq = Counter
13         < gp, p, l > = Search(k, seq)
14         < validated, −, − > = ValidateLeaf(gp, p, l, k)
15         if validated then {
16             if LeafContainsKey(l, k) then return l
17             else return ⊥
18         }
19     }
20 }

▷ Performs binary search of key k in the array of leaf l
21 LeafContainsKey(Leaf * l, Key k): Boolean {
22     int i, a = 0, b = ( length of l → keys − 1)
23     while (a ≤ b) {
24         i = (a + b)/2
25         if k == l → keys[i] then
26             return TRUE
27         if k < l → keys[i] then
28             b = i − 1
29         else
30             a = i + 1
31     }
32     return FALSE
33 }

34 Insert(Key k): Boolean {
35     while TRUE {
36         seq = Counter
37         < gp, p, l > = Search(k, seq)
38         < validated, −, pupdate > = ValidateLeaf(gp, p, l, k)
39         if validated then {
40             if LeafContainsKey(l, k) then
41                 return FALSE        ▷ unsuccessful Insert
42             if l → keys[0] == ∞1 then {
43                 ▷ Special case - insert the very first node
44                 newLeaf = new Leaf node
45                 newLeaf →< keys, update, prev, seq >=< [k], < FLAG, Dummy >, ⊥, seq >
46                 newSibling = new Leaf node
47                 newSibling →< keys, update, prev, seq >=< [∞1], < FLAG, Dummy >, ⊥, seq >
48                 newInternal = new Internal node
49                 newInternal →< key, update, left, right, prev, seq >=
50                     < ∞1, < FLAG, Dummy >, newLeaf, newSibling, l, seq >
51             }
52             else if length of l → keys == m then
53                 ▷ l is full - create a new subtree of three nodes
54                 newInternal  = Split(l, k, seq)
55             else    ▷ l is not full - add k to the keys of l
56                 newInternal = Add(l, k, seq)
```

```
57              if Execute([p, l], [pupdate, l → update], [l], p, l, newInternal, seq) then
58                  return TRUE          ▷ successful Insert
59          }
60      }
61  }

62  ScanHelper(Node * node, int seq, int a, int b): Set {
63      Info * infp
64      if node points to a leaf then
65          return node → keys ∩ [a, b]
66      else {
67          infp = node → update.info
68          if infp → state ∈ {⊥, TRY} then Help(infp)
69          if a ≥ node → key then return ScanHelper(ReadChild(node,FALSE,seq),seq, a, b)
70          else if b < node → key then return ScanHelper(ReadChild(node,TRUE,seq),seq, a, b)
71          else return ScanHelper(ReadChild(node,FALSE,seq),seq, a, b) ∪
72                      ScanHelper(ReadChild(node,TRUE,seq),seq, a, b)
73      }
74  }

75  Delete(Key k): Boolean {
76      while TRUE {
77          seq = Counter
78          < gp, p, l > = Search(k, seq)
79          < validated, gpupdate, pupdate > = ValidateLeaf(gp, p, l, k)
80          if validated then {
81              if not LeafContainsKey(l, k) then
82                  return FALSE    ▷ unsuccessful Delete
83              if length of l → keys == 1 then {
84                  ▷ l has only one key - substitute with sibling
85                  sibling = ReadChild(p, l → keys[0] ≥ p → key, seq)
86                  < validated, − > = ValidateLink(p, sibling, l → keys[0] ≥ p → key)
87                  if validated then {
88                      if sibling is Internal then {
89                          newSibling = new Internal node
90                          newSibling →< key, left, right >=
91                              < sibling → key, sibling → left, sibling → right >
92                      }
93                      else {
94                          newSibling = new Leaf node
95                          newSibling → keys = sibling → keys
96                      }
97                      newSibling →< update, prev, seq >=
98                          << FLAG, Dummy >, p, seq >
99                      if sibling is Internal then {
100                         < validated, supdate > = ValidateLink(sibling, newSibling → left, TRUE)
101                         if validated then
102                             < validated, − > = ValidateLink(sibling, newSibling → right, FALSE)
103                     }
104                     else
105                         supdate = sibling → update
106                     if validated and Execute([gp, p, l, sibling], [gpupdate, pupdate, l → update, supdate],
107                         [p, l, sibling], gp, p, newSibling, seq) then
```

```
108                     return TRUE        ▷ successful Delete
109                 }
110             }
111             else {    ▷ l contains more than one key - remove k from l
112                 newLeaf = Subtract(l, k, seq)
113                 if Execute([p, l], [pupdate, l → update], [l], p, l, newLeaf, seq) then
114                     return TRUE        ▷ successful Delete
115             }
116         }
117     }
118 }

    ▷ Returns a new leaf containing all keys stored in l plus k
119 Add(Leaf l, Key k, int seq): Leaf * {
120     int i, a = 0, b = ( length of l → keys − 1)
121     while (a ≤ b) {
122         i = (a + b)/2
123         if k < l → keys[i] then
124             b = i − 1
125         else
126             a = i + 1
127     }
128     ▷ Create and return a leaf that contains the keys of l plus k
129     oldSize = (length of l → keys)
130     newSize = oldSize + 1
131     Leaf * newLeaf = new Leaf node
132     newLeaf → keys[0...a − 1] = l → keys[0...a − 1]
133     newLeaf → keys[a] = k
134     newLeaf → keys[a + 1...newSize − 1] = l → keys[a + 1...oldSize − 1]
135     newLeaf →< update, prev, seq >=
136         << FLAG, Dummy >, l, seq >
137     return newLeaf
138 }

    ▷ Returns a new leaf containing all keys stored in l except k
139 Subtract(Leaf l, Key k, int seq): Leaf * {
140     int i, a = 0, b = ( length of l → keys − 1)
141     while (a ≤ b) {
142         i = (a + b)/2
143         if k < l → keys[i] then
144             b = i − 1
145         else
146             a = i + 1
147     }
148     ▷ Create and return a leaf that contains the keys of l except k
149     oldSize = (length of l → keys)
150     newSize = oldSize − 1
151     Leaf * newLeaf = new Leaf node
152     newLeaf → keys[0...b − 1] = l → keys[0...b − 1]
153     newLeaf → keys[b...newSize − 1] = l → keys[b + 1...oldSize − 1]
154     newLeaf →< update, prev, seq >=
155         << FLAG, Dummy >, l, seq >
156     return newLeaf
```

```
157 }

▷ Splits the array of keys stored in l in two new sets, adding k
158 Split(Leaf l, Key k, int seq): Internal * {
159     ▷ Create a leaf that contains the keys of l plus k
160     Leaf * tempLeaf = Add(l, k, seq)
161     oldSize = (length of tempLeaf → keys)

162     ▷ Create a leaf that contains the left half of tempLeaf keys
163     Leaf * newLeft = new Leaf node
164     newLeft →< keys, update, prev, seq >=
165         < tempLeaf → keys[0...(oldSize/2 − 1)],
166         < FLAG, Dummy >, ⊥, seq >

167     ▷ Create a leaf that contains the right half of tempLeaf keys
168     Leaf * newRight = new Leaf node
169     newRight →< keys, update, prev, seq >=
170         < tempLeaf → keys[oldSize/2...(oldSize − 1)],
171         < FLAG, Dummy >, ⊥, seq >

172     ▷ Create and return the parent of the two new leaves
173     Internal * newInternal = new Internal node
174     newRight →< key, update, left, right, prev, seq >=
175         < newRight → keys[0], < FLAG, Dummy >,
176         newLeft, newRight, l, seq >
177     return newInternal
178 }
```

Chapter 7

Experimental Analysis

This chapter presents our experimental analysis. With this analysis, we illustrate the good performance of BPNB-BST and compare it with the performance of other state-of-the-art algorithms that support range queries, revealing the tradeoffs that exist among different design decisions. Moreover, the experimental analysis provides insight and explanations for the performance differences that the studied algorithms exhibit. Finally, some concluding remarks and lessons learned are discussed.

