8-1-2019

Coarse-Grained, Fine-Grained, and Lock-Free Concurrency Approaches for Self-Balancing B-Tree

Edward R. Jorgensen II
ed.jorgensen1@gmail.com

COARSE-GRAINED, FINE-GRAINED, AND LOCK-FREE CONCURRENCY APPROACHES FOR SELF-BALANCING B-TREE

By

Edward R. Jorgensen II

Bachelor of Science – Computer Science
University of Nevada, Las Vegas
1984

Master of Science – Computer Science
University of Nevada, Las Vegas
1991

A dissertation submitted in partial fulfillment
of the requirements for the

Doctor of Philosophy - Computer Science

Department of Computer Science
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
May 2019

UNLV | GRADUATE COLLEGE

Dissertation Approval

The Graduate College
The University of Nevada, Las Vegas

April 11, 2019

This dissertation prepared by

Edward R. Jorgensen II

entitled

Coarse-Grained, Fine-Grained, and Lock-Free Concurrency Approaches for Self-Balancing B-Tree

is approved in partial fulfillment of the requirements for the degree of

Doctor of Philosophy - Computer Science
Department of Computer Science

Ajoy Datta, Ph.D.
Examination Committee Chair

Kathryn Hausbeck Korgan, Ph.D.
Graduate College Dean

John Minor, Ph.D.
Examination Committee Member

Laxmi Gewali, Ph.D.
Examination Committee Member

Sidkazem Taghva, Ph.D.
Examination Committee Member

Emma Regentova, Ph.D.
Graduate College Faculty Representative

Abstract

This dissertation examines the concurrency approaches for a standard, unmodified B-Tree which is one of the more complex data structures. This includes the coarse grained, fine-grained locking, and the lock-free approaches. The basic industry standard coarse-grained approach is used as a base-line for comparison to the more advanced fine-grained and lock-free approaches. The fine-grained approach is explored and algorithms are presented for the fine-grained B-Tree insertion and deletion. The lock-free approach is addressed and an algorithm for a lock-free B-Tree insertion is provided. The issues associated with a lock-free deletion are discussed. Comparison trade-offs are presented and discussed. As a final part of this effort, specific testing processes are discussed and presented.

Acknowledgments

This work would not have been possible without the support and mentoring of Dr. Ajoy Datta over many years. His patience, guidance, and support allowed me to succeed.

I would also like to thank the members of my committee who have supported and guided me on topics well beyond this project.

And finally, I wish to thank my loving and supportive wife, Rudolpha, who has provided unending inspiration.

Table of Contents

List of Figures

List of Algorithms

1.0 Chapter 1,

Introduction

1.0 Chapter 1, Introduction

The ubiquitous available multi-core resources must be effectively applied to the increasing demand brought on by the performance requirements from more complex applications and expanding data sets. Such applications are often heavily dependent upon concurrent data structures. To meet this demand, effectual concurrency must be applied to more efficiently utilize the applicable data structures.

This effort focuses on concurrency approaches for implementation of a B-Tree [1][2] which is a fairly complex data structure. In their 1992 text, J. Gray and A. Reuter claim that B-Trees are the most important access path structure in database and file systems [3]. In this context, the complexity of the B-Tree data structure is associated with the localization of the changes when modifying the data structure. A more straight-forward data structure, such as a linked list or hash-table, can be altered (i.e., insertion or deletion) by modifying a very limited subset of directly adjacent elements. For a more complex data structure, such modify operations may impact additional elements at multiple, varying non-adjacent levels within the hierarchical structure. The specifics of B-Tree modification processes are addressed in the subsequent sections.

There are many variants for B-Trees (B^{+}-Trees [2], B*-Trees [4], and B^{link}-Trees [5]) and this effort focuses on the standard, unmodified B-Tree structure as fully described in subsequent sections. B-Trees are typically used for large-block storage systems including file and database

systems. Concurrent operations on B-Trees have been studied to improve overall performance and address multi-user functionality. A B^+-Tree is a B-Tree to which an additional level containing all the keys is added at the bottom level with those leaves forming a linked-list. The redundant copies of the keys in the interior of the B^+-Tree provide additional options for addressing concurrency at the cost of additional space.

The concurrency environment being addressed herein is multiple, simultaneously executing threads on a shared memory machine. The specific concurrency approaches for this environment are summarized in the following sections.

1.1 B-Tree Description

A B-Tree is a self-balancing tree-based data structure that stores sorted data and allows the standard search, insert, and delete operations in logarithmic time.

The literature is not consistent regarding the terminology associated with B-Tree definitions [6]. Some researchers define a B-Tree in terms of the minimum numbers of keys, referred to as order [7]. Other researches, particularly for concurrent research activities, have define B-Trees in terms of the maximum number of child pointers referred to as degree [8]. To maintain consistency with similar research, the B-Tree definition herein uses the degree terminology.

The degree, ***degree***, of a B-Tree is the maximum number of links (pointers) per node. The B-Tree maintains the basic binary search tree properties with the primary generalization that a node can have ***degree***-1 keys with up to ***degree*** links. Once instantiated, the degree is fixed.

For example, the layout for a partially populated B-Tree with a degree of six is as follows:

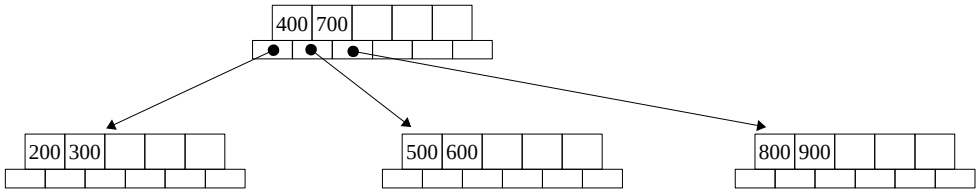


Figure 1: Simple B-Tree Example

In a more populated B-Tree, there could be many additional levels than shown in the figure above. The B-Tree always has a single root node.

The complete properties for a standard B-Tree include:

- All the leaf nodes must be at same level
- All nodes except root must have at least (***degree***/2-1) keys and maximum of ***degree***-1 keys
- All non-leaf nodes except root (i.e., all internal nodes) must have at least ***degree***/2 children
- If the root node is a non leaf node, then it must have at least $\lceil \boldsymbol{degree/2} \rceil - 1$ children
- A non-leaf node with ***k*** keys must have ***k***+1 number of children
- All the key values within a node must be in order, typically ascending
- The height of a B-Tree is:

height: $$\boldsymbol{height} \leq \left\lfloor \log_{degree} \left(\frac{\boldsymbol{degree + 1}}{\boldsymbol{2}} \right) \right\rfloor$$

The degree is configurable, but fixed in a specific implementation.

The minimum degree for a B-Tree is not explicitly defined in the technical literature. From a practical standpoint, a degree of one would be a linked list, and a degree of two would be a standard binary tree. A degree of three would could be considered a B-Tree, however such a

configuration is typically referred to as a 2-3 tree [9] and a degree of four is usually referred to as a 2-3-4 tree [9]. While such smaller degree B-Tree's would be technically valid, they do not represent the more general case. In this effort, a more practical minimum is considered five.

In the face of active on-going modify operations, B-Trees typically do not require re-balancing as frequently as other types of self-balancing trees (i.e., Red-Black trees and AVL trees). However, re-balancing may require access to nodes at multiple non-adjacent levels which can present additional challenges for concurrent implementations.

1.2 Previous Work

The initial approach for concurrency in B-Tree's was proposed by Samdi [10]. A variant on coarse-grained locking which uses semaphores with the impacted section of the B-Tree being exclusively locked. This approach blocks all other operations including searching.

More recent work on concurrent operations for B-Trees have primarily focused on B-Tree variants: B*-Trees, B^{+}-Trees, B^{link}-Trees, and prefix B-Trees [11]. Such variants alter the problem and require that, in addition of the B-Tree structure, all the keys be stored in leaf nodes (often redundantly). The non-leaf internal nodes contain key values and applicable pointers which are used in traversing downward. As another variant, the leaf nodes (all at the same level) form a linked list. Broadly, these approaches trade additional memory for implementation speed and simplification of concurrency by allowing concurrent threads the option of "going-around" by taking a path from a previous nodes unblocked left link down to the bottom of the tree and then using the linked list to find the key. This allows updates to be done to the internal node structure while still allowing access to all the keys contained at the leaf level.

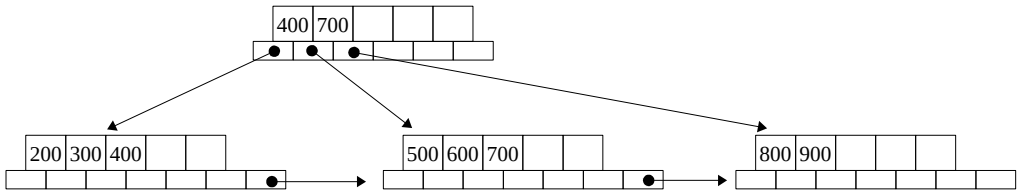


Figure 2: Simple B+-Tree Example

For example, the layout for a B+-Tree with a degree of six would be as follows:

In a more populated B+-Tree, there would be many additional levels than shown. This approach uses additional memory related to the redundant storage of keys and the associated pointers.

Ellis [12] presented a concurrency solution for 2-3 trees. Several methods are used to increase the concurrency possible, and it was claimed these are extendable to B-trees. No algorithms for concurrent B-Trees were presented.

The B^{link}-Trees [13] also use an additional pointer field to each node that points to the next node at the same level of the tree forming a linked-list. However, the algorithms were simplified and latches used, instead of locks, to improve performance.

Previous work on lock-free trees include Fraser's lock-free balanced tree [14] which utilizes a transactional memory subsystem. This effort simulated the transactional memory in software which incurred a performance penalty.

