B.Comp. Dissertation

# Performance Comparison of Lock-Based and Lock-Free Data Structures in Modern Programming Languages

By

Ryan Chung Yi Sheng

Department of Computer Science

School of Computing

National University of Singapore

2023/24

B.Comp. Dissertation

# Performance Comparison of Lock-Based and Lock-Free Data Structures in Modern Programming Languages

By

Ryan Chung Yi Sheng

Department of Computer Science

School of Computing

National University of Singapore

2023/24

Project No: H201540
Advisor: Senior Lecturer Cristina Carbunaru
Deliverables:
Report: 1 Volume
Source Code: 1 Github Repository

## Abstract

Traditional mutual exclusion mechanisms, vital for ensuring the correctness of concurrent data structures, bring forth challenges such as deadlocks, priority inversion, and non-deterministic access latency. As an alternative, the community is exploring alternative mechanisms that avoid mutual exclusion to mitigate these issues. This study builds upon ongoing research, comparing lock-based and lock-free implementations of data structures, specifically binary search trees (BSTs) and balanced binary search trees (BBSTs), employing C++ as the implementation language. Moreover, this study addresses certain flaws identified in previous work and offers an implementation that rectifies these issues through algorithm redesign. Through comprehensive testing and benchmarking, we assess the performance variations among different concurrent BSTs and BBSTs implementations. Finally, by using the two implementations of lock-free binary search trees as references, we delve into the design guidelines of such lock-free data structures.

Subject Descriptors:
Theory of computation - Design and analysis of algorithms - Concurrent algorithms
General and reference - Cross-computing tools and techniques - Performance

Keywords:
Lock-free; Data structures; Performance; Concurrency; Binary Search Tree; Balanced Binary Search Tree; CAS;

**Acknowledgement**

I am deeply grateful for the support and guidance provided by Cristina Carbunaru throughout the course of this project. Her guidance and advice have been extremely helpful throughout this project. Additionally, I would like to thank Grace for her valuable contributions, which included insightful exchanges of ideas and assistance with C++ and benchmarking-related topics. I would also like to express my gratitude to Lai Mei Tin for his invaluable insights regarding the verification of the rebalancing algorithm.

# Table of Contents

**Title** **i**

**Abstract** **ii**

**Acknowledgement** **iii**

**1 Introduction** **1**

**2 Preliminaries and Related Work** **3**

2.1 Lock-Based Synchronization . . . 3
2.2 Lock-Free Synchronization . . . 4
2.3 Compare-And-Swap . . . 5
2.4 Linearizability . . . 5

**3 Data Structures and Implementation Details** **7**

3.1 Binary Search Trees (BSTs) . . . 7
3.1.1 Coarse-Grained Locking BST . . . 7
3.1.2 Fine-Grained Locking BST . . . 8
3.1.3 Lock-Free BST . . . 11
3.2 Balanced Binary Search Trees (BBSTs) . . . 13
3.2.1 Coarse-Grained Locking BBST . . . 14
3.2.2 Lock-Free BBST . . . 14
3.3 Testing . . . 21
3.4 Benchmarking . . . 23
3.4.1 Initial Balanced State Benchmarks . . . 24
3.4.2 Initial Imbalanced State Benchmarks . . . 24

**4 Discussion and Results** **25**

4.1 Benchmarking Results . . . 25
4.1.1 Results of Initial Balanced State Benchmarks . . . 25
4.1.2 Results of Initial Imbalanced State Benchmarks . . . 27
4.2 Design Guidelines for Intention-Swapping . . . 28

**5 Conclusion** **30**

5.1 Contributions . . . 30
5.2 Future Work . . . 30

**References** **32**

# Chapter 1

# Introduction

In recent years, there has been a growing emphasis on performance and scalability. Driven by this imperative, the research community has prioritized leveraging concurrent programming to improve specific data structures by supporting concurrent operations.

The preservation of data structure invariants amidst the interleaving of operations is important to ensure correctness. The common approach for maintaining these invariants typically involves the application of locking mechanisms, thereby restricting access to critical sections to a single process at any given time. Nevertheless, locking comes with inherent limitations, including the potential for deadlocks, priority inversion, and convoying (Herlihy, Shavit, Luchangco, & Spear, 2020).

Consequently, the pursuit of alternative, non-blocking implementations has gained prominence. Such implementations aim to provide guarantees of wait-freedom and lock-freedom. Wait-freedom assures that every thread continues to make progress, even in the face of arbitrary delays incurred by some threads. Lock-freedom, on the other hand, provides the more modest assurance that at least one thread always makes progress. (Herlihy, Luchangco, & Moir, 2003).

This project builds on prior work by Teh (2022), exploring lock-free implementations of fundamental tree-based data structures like balance binary search trees (BSTs) and balanced binary search trees (BBSTs). We aim to delve into the intricacies of BBSTs and address design challenges that plague lock-free data structures. Three concurrent BST implementations and two BBST implementations, supporting simultaneous read and write operations, will be explored. Lock-based implementations will employ coarse-grained or hand-over-hand locking, while lock-free implementations will use intention-swapping with Compare-And-Swap (CAS) operations.

The discussion will cover testing methodologies, benchmarking, and the design guidelines behind these structures. We will conclude by demonstrating the lock-free BST and BBST's comparable performance to their lock-based counterparts across various workloads through microbenchmarking whilst providing theoretical guarantees.

# Chapter 2

# Preliminaries and Related Work

In this section, we will examine the preliminary concepts necessary for understanding the content of this report. Firstly, we will explore the distinctions between lock-based and lock-free synchronization techniques. Following this, we will then discuss the Compare-And-Swap instruction, which serves as a building block in the realm of lock-free synchronization. Lastly, we will provide a concise overview of the correctness condition essential for our data structures, namely linearizability.

## 2.1 Lock-Based Synchronization

Lock-based data structures fundamentally employ locks or mutexes, to ensure mutual exclusion within designated critical segments. This approach can be classified into distinct categories based on the granularity of the locking used, primarily encompassing coarse-grained locking and fine-grained locking strategies.

Coarse-grained locking is characterized by the application of a singular lock, responsible for safeguarding the entirety of the data structure. Consequently, any operation necessitates the acquisition of this sole lock, which may potentially impact the degree of parallelism due to its inherent contention control mechanism. In contrast, fine-grained locking strategies entail the utilization of multiple locks, designed to protect different components of the concurrent object. In the context of our specific case study, each node contains a mutex, and using hand-over-hand locking, ensures the invariants of the BST hold.

Similar works have been done by Pimpale and Kudtarkar (2015), however, this project provides an implementation that supports generics and uses the concurrency library C++ has to offer. We also provide a proof sketch as to why the hand-over-hand locking method in this case provides the linearizability property.

Lock-based data structures are easier to reason when it comes to correctness, however, they come with a plethora of downsides. The notable ones include:

1. Convoying: When a thread holding a lock is descheduled whilst holding a high-traffic lock, all other processes that require lock L will find lock L busy and will wait for it (Blasgen, Gray, Mitoma, & Price, 1979).

2. Priority inversion: Lower-priority thread is preempted while holding a lock needed by higher-priority threads, hence the higher-priority threads cannot proceed.