7.1 Methodology

System setup To run the benchmarks, we used an established methodology and extended the code available at [6]. Benchmarks were performed on two machines. The first machine has two Intel(R) Xeon(R) E5-2630 v3 @ 2.40GHz CPUs with 8 cores each and hyperthreading enabled, yielding a total of 32 hardware threads. The machine is equipped with 256GB of RAM and runs CentOS Linux 7.3.1611 with kernel version 4.4.44. The second machine has four Intel(R) Xeon(R) E5-4610 v3 1.70GHz CPUs with 10 cores each and hyperthreading enabled, yielding a total of 80 hardware threads. The machine is equipped with 256GB of RAM and runs CentOS Linux 7.5.1804 with kernel version 3.10.0-862.14.4.el7.x86_64. All implementations of the algorithms are in Java; OpenJDK 64-Bit Server VM with version 1.8.0_191 was used, with the flags -server and -d64 enabled. Garbage collection was virtually deactivated by enabling the flags -Xms200G and -Xmx200G. This is done to avoid the performance penalty imposed on the algorithms by the garbage collection mechanism.

Operation mixes Suppose there is a first group of threads that executes only Insert operations, and a second group of threads that executes only Delete operations. Due to the producer-consumer problem, if Inserts are faster than Deletes, the tree will be full during the whole experiment. On the other hand, if Deletes

are faster than Inserts, the tree will be empty the whole time. To be able to measure a consistent operation performance, the tree has to first reach its *steady state*, in which its size should be approximately constant during the whole time of the experiment. Therefore, in our experiments, operation *mixes* are used, in which every thread decides at random (using predefined probabilities) which operation to execute each time. More specifically, the group of values i, d, f, rq and r is called an operation *mix*. In every operation mix, i, d, f and rq are the probabilities for a thread to execute an Insert, a Delete, a Find and a range query operation, respectively, and keys are integers drawn uniformly at random from the range $[1, r]$. Before the start of each run, the tree is prefilled with randomly selected keys until reaching its steady state, in which the tree maintains an approximately constant size of $\frac{ri}{i+d}$ keys throughout the whole experiment.

Measuring performance The *throughput* of an implementation is defined as:

$$\text{Throughput} = \frac{\text{Completed Operations}}{\text{Time of completion}}$$

Throughput is the most commonly used performance metric of a concurrent data structure. In our experiments, we let all threads run for a specific time interval and count the total number of operations that are completed by all threads in that time interval. This is done to ensure that the throughput we measure corresponds to the time when all threads are running concurrently in the system. An alternative setting would be to give each thread a specific amount of work to do (i.e., a specific number of operations to execute) and measure the time it takes for all threads to complete the execution of their operations. This could be problematic for the following reason: In some of our experiments we have groups of threads executing a single operation (such as range queries), and thus not all threads use the same operation mix. In such cases, using the "fixed amount of operations" approach, some threads (e.g. those that execute Insert, Delete and Find) may finish their work sooner than others.

To compute each data point of a diagram, 10 runs of the experiment with a duration of 6 seconds each were executed. Because of JIT compilation performed by the JVM, the first five runs were deliberately discarded (they were used solely to warm up the JVM) and the average from the last five runs was used as the data point value. The vertical line that crosses the horizontal axis at 32 or 80 threads in the Throughput vs Number of Threads diagrams indicates the maximum number of hardware threads of the machine on which the experiment was performed.

Tools To compare the space complexity of an algorithm with that of others, its *memory footprint*, i.e. the amount of physical memory that the system has allocated for the benchmark process, is measured over time. To accomplish this, the *ps* tool was used. Specifically, the command "ps h -p $java pid -o rssize" was executed by a script every second, and the result was written to a file. In

this command, $java is the pid of the benchmark process. In contrast with most profilers, ps does not impose any slowdown on the benchmark itself. This was the main reason that this method was chosen.

The performance of an implementation can depend on many factors, including preemption by the operating system, page faults, cache misses, and algorithmic reasons [1]. To avoid operating system preemption, no other user processes were running in the system at the same time with the benchmark. To avoid page faults, the benchmark was run on a machine with sufficient amount of memory for the whole execution of each experiment.

As experiments have shown, one of the most important factors that affects the performance of an implementation is its *cache miss rate*, i.e. the number of cache misses per algorithmic operation (such as Insert, Delete, Find or RangeQuery) the implementation produces. To measure the cache miss rate, the *linux perf* tool was used, with the "cache misses" performance counter. After the end of each experiment, the total number of cache misses reported by perf was divided with the total number of operations reported by the benchmark.

The keysum test The keysum test is a correctness check for concurrent implementations included in the benchmark. It works as follows: Every thread t maintains a thread local integer variable, called *keysum* counter. Every time t performs a successful Insert(k) (or Delete(k)) call, t adds k to (or subtracts k from) this counter. After the end of the experiment, the main thread sums up the counters of all threads, and the result should be equal to the sum of keys that the tree contains at the end of the experiment, which is found by a simple traversal of the tree by the main thread. Note that the keysum check is performed at the end of the execution and the time it requires to execute does not count as execution time of the experiment. Moreover, the addition and subtraction happen on thread local variables and therefore they do not incur any overhead to a thread.

7.2 Experimental Evaluation

In section 7.2.1, we start by presenting a set of diagrams (Fig. 7.1 - 7.5) showing preliminary results that we got at the beginning of our analysis. Based on these diagrams, we extracted useful knowledge which we then used to perform a number of meaningful optimizations on PNB-BST. These optimizations and the impact they have on performance are discussed in section 7.2.2.

We then present a set of experiments to study the impact of batching on LFCA tree (section 7.2.3, Fig. 7.8 and 7.9). Then, we present the performance of BPNB-BST in comparison with other state-of-the-art algorithms which employ batching (Fig. 7.10 and 7.18), and conclude with a comparison of the performance of the algorithms discussed in this thesis (section 7.3).

7.2.1 NB-BST and PNB-BST

As already mentioned, PNB-BST (presented in Chapter 4) builds on top of NB-BST (presented in Chapter 3) providing support for range query operations. Thus, we are interested in measuring the performance overhead of PNB-BST operations in relation to the corresponding ones of NB-BST. We start with an experiment of a mix of all operations (Fig. 7.1).

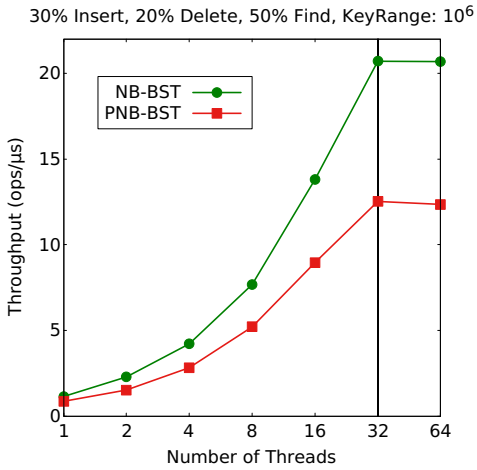


Figure 7.1: Performance of updates in NB-BST and PNB-BST.

In both NB-BST and PNB-BST, the performance of Inserts and Deletes depends on the performance of Finds. This is because every Insert or Delete operation locates at first the three last nodes of a path from the root to a leaf (namely the leaf, its parent and grandparent) before making any changes to the tree. Thus, it is reasonable to first study the performance of Find operations alone.

We perform experiments for both small key ranges (this results in small trees with about 10^4 nodes, that may fit in the cache) and large key ranges resulting in big trees (10^6 nodes) that do not fit in the cache. In this section, we focus on experiments for big trees where the performance differences between algorithms are more emphasized.

The performance difference shown in Fig. 7.2 is quite unexpectable, because the implementations of Find in NB-BST and PNB-BST are similar: They both walk down a path of the tree and return the leaf that contains the key to be found, or null if the key was not found.