Braginsky and Petrank present a lock-free B+-Tree [15]. For a B+-Tree, the leaves contain all the keys. The redundant copies of the keys in the internal nodes are used for navigating through the tree. Keys in internal nodes are used only as guides to the correct leaf nodes. This redundant storage simplifies the B+-Tree concurrency operations by allowing changes to the internal nodes and changes to the leaf nodes to be performed independently.

1.3 Contributions

In this effort, a series of algorithms and implementations for concurrent B-Tree's are provided. Due to the relative complexity, B-Tree implementations are not commonly available. As such, a fully polymorphic coarse-grained B-Tree implementation using C++11/17 primitives was developed and is provided as a reference.

The fine-grained locking approach is thoroughly explored and also includes a fully polymorphic coarse-grained B-Tree implementation using C++11/17 primitives for both the insertion and deletion operations.

A lock-free insertion algorithm and fully polymorphic coarse-grained B-Tree implementation using C++11/17 primitives is provided.

A series of performance comparisons are presented. This addresses the trade-offs between algorithm complexity, implementation ease, and overall performance and throughput.

To complete the overall effort, a comprehensive testing methodology is presented.

2.0 Chapter 2,

Summary of Concurrency Approaches

2.0 Chapter 2, Summary of Concurrency Approaches

Broadly speaking, there are two distinct approaches for concurrency. The distributed approach moves parallel computations to different, connected computers [16]. This has some key advantages including access to a potentially extremely large number of different computers. The code and the applicable data must be passed between the various computers performing the calculations. The communication speed becomes a limiting factor in the distributed approach. Shared data structures also present additional challenges. The application interface has been well-developed and embedded into libraries such as the Message Passing Interface (MPI) [17].

The other common approach is a shared memory environment on a multi threaded processor. In such an environment, a processor will simultaneously issue instructions from multiple threads in a single cycle [18]. This approach is limited by the number of available cores. Each simultaneously executing thread has immediate access to shared memory data structures. Such access presents challenges for ensuring safe access and correct manipulation during modifications.

For this research project, the focus is a shared memory environment (i.e., multi-core computing). A shared memory, concurrent data structure object must allow safe access for multiple, simultaneously executing threads performing an on-going series of various operations (search, insert, and/or delete) that may alter the data structure in indeterminate ways.

The approaches to shared memory concurrency [19] include

- Coarse-Grained Locking
- Fine-Grained Locking
- Lock-Free
- Transactional Memory

The coarse-grained [20] and fine-grained [21] locking are traditional approaches with the lock-free [22] approach being relatively new and a current research focus. These approaches are often implemented for basic data structures such as linked lists, queues, and stacks.

The transactional memory [23] approach typically relies on hardware to assist with the locking of low-level load and store operations to support the higher-level more complex software operations. Such exotic hardware is not generally available. A software implementation of transactional memory is possible with a significant performance penalty. Due to the lack of accessible hardware, the transactional memory approach is not addressed in this effort.

2.1 Coarse-Grained Locking

The most basic approach for handling concurrent data structure access is coarse-grained locking. This approach involves locking the entire data structure, performing the operation, and when complete, unlocking the entire data structure. This approach is fairly straight-forward and is commonly used.

A coarse-gained polymorphic implementation was completed. The implementation is done in C++11/17 using the relatively new concurrency features (e.g., threading library, mutexes as primitives). There are no known C++11/17 reference implementations. Such a coarse-grained implementation is not new work and is not a core part of the research. It is only included for a baseline reference and for further comparisons. Additionally, the typical implementation approach will be adjusted from a recursion to a top-down, proactive splitting process as suggested by Cormen [24].

2.2 Fine-Grained Locking

The next approach is fine-grained locking. This approach locks a minimal subset of the data structure [18]. As such, operations on different subtrees of the data structure may continue simultaneously unimpeded. Fine-grained locking can improve the overall throughput of a concurrent system. However, this increases the complexity and requires addressing potential deadlock, live-lock, and starvation issues. For a B-Tree, the scope of that locking may be a non-trivial subsection of the data structure which negatively impacts the performance and adds to the overall complexity. Additionally, fine-grained locking adds additional run-time overhead as compared to coarse-grained locking due to the lock acquire and lock release operations. For a no or limited contention environment this additional locking overhead typically causes the coarse-grained implementation to provide better performance.

A complete custom, fine-grained polymorphic implementation is given that addresses the potential deadlock, live-lock, and starvation issues. The implementation will be done in C++11/17 using the relatively new concurrency features (e.g., threading library, mutexes as primitives). This is compared with the coarse-grained locking implementation to determine and quantify the performance and complexity trade-offs associated with such complex concurrent data structure implementations. There are no known B-Tree fine-grained reference implementations.

2.3 Lock-Free

The previously outlined approaches for handling concurrency use locks to synchronize access to the data structure or subsections of the data structure. If one thread attempts to acquire a lock that is already held by another thread, the thread will block until the lock is free. This blocking can be undesirable. When a thread is blocked, it cannot accomplish useful work.

Lock-free programming is concurrent programming without locks. More formally, a lock-free implementation of a concurrent data structure is one that guarantees that some thread can complete an operation in a finite number of steps regardless of the execution of the other threads. Lock-free programming uses atomic operations, such as compare-and-swap to maintain a

consistent state. The lock-free approach has the potential to improve overall system throughput, especially for high contention environments and has the desirable liveness property. A lock-free data structure can improve performance on multi-core processors because access to the shared data structure does not need to be serialized to stay coherent. Lock-free implementations are very difficult and not always possible. The complexity of lock-free can negatively impact performance, particularly for no or limited contention environments.

A polymorphic lock-free B-Tree insertion algorithm was developed and implemented. The implementation will be done in C++11/17 using the relatively new concurrency features. The issues associated with a lock-free delete are addressed. This implementation will be compared with the coarse-grained and fine-grained implementations to quantify the performance and complexity trade-offs associated with the lock-free approach. There are no known lock-free B-Tree implementations.

2.3.1 Lock-Free Primitive

The primary primitive for lock-free algorithms is the compare-and-swap (CAS) instruction [22]. The CAS instruction is atomic in that no other concurrently executing instructions can interrupt or interfere with this operation. More specifically, the CAS instruction compares the contents of a single memory location with an expected value and, only if the expected value and current value are the same, modify the contents of the single memory location to the set the new value. If the actual value does not match the expected value, another thread has changed the value which indicates the status of the current has been changed. The change means the current action must be re-evaluated before proceeding. This typically requires repeating the in-process action from the beginning.

2.3.2 Levels of Lock-Freedom

Maurice Herlihy introduced a wait-free hierarchy [22] which classifies multiprocessor synchronization primitives and is further detailed in his text [19]. The hierarchy is summarized in the following sections.

2.3.2.1 Wait-Free

A wait-free concurrent implementation guarantees that any thread can complete an operation in a finite number of steps, fully independent of other on-going concurrent operations. This is the strongest level of lock-freedom, but is often not possible. Wait-freedom provides guaranteed system-wide throughput and starvation-freedom. Since there is no locking, the inherent problems of locks, including deadlock and priority-inversion, are avoided.

2.3.2.2 Lock-Free Approach

A lock-free or non-blocking approach to addressing shared access to mutable objects guarantees that if there is an active thread performing an operation on the object, some operation, by the same or another thread, will complete within a finite number of steps regardless of other threads' actions. Wait-freedom provides guaranteed system-wide throughput. The lock-free approach is inherently immune to deadlock and priority inversion. A wait-free algorithm is lock-free but a lock-free algorithm may not be wait-free.

2.3.2.3 Obstruction-Free

An implementation is obstruction-free if, starting from any reachable configuration, any thread can finish in a bounded number of steps if all of the other processes stop [25]. No thread can be blocked by delays or failures of other threads. Obstruction-free does not guarantee progress while two or more threads run concurrently so while deadlock is not possible, livelock is possible. This is the weakest form of lock-freedom. A lock-free algorithm is obstruction-free but an obstruction-free algorithm may not be lock-free.

2.3.3 ABA Problem

The ABA problem [26] can occur in multi-threaded algorithms, especially lock-free algorithms, particularly in high-contention environments. The ABA problem can happen when two threads that are attempting to simultaneously alter a mutable shared data object interleave execution.

For example,

- An initial thread, t_i, begins the process of updating a node, n_0, in a data structure and is preempted before completing its action.
- Another thread, t_n, finds and deletes node, n_0, from the data structure.
- Thread, t_n, or possibly another thread, allocates a new node, n_1, and inserts the new node into the data structure. The new node is placed into the data structure at the same location as the previously deleted node.
    - By chance, the new node has the same address as the old node.
- The initial thread, t_i, resumes execution and sees the same address and is unable to determine that the data structure has been changed during it preemption, potentially resulting in invalid operations (since the node has been changed).

The common methods to address the ABA problem include [27]:

- Tagging
- Intermediate Nodes
- Deferred Deletion

The deferred deletion is utilized as the most straight-forward and robust technique in this specific application.

3.0 Chapter 3,

B-Tree Operations

3.0 Chapter 3, B-Tree Operations

This chapter outlines the standard B-Tree insert, search, and delete operations. These basic non-concurrent approaches form the foundation for the concurrent algorithms. Specific algorithms for each operation and locking strategy are provided in the applicable subsequent sections.

3.1 Standard Insertion Approach

The standard B-Tree insertion approach uses a typical recursive descent process in a similar manner as a Binary Search Tree insertion [28]. This involves recursively traversing the B-Tree to the appropriate leaf node. As needed, split the node and move a value up to the preceding node progressing all the way to the root if needed.