3. Deadlock: Possible to occur if threads attempt to lock the same objects in different orders.

## 2.2 Lock-Free Synchronization

The limitations mentioned above underscore the need for lock-free data structures, which offer the advantage of providing a theoretical guarantee that at least one thread will always make progress. However, the challenge lies in the complexity of designing such structures, primarily due to the constraints of synchronization primitives operating on single words, thus imposing intricate structures on these objects (Herlihy et al., 2020).

While several studies have explored lock-free BSTs Ellen, Fatourou, Ruppert, and van Breugel (2010), Howley and Jones (2012), Natarajan and Mittal (2014), lock-free BBSTs are less common due to the intricate design considerations arising from multiple operations lacking a single commit point. This study focuses on the lock-free BST proposed by Natarajan and Mittal (2014) and the lock-free BBST proposed by Singh, Groves, and Potanin (2021). These implementations offer unique architectural approaches, providing alternatives to traditional lock-based BSTs and BBSTs, and will be compared against their lock-based counterparts.

## 2.3 Compare-And-Swap

To understand the lock-free data structures better, we offer an introductory explanation of their fundamental building blocks, the Compare-And-Swap instruction.

Compare-And-Swap is typically in the form of a `lock cmpxchg` instruction, which compares the value in the AL, AX, EAX, or RAX register with the first operand (destination operand). If the two values are equal, the second operand (source operand) is loaded into the destination operand. Otherwise, the destination operand is loaded into the AL, AX, EAX or RAX register. Crucially, by using the `lock` prefix, this instruction is executed atomically, (i.e. executes as a single indivisible unit).

The difficulty of lock-free data structures essentially boils down to reducing the critical section of the code to a single atomic instruction, whilst ensuring the invariants of such data structure holds. In the subsequent discussion, we will delve deeper into why Compare-And-Swap is a fitting choice for the design of such data structures.

## 2.4 Linearizability

Linearizability is a correctness condition defined for concurrent objects. This is also the main correctness condition this study is using when discussing the correctness of the data structures.

We first define an operation to be a pair of an invocation event, $inv(e)$ and its corresponding response event, $res(e)$. An execution of a concurrent system is modelled by a history, which is a finite sequence of operation invocation and response events.

A history $H$ is sequential if the first event of $H$ is an invocation and each invocation, except possibly the last, is immediately followed by a matching response. A history $H$ induces a partial ordering $<_H$ if $e_0 <_H e_1$ if $res(e_0)$ precedes $inv(e_1)$ in $H$.

A history $H$ is linearizable if there exists a legal sequential history $H'$ such that they have the same set of events and $<_H \subseteq <_{H'}$ (Herlihy & Wing, 1990). Equivalently and perhaps more intuitively, linearizability describes the notion of the execution being equivalent to some execution such that each operation happens instantaneously at some linearization point between the invocation and response, formalizing the notion of atomicity, as all operations can be seen as instantaneous.

# Chapter 3

# Data Structures and Implementation Details

We will now delve into the implementation details of various data structures, including 3 BSTs (2 lock-based and 1 lock-free) and 2 BBSTs (1 lock-based and 1 lock-free). Additionally, we will provide concise analyses of the proof of correctness and lock-freedom, along with examinations of specific operations and necessary modifications for certain implementations. Furthermore, our testing and benchmarking methodology will be outlined in this section.

## 3.1 Binary Search Trees (BSTs)

Initially, we analyze three distinct implementations of binary search trees, given their relatively simpler design. This holds particularly true for lock-free BSTs, where each operation typically revolves around a single commit point, specifically a pointer of a given node. However, the design of deletions warrants additional consideration due to its inherent complexity.

### 3.1.1 Coarse-Grained Locking BST

#### Implementation Details

The coarse-grained locking BST goes with a straightforward approach: a single mutex to control access to the entire tree. To mitigate lock contention for specific tree operations, we utilize a

Readers-Writers Lock, implemented with a `std::shared_mutex`.

```cpp
template <class T>
struct CGLBST {
  CGLBSTNode<T>* root;
  std::shared_mutex mut;
}

template <class T>
struct CGLBSTNode {
  T key;
  CGLBSTNode<T> *left, *right;
};
```

Listing 3.1: Definition of Coarse-Grained Locking BST and its node

Since `find` operations are read-only, they only require a `std::shared_lock`. Conversely, `insert` and `remove` operations, being write operations, necessitate a `std::unique_lock`. Our implementation offers a simplistic baseline for comparison with other approaches.

#### Proof of Correctness

The correctness proof for this data structure is relatively straightforward. By setting the linearization points of all operations at the moment when the mutex is released, we ensure linearizability. This approach guarantees that the data structure behaves as expected under concurrent operations.

### 3.1.2 Fine-Grained Locking BST

#### Implementation Details

The lock-based BST implemented here adopts a fine-grained locking mechanism known as hand-over-hand locking. In this strategy, each node is equipped with a mutex to ensure exclusive access when a particular node is involved in an operation.

All three operations, namely `find`, `insert`, and `remove`, follow the same procedure as a sequential BST. However, during traversal, the lock of the next node is acquired before releasing the lock of the current node. For `find` and `insert`, this only requires acquiring the lock of a single node eventually (maximum of 2 nodes simultaneously due to hand-over-hand locking). However, for `remove`, up to 4 nodes may be involved (maximum of 5 nodes simultaneously due to hand-over-hand locking).

```cpp
template <class T>
struct FGLBSTNode {
  T key;
  std::shared_mutex mut;
  FGLBSTNode<T> *left, *right;
  ...
};

void example_traversal() {
  // Acquiring initial locks
  std::unique_lock<std::shared_mutex> lk{root->mut};
  FGLBSTNode<T>* cur = root, *child = root->left;

  while (true) {
    // Hand-over-hand locking, create a unique lock to lock child mutex
    // before moving it to lk so that the previous mutex is unlocked.
    std::unique_lock<std::shared_mutex> childLk{child->mut};

    // Requires 2 locks at this point
    if (child->key == key) ...;
    else if (key < child->key) {
      if (child->left == nullptr) ...;
      cur = child; child = child->left;
      lk = std::move(childLk);
    } else {
      if (child->right == nullptr) ...;
      cur = child; child = child->right;
      lk = std::move(childLk);
    }
  }
}
```

Listing 3.2: Example of hand-over-hand locking with RAII

Finally, we implement the algorithm using 2 sentinel keys, with the root having value $\infty_1$ and the root's left child having the value $\infty_0$, where $\infty_0 < \infty_1$.

### Analysis of the Remove Operation

In this section, we delve deeper into the `remove` operation, as its correctness may not be immediately apparent due to potential interference among concurrent `remove` operations.

When the node to be deleted has 0 or 1 child, the `remove` operation simply replaces the pointer to the node with its successor, which is the non-empty child (possibly none). However, when the node to be deleted has 2 children, selecting a suitable successor node becomes crucial to maintain our invariants. In our current implementation, we search for the rightmost node of the left subtree to serve as the successor.