We got a better understanding of why this happened by plotting the cache miss rate (Fig. 7.3a) and the memory footprint (Fig. 7.3b) of each implementation. Experiments shown in figures 7.3 - 7.5 illustrate the impact of cache miss rate to performance, and led us to apply optimizations that were necessary to reduce this rate. As we can see, the memory footprint of PNB-BST is similar to that of NB-BST. On the other hand, PNB-BST produces twice as many cache misses per

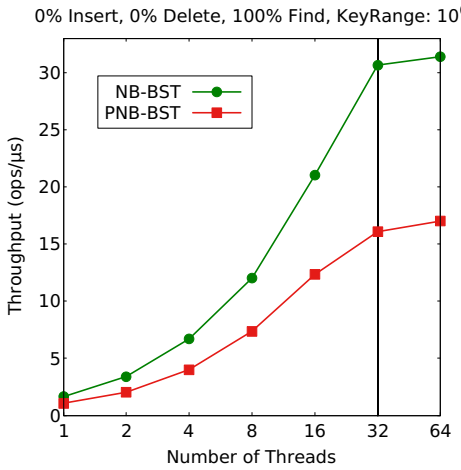
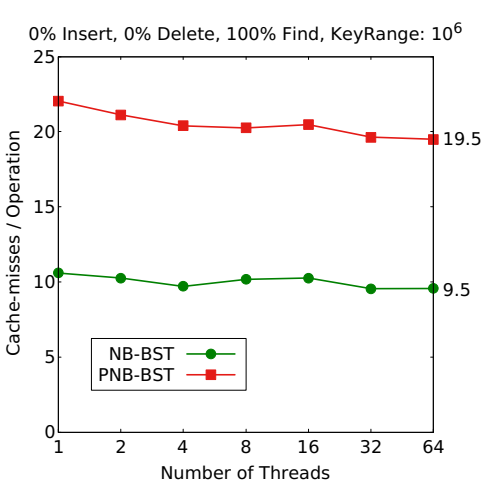
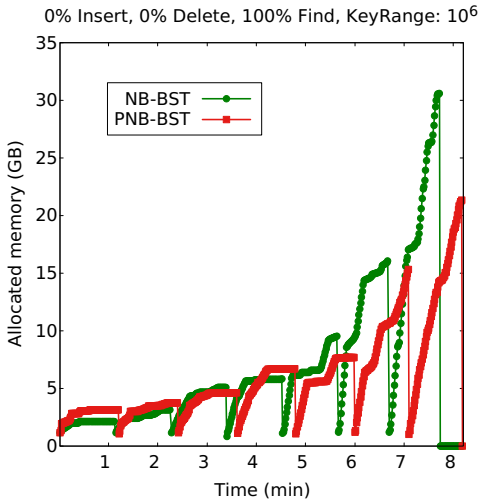


Figure 7.2: Performance of Finds in NB-BST and PNB-BST.

operation than NB-BST. So we focus on how to reduce these cache misses.



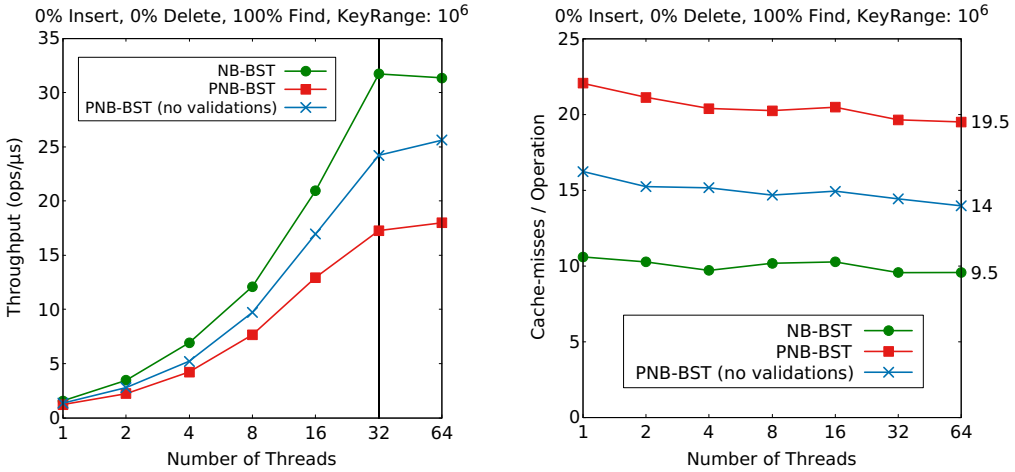
(a) Cache misses of Finds



(b) Memory footprint of Finds

Figure 7.3: Finds in PNB-BST.

To further investigate where the cache misses come from, a Java profiler was used. The results indicated that most of the execution time was devoted to a function that PNB-BST Find() calls, namely ValidateLeaf(), so that function was a source of excessive cache misses. To evaluate this, a special version of PNB-BST was constructed in which this function was not called, without breaking correctness. The results are shown in Fig. 7.4.



(a) Performance of Finds (b) Cache misses of Finds

Figure 7.4: Finds in PNB-BST after the removal of validations.

In a similar manner, we experimented with additional special versions of PNB-BST where we gradually remove lines of code, until we figured out which parts of the code were causing the excessive cache misses (Fig. 7.5). The versions that appear in Fig. 7.5 are the following:

a) The NB-BST implementation.

b) The PNB-BST implementation.

c) The PNB-BST implementation, after the removal of Frozen() function.

d) The implementation of (c), after the removal of ValidateLeaf() function.

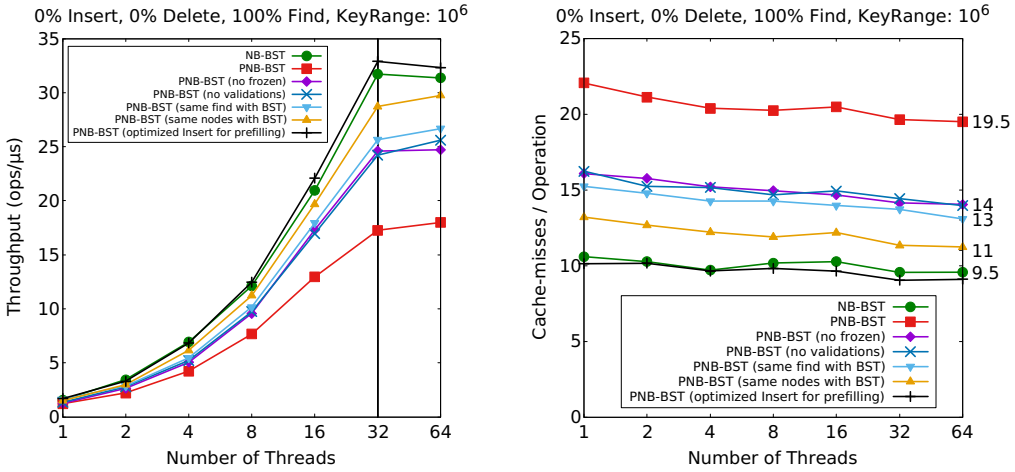
e) The implementation of (d), with the Find() function substituted by the Find() of NB-BST.

f) The implementation of (e), with the node objects substituted by the node objects of NB-BST.

g) The implementation of (f), with the Insert() function made similar to the Insert() of NB-BST.

The diagrams of Fig. 7.5 indicate that the performance penalty of PNB-BST is imposed by the algorithm itself. This is so because versions (c), (d), (e), (f) and (g) were manufactured solely for testing purposes, as they break correctness in the general case. Thus, the slowdown seems to be due to algorithmic reasons.

In Fig. 7.5, the inversely proportional relationship of throughput and cache miss rate becomes apparent. NB-BST Finds produce 9.5 cache misses/operation whereas PNB-BST Finds produce 19.5 cache misses/operation. This 10 cache



(a) Performance of Finds

(b) Cache misses of Finds

Figure 7.5: Finds in different versions of PNB-BST.

misses/operation difference seems to be the cause of the performance drop of PNB-BST compared to NB-BST. It can be analyzed as follows:

- About half of the cache misses are performed when executing the Frozen() function. During a Find(), Frozen() is called two times, accessing both the update struct and the info struct of the update struct each time it is called.