A simple sketch of the standard B-Tree insertion approach to insert a value, ***key***, into a B-Tree of degree, ***degree***, is as follows:

1. If empty, create new root, insert key and terminate.
2. From the root node, traverse the B-Tree to find the leaf node to which ***key*** should should be added.
3. Add ***key*** to this node in the appropriate place among the existing values (sliding existing keys as needed). Since this is a leaf node, there are no child pointers to adjust.
4. Check key count

1. If the leaf node is not full (e.g., all key slots filled), then terminate.
2. If the leaf node is full (e.g., all key slots filled), split the node into three parts:
    1. Left: the first (***degree***-1)/2 values
    2. Middle: the middle value (position 1+((***degree***-1)/2)
    3. Right: the last (***degree***-1)/2 values

    The left and right will have the minimum necessary number of key values (***degree***-1)/2 required and made into individual nodes. Additionally, the middle value is placed in the parent. The parent node is adjusted to reflect these updates. Due to the recursive descent, the parent node is accessible as the recursion unwinds.

This process is continued until a non-full node is found or until the root itself overflows. If the root overflows, it is split in this manner and a new root is created containing only the middle value. As such, a B-Tree grows in height only at the root.

3.2 Proactive Insertion Approach

The standard recursive descent insertion approach is especially challenging for concurrent operations due to the large number of nodes that are essentially locked, including the root. This effectively blocks other concurrent operations until the insertion process is completely finished. Additionally, stack space is used for the recursion which requires additional memory resources resulting in extra overhead. In no or very low-contention environments this is acceptable. In high-contention environments, the top-down recursive approach effectively limits concurrency to single threaded.

To address these issues, Corman [24] suggested a top-down proactive splitting approach. During the top-down traversal, if a node is encountered that is full, it is split immediately or proactively. When the final leaf node insertion is performed, this ensures that the parent is not full and can accept a key if the leaf node must be split.

A simple sketch of the proactive B-Tree insertion approach is as follows:

1. If empty, create new root, insert key and terminate.
2. If root node is full, split root node.
3. Initialize current node to root.
4. While the current node is not a leaf.
    1. Find the appropriate child that is going to to be traversed next.
    2. If that child node is full.
        1. Split child node into three parts:
            1. Left: the first (***degree***-1)/2 values
            2. Middle: the middle value (position 1+((***degree***-1)/2)
            3. Right: the last (***degree***-1)/2 values

        The left and right will have the minimum necessary number of key values (***degree***-1)/2 required and made into individual nodes. Additionally, the middle value is placed in the parent. The parent node is adjusted to reflect these updates.
    3. Set current node to applicable child.
5. Inset key into non-full leaf node.

The process ensures that the final insertion into the leaf node can be performed directly without splitting since if the leaf node was full, it would have already been split in the previous step.

3.3 Search Approach

The B-Tree search uses a modified descent process in a similar manner as a binary tree. The search is started at the root and progresses downward until either the key is found or the downward path is exhausted (i.e., a NULL is found).

A simple sketch of the general B-Tree search approach is as follows:

1. If root empty
    1. Return key is not found.
    2. Terminate.
2. Initialize current node to root.
3. While the current node is not a leaf.
    1. Find first key greater than or equal to the key
    2. If key is found
        1. Return key found
    3. Set current node to applicable child.
4. Return key not found.

A simple approach would be for the search to return either true or false if the key is found or not found. Depending on the specific requirements, the key node itself may be returned or a boolean returned.

3.4 Deletion Approach

Deletion of a key from a B-tree is more complicated than insertion. The additional complexity is primarily because a key cannot be directly deleted when it is in an internal (i.e., a non-leaf node). Deletion from an internal node requires rearranging the node's children which may involve a traversal to the applicable leaf node in that subtree. A further complication is that as keys are removed, nodes may lose keys and the key count could fall below the (***degree***/2-1) property which must be addressed by either borrowing keys from neighboring nodes or merging sibling nodes and then deleting the freed node. The merging must maintain consistency for the key values and the applicable child points.

The general deletion approach is as follows:

1. If tree is empty, terminate.
2. Initialize current node to root.
3. While the current node is not NULL.
    1. Find the appropriate key array index based on key value.
    2. Check if the key is in current node.
        1. If node is a leaf, delete key and slide remaining key as needed and terminate.
        2. If non-leaf node
            1. Case A: If the child, ***y***, that precedes the key index in the current node has at least ***degree***/2 keys, then;
                1. Find the predecessor in the subtree rooted at ***y***. *Note*, $\boldsymbol{k_0}$ can be found in a single downward pass.
                2. Replace the key value with the predecessor value
                3. Change the key value to be deleted to the predecessor value and continue the deletion process from the applicable current node child (in order to remove the duplicate key).
            2. Case B: If the child, ***z***, that succeeds the key index in the current node has at least ***degree***/2 keys, then;
                1. find the successor in the subtree rooted at ***z***. *Note*, $\boldsymbol{k_0}$ can be found in a single downward pass.
                2. Replace the key value with the successor value
                3. Change the key value to be deleted to the successor value and continue the deletion process from the applicable current node child (in order to remove the duplicate key).

3. Case C: If both ***y*** and ***z*** have only less than (***degree***/2) keys, merge ***z*** into ***y*** and delete ***z*** when done pushing the key into the updated ***y*** node. *Note*, this moves the key down one level.

3. If current node is a leaf
    1. Report key not found and terminate.
4. If the applicable child has less than ***degree***/2 keys, fill the child node.
    1. If the left sibling exists and has at least ***degree***/2 keys, then borrow a key from the previous node.
    2. If the right sibling exists and has at least ***degree***/2 keys, then borrow a key from the next node.
    3. If both left and right siblings have less than (***degree***/2) keys, merge right sibling into the left sibling and and delete the right sibling.
5. Set current node to applicable child node based on index.

Due to the general structure, the majority of the keys in a B-Tree are in the leaves, deletion operations are most often deleted from keys in leaf nodes. When deleting a key in an internal node, the general approach is to make a downward pass through the tree to replace the key with its predecessor or successor (cases A and B) key values (from an applicable leaf node). This process may impact a series of non-adjacent nodes which presents additional challenges for concurrency.

4.0 Chapter 4,

Coarse-Grained B-Tree Algorithms

4.0 Chapter 4, Coarse-grained B-Tree Algorithms

This chapter presents the specific customized coarse-grained B-Tree insert, search, and delete algorithms. The general approach for coarse-gained locking is to use a single, globally shared lock at the beginning of each function. The proactive node splitting approach is used which eliminates the recursive element. This is advantageous since a node is never traversed twice and no additional stack space is required. A disadvantage of the proactive insertion approach is that some unnecessary splits may be performed.

This coarse-grained approach and implementation is fully polymorphic using the C++11/17 mutex primitives and used as a baseline. Available existing reference implementations do not use the more current C++11/17 (or better) standard, are not polymorphic, and typically use the recursive descent approach.

4.1 Main Ideas

This section presents a conceptual overview of the algorithms and the coarse-grained locking strategy. By using a single global coarse-gained lock, only one operation (insert, search, or delete) is being performed at any given time. This ensures that the modify operations cannot interfere with each other. The search thread also requires the lock in order to ensure that a search cannot not performed while the data structure is being modified and thus not be in a valid state. For modifying operations, once the lock is obtained, the changes to the data structure are identified and performed. Subsequent operations are only allowed to start after the current

operation has fully completed. The contention point is at the main global lock. For no or minimal contention, this works very well and represents the least amount of additional overhead. However, as the contention increases, more threads will be competing for the single lock. A mutex does not provide any fairness, so if a set of threads are waiting to obtain the lock causing a hot spot, the selected thread will be chosen arbitrarily instead of a more fair first come, first served order.

The following sections provide the technical details of the insertion, search, and deletion algorithms including the specific lock acquire point for each algorithm.

4.2 Coarse-Grained B-Tree Configuration

This section summarizes the coarse-grained B-Tree configuration with regard to the specific structure fields and the sentinel root node.

4.2.1 B-Tree Node Definition

The B-Tree node definition includes fields for the count, leaf status (true/false), a pointer to an array for the keys, and a pointer to an array of pointers for the child pointers.

```
struct treeNode
{
      myType             *keys;      // key values
      treeNode<myType>   **ptrs;     // child ptrs
      int                cnt;        // current number of keys
      bool               leaf;       // true if leaf, else false
};
```

Figure 3: Coarse-Grained B-Tree Node Configuration

The ***myType*** is the passed type for the polymorphic implementation.

Visually, a node of degree 6 can be logically viewed as shown below.

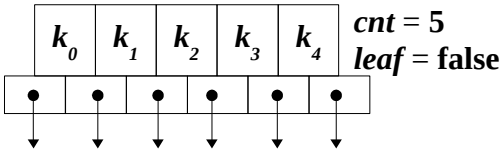


Figure 4: B-Tree Node Visualization (degree=6)

In addition to the node structure, some global class variables include the instantiated degree, root, and the single mutex. These are individual global variables defined for the entire class (and not available external to the class).

4.2.2 Sentinel Node

A sentinel node is used in this implementation. The sentinel node is the root node which is created as part of the initialization process (i.e., in the constructor). This simplifies the insertion algorithm by eliminating the need to check for the existence of a root node, and if not found, creating one. The use of a sentinel node in a coarse-gained implementation is a matter of convenience and provides some consistency with the fine-grained approach. Since keys are stored in the sentinel node, it does not use any additional space for a non-empty B-Tree.

4.3 Insertion Algorithm

The coarse-grained B-Tree insertion algorithm uses a root to leaf-node downward traversal algorithm with proactive splitting during the traversal for the final leaf node insertion.

To ensure there is no interference from other active threads, a global lock is obtained. Once the lock is obtained, the insertion is allowed to proceed. Other threads will wait until the lock is released before continuing. In this implementation, the C++11/17 lock guard mechanism [29] is used to improve safety and correctness by automatically releasing the lock when the applicable

mutex goes out of scope. This is an accordance with C++11/17 recommended coding standards [29].