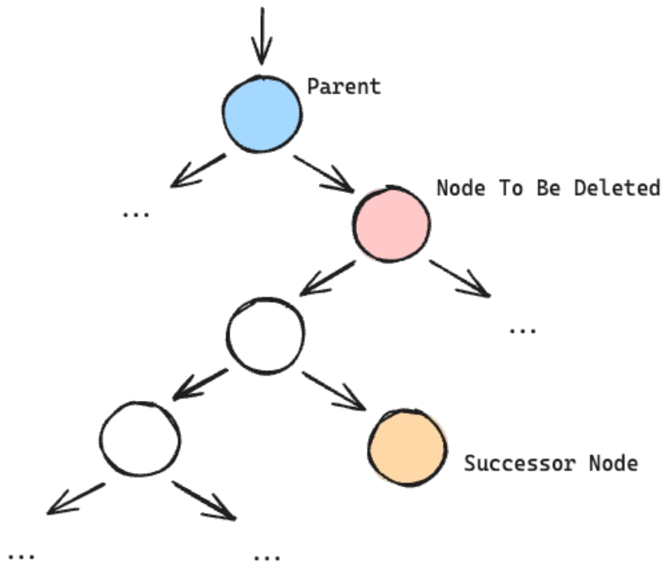


Figure 3.1: Illustration of a sample Fine-Grained Locking BST depicting the remove operation

Figure 3.1 displays a sample tree used to illustrate the `remove` operation. It's important to note that the successor node may be removed from the tree during the `remove` operation's search process. However, a crucial observation is that the left child cannot be removed while the lock for the node to be deleted is held by the current operation. Thus, there always exists a valid successor node. In the scenario depicted above, the `remove` operation would eventually designate the left child of the node to be deleted as its successor.

#### Proof of Correctness

Establishing the correctness proof is more complex. We define the linearization points of operations as the instances where the last required lock is acquired, signifying that all operations seem to occur instantaneously upon acquiring the final required mutex. It can be shown that when operations are arranged according to their linearization points, the resultant sequence of operations is legal.

### 3.1.3 Lock-Free BST

#### Implementation Details

We will offer a concise summary of the lock-free BST implementation, with more comprehensive explanations provided in the work by Natarajan and Mittal (2014). In summary, the BST utilizes tagged pointers, allocating the last 2 bits of a node's address for flags. This implementation is achieved through alignment in C++, and the current code snippet is depicted in Listing 3.3.

```
template <class T, std::size_t ALIGN = std::max(
                        {alignof(T), std::size_t(1 << 2), alignof(void*)})>
struct alignas(ALIGN) Node {
  ...
  T key;
  std::atomic<uintptr_t> left, right;
  ...
};
```

Listing 3.3: Implementation of tagged pointers

This lock-free BST, much like its lock-based counterpart, supports the operations `find`, `insert`, and `remove`, with all three operations relying on the `seek` operation. A crucial aspect to note about this specific tree is that it operates as an external binary search tree

(BST). In this context, the actual values are stored only in the leaf nodes, while the internal nodes function as guides for search operations.

The `seek` operation traverses the tree once, collecting metadata required for other operations, without performing any writes on the tree. The `find` operation utilizes this `seek` operation, returning true only when the leaf node found matches the key, making it wait-free (i.e., every thread continues to make progress as long as they only call `find`).

Both the `insert` and `delete` operations attempt to perform the required writes using the metadata from `seek`. In case of failure, they try to clean up the node required to be deleted before performing another `seek` operation.

### Analysis of the Remove and Cleanup Operations

In this section, we delve deeper into the `remove` and `cleanup` operations, shedding light on their intricacies and offering insights for designing effective guidelines.

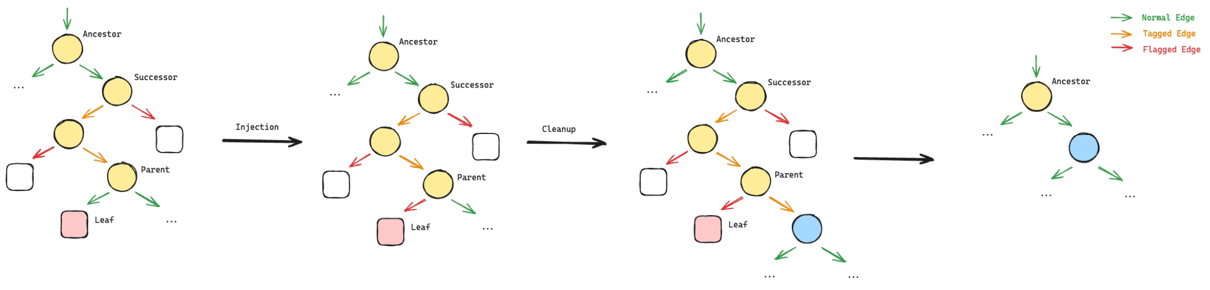


Figure 3.2: Illustration of a sample Lock-Free BST depicting the remove operation

The `deletion` process is divided into two distinct phases: the *injection* phase and the *cleanup* phase. During the *injection* phase, the leaf node is located, and the edge connected to it is marked for deletion. Following this, the *cleanup* phase is initiated, where the sibling node is tagged. Once this edge is marked, any further operations on it are prohibited, ultimately leading to the replacement of the successor, thereby completing the cleanup process.

Several key observations emerge from this discussion, serving as valuable guidelines for designing lock-free data structures:

1. The `cleanup` operation may be invoked by multiple threads to ensure lock-freedom.

Therefore, it is crucial for this operation to be idempotent, ensuring that executing it multiple times for the same set of arguments on a fixed tree structure yields the same result as executing it once. Leveraging Compare-And-Swap ensures that only one of the concurrent `cleanup` operations succeeds.

2. The `remove` operation is split into two phases, possibly to ensure that each phase succeeds only once. This design choice prevents attempts to mark a node for deletion after it has already been successfully flagged (which may violate correctness). Generally, breaking down an operation into multiple phases, each intended to succeed once, proves to be a useful strategy in designing lock-free data structures.

**Proof of Correctness and Lock-Freedom**

The proof of correctness is documented in the proposal by Natarajan and Mittal (2014) and will not be reiterated here.

However, we offer an intuitive argument for the lock-freedom of the data structure, grounded in the following observations:

1. If no `insert` or `delete` operations are successfully completed, every `find` operation will ultimately succeed, as there are no changes to the structure of the tree.
2. If no `insert` or `delete` operations are successfully completed, a `cleanup` operation is triggered, which will eventually succeed for a single thread. Since the tree possesses a finite number of nodes, this process can only repeat a finite number of times, implying that either an `insert` or `delete` operation will eventually reach completion.

Based on these observations, it is evident there will always be progress for a single operation after a finite number of steps.

## 3.2 Balanced Binary Search Trees (BBSTs)

We will now explore in greater detail the analysis of two distinct implementations of binary search trees, placing significant emphasis on the lock-free BST proposed by Singh et al. (2021).

Additionally, we will address the challenging task of balancing trees in a lock-free manner, while also discussing various modifications and their necessity.

### 3.2.1 Coarse-Grained Locking BBST

#### Implementation Details

The coarse-grained locking BBST follows a straightforward approach, employing a single mutex to manage the entire tree. To benchmark its performance, we utilized `std::set`, a standard library construct widely implemented as a red-black tree.

```
template <typename T>
struct CGLBBST {
  std::set<T> tree;
  std::shared_mutex mut;
}
```

Listing 3.4: Definition of Coarse-Grained Locking BBST

Similar to the coarse-grained locking BST, the `find` operations in the coarse-grained locking BBST are read-only, requiring only a `std::shared_lock`. In contrast, `insert` and `remove` operations, being write operations, necessitate a `std::unique_lock`.