- About 10% of the cache misses come from PNB-BST Find(), excluding the call to ValidateLeaf().
- About 20% of the cache misses come from the fact that PNB-BST nodes are of bigger size than NB-BST nodes. This can be important especially for big trees (with at least 10^6 nodes, as in the current case) because as the nodes get larger, fewer of them can fit in the cache. This is also true for other structs of the algorithm that are frequently accessed, such as the Info structs.
- About 20% of the cache misses come from the Insert() function, which is used to prefill the tree with 10^6 nodes ($\sim 5 \cdot 10^5$ leaves and $\sim 5 \cdot 10^5$ internal nodes) before the actual experiment begins. In version, Insert() performs similarly as the Insert() of NB-BST.

7.2.2 PNB-BST optimizations

Since the performance drop of PNB-BST with respect to NB-BST is attributed to algorithmic reasons, we started investigating possible algorithmic optimizations. In the end, the following were applied to PNB-BST:

1) After helping, the search is restarted from the grand-grandparent of the leaf (ggp) instead of the root node, provided that ggp is not marked. This has also been applied on NB-BST implementation.

2) The variables par, oldChild and the mark[ ] array of the Info struct are removed. The rationale is that these are immutable duplicated values which exist in nodes[ ] array as well, so there is no need to pass them twice. This way the memory footprint will decrease and that may lead to fewer cache misses.

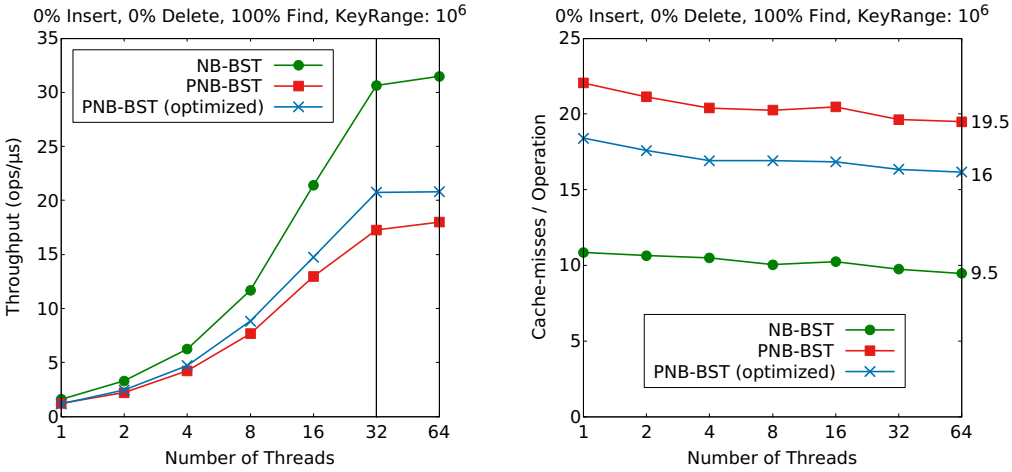
3) The Update field has been replaced by an Info pointer. At the same time the type enum is removed. This is based on the following observations:

   - There is no need for both FLAG and MARK, since everything that is not MARK is considered FLAG. Therefore, only MARK is useful.
   - In nodes[ ] array, the first node is always flagged and the rest are always marked. Therefore, in Fig. 4.6 of chapter 4, line #118 is never executed, and thus lines #116 and #118 are removed.
   - A node is always either flagged or marked *for some Info object*. Therefore, the information of whether a node is flagged or marked can be drawn from the Info object that the node's Info pointer points to. More specifically, if the Info pointer is set to the Dummy Info object or the node is first in the nodes[ ] array, then the node is considered flagged for that Info object. On the other hand, if the node is not placed first in the nodes[ ] array, then it is considered marked for that Info object.

4) As explained in Table 4.3 of Chapter 4, Frozen() returns True whenever a node either needs help (i.e., its Info object's state is NULL or TRY) or is deleted from the tree (i.e., its Info object's state is COMMIT and the node is marked for that Info object). There is no need to help when the node is deleted, so an optimization is applied to lines #53-55 (Fig. 4.5, chapter 4) to avoid helping in that case.

Out of these optimizations, only optimization (3) is directly related to a decrease in cache misses. This is because, after the transformation of Update fields to Info pointers, a double dereference becomes a single dereference. This happens in Frozen(), which is called twice per operation. Thus, after these optimizations, we expect a decrease of at least ~ 2 cache misses/operation, and we also expect the performance difference to decrease accordingly. This is evaluated in the diagrams of Fig. 7.6.

We can see that these optimizations increased the performance of PNB-BST Find() by 15%.



(a) Performance of Finds (b) Cache misses of Finds

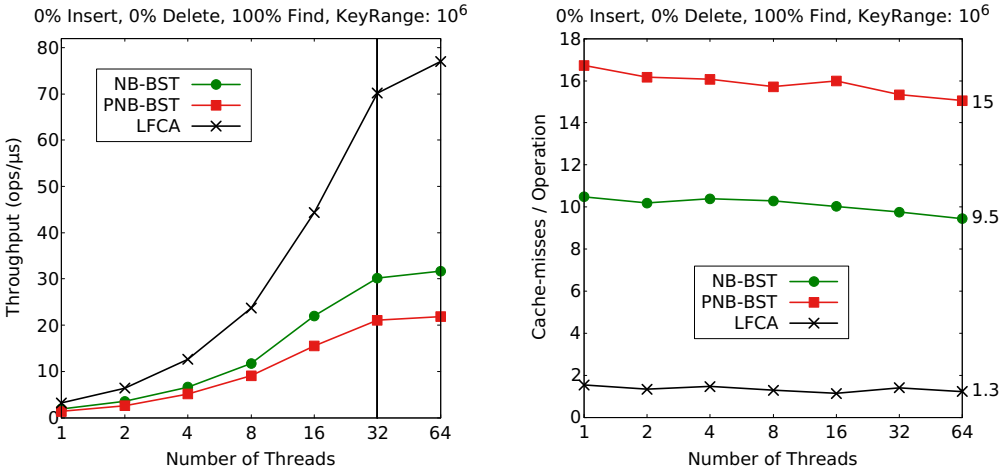
Figure 7.6: Finds in optimized PNB-BST.

7.2.3 LFCA tree and key batching

After optimizing PNB-BST, we now bring LFCA tree into the picture. Following the approach of section 7.2.1, we first examine what happens in a 100% Find() scenario, before adding update operations to the mix.

The result shown in Fig. 7.7a indicates that the performance of the Find() operation of LFCA tree is about 2.5 times better compared to the Find() of NB-BST. This can seem quite strange, because the code of Find() operation of LFCA tree is similar to that of NB-BST. However, LFCA tree uses treaps to store its keys inside them. A closer look on the implementation of these treaps [13] shows that keys are stored sequentially in arrays, and each array is stored inside a leaf of a treap. That is, the LFCA tree utilizes key batching in the leaves of these treaps. The maximum number of keys that each array can store is controlled by a constant. We refer to this constant as the *batching degree* of LFCA tree. The default value of the batching degree of LFCA tree is 64. We repeat the previous experiment, for different versions of the LFCA tree, each one having a different batching degree, which is denoted as an integer besides its name in the diagrams. The results are shown in Fig. 7.8.

Fig. 7.8a shows that, for a batching degree of 2, Find of LFCA tree has lost 43% of its performance. Now it is only 0.3 times faster than that of NB-BST. Thus, key batching seems to play an important role for the performance of Finds. We are interested to see whether the same is true for the performance of range queries. To achieve this, we perform an experiment where half of the threads are updaters (they execute only Insert and Delete operations) and the other half are RangeQuerers (they execute only range query operations). The throughput achieved by RangeQuerers for different range query sizes is shown in Fig. 7.9.