As part of the downward traversal, a node discovered containing the maximum allowed keys is split into two nodes. This includes the splitting the root node if needed. The root split is a special operation and is how the B-Tree grows in height. Keys are always added to a leaf node. If that fills the leaf node, it would be split by a subsequent insertion operation (if there is one).

Below is the coarse-grained B-Tree insertion algorithm.

```
1.  void bTree<myType>::insert(myType key)  {
2.          treeNode<myType>     *curr, *currChld;
3.          int       idx, nIdx;
4.          bool      insertDone = false;
5.
6.          // coarse-gained lock
7.          lock_guard<std::mutex> lock(cgMtx);
8.
9.          // if root full, must split
10.         if (root→cnt == degree-1)
11.                 splitRootNode(root);
12.
13.         curr = root;
14.         while (!insertDone) {
15.
16.                 //  traverse downward to find applicable leaf node, split full nodes along the way
17.                 while (!curr→leaf) {
18.                         // find next ptr based on key
19.                         idx = 0;
20.                         while ((idx < curr→cnt) && (key > curr→keys[idx]))
21.                                 idx++;
22.
23.                         currChld = curr→ptrs[idx];
24.
25.                         // if that child is full, split it...
26.                         if (currChld != NULL && currChld→cnt == (degree-1)) {
```

```
27.                                 splitChildNode(curr, currChld);
28.
29.                                 // re-find where key goes (due to split)
30.                                 nIdx = 0;
31.                                 while ((nIdx < curr→cnt) && (key > curr→keys[nIdx]))
32.                                         nIdx++;
33.
34.                                 if (idx != nIdx)
35.                                         idx = nIdx;
36.                         }
37.                         curr = curr→ptrs[idx];
38.                 }
39.
40.                 //  insert key into non-full leaf node, slide all keys > k over one place over
41.                 idx = curr→cnt - 1;
42.                 while (idx >= 0 && curr→keys[idx] > key) {
43.                         curr→keys[idx+1] = curr→keys[idx];
44.                         idx--;
45.                 }
46.
47.                 // insert the new key at appropriate location
48.                 curr→keys[idx+1] = key;
49.                 curr→cnt = curr→cnt + 1;
50.
51.                 insertDone = true;
52.         }  // end main while
53.  }
```

Algorithm 1: Coarse-Grained B-Tree Insertion Algorithm

During the update operation, the data structure may not be in a valid, traversable state. As such, other threads, including search, must be and are blocked (Coarse-grained B-Tree Insertion Algorithm, line 7).

Each of the major insertion operations are explained in the following sections.

4.3.1 Leaf Node Insertion

For insertion into a non-full leaf node, the keys may need to shifted as required in order to place

the new key into the appropriate position. A lock is required since the during this process the node may not be in a valid state.

For example, inserting 250 into the following leaf node would be performed as shown.

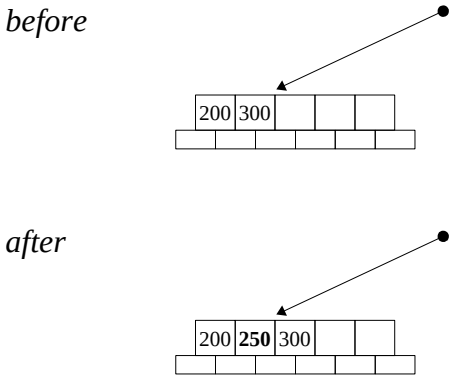


Figure 5: B-Tree Insert Example, Leaf Node

Keys are shifted within the node in order to place the new key into the correct position. Since this is a leaf node, there are no child pointers.

If a leaf node already has the maximum number of allowable keys, it will have been previously split during the while loop by the *splitChildNode()* function (Coarse-grained B-Tree Insertion Algorithm, line 27). This proactive approach ensures that when the final key insertion is performed, the leaf node will always have space for the new key.

4.3.2 Split Root Node Operation

If the root node has the maximum number of allowable keys (***degree***-1), it must be split. Since it is the root node, the B-Tree will grown in height. Upon completion, this operation allows a single key to be in the root node, an exception to the invariant of (***degree***/2-1) keys per node. The key count for the root node is checked (Coarse-grained B-Tree Insertion Algorithm, line 10) before the downward traversal is initiated.

The coarse-grained split root operation algorithm is shown below.

```
1.  void bTree<myType>::splitRootNode(treeNode<myType> *currRoot) {
2.          treeNode<myType>    *lftSib=NULL, *rhtSib=NULL;
3.          int     siz;
4.
5.          // create new sibling nodes
6.          lftSib = makeNewNode();
7.          lftSib→leaf = currRoot→leaf;
8.          rhtSib = makeNewNode();
9.          rhtSib→leaf = currRoot→leaf;
10.
11.         // set size based on degree
12.         siz = (degree-1) / 2;
13.
14.         // copy the first (degree-1)/2 keys of current root left sibling
15.         for (int j = 0; j < siz; j++)
16.                 lftSib→keys[j] = currRoot→keys[j];
17.
18.         // copy the first degree/2 ptrs of current root over to left sibling
19.         if (!currRoot→leaf)
20.                 for (int j=0; j < degree/2; j++)
21.                         lftSib→ptrs[j] = currRoot→ptrs[j];
22.
23.         // copy the last (degree-1)/2 keys of current root to right sibling
24.         for (int j = 0; j < siz; j++)
25.                 rhtSib→keys[j] = currRoot→keys[j+siz+1];
26.
27.         // copy the last degree/2 ptrs of current root over to right sibling
28.         if (!currRoot→leaf)
29.                 for (int j=0; j < degree/2; j++)
30.                         rhtSib→ptrs[j] = currRoot→ptrs[j+degree/2];
31.
32.         // set sibling sizes
33.         lftSib→cnt = siz;
34.         rhtSib→cnt = siz;
35.
36.         // update root, shift key
37.         currRoot→leaf = false;
38.         currRoot→cnt = 1;
```

```
39.         currRoot→keys[0] = currRoot→keys[siz];
40.         for (int i=1; i<degree-1; i++)
41.             currRoot→keys[i] = 0;
42.
43.         // set root pointers
44.         currRoot→ptrs[0] = lftSib;
45.         currRoot→ptrs[1] = rhtSib;
46.         for (int i=2; i<degree; i++)
47.             currRoot→ptrs[i] = NULL;
48. }
```

Algorithm 2: Coarse-Grained Split Root Algorithm

If the root is full, it will be split. This ensures the traversal is initiated from a non-full node.

The split root operation is only initiated after the primary lock is obtained (Coarse-grained B-Tree Insertion Algorithm, line 7). Since this is a global lock, no additional locking is required.

4.3.3 Split Child Node Operation

The split of a full non-root node is referred to as split-child node since only the child on the direct traversal path of the current node is checked. The split operation updates the parent by moving a key up to the parent and thus requires the parent to have space for a key (i.e., must not be full). The traversal is initiated from a non-full root node which is ensured by the Split Root Node Operation, 4.3.2. As the traversal progresses, only the applicable child node is checked for key count. Other nodes, not in the traversal path, are not checked as they are not directly impacted.

The coarse-grained split child operation algorithm is shown below.

```
1.  void bTree<myType>::splitChildNode(treeNode<myType> *parnt,
2.                                     treeNode<myType> *chld)  {
3.          int       siz, idx;
4.          treeNode<myType>      *sib = NULL;
5.
```

```
6.          // create a new sibling node
7.          sib = makeNewNode();
8.          sib→leaf = chld→leaf;
9.
10.         // set size based on degree
11.         siz = (degree-1) / 2;
12.         sib→cnt = siz;
13.
14.         // copy the last (degree-1)/2 keys of child to sibling
15.         for (int j=0; j < siz; j++)
16.                 sib→keys[j] = chld→keys[j+siz+1];
17.
18.         // copy the last degree/2 ptrs of child over to sibling
19.         if (!chld→leaf) {
20.                 for (int j=0; j < degree/2; j++) {
21.                         sib→ptrs[j] = chld→ptrs[j+degree/2];
22.                         chld→ptrs[j+degree/2] = NULL;
23.                 }
24.         }
25.
26.         // reduce the number of keys in child
27.         chld→cnt = chld→cnt / 2;
28.
29.         // find where new key is going in parent
30.         idx = 0;
31.         while ((idx < parnt→cnt) && (parnt→keys[idx] < chld→keys[siz]))
32.                 idx++;
33.
34.         // slide parent child ptrs over to make room for new child
35.         for (int j = parnt→cnt; j >= idx+1; j--)
36.                 parnt→ptrs[j+1] = parnt→ptrs[j];
37.
38.         // add new child to parent
39.         parnt→ptrs[idx+1] = sib;
40.
41.         // slide parent child keys over to make room for new key
42.         for (int j = parnt→cnt-1; j >= idx; j--)
43.                 parnt→keys[j+1] = parnt→keys[j];
44.
45.         // copy middle key of child to parent
46.         parnt→keys[idx] = chld→keys[siz];
```

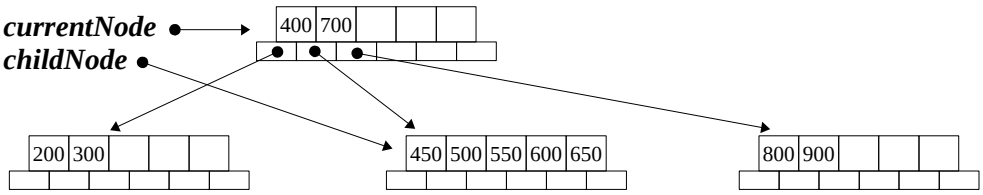
```
47.
48.         // increment count of keys in this node
49.         parnt→cnt += 1;
50.  }
```

Algorithm 3: Coarse-Grained Split Child Node Algorithm

In this manner, keys are gradually moved up from lower nodes to higher nodes. This will occur more frequently during a heavy insert load.