#### Proof of Correctness

One could show that the coarse-grained locking BBST obeys linearizability analogous to its BST counterpart and hence is not further elaborated here.

### 3.2.2 Lock-Free BBST

#### Implementation Details

The existing implementation draws inspiration from the pseudocode outlined by Singh et al. (2021), although certain adjustments have been made. This section will provide a comprehensive overview of the implementation of each operation.

The lock-free BBST consists of a root node using a sentinel key, with the root having value $\infty$. The definition of the node for this lock-free BBST is detailed in Listing 3.5. A few important details are highlighted here:

1. The `OperationFlaggedPointer` is a flagged pointer pointing to an `Operation`, which could be an `InsertOp` or a `RotateOp`. We used the last 2 bits of its address to signify what type of operation it is.

2. If the `deleted` flag's last bit is 1, a node is considered as *deleted*. If the `removed` flag is true, the node is marked as *removed*.

3. A node is marked *deleted* when it is no longer logically in a tree. In contrast, a node is marked *removed* when it is prepared for physical removal from the tree. For simplicity, we have omitted the feature of physically removing a node from the tree.

```
template <class T>
struct Node {
  T key;
  std::atomic<Node<T>*> left, right;
  std::atomic<OperationFlaggedPointer> op;
  int local_height, lh, rh;
  std::atomic<uint8_t> deleted;
  std::atomic<bool> removed;
}
```

Listing 3.5: Definition of Lock-Free BBST node

One key aspect of this lock-free BBST is how it differs from a sequential one. In this BBST, a special maintenance thread is responsible for keeping track of the tree's height. It constantly moves through the tree, updating height values, and making adjustments by performing a Compare-And-Swapping a `RotateOp` and `help`ing the operation to ensure the tree remains balanced when needed.

**Details of the Find, Insert and Remove operation**

Next, we dedicate a section to the details of all three operations, as they are notably more intricate compared to the preceding algorithms.

The `find` operation is relatively straightforward compared to the other two. It traverses the tree to locate the desired node. The variable `result` indicates whether a node with the matching key is found in the tree. If `result` is false, or if `result` is true and the node is not marked as *deleted*, the operation returns `result`. Otherwise, it returns true only if there is an `InsertOp` with the same key.

The `seek` operation is slightly different from the lock-free BST implementation. However, it is once again required by both `insert` and `remove`. It essentially traverses the tree in a similar manner as `find`, but it keeps track of the required metadata. If there is an ongoing operation on the resulting node, it attempts to `help` the operation before retrying.

The `insert` operation calls `seek` and checks if the node is already in the tree and not marked as *deleted*. If it is, `insert` returns false as duplicates are disallowed. Conversely, we create an `InsertOp` and perform a Compare-And-Swap into the node's `op` field. If it succeeds, it tries to `help` the operation and returns true. Otherwise, it retries from the `insert` operation.

The `remove` operation also calls `seek`, returning false if no node with a matching key is located. It further examines the node, returning false only if the node with a matching key is marked as *deleted* and does not have an `InsertOp` in its `op` field. In cases where the node operation is not marked as *deleted* and it does not have an ongoing operation, it attempts a Compare-And-Swap on the `delete` field, returning true if successful. All other scenarios necessitate a retry of the operation.

**Analysis of the HelpInsert Operation**

We previously mentioned the notion of `help`ing an operation. Following a successful Compare-And-Swap operation for either an `InsertOp` or `RotateOp`, the `help` operation is invoked to finalize the operation. It's noteworthy that we flag the operation address based on its type.

A crucial characteristic of such an operation is its ability to be invoked by multiple threads to ensure lock-freedom. As previously discussed in Section section 3.1.3, it's imperative that such an operation remains idempotent. Here, we provide a brief overview of the `helpInsert` operation.

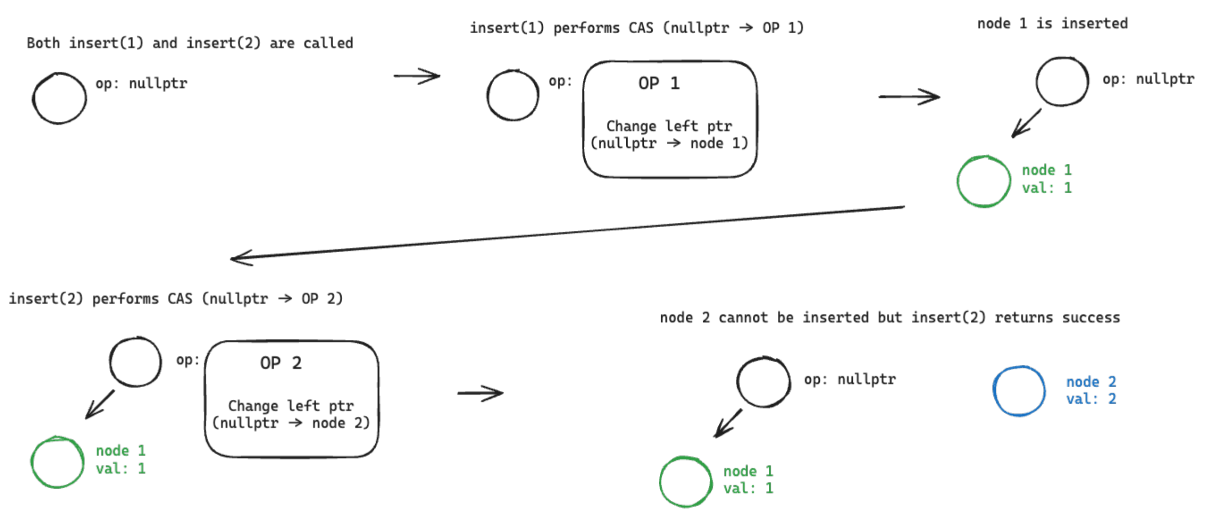
The definition of the `InsertOp` is outlined in Listing 3.6. The boolean flags `isLeft` and `isUpdate` denote whether the new node should be inserted as the left or right child and if the `insert` operation should only involve unmarking the node from *deleted*, respectively.

```
template <typename T>
struct InsertOp {
  bool isLeft;
  bool isUpdate;
  Node<T>* expectedNode;
  Node<T>* newNode;
}
```

Listing 3.6: Definition of InsertOp

The `helpInsert` operation primarily utilizes multiple Compare-And-Swap operations, either on the `delete` field or the children pointers of the node, followed by a Compare-And-Swap on the `op` field to signify completion. Notably, nearly all atomic operations employ Compare-And-Swap to ensure idempotency. Another crucial implementation detail is that we never Compare-And-Swap a `nullptr` for the `op` field to indicate completion; instead, we utilize the unflagged address of the previous `op`. Failure to do so could lead to an ABA problem, as illustrated in Figure 3.3.