(a) Performance of Finds

(b) Cache misses of Finds

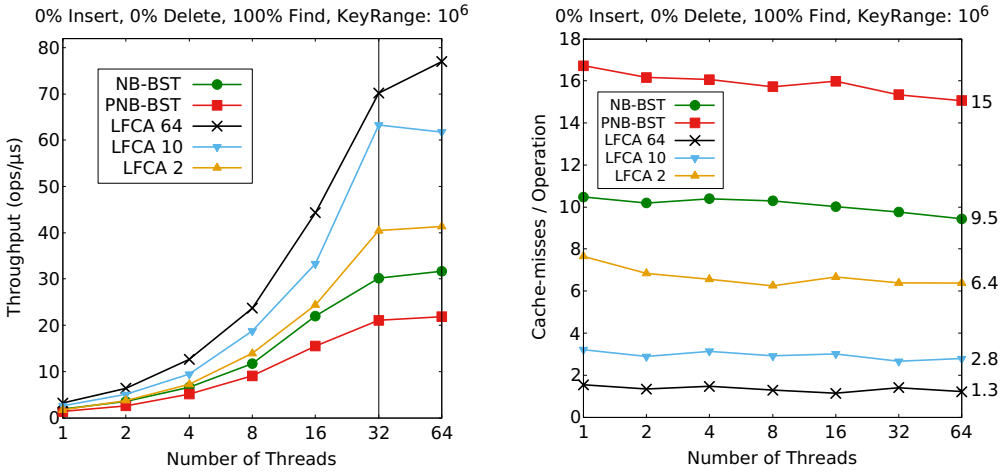
Figure 7.7: Finds in LFCA tree.

In Fig. 7.9 we can see that, with a batching degree of 2, RangeScan of LFCA tree has got an order of magnitude slower. Its performance is now equal to RangeScan of PNB-BST. For LFCA tree and PNB-BST, the performance of range queries decreases as the range query size increases. This is expected, because range queries have to search for a lot more keys when the range query size is large. What is more noteworthy is that, as the batching degree of the LFCA tree approaches that of PNB-BST (which can be thought to have a batching degree of 1), the performance of range queries is always the same in both algorithms, for all range query sizes. Therefore, key batching seems to be a crucial factor for range query performance.

7.2.4 BPNB-BST

As we saw in section 7.2.3, batching is crucial for performance. For this reason, we come up with a batched version of PNB-BST, called BPNB-BST, which stores arrays of keys in the leaf nodes. This way, BPNB-BST achieves a reduced rate of cache misses: an entire array of keys are moved into the cache every time a cache miss occurs to access a leaf. In contrast, when PNB-BST pays a cache miss it brings just one key into the cache. Thus, the cache miss rate of BPNB-BST is lower, and the performance of range queries and Finds is higher. These observations are illustrated in Fig. 7.10. The degree of batching in BPNB-BST is denoted as an integer besides its name in the diagrams. Fig. 7.10a shows the throughput and Fig. 7.10b shows the cache miss rate for the case of Finds.

In Fig. 7.10a we can see that the Finds of BPNB-BST achieve two times better performance than the Finds of NB-BST, and three times better performance than the Finds of PNB-BST. This is attributed to the low cache miss rate of BPNB-BST Finds (Fig. 7.10b), which is due to the employment of batching. Batching has two



(a) Performance of Finds (b) Cache misses of Finds

Figure 7.8: Finds in LFCA tree, for different batching degrees.

effects. First, it reduces the number of levels of the tree, so a path from the root to a leaf node gets shorter. And second, because keys are brought into cache in arrays, some keys that will be searched by future Finds are already in the cache.

The results for range queries are shown in Fig. 7.18b where BPNB-BST is compared to other algorithms as well.

7.3 Performance Comparison

This section analyzes the performance of BPNB-BST and compares it with that of the algorithms that have been discussed (and others).

In the first experiment (Fig. 7.11 and 7.12), the total throughput of operations is plotted against the number of threads, without range queries. We examine the cases of a read-dominated workload (30% updates and 70% Finds) and a write-dominated workload (70% updates and 30% Finds) for big and small trees (Fig. 7.11 and 7.12). We can see that NB-BST is the best performing algorithm in these experiments. This can be explained by the analysis of section 7.2.1: Find, Insert and Delete routines are much simpler and lightweight in NB-BST compared to the other algorithms, which have to synchronize with range queries as well. In Fig. 7.11a and 7.11b, contention is low because the tree size is big. The performance of LFCA tree in these diagrams is similar to that of PNB-BST. This is because, despite the fact that LFCA tree employs batching, which is expected to give a boost in performance, its Insert and Delete routines are heavier than those of PNB-BST because, in case of low contention, treaps get bigger, so an update has to copy longer paths from the root of a treap to a leaf. In these diagrams, batching seems to be an advantage for BPNB-BST and can explain its good performance.

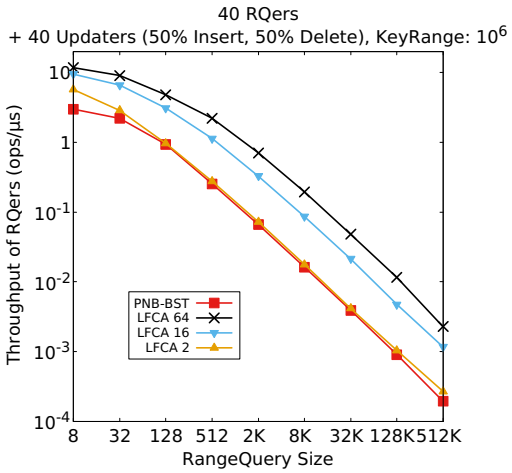
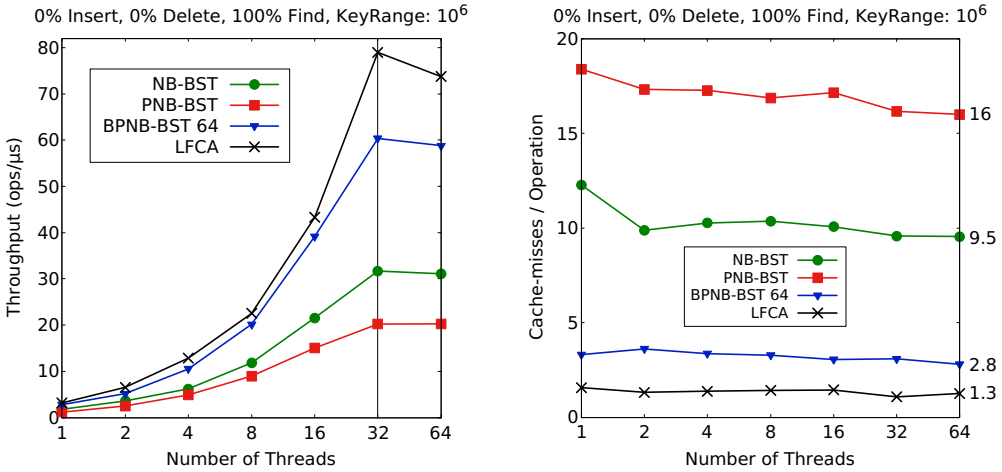


Figure 7.9: Performance of RangeQueries in LFCA tree, for different batching degrees.

In Fig. 7.12a and 7.12b, contention is higher because the tree size is smaller. In that case, Finds and updates in BPNB-BST are forced to help other updates that block their own progress, and this results to a performance penalty for BPNB-BST. BPNB-BST 64 exhibits an additional slowdown compared to BPNB-BST 16 due to its excessive batching, which further increases contention. KIWI exhibits a performance degradation in most cases (especially when contention is high). This can be attributed to its heavy rebalancing mechanism, and the fact that updaters are forced to help the rebalance operation before they can make progress with their own work.