The split child node operation splits the child node into two nodes, moving the middle key up to the parent. The parent is updated accordingly which includes shifting the keys and impacted child pointers to make room for the new key and the new child node pointer. Once the split is completed, the traversal is resumed. This process ensures that the applicable leaf node is not full (since it would have been split if it was full). At the end of the traversal, the new key is added to the applicable non-full leaf node. For example, given the below configuration, and an insert operation of key 525, the current node and applicable child node pointers are shown. Since the child node is full, it will be split before the leaf insertion as shown.

Before split



After split

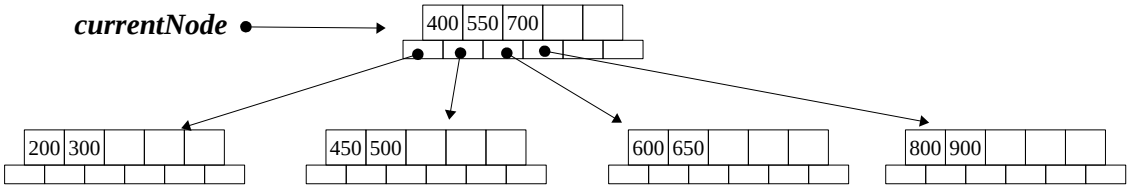


Figure 6: B-Tree Insert Example, Full Node

In this example, the middle key, 550, is moved up to the parent. This requires an additional pointer to be inserted into the parent node (as shown in the "after split" configuration). If the child node is not a leaf node, the traversal would continue. Here, the child node is a leaf node, so the traversal loop is ended at the leaf node and the leaf node insertion is performed.

The split child operation is only initiated after the global lock is obtained. No additional locking is required.

4.4 Search Algorithm

The coarse-grained search algorithm is shown below.

```
1.  bool bTree<myType>::search(myType key) const  {
2.          treeNode<myType>     *curr;
3.          int       i;
4.
```

```
5.      // coarse-gained lock
6.      lock_guard<std::mutex> lock(cgMtx);
7.
8.      curr = root;
9.      if (curr == NULL)
10.             return  false;
11.
12.     while (curr != NULL) {
13.             // find the first key greater than or equal to key
14.             i = 0;
15.             while ( (i < curr→cnt) && (key > curr→keys[i]) )
16.                     i++;
17.
18.             // see if key is found
19.             if (i < curr→cnt && curr→keys[i] == key)
20.                     return  true;
21.
22.             // traverse downward
23.             curr = curr→ptrs[i];
24.     }
25.     return  false;
26. }
```

Algorithm 4: Coarse-Grained B-Tree Search Algorithm

In-process insertion and deletion operations may cause the B-Tree to temporarily be in an invalid, illegal state. As such, the search algorithm requires locking (Coarse-Grained B-Tree Search Algorithm, line 6) since traversing the data structure while it is being updated could result in an exception. The lock is automatically released after the key is either found or it is determined that they is is not in the B-Tree.

4.5 Deletion Algorithm

As the deletion process continues, some nodes key counts may be decreased to the minimum number of keys allowed (***degree***/2-1). In a similar manner to the insertion, the deletion process checks for and, as needed, performs a proactive adjustment of such nodes. The adjustment options include borrowing a key from the right or left sibling if either has sufficient keys (e.g.,

greater than the minimum number of keys allowed). If neither sibling has sufficient keys, it is possible to merge two nodes into a single node, releasing a node and reducing the width of the tree. The merge operation also moves a key down from the current node into child node. In this manner, under repetitious deletions, keys will be gradually migrated downward in the B-Tree.

This process can eventually drain the root the of keys. If the root node key count is depleted to 0, the empty root node will be deleted and a child node prompted to root. In this manner, the B-Tree will shrink in height.

When deleting keys, the key might be found in either a leaf node or an internal node. Deletion of a key from a leaf node can be performed directly. Due to the proactive adjustment, a leaf node containing the key to be deleted will have sufficient keys for the deletion operation to proceed. If a key to be deleted is identified in a internal node, it must be replaced. Removing the key directly is not possible due to the child pointers. The process of obtaining a replacement key involves identifying and relocating a key from a successor or predecessor leaf node. This process is fully explained in section 4.5.2, Internal Node Deletion Operation.

The coarse-grained deletion algorithm is shown below.

```
1.  void bTree<myType>::remove(myType key)  {
2.          treeNode<myType>     *curr, *cTmp, *rTmp, *tmpChld;
3.          int       idx, pred, succ, t=degree/2;
4.          bool      wasLastChild;
5.
6.          lock_guard<std::mutex> lock(cgMtx);
7.
8.          if (root == NULL) {
9.                  cout << "Tree is empty." << endl;
10.                 return;
11.         }
12.
13.         curr = root;
14.         while (curr != NULL) {
15.                 idx=0;                          // find appropriate index for key
16.                 while ((idx < curr→cnt) && (curr→keys[idx] < key))
```

```
17.                     idx++;
18.
19.             // does this node contain the key?
20.             if ((idx < curr→cnt) && (curr→keys[idx] == key)) {
21.                     if (curr→leaf) {
22.                             // leaf node, just remove key, slide keys backwards one place
23.                             for (int i=idx+1; i < curr→cnt; i++)
24.                                     curr→keys[i-1] = curr→keys[i];
25.                             curr→cnt--;
26.                             curr = NULL;
27.                     } else {
28.                             // key found, but not leaf
29.                             cTmp = curr→ptrs[idx];
30.                             rTmp = curr→ptrs[idx+1];
31.
32.                             if (cTmp→cnt >= t) {                    // case A
33.                                     pred = getPred(curr, idx);
34.                                     curr→keys[idx] = pred;
35.                                     key = pred;
36.                                     curr = curr→ptrs[idx];
37.                             } else if (rTmp→cnt >= t) {             // case B
38.                                     succ = getSucc(curr, idx);
39.                                     curr→keys[idx] = succ;
40.                                     key = succ;
41.                                     curr = curr→ptrs[idx+1];
42.                             } else {                                // case C
43.                                     merge(curr, idx);
44.                                     curr = curr→ptrs[idx];
45.                             }
46.                     }
47.             } else {
48.                     // key not found yet, keep going down...
49.
50.                     // check for not in tree
51.                     if (curr→leaf) {
52.                             cout << "Key " << key << " does not exist in the tree." << endl;
53.                             return;
54.                     } else {
55.                             wasLastChild = ( (idx == curr→cnt) ? true : false );
56.
57.                             // if the child has less than t keys, fill child
```

```
58.                                     if (curr→ptrs[idx]→cnt < t)
59.                                             fill(curr, idx);
60.
61.                                     // keep traversing downward
62.                                     if ( wasLastChild && (idx > curr→cnt) )
63.                                             curr = curr→ptrs[idx-1];
64.                                     else
65.                                             curr = curr→ptrs[idx];
66.                             }
67.                     }
68.             } // end main while
69.
70.             // if root is leaf with no keys, empty tree
71.             if (root→cnt==0 && root→leaf)
72.                     return;
73.
74.             // if root has no keys, tree shrinks -> move up info from child
75.             if (root→cnt == 0) {
76.                     tmpChld = root→ptrs[0];
77.
78.                     // copy keys up
79.                     for (int i=0; i < tmpChld→cnt; i++)
80.                             root→keys[i] = tmpChld→keys[i];
81.
82.                     // copy child pointers up
83.                     if (!tmpChld→leaf)
84.                             for(int i=0; i <= tmpChld→cnt; i++)
85.                                     root→ptrs[i] = tmpChld→ptrs[i];
86.                     root→cnt = tmpChld→cnt;
87.                     root→leaf = tmpChld→leaf;
88.
89.                     if (tmpChld→keys != NULL)
90.                             delete [] tmpChld→keys;
91.                     if (tmpChld→ptrs != NULL)
92.                             delete [] tmpChld→ptrs;
93.                     delete   tmpChld;
94.             }
95.  }
```

Algorithm 5: Coarse-Grained B-Tree Deletion Algorithm

During this process, the B-Tree may be temporarily in an invalid state. To ensure no interference from other active simultaneously executing threads (insertion or deletion), the deletion process must be locked. Before the process is allowed to proceed, the lock is obtained (Coarse-Grained B-Tree Deletion Algorithm, line 6).

Each of the major deletion operations are explained in the following sections.

4.5.1 Key Deletion from Leaf Node

The most straight-forward case is when a key is removed from a leaf node. For deletion from a leaf node with greater than (***degree***/2) keys, the key is removed and the remaining keys are shifted as required. This represents the most basic and straightforward possibility. For example, deleting 500 from the following subtree would be performed as shown.

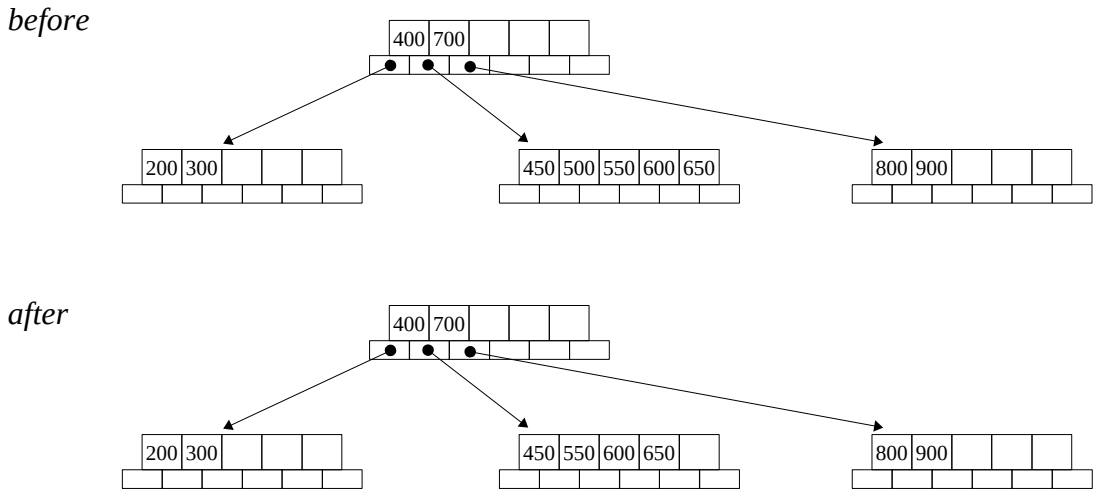


Figure 7: B-Tree Delete Example, Leaf Node

The entire B-Tree data structure is locked (Coarse-Grained B-Tree Deletion Algorithm, line 6) so no other threads are allowed to access the data structure until the changes are completed. As such, no additional locking is required as part of this operation.