This occurrence stems from the inability of `insert(2)` to ascertain whether an operation has completed in between its `seek` operation and and its execution of the Compare-And-Swap operation. However, the utilization of the unflagged pointer to the previous operation effectively mitigates this issue. By avoiding garbage collecting the `Operation`s, only one successful Compare-And-Swap is permitted for a given address. Although there exist methods to conduct garbage collection while avoiding the ABA problem, elaboration on such techniques is beyond the scope of this project and thus will not be expounded upon here.



Both insert(1) and insert(2) are called
op: nullptr
insert(1) performs CAS (nullptr → OP 1)
op:
OP 1
Change left ptr (nullptr → node 1)
node 1 is inserted
op: nullptr
node 1 val: 1
insert(2) performs CAS (nullptr → OP 2)
op:
OP 2
Change left ptr (nullptr → node 2)
node 1 val: 1
node 2 cannot be inserted but insert(2) returns success
op: nullptr
node 1 val: 1
node 2 val: 2

Figure 3.3: Illustration of the ABA problem using nullptr

### Analysis of the HelpRotate operation

In this section, significant modifications to the algorithm will be discussed. We will clarify the functionality of the original algorithm and outline the reasons necessitating changes in our implementation. The complete algorithm can be accessed here.

The `RotateOp` can only be Compare-And-Swap into a node's `op`'s field by the maintenance thread. Once a `RotateOp` is successfully Compare-And-Swapped, the `helpRotate` operation will be called.

In the original algorithm, we keep track of the initial state the `RotateOp` is in, as other threads might also call `helpRotate`. We then try to grab the appropriate the participating nodes and retrying if we failed. Here is where we encounter our first issue:

```
if GETFLAG(node->op) == ROTATE then
nodegrabbed:
    if GETFLAG(child->op) == NONE || GETFLAG(child->op) == ROTATE then
        if GETFLAG(child->op) == NONE then
            CAS(&(op->rotate_op.child->op), child->op, FLAG(op,ROTATE))
        if GETFLAG(child->op) == ROTATE then
            CAS(&(op->rotate_op.state), UNDECIDED, GRABBED)
            seen_state = GRABBED
        else goto nodegrabbed
```

Listing 3.7: Original Pseudocode of lines 7-16 of helpRotate

We present the aforementioned snippet of pseudocode from the original paper for clarity, highlighting the conditional check on `child->op` to ascertain its state before initiating the Compare-And-Swap. This introduces a dilemma:

1. If we constantly load `child->op`'s value to attempt Compare-And-Swap, there's a risk that another thread may have already completed the `helpRotate`, resulting in a Compare-And-Swap that mistakenly swaps in the `RotateOp` despite the operation being completed.

2. If we do not load `child->op`'s value to attempt a Compare-And-Swap (i.e., load once and use that value), a different `Operation` may be Compare-And-Swapped in and completed, causing the current Compare-And-Swap to fail repeatedly.

The solution to this predicament lies in finding a middle ground: we continuously load the value, but upon loading it, we check if the current state in `RotateOp` still pertains to grabbing the child node. This approach works because:

1. If the state has changed, we forgo the Compare-And-Swap operation, as it implies that another thread has already executed it.

2. If the state remains unchanged, we can safely conclude that the loaded value for `child->op` predates the completion of `RotateOp`, ensuring that the Compare-And-Swap will only succeed if the operation has not yet been completed.

After grabbing the node, we will perform multiple Compare-And-Swaps to perform correct pointer changes. Figure 3.4 serves to illustrate the pointer changes.

However, given that `helpRotate` can be invoked by multiple threads, it is imperative to ensure the correct handling of both the copy of the node and the Compare-And-Swap operations.

The initial proposal neglects to store the pointer to the grandchild (i.e., the pointer to node L in the diagram) anywhere. Hence we can only assume that a load will suffice when the value is required. This approach presents a problem, as the child node may point to a different

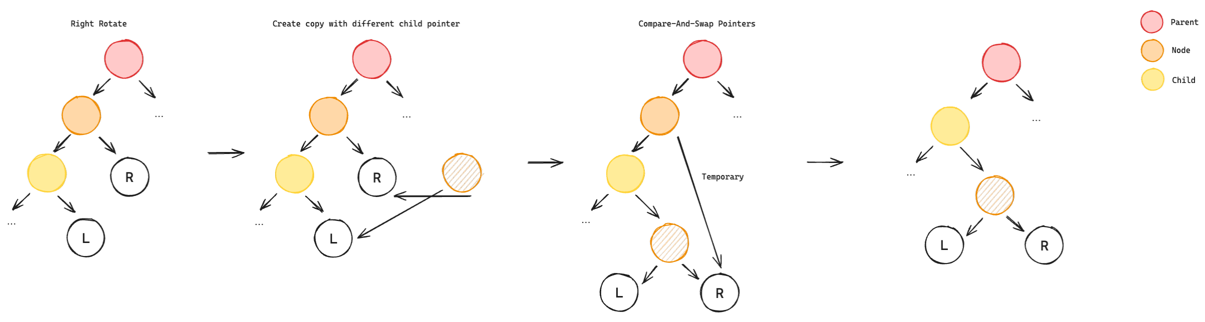

Right Rotate
Create copy with different child pointer
Compare-And-Swap Pointers
Temporary
R
L
Parent
Node
Child

Figure 3.4: Illustration of Right Rotate

grandchild during the rotation process, as depicted in Figure 3.5. Although it may be tempting to apply our previous fix—conducting a conditional check to verify if the state has progressed beyond this stage—this approach is flawed. The current issue may still occur in the interim between the successful Compare-And-Swap of the grandchild and the subsequent change in state.

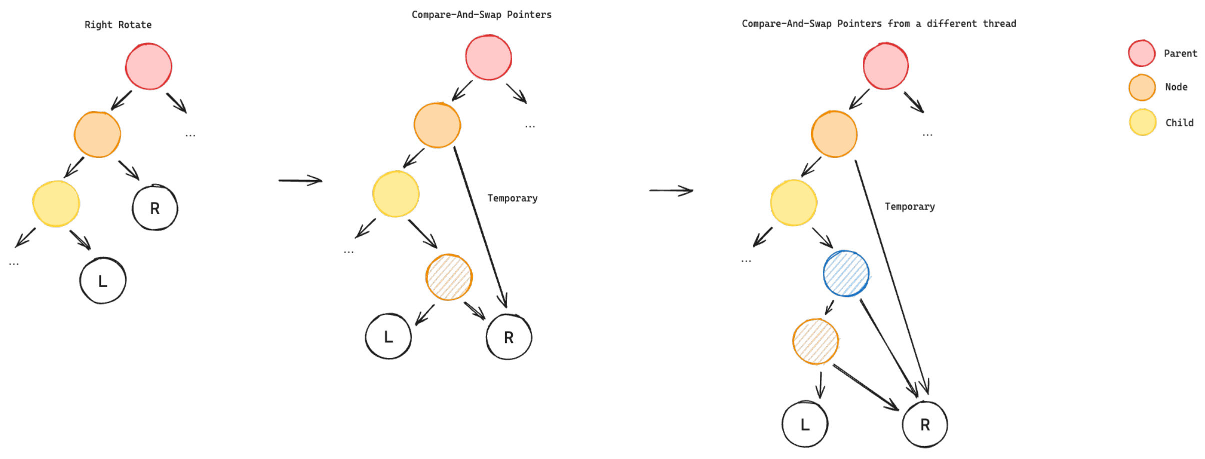

Right Rotate
Compare-And-Swap Pointers
Compare-And-Swap Pointers from a different thread
Temporary
R
L
Parent
Node
Child

Figure 3.5: Illustration of Issue with Right Rotate

The fix circumvented this issue in the previous case by allowing us to detect whether the current `Operation` is between the Compare-And-Swap of the grandchild and the state change. This detection relied on the `Operation` in the `op` field being flagged with a `ROTATE` flag (as `op` now points to a `RotateOp`), whereas no such guarantee exists for the grandchild. However, the remedy is straightforward: we simply store the address of the grandchild in the RotateOp just before transitioning from `GRABBED_FIRST` to `GRABBED_SECOND`, as demonstrated here.