After examining the throughput of threads when no range queries are executed, we now add range queries running in separate cores, to study the performance penalty imposed by range queries on the other operations. Figures 7.13, 7.14 and 7.15 shows the total throughput of threads performing Inserts, Deletes and Finds when an additional set of threads (called RangeQuerers) perform only range queries of size 10^3. For PNB-BST (Fig. 7.13a and 7.13b), it is apparent that range queries do not significantly affect the other operations' performance. This is because, in PNB-BST, the contention between range queries and other operations is minimal: updates and Finds do not help range queries to complete, whereas range queries might spend most of their time traversing older versions of the tree (i.e. deleted nodes), this way avoiding interference with updates. For LFCA tree (Fig. 7.14a and 7.14b), the performance of the other operations is significantly affected by RangeQuerers when the size of the range being scanned (namely the range query size, denoted as RQSize in the diagrams) approaches the size of the tree, as in Fig. 7.14b. This is because, in LFCA tree, updaters are forced to help range queries to complete. Whenever a range query with a large range query size is performed, almost all base nodes of the tree become irreplaceable (i.e., they



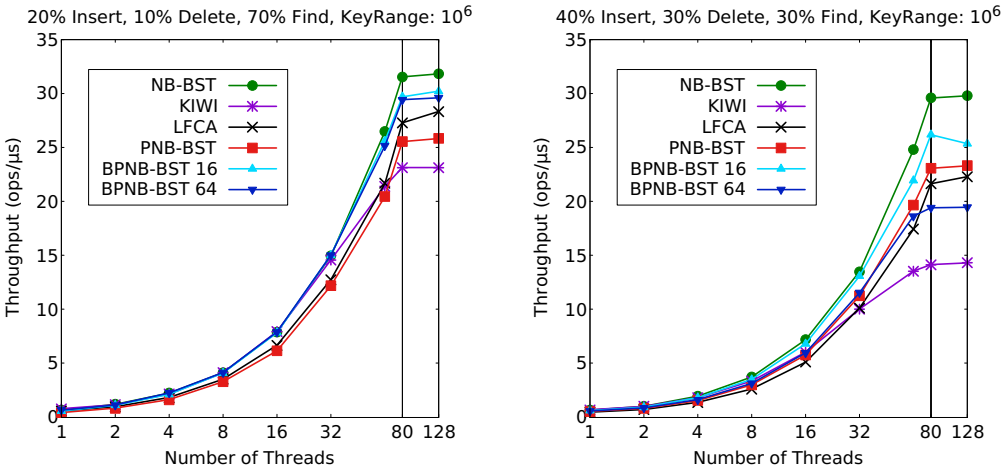
(a) Performance of Finds (b) Cache misses of Finds

Figure 7.10: Finds in BPNB-BST for a batching degree of 64.

become effectively locked). Thus, many updaters have to help that range query to complete. This way, their progress is delayed. For BPNB-BST (Fig. 7.15a and 7.15b), the overhead of RangeQuerers is bigger because, due to batching, the RangeQuerers are faster. When the RangeQuerers are fast, the *Counter* variable that is used for assigning versions to nodes is being rapidly incremented. Thus, handshaking fails more often, and this way updaters cannot easily make progress since they are more prone to restart their operations.

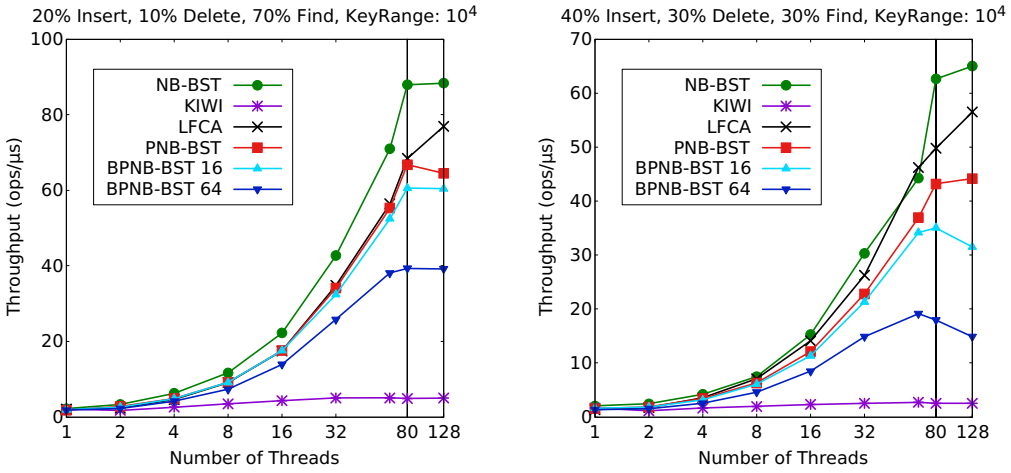
Next, the total throughput of operations is measured when range queries are added to the operation mix (Fig. 7.16 and 7.17). This diagram clearly shows the importance of batching degree: PNB-BST can be thought to have a batching degree of 1, and the algorithms that use batching perform significantly better. Experiments of the same type, with more than 10% range queries (as in Fig. 7.16a and 7.16b), produce similar results. This is because, since range queries are slower than the other operations, threads spend a great fraction of their time performing range queries in such experiments. Figures 7.17a and 7.17b show the performance of the different algorithms for reduced values of this fraction, as many applications spend a smaller portion of their time executing range queries.