4.5.2 Internal Node Deletion Operation

If a key is found in an internal node, the key cannot be removed directly as this would create an illegal imbalance in the key and child pointers. The key must be replaced with either a successor or predecessor key value from a corresponding leaf node. The algorithm checks for a potential predecessor (Coarse-Grained B-Tree Deletion Algorithm, line 32) or a potential successor (Coarse-Grained B-Tree Deletion Algorithm, line 37). Once the key in the internal node has been replaced, the predecessor or successor replacement key is in the tree twice and the redundant, second key in the leaf node must be removed. This is accomplished by updating the key to be deleted and continuing the downward traversal from the current location ensuring that only the second one will be deleted.

The internal node deletion can potentially make significant changes in the data structure. While the changes are in process, the data structure may not be in a valid, traversalable state. Other active threads are already blocked and as such no additional locking is required.

4.5.2.1 Predecessor Operation

From the deletion algorithm, (Coarse-Grained B-Tree Deletion Algorithm, line 32), case A occurs when the child that precedes the key index in the current node has more than the minimum number of keys (i.e., > ***degree***/2-1) in which case the predecessor can be found in the predecessor subtree.

The predecessor operation algorithm is shown below.

```
1.  myType bTree<myType>::getPred(treeNode<myType> *nd, int idx)  {
2.          treeNode<myType>      *cur = nd→ptrs[idx];
3.
4.          // keep moving to the right-most until leaf
5.          while (!cur→leaf)
6.                  cur = cur→ptrs[cur→cnt];
7.
8.          return   cur→keys[cur→cnt-1];
9.  }
```

Algorithm 6: Coarse-Grained B-Tree Get Predecessor Algorithm

For example, to delete the key 900 from the internal node in the below figure, a predecessor key value must be found as a replacement. This involves a right-most traversal from the current node to the right-most key value, 890 in the below example. The traversal is initiated from the key value left child pointer (as shown).

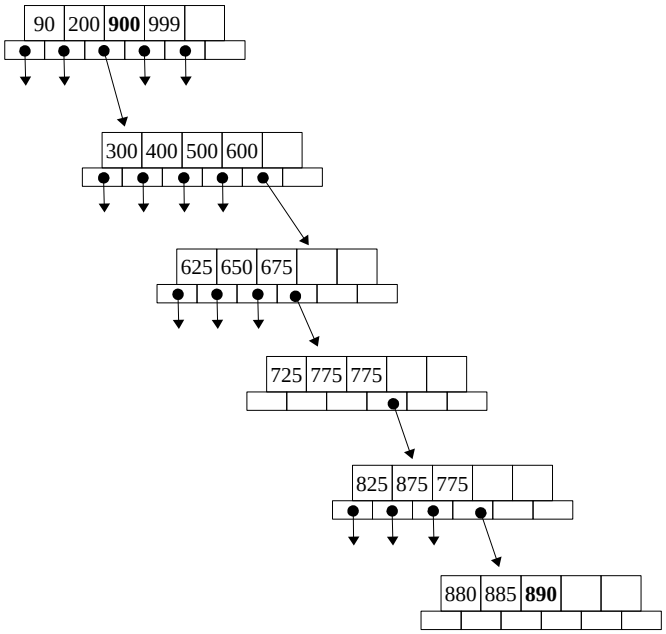


Figure 8: B-Tree Internal Delete Example, Predecessor (before)

While only six levels are shown in this example, there could be arbitrarily many.

The predecessor key value, 890 in the example figure, is used to overwrite the 900, thus removing 900 from the B-Tree as shown in the following figure. By using this predecessor value, it ensures that the child pointers are still valid and thus no further re-structuring of the B-Tree is required. Such restructuring would not be efficiently feasible given the overall complexity of the B-Tree data structure.

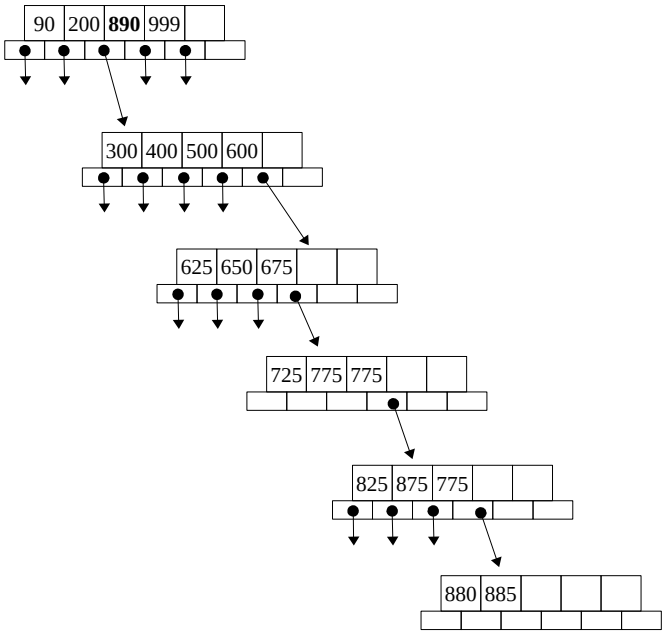


Figure 9: B-Tree Internal Delete Example, Predecessor (after)

After the replacement has been performed, the key to be deleted is changed to 890 and the deletion traversal continues from the current node to the leaf node. The continuation of the downward traversal is required in order to ensure that the leaf node still contain the minimum number of keys even after the key deletion. The duplicate key, 890 here, is deleted from the leaf node as outlined in Section 4.5.1, Key Deletion from Leaf Node.

4.5.2.2 Successor Operation

From the deletion algorithm, (Coarse-Grained B-Tree Deletion Algorithm, line 37), case B occurs when the child that precedes the key index in the current node has more than the minimum number of keys (i.e., > ***degree***/2-1) in which case the predecessor can be found in the successor subtree.

The successor algorithm is similar and is shown below.

```
1.  myType bTree<myType>::getSucc(treeNode<myType> *nd, int idx)  {
2.          treeNode<myType>      *cur = nd→ptrs[idx+1];
3.
4.          // keep moving the left until leaf
5.          while (!cur→leaf)
6.                  cur = cur→ptrs[0];
7.
8.          return   cur→keys[0];
9.  }
```

Algorithm 7: Coarse-Grained B-Tree Get Successor Algorithm

For example, to delete the key 900 from the internal node in the below figure, a sucessor key value must be found as a replacement. This involves a left-most traversal from the current node to the left-most key value. 325 in this example. The traversal is initiated from the key value right child pointer (as shown).

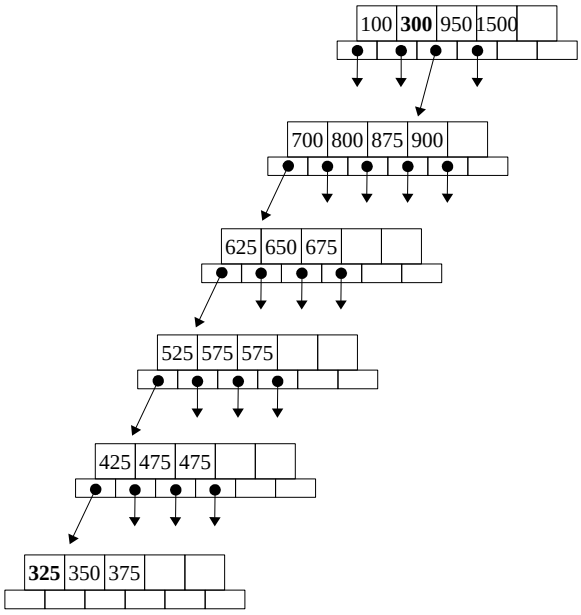


Figure 10: B-Tree Internal Delete Example, Successor (before)

While only six levels are shown in this example, there could be arbitrarily many.

The successor key value, 325 in the example figure, is used to overwrite the 300, thus removing 300 from the B-Tree as shown in the following figure. By using this predecessor value, it ensures that the child pointers are still valid and thus no further re-structuring of the B-Tree is required. Such restructuring would not be efficiently feasible given the overall complexity of the B-Tree data structure.

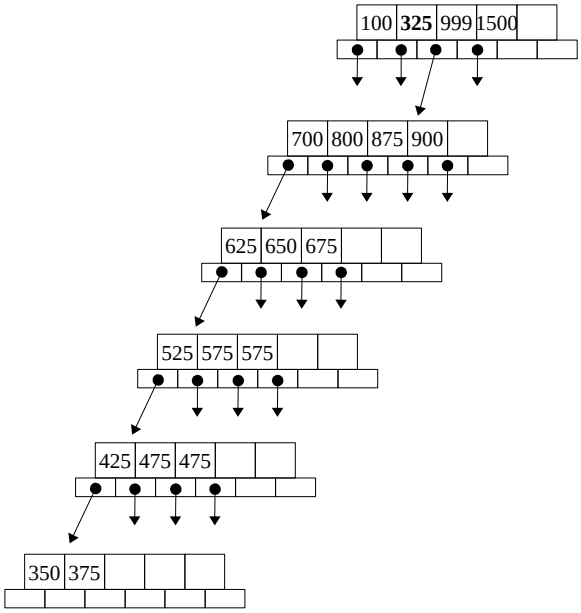


Figure 11: B-Tree Internal Delete Example, Successor (after)

After the replacement has been performed, the key to be deleted is changed to 325 and the deletion traversal continues to the leaf node. The continuation of the downward traversal is required in order to ensure that the leaf node still contain the minimum number of keys even after the key deletion. The duplicate key, 325 here, is deleted from a leaf node as outlined in section 4.5.1, Key Deletion from Leaf Node.