Now, we can guarantee that the pointer to the grandchild is successfully updated only once by employing the address stored in our `RotateOp` as the expected value during the Compare-And-Swap operation.

The final issue to be addressed pertains to the copying of the node. This presents a challenge due to the `remove` operation. There's a possibility for a node to be marked as *deleted* after the copy process, which could potentially lead to the new node not being marked as *deleted*. To mitigate this issue, we perform a bitwise OR operation on the `deleted` field value before the copy, ensuring any Compare-And-Swap from `remove` will fail. This guarantees that the copy process accurately copies the `deleted` field, as it remains unchanged once the bitwise OR operation is executed.

#### Proof of Correctness and Lock-Freedom

The proof of correctness and lock-freedom is outlined in the proposal by Singh et al. (2021) and will not be reiterated here.

Nevertheless, we will delve into some potential concerns regarding our modification. The current implementation of `helpRotate` solely aims to maintain the invariant mentioned in the original paper, ensuring that any operation remains oblivious by node movements due to rotations. Hence, there is no need to re-evaluate the proof of correctness.

However, a notable concern arises when scrutinizing the proof of lock-freedom. Specifically, does the bitwise OR operation on the `deleted` field jeopardize the lock-free property of the tree? The answer is no. In the event of a failure in the `remove` operation, it will retry, and the `seek` operation will either ascertain that the operation has been completed or attempt to `help` the operation, thereby guaranteeing eventual completion.

## 3.3 Testing

Testing the two data structures is a non-trivial undertaking, as it is imperative not only to rely on the theoretical proof of correctness but also to rigorously validate the practical implementation.

To assess the correctness and functionality of these data structures, we have devised a series of sanity checks. These checks ensure the correctness of the data structure when operations are executed sequentially. The 3 primary sanity checks are as follows:

1. Insertion Test:

   This test starts by validating that the initial state of the tree is empty. It then proceeds to insert 1000 elements. Subsequently, it verifies whether all 1000 elements have been successfully inserted into the tree.

2. Deletion Test:

   The deletion tests test the data structures under 2 different scenarios to comprehensively evaluate the data structures.

   The first scenario assesses deletion in an environment where the tree is highly imbalanced. For the lock-based BST, this translates to all nodes initially having either 0/1 child.

   The second scenario focuses on deletion from a BST that is balanced, where, in the case of the lock-based BST, nearly all nodes initially have two children. In both scenarios, the test proceeds by sequentially deleting elements from the tree. Each `remove` operation confirms that all deleted nodes are successfully removed from the tree, while the nodes that are not intended for deletion remain intact within the structure.

3. Linearizability Test:

   The linearizability tests essentially tests whether the linearizability property holds for the data structure.

   This test starts by performing concurrent insertions and deletions. If an `insert` and `remove` operates on the same key concurrently, `remove` can return either true or false. More importantly, if `remove` returns false, the corresponding key must still be in the tree, and the opposite is true if it returns true.

   We place a barrier after this concurrent operations to ensure every operation is completed. Then using the observation above, we check whether the key is still in the tree depending on what is returned by `remove`.

A series of tests is also written in order to ensure the data structure works correctly under a concurrent setting. As such, the tests are written in such a way that it attempts to check for certain race conditions. The tests were each run with 10 iterations, with ThreadSanitizer to ensure no unwanted races were happening. The 3 race conditions the tests attempt to check for are:

1. Insertion-Insertion Race:

   The tests create 10 threads and each thread performs 1000 insertions. The tree is then sequentially checked to make sure all 10000 elements are in the tree. This is to check if an insertion of an element could potentially overwrite another insertion.

2. Deletion-Deletion Race:

   The test first creates a BST that is balanced with 25600 elements. 50 threads are then created and each thread performs 400 deletions on non-overlapping elements. The tree is then checked to make sure the correct elements are removed.

3. Insertion-Deletion Race:

   The test first creates a BST that is balanced with 16384 elements. 128 threads are then created, half of the threads perform 256 deletions on non-overlapping elements, whilst the other half performs 500 insertions. The tree is then checked to make sure the correct elements are removed and inserted.

## 3.4 Benchmarking

To ensure consistency with previous work by Teh (2022), we opted to conduct benchmarking using Google's MultiThreaded Benchmarks. The benchmarks were executed on a system equipped with 2 Intel Xeon Gold 6230 processors (2.1GHz) and varying thread counts (2, 4, 8, 16, 32) to allow each thread to run on a physical core. Each benchmark involved creating a BST with 32768 elements, with further details provided in each subsection.

### 3.4.1 Initial Balanced State Benchmarks

In this benchmark, the tree is initially configured as a perfectly balanced binary search tree. We conducted three distinct benchmarks with different data structures:

1. After setup, each thread performed find operations. Each thread was tasked with checking $\frac{524288}{N_{threads}}$ elements, some of which may or may not exist in the tree.

2. Write Intensive Benchmark: Following setup, the elements and their insertion order were fixed for every thread ($\frac{524288}{N_{threads}}$ elements) to maintain tree balance. Randomly generated integers were not used to prevent significant height variations. Each thread was then tasked with inserting and removing elements in every loop.

3. Read Write Benchmark: The setup mirrored the previous benchmark, with the addition of two find operations between insertion and deletion operations. This benchmark assessed the performance of the binary search tree under a balanced 50% read and 50% write workload.

### 3.4.2 Initial Imbalanced State Benchmarks

For this benchmark, the tree was intentionally initialized to the worst possible imbalanced state based on insertion order (sequential BBSTs remained unaffected). We conducted the same benchmarks as in the previous section. However, the fine-grained locking BST was excluded due to its considerably slower performance, making meaningful benchmarking impractical. This benchmark aimed to quantify the performance difference between a BST and BBST, particularly under different initial conditions.