In the last experiment (Fig. 7.18), we consider how the size of the range being scanned affects performance. Half of the threads are RangeQuerers and the other half are updaters, which perform Inserts and Deletes. The throughput of the two groups is depicted in separate diagrams. For the throughput of updaters (Fig. 7.18a), LFCA tree performs better for small range query sizes, whereas PNB-BST and BPNB-BST outperform LFCA tree for large range query sizes. This diagram supports the claim that we've made about the diagrams of Fig. 7.13, 7.14 and 7.15: For small range query sizes, the rate of change of *Counter* variable in



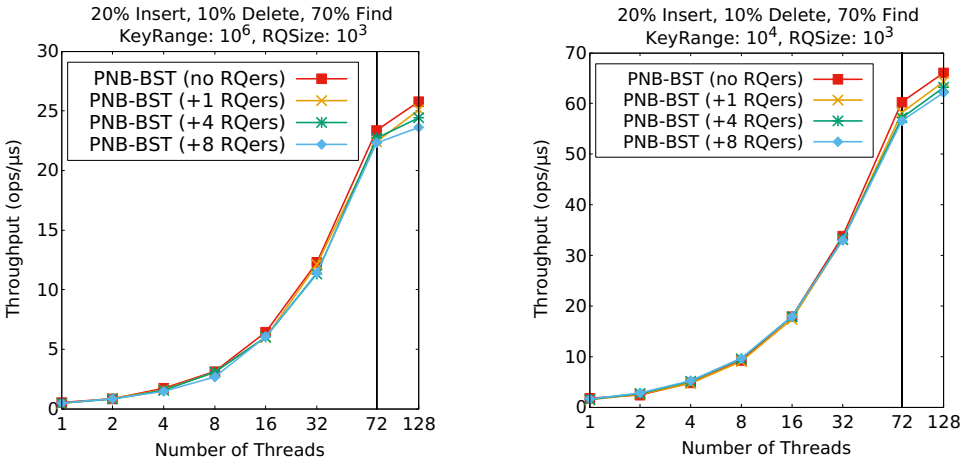
(a) (b)

Figure 7.11: Total throughput without RangeQueries in big trees



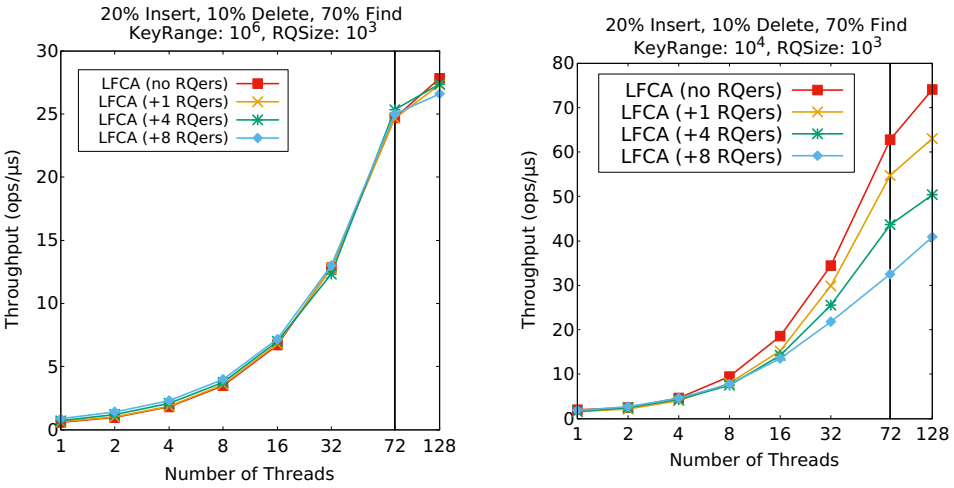
(a) (b)

Figure 7.12: Total throughput without RangeQueries in small trees



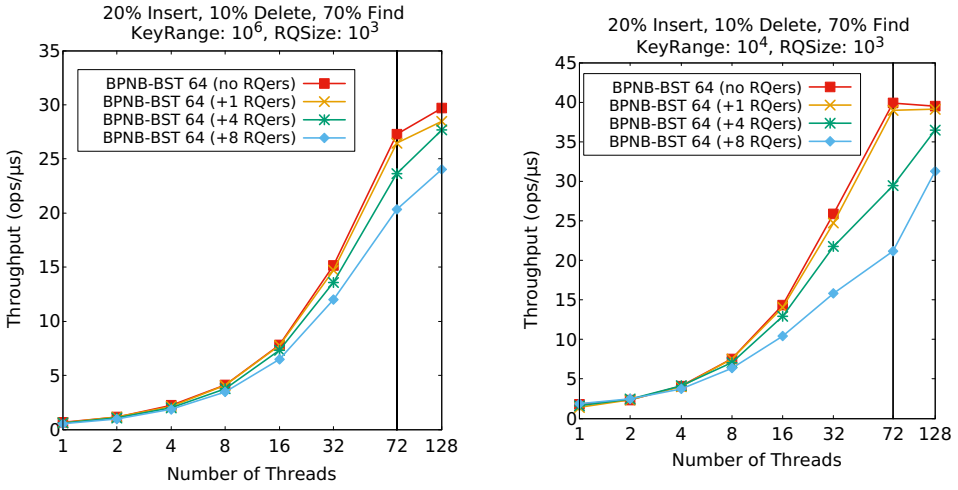
(a) (b)

Figure 7.13: RangeQuery Overhead in PNB-BST



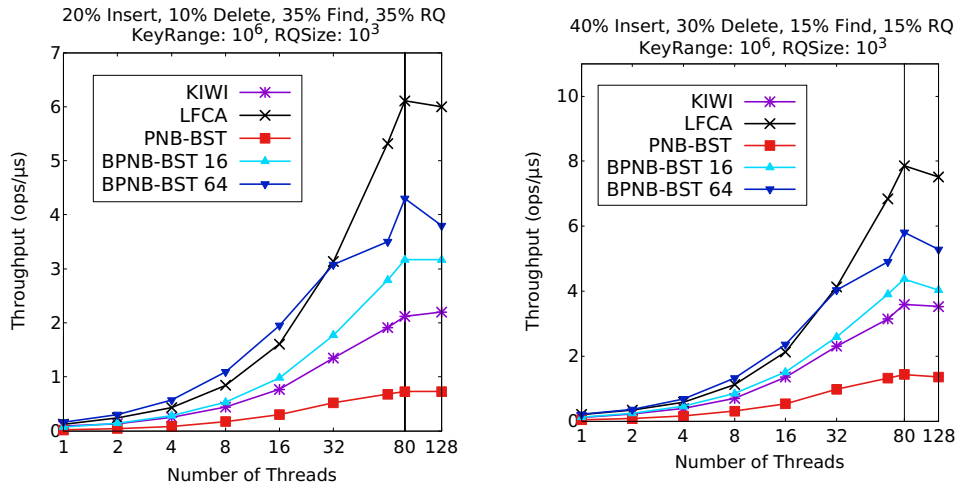
(a) (b)

Figure 7.14: RangeQuery Overhead in LFCA tree



(a) (b)

Figure 7.15: RangeQuery Overhead in BPNB-BST 64



(a) (b)

Figure 7.16: Total throughput with RangeQueries - high RQ percentage

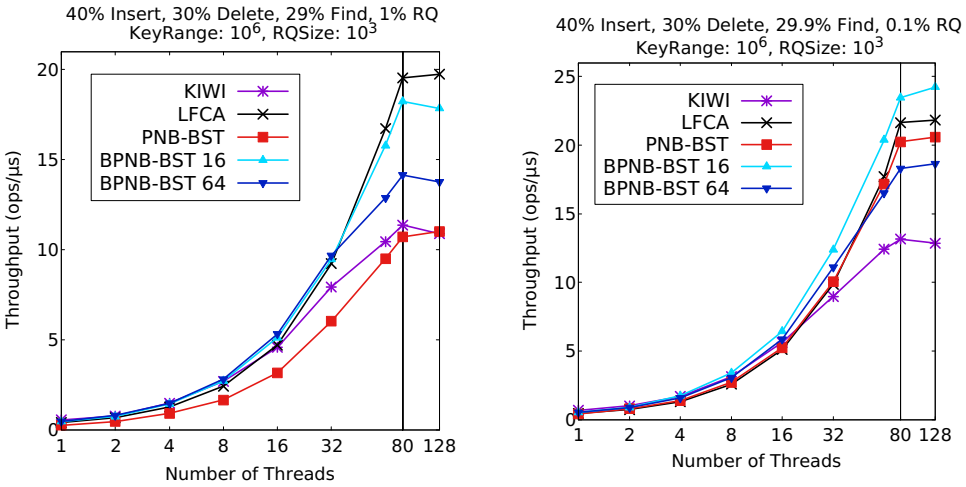
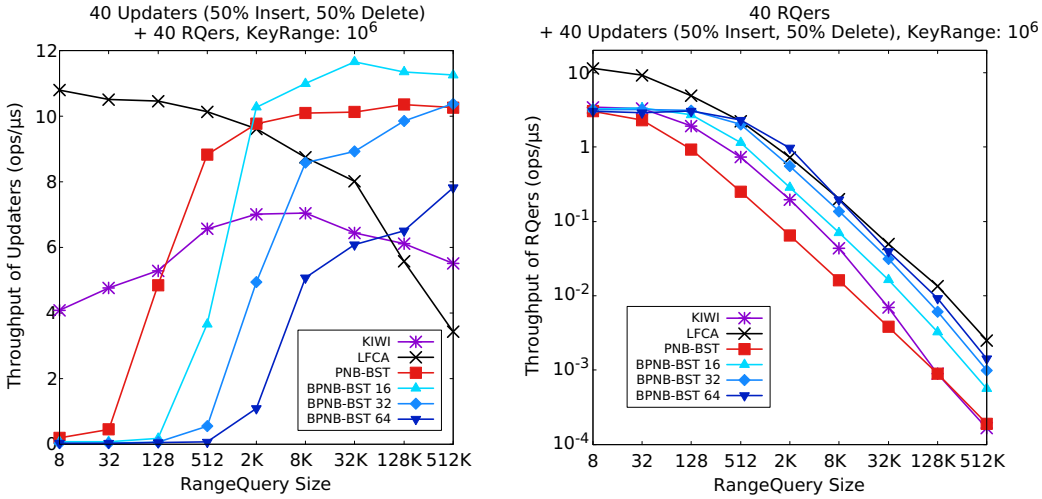


Figure 7.17: Total throughput with RangeQueries - low RQ percentage

PNB-BST and BPNB-BST (which is the synchronization point between updaters and RangeQuerers) is greater because the throughput of range queries is high, as they only have to traverse a small portion of the tree. Thus, handshaking fails more often and updaters have to restart their operation quite frequently, causing their performance to drop. From Fig. 7.18b we can deduce that RangeQuerers perform faster when batching is employed. This is why BPNB-BST 16 is better than BPNB-BST 32, and BPNB-BST 32 is better than BPNB-BST 64 in Fig.7.18a: the faster the RangeQuerers, the more likely the updaters are to restart their operation. On the other hand, LFCA tree cannot keep its high performance for updates when range queries of large range query sizes are executed, because it employs a heavy helping mechanism: many updates are forced to help a large range query to complete, before they make progress themselves. As we can see in Fig. 7.18, updates and range query operations of BPNB-BST 16 scale better with range query size than the corresponding operations of PNB-BST. Furthermore, by selecting a proper value of the batching degree for BPNB-BST, one can trade update performance for range query performance. As expected, Fig. 7.18b shows that, for every algorithm, the throughput of RangeQuerers decreases as the range query size increases, because the range query has to traverse a bigger portion of the tree to complete. Depending on the batching degree, range queries of BPNB-BST can be up to an order of magnitude faster than those of PNB-BST. The performance of RangeQuerers in BPNB-BST 64 and LFCA tree (which also has batching degree 64) is almost identical for all range query sizes. This provides further evidence that the high performance of range queries in LFCA tree (illustrated in Figure 7.16) comes mainly from the key batching.