4.5.3 Merge Operation

From the deletion algorithm (Coarse-Grained B-Tree Deletion Algorithm, line 43), case C occurs when both the left and right siblings have the minimum number of allowable keys (each ***degree***/2-1 keys). Upon completion of the merge, the removed unnecessary right node is deleted and the memory recovered.

The coarse-grained merge nodes algorithm is shown below.

```
1.  void bTree<myType>::merge(treeNode<myType> *nd, int idx)  {
2.          treeNode<myType>     *leftChld = nd→ptrs[idx],  *rightChld = nd→ptrs[idx+1];
3.          int       t = degree / 2;
4.
5.          // put key from the current node to left child
6.          leftChld→keys[t-1] = nd→keys[idx];
7.
8.          // copy keys from right to left
9.          for (int i=0; i < rightChld→cnt; i++)
10.                 leftChld→keys[i+t] = rightChld→keys[i];
11.
12.         // copy the child pointers from right to left
13.         if (!leftChld→leaf)
14.                 for(int i=0; i <= rightChld→cnt; i++)
15.                         leftChld→ptrs[i+t] = rightChld→ptrs[i];
16.
17.         // slide keys after idx one step before
18.         for (int i=idx+1; i < nd→cnt; i++)
19.                 nd→keys[i-1] = nd→keys[i];
20.
21.         // slide child pointers one step before
22.         for (int i=idx+2; i <= nd→cnt; i++)
23.                 nd→ptrs[i-1] = nd→ptrs[i];
24.
25.         nd→ptrs[nd→cnt] = NULL;
26.
27.         // update key counts
28.         leftChld→cnt += rightChld→cnt+1;
```

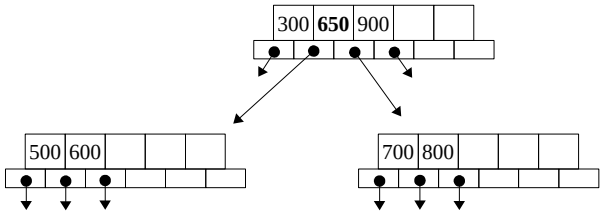
```
29.         nd→cnt--;
30.
31.         // free memory
32.         if (rightChld→keys != NULL) {
33.                 delete [] rightChld→keys;
34.                 rightChld→keys = NULL;
35.         }
36.         if (rightChld→ptrs !=NULL) {
37.                 delete [] rightChld→ptrs;
38.                 rightChld→ptrs = NULL;
39.         }
40.         delete   rightChld;
41.         rightChld = NULL;
42.
43.         return;
44. }
```

Algorithm 8: Coarse-Grained B-Tree Merge Nodes Algorithm

The case C occurs when both child nodes have the minimum allowable keys. This allows the two nodes to be merged into a single node.

To delete 650 in the following example, the two child nodes are merged and the parent key values updated (removing the 650) and the child pointers are shifted to reflect the node removal.

before



after

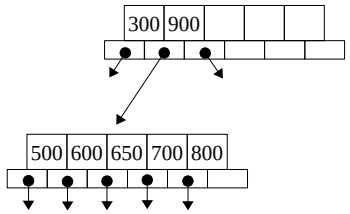


Figure 12: B-Tree Internal Node Delete Example, Merge

The removed right-node, originally containing the 700 and 800 keys, must be deleted.

4.5.4 Node Fill Operation

In accordance with the B-Tree definition, a B-Tree node should have a minimum number of keys (***degree***/2-1) in each node. During a traversal, when a child node is found to have the minimum number of keys, it will be addressed pro-actively (Coarse-Grained B-Tree Deletion Algorithm, line 58) by calling the *fill()* routine. The coarse-grained node fill operation will either add a key to the current child node by borrowing a key from a sibling node with excess keys (**> *degree***/2-1) or by merging the node with another child node that also has the minimum number of keys. The left sibling node and then right sibling node are checked to see if either has excess keys that can be borrowed or shifted into the current child node. If a borrow is possible, the parent node and applicable child pointers must be updated accordingly.

It may not be possible to borrow key if the the left or right siblings either do not exist or do not have more than the minimum number of key values. If the applicable child node is the left-most or right most child, it will only have one left or right sibling. If no borrow is possible, two nodes

are merged into a single node, thus contracting the width of the B-Tree.

The coarse-grained node fill algorithm is shown below.

```
1.  void bTree<myType>::fill(treeNode<myType> *nd, int idx)  {
2.          treeNode<myType>    *prevSib = NULL, *nextSib = NULL;
3.          int         t = degree / 2;
4.
5.          if (idx > 0)
6.                  prevSib = nd→ptrs[idx-1];
7.          if (idx < nd→cnt)
8.                  nextSib = nd→ptrs[idx+1];
9.
10.         if (idx != 0 && prevSib→cnt >= t) {
11.                 borrowFromPrev(nd, idx);
12.         } else if (idx != nd→cnt && nextSib→cnt>=t) {
13.                 borrowFromNext(nd, idx);
14.         } else {
15.                 if (idx != nd→cnt)
16.                         merge(nd, idx);
17.                 else
18.                         merge(nd, idx-1);
19.         }
20.         return;
21. }
```

Algorithm 9: Coarse-Grained B-Tree Node Fill Algorithm

Initially, the previous or left sibling node is checked. The left sibling node might not exist if the current child node is already the left-most key. If the left sibling node exists and has excess keys, the *borrowFromPrev()* routine will shift a key from the previous child node. This will move a key from the left sibling into the parent and the key from the parent into the applicable child node.

If the left sibling does not exist or does not have more than the minimum number of keys, the right sibling node is checked. The right sibling node might not exist if the current child node is

already the right-most node. If the right sibling node exists and has sufficient keys, the *borrowFromNext()* routine will shift a key from the right sibling child node. This will move a key from the right sibling into the parent and the key from the parent into the applicable child node.

In accordance with the B-Tree invariants, a child node will always have at least a right sibling or a left sibling.

If neither the left or right sibling nodes has more than the minimum number of keys, then a merge operation is possible. The merge operation is fully described in section 4.5.3, Merge Operation.

The borrow operation is fully described in the following section 4.5.4.1, Borrow Key Operation.

4.5.4.1 Borrow Key Operations

The coarse-grained borrow key operations perform the borrowing or shifting of a key from a left or right sibling with more than the minimum number of keys into the current child node. When shifting a key, the applicable child node pointers, if they exist, must also be shifted. While the shifting is in process, the data structure may not be traversable. As such, other threads must be and are blocked.

The coarse-grained borrow from previous (left sibling node) algorithm is shown below.

```
1.  void bTree<myType>::borrowFromPrev(treeNode<myType> *nd, int idx)  {
2.          treeNode<myType>      *currChld = nd→ptrs[idx],  *rightChld = nd→ptrs[idx-1];
3.
4.          // slide all keys in current child node one step
5.          for (int i=currChld→cnt-1; i >= 0; i--)
6.                  currChld→keys[i+1] = currChld→keys[i];
7.
8.          // if current child node is not a leaf, slide child pointers one step
9.          if (!currChld→leaf)
10.                 for(int i=currChld→cnt; i >= 0; i--)
11.                         currChld→ptrs[i+1] = currChld→ptrs[i];
12.
```

```
13.         // set current child's first key to keys[idx-1] from parent node
14.         currChld→keys[0] = nd→keys[idx-1];
15.
16.         // moving sibling's last child as ptrs[idx]'s first child
17.         if (!nd→leaf)
18.                 currChld→ptrs[0] = rightChld→ptrs[rightChld→cnt];
19.
20.         // moving the key from the sibling to the parent
21.         // this reduces the number of keys in the sibling
22.         nd→keys[idx-1] = rightChld→keys[rightChld→cnt-1];
23.
24.         currChld→cnt++;
25.         rightChld→cnt--;
26.
27.         return;
28.  }
```

Algorithm 10: Coarse-Grained B-Tree Borrow from Previous Algorithm

As shown in the following figure with a ***degree*** of 6, the current child has the minimum number of keys and the left sibling has more than the minimum number of keys which meets the conditions for a borrow from the left sibling node operation. After the borrow, both nodes will still meet the basic B-Tree invariant for minimum number of keys.

For example, given the following initial configuration, the key 550 can be borrowed from the left sibling.

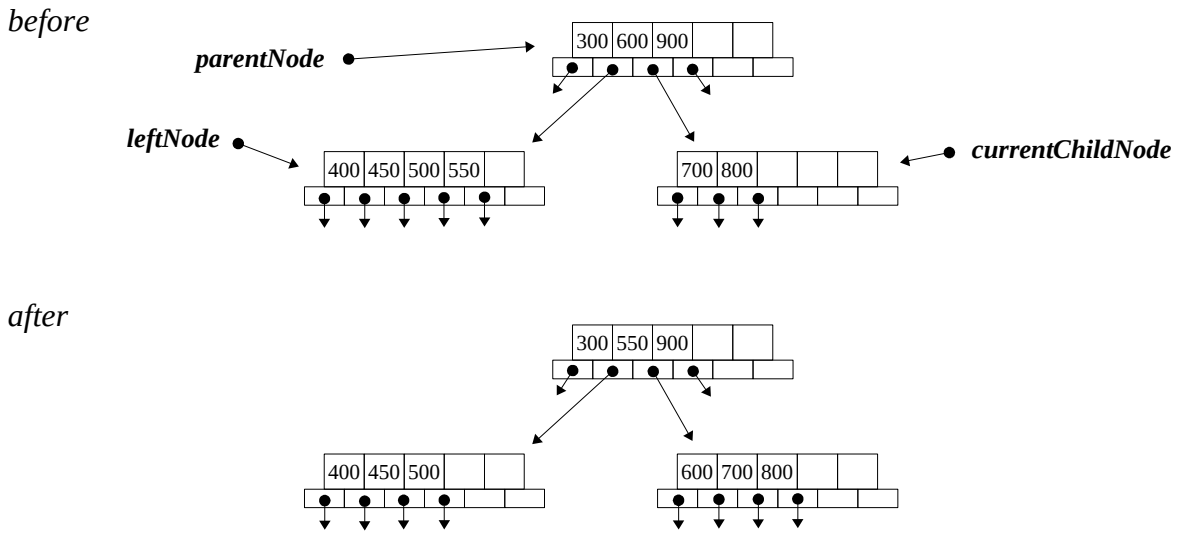


Figure 13: B-Tree Borrow From Previous Operation

In order to ensure the B-Tree invariants are maintained, the key must be shifted through the parent node. The keys, and if needed the child pointers, of the current child are shifted one place to the right in order to make room for the new key. The current key, and if needed the associated child pointer, from the parent is moved into the current child node (600 in this example). Finally, the last or right most key of the left sibling node is move into the parent, replacing the previous key value which was moved down to the current child node. The key counts are updated for both the left sibling and current child nodes.

The coarse-grained borrow from next (right sibling node) algorithm is shown below.

```
1.  void bTree<myType>::borrowFromNext(treeNode<myType> *nd, int idx)  {
2.          treeNode<myType>      *currChld = nd→ptrs[idx], *rightChld = nd→ptrs[idx+1];
3.
4.          // copy key from parent to current child node
5.          currChld→keys[currChld→cnt] = nd→keys[idx];
6.
7.          // current child insert as the last into left child
8.          if (!currChld→leaf)
```

```
9.                  currChld→ptrs[currChld→cnt+1] = rightChld→ptrs[0];
10.
11.         // first key from right child is inserted into parent
12.         nd→keys[idx] = rightChld→keys[0];
13.
14.         // slide keys in sibling one step
15.         for (int i=1; i < rightChld→cnt; i++)
16.                 rightChld→keys[i-1] = rightChld→keys[i];
17.
18.         // slide child pointers one step
19.         if (!rightChld→leaf)
20.                 for(int i=1; i <= rightChld→cnt; i++)
21.                         rightChld→ptrs[i-1] = rightChld→ptrs[i];
22.
23.         // update counts
24.         currChld→cnt++;
25.         rightChld→cnt--;
26.
27.         return;
28. }
```

Algorithm 11: Coarse-Grained B-Tree Borrow from Next Algorithm

As shown in the following figure, the current child has the minimum number of keys and the right sibling has more than the minimum number of keys which meets the conditions for a borrow from the right sibling node operation. After the borrow, both nodes will still meet the basic B-Tree invariant for minimum number of keys.

For example, given the following configuration, the key 700 can be borrowed from the right sibling.

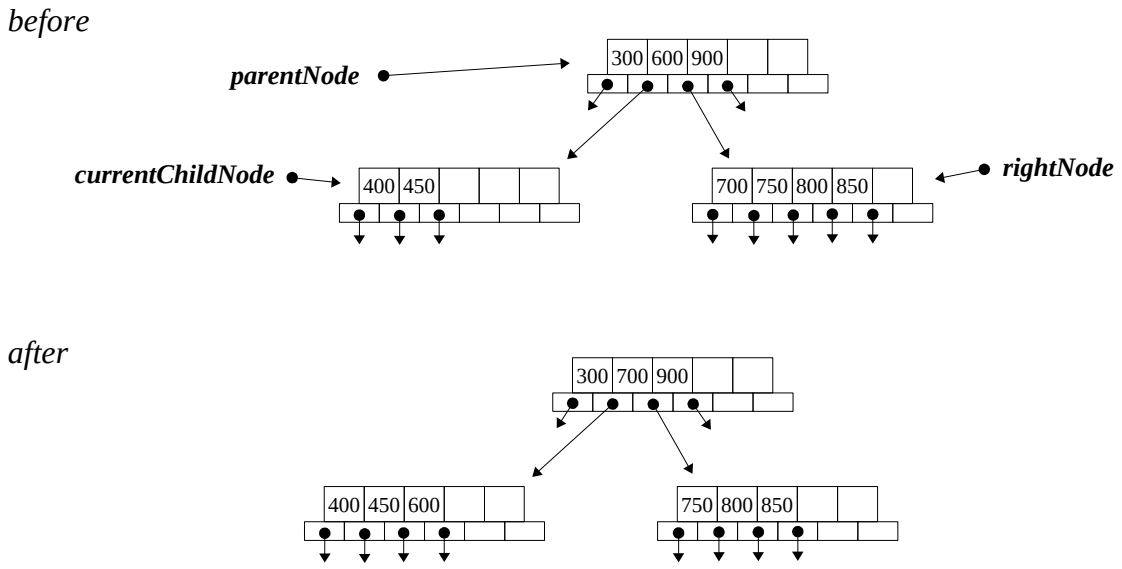


Figure 14: B-Tree Borrow From Next Operation

As per the B-Tree invariants, the key must be shifted through the parent node. The current key, and if needed the associated child pointer, from the parent are copied into the current child node (600 in this example). The first key of the right child node is moved into the parent (700 in this example), replacing the key previously moved down. The keys, and if needed the child pointers, of the right child node are shifted one place to the left in order fill the hole from the key that was moved up. The key counts are updated for both the current child and right sibling nodes.

4.6 Correctness

This section addresses correctness issues for the algorithms.

4.6.1 Deadlock Freedom

The single global lock provides the necessary functionality to ensure freedom from deadlock. The lock is either acquired or the process is blocked waiting for that single lock. The lock acquire operation is limited to the initial steps of the insert, search, and delete operations. None of the associated sub-functions perform any lock operations. As no other resources are sought by

the thread, once the single lock is acquired the process can complete is intended operation thus ensuring deadlock freedom.

4.6.2 Starvation Freedom

In a high contention environment, many threads may be competing for the single global lock. The final selected thread that obtains the lock is chosen arbitrarily. This allows the possibility that under a constant sustained load a specific thread may be repeatedly not be chosen indefinitely. When the load is decreased the thread will be selected.

5.0 Chapter 5,

Fine-Grained B-Tree Algorithms

5.0 Chapter 5, Fine-Grained B-Tree Algorithms

This chapter presents the fine-grained B-Tree insert, search, and delete algorithms. The approach for fine-gained locking is to only lock a minimal subset of the data structure. By locking a limited subset of the data structure, other concurrent operations in different parts of the data structure can be processed simultaneously. The proactive node splitting approach is used which eliminates the recursive element and ensures that nodes are not traversed twice. This also eliminates use of additional stack space.

The repeated lock acquire and lock release overhead during the traversal increases the run-time overhead. Due to this overhead, in no or low contention environments, the fine-grained approach is slower than the coarse-grained approach. A fine-grained implementation is more appropriate in a concurrent environment with a high degree of contention.

The fine-grained implementation presented is fully polymorphic using the C++11/17 mutex primitives. There are no known existing C++11/17 fine-grained locking reference implementations. The coarse-gained implementation is used as a performance base-line.

5.1 Main Ideas

This section presents a conceptual overview of the B-Tree fine-grained algorithms and a summary of the specific locking strategy. As detailed in subsequent sections, a sentinel root node is used. All nodes include a field for a node lock (node specific mutex). This node specific lock allows nodes to be locked individually providing the fundamental technical functionality for

the fine-grained approach. All operations always start from the root. As the initial step, all threads must compete for the root node lock. The use of a sentinel node ensures that even operations on an empty B-Tree will compete for the root lock in a consistent manner.

A parent then child lock-coupling technique [30] is used requiring that a thread must acquire a parent node lock before being allowed to attempt a lock acquire for a child node in a strictly enforced parent then child manner. As a thread proceeds on its downward traversal, it will release the previous node lock (above the parent). This allows a thread to release node locks that are no longer necessary. For example, once the root lock is released, another thread may obtain the root lock and proceed down a different traversal path. If the subsequent thread is attempting to traverse the same path, it will be blocked from overtaking. For high contention environments, this will allow more concurrency. For no or minimal contention environments, the additional lock acquire/release requirements will increase overhead.

Broadly, the insertion algorithm traverses downward and inserts all new keys into a leaf node. During the downward traversal, if a child node is found to be full (i.e., already has the maximum allowed keys), it is split into two nodes. The split node process will move one of the keys from the child into the parent. The proactive split node process starts at the root. If the root is full, it will be split which increases the height of the tree. This ensures that if a full child node is found, the parent will have room for the additional key. The specific technical details of the root and internal node split operations are detailed in the following sections. When the insertion operation finds the applicable leaf node and it is not full it will insert the new key into the appropriate location and terminate. If that leaf node is full, the leaf must be split into two nodes. Once the split is performed and the middle child key is moved up to the parent, the new key is inserted into the appropriate new child node (left or right). Once the new key is placed in the leaf into its proper position, the process is complete. In this manner, successive insertion operations gradually move keys up from leaf to root in the B-Tree. Since the keys only move one level at a time (to parent from child), the fine-grained locking is sufficient.

Since the modify operations change the data structure, any search threads must also acquire and release the locks in the same parent then child manner during