# Chapter 4

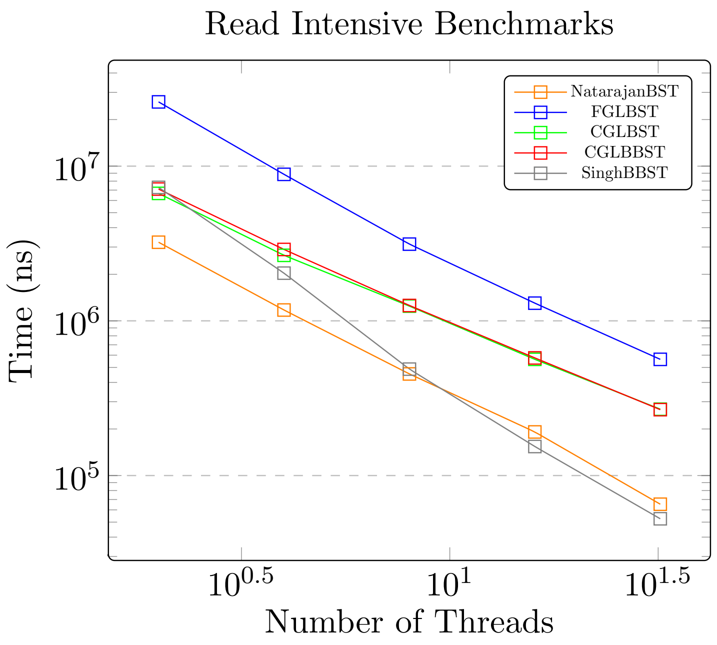
# Discussion and Results

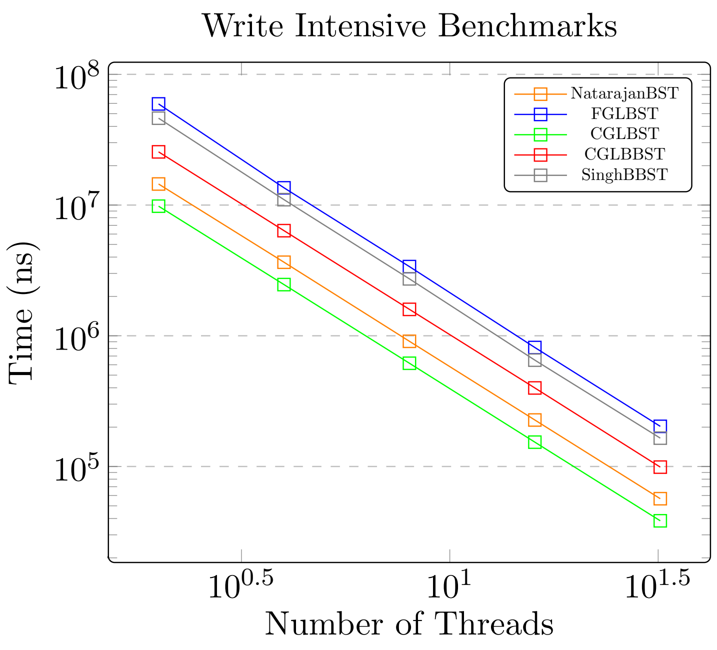
In this section, we will present our benchmarking results. Furthermore, we will explore design guidelines for lock-free data structures, focusing on the intention-swapping technique.

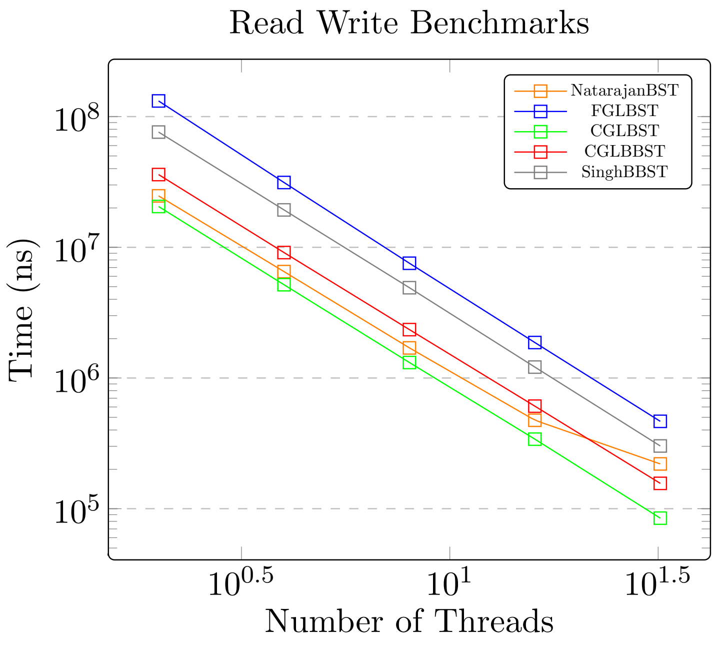
## 4.1 Benchmarking Results

### 4.1.1 Results of Initial Balanced State Benchmarks

The results of the benchmarks conducted on the initial balanced state are presented below:



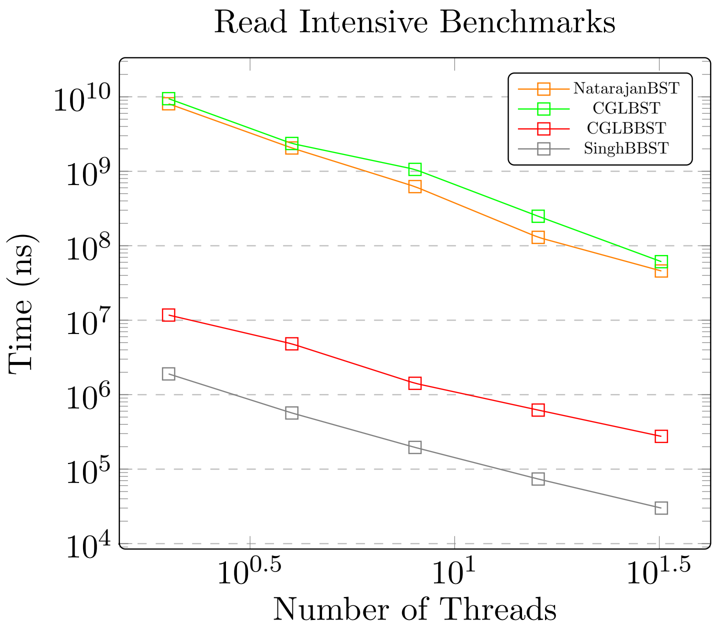


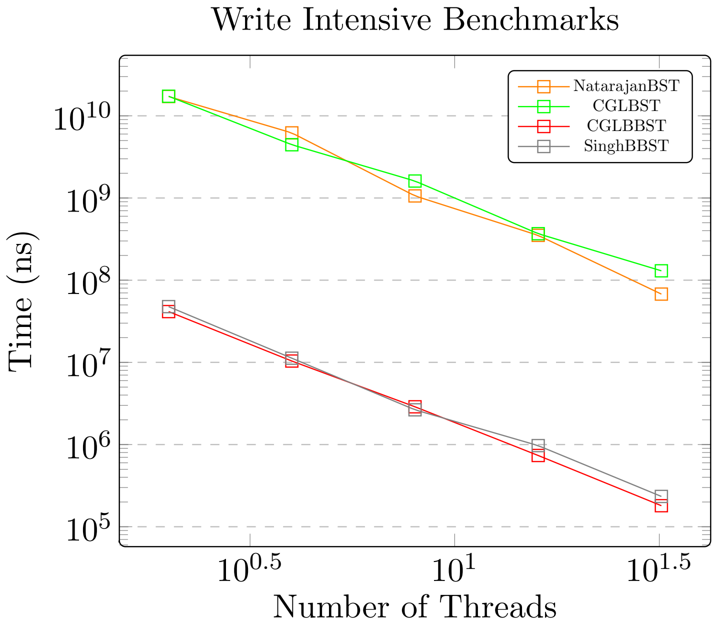


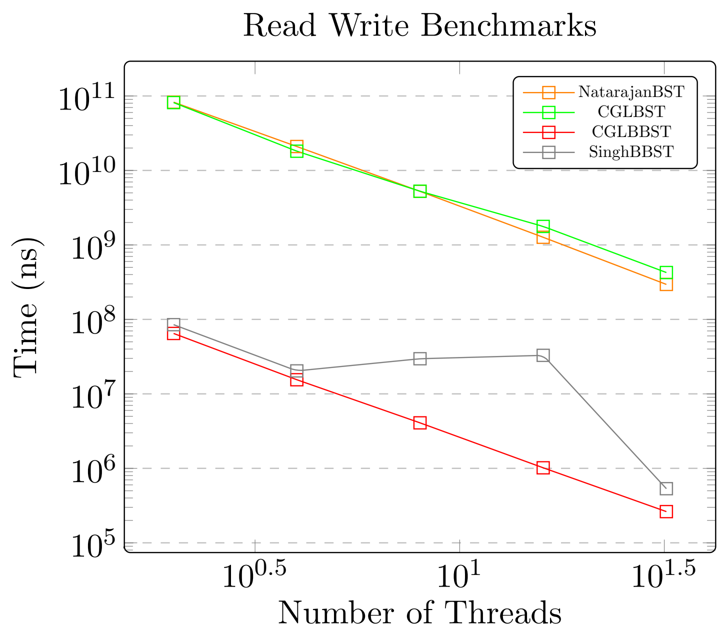
In summary, our findings indicate significant improvements across all data structures as the number of threads increased, demonstrating scalability. It is noteworthy that both the coarse-grained locking BST and BBST outperformed their counterparts, albeit not by a substantial margin. Additionally, we speculate that the additional work performed by the BBSTs for balancing may have contributed to slightly inferior performance, particularly as we have already ensured that the insertion order maintained the balance of the tree.

### 4.1.2 Results of Initial Imbalanced State Benchmarks

The results of the benchmarks conducted on the initial imbalanced state are presented below:







In summary, despite an initial imbalanced state, we observed significant improvements across all data structures as the number of threads increased. Moreover, we observed comparable performance between the coarse-grained locking BST and BBST, as well as their lock-free counterparts, albeit with some fluctuations in certain data points. However, we attribute these fluctuations to outliers. This benchmark also underscores the importance of the initial degree of balance of the tree. While the lock-free structures exhibited resilience to imbalance, the BST suffered a significant performance impact, highlighting the critical role of initial tree balance.

## 4.2 Design Guidelines for Intention-Swapping

As demonstrated in chapter 3, the design of lock-free data structures is far from trivial. It demands meticulous attention to detail, and in this section, we will outline the intention-swapping technique and some guidelines to keep in mind when creating such data structures.

The core of intention-swapping lies in establishing a representation of an intended operation, which must encapsulate all necessary information for the `help` function. Subsequently, a Compare-And-Swap is attempted using the pointer associated with this intention, with the entire operation retried in case of failure. When employing this approach, it's paramount to remain vigilant of the ABA problem and implement measures to mitigate its occurrence.

Additionally, it's imperative to include a `help` function that any thread can invoke to aid in completing the operation. This ensures that the operation remains lock-free. A key attribute of such a `help` function is its idempotence, which is essential for maintaining the correctness of the algorithm.

To determine when intention-swapping is necessary, we can refer to the lock-free stack and queue described by Teh (2022), which do not utilize this technique. The crucial distinction lies in the fact that our trees must prohibit certain operations while others are ongoing, whereas no such requirement exists for the stack and queue. If such a requirement is present, intention-swapping ensures that only one operation can proceed at a time, with all other operations needing to `help` this operation before retrying.

In summary, if certain operations need to be disallowed while another operation is in progress, intention-swapping may be a beneficial design choice. If intention-swapping is chosen, it's important to:

1. Ensure that the class representing such an intention encapsulates all necessary data.
2. Create a `help` function for each intention, ensuring that all `help` functions are idempotent.

# Chapter 5

# Conclusion

We will wrap up our project by summarizing our contributions and discussing limitations of our project as well as potential avenues for future research.

## 5.1 Contributions

This project offers three distinct BST implementations and two distinct BBST implementations. We introduced modifications to certain implementations, providing reasoning as well as a proof sketch for lock-freedom despite these modifications. Additionally, we conducted benchmarks for all implementations to provide a comprehensive perspective on their performance across diverse workloads. Our findings demonstrate that lock-free data structures exhibit performance comparable to their lock-based counterparts while simultaneously providing a theoretical guarantee of progress. Furthermore, we present a systematic guideline for the intention-swapping technique in lock-free data structures. This concludes our contribution to this project.

## 5.2 Future Work

At its current stage, this project does not delve into garbage collection, as its complexity exceeds the scope of this project. However, by laying the foundation, future implementations may integrate garbage collection into our implementation and benchmarks.

Similar to the approach taken by Teh (2022), our implementations rely on operating system-defined memory allocators and deallocators, which may not be lock-free. Therefore, future work could involve analyzing these algorithms with a lock-free allocator.

Additionally, future work may explore different specialized algorithms and design choices for data structures similar to ours. Alternatively, a more general approach, such as the lock-free locks proposed by Ben-David, Blelloch, and Wei (2022), could be investigated.

# References

Ben-David, N., Blelloch, G. E., & Wei, Y. (2022). Lock-free locks revisited. *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (pp. 278–293), 2022.

Blasgen, M., Gray, J., Mitoma, M., & Price, T. (1979). The convoy phenomenon. *ACM SIGOPS Operating Systems Review*, *13*(2), 1979, 20–25.

Ellen, F., Fatourou, P., Ruppert, E., & van Breugel, F. (2010). Non-blocking binary search trees. *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing* (pp. 131–140), 2010.

Herlihy, M., Luchangco, V., & Moir, M. (2003). Obstruction-free synchronization: Double-ended queues as an example. *23rd International Conference on Distributed Computing Systems, 2003. Proceedings.* (pp. 522–529), IEEE, 2003.

Herlihy, M., Shavit, N., Luchangco, V., & Spear, M. (2020). *The art of multiprocessor programming.* Newnes.

Herlihy, M. P., & Wing, J. M. (1990). Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *12*(3), 1990, 463–492.

Howley, S. V., & Jones, J. (2012). A non-blocking internal binary search tree. *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures* (pp. 161–171), 2012.

Natarajan, A., & Mittal, N. (2014). Fast concurrent lock-free binary search trees. *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming* (pp. 317–328), 2014.

Pimpale, S., & Kudtarkar, R. (2015). *Concurrent lock-free bst.*

Singh, M., Groves, L., & Potanin, A. (2021). A relaxed balanced lock-free binary search tree. *Parallel and Distributed Computing, Applications and Technologies: 21st International Conference, PDCAT 2020, Shenzhen, China, December 28–30, 2020, Proceedings 21* (pp. 304–317), Springer, 2021.

Teh, X. Y. (2022). *Performance comparison of lock-based and lock-free data structures in modern programming languages.*