(a) (b)

Figure 7.18: Throughput of (a) Updaters and (b) RangeQuerers

7.4 Conclusions

The completion of this thesis resulted to the following lessons learned:

- **Two of the most important factors that affect the performance of an implementation are: 1) the contention of threads, and 2) the cache miss rate of operations.** While the cache miss rate has the same meaning for every implementation, contention can have different meanings, depending on the specific algorithmic choices of each implementation. More specifically, the amount of helping every thread has to perform seems to be a source of contention for all presented algorithms. For PNB-BST and BPNB-BST in particular, the global integer variable used for versioning nodes seems to be an additional source of contention. Performance seems to degrade when either contention, or the cache miss rate, or both, increase. Memory footprint does not affect performance in general, although it may affect performance indirectly, by increasing the cache miss rate.

- **Key batching can play a critical role in performance, as it significantly decreases the cache miss rate.** This is true especially for Find and Range Query operations. However, there is the following trade-off: Excessive batching may increase contention, and can thus have a negative effect on performance in some cases.

- **Algorithmic choices can seriously affect performance, since they determine both the contention and the cache miss rate.** The persistence property of PNB-BST and BPNB-BST reduces contention between

updates and large range queries. The same is not true for LFCA tree: Because updates are forced to help range queries, this becomes detrimental to performance as the range query size approaches the tree size. On the other hand, Finds and updates in PNB-BST and BPNB-BST are forced to help other updates that block their own progress. This can cause an additional performance penalty in cases of high contention.

To conclude, we showed that BPNB-BST is an algorithm that performs better than PNB-BST in most cases and compares well with other state-of-the-art implementations. It provides up to an order of magnitude faster range queries than PNB-BST. It also provides updates whose performance scale well with range query size. This is without sacrificing progress for performance, since BPNB-BST maintains the same progress guarantees as PNB-BST.

Bibliography

[1] M. Herlihy, N.Shavit. The Art of Multiprocessor Programming. Morgan Kaufmann Publishers, 2008.

[2] H. Attiya, J. Welch. Distributed Computing: Fundamentals, Simulations and Advanced Topics. Morgan Kaufmann Publishers, 1998.

[3] N. Lynch. Distributed Algorithms. Morgan Kaufmann, 1996.

[4] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking binary search trees. In *Proc. 29th ACM Symposium on Principles of Distributed Computing (PODC 10)*, pages 131-140, 2010.

[5] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking binary search trees. Technical Report CSE-2010-04, York University, 2010.

[6] T. Brown. Software Artifacts. `https://bitbucket.org/trbot86/implementations/src/master/`

[7] P. Fatourou, E. Papavasileiou and E. Ruppert. Persistent non-blocking binary search trees supporting wait-free range queries. In *the 31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '19). ACM, New York, NY, USA*, pages 275-286, `https://doi.org/10.1145/3323165.3323197`, 2019.

[8] P. Fatourou, E. Papavasileiou and E. Ruppert. Persistent non-blocking binary search trees supporting wait-free range queries. `https://arxiv.org/abs/1805.04779`, 2019.

[9] E. Papavasileiou. Implementations of concurrent data structures. `https://github.com/elias-pap/concurrent-data-structures`

[10] K. Winblad, K. Sagonas, and B. Jonsson. Lock-free contention adapting search trees. In *Proceedings of the 30th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '18, New York, NY, USA, 2018.

[11] R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica* 16, 4 (01 Oct. 1996), 464-497. `https://doi.org/10.1007/BF01940876`, 1996.

[12] K. Winblad, K. Sagonas, and B. Jonsson. Lock-free contention adapting search trees (extended version), `http://www.it.uu.se/research/group/languages/software/ca_tree`, 2018.

[13] K. Winblad. JavaRQBench. `https://github.com/kjellwinblad/JavaRQBench`

[14] D. Basin, E. Bortnikov, A. Braginsky, G. Golan-Gueta, E. Hillel, I. Keidar, and M. Sulamy. KiWi: A key-value map for scalable real-time analytics. In *Proc. 22nd ACM Symposium on Principles and Practice of Parallel Programming*, pages 357-369, 2017. `https://github.com/sdimbsn/KiWi`.

[15] T. Brown and H. Avni. Range queries in non-blocking k-ary search trees. In *Proc. 16th International Conference on Principles of Distributed Systems*, pages 31-45, 2012.

[16] A. Prokopec, N.G. Bronson, P. Bagwell and M. Odersky. Concurrent Tries with Efficient Non-blocking Snapshots. In *Proc. 17th ACM Symp. on Principles and Practice of Parallel Programming*, pages 151-160, 2012.

[17] E. Petrank and S. Timnat. Lock-Free Data-Structure Iterators. In *Proc. 27th International Symposium on Distributed Computing*, pages 224-238, 2013.

[18] B. Chatterjee. Lock-free Linearizable 1-Dimensional Range Queries. In *Proc. 18th Intl. Conference on Distributed Computing and Networking*, pages 9:1–9:10, 2017.

[19] M. Arbel-Raviv and T. Brown. Harnessing Epoch-based Reclamation for Efficient Range Queries. In *Proc. 23rd ACM Symposium on Principles and Practice of Parallel Programming*, pages 14-27, 2018.

[20] H. Avni, N. Shavit and A. Suissa. Leaplist: Lessons Learned in Designing TM-supported Range Queries. In *Proc. 2013 ACM Symposium on Principles of Distributed Computing*, pages 299-308, 2013.

[21] N.G. Bronson, J. Casper, H. Chafi and K. Olukotun. A Practical Concurrent Binary Search Tree. In *Proc. 15th ACM Symposium on Principles and Practice of Parallel Programming*, pages 257-268, 2010.

[22] Y. Nikolakopoulos, A. Gidenstam, M. Papatriantafilou and P. Tsigas. Of Concurrent Data Structures and Iterations. *Algorithms, Probability, Networks and Games: Scientific Papers and Essays Dedicated to Paul G. Spirakis on the Occasion of his 60th Birthday.* pages 358-369, 2015.

[23] Y. Nikolakopoulos, A. Gidenstam, M. Papatriantafilou and P. Tsigas. A Consistency Framework for Iteration Operations in Concurrent Data Structures. In *Proc. IEEE International Parallel and Distributed Processing Symposium*, pages 239-248, 2015.

[24] P. Fatourou, Y. Nikolakopoulos and M. Papatriantafilou. Linearizable Wait-Free Iteration Operations in Shared Double-Ended Queues. In *Parallel Processing Letters*, 27(2), pages 1-17, 2017.

[25] N.D. Kallimanis and E. Kanellou. Wait-free Concurrent Graph Objects with Dynamic Traversals. In *Proc. 19th International Conference on Principles of Distributed Systems*, 2015.

[26] A. Spiegelman and I. Keidar. Dynamic Atomic Snapshots. In *Proc. 20th International Conference on Principles of Distributed Systems*, 2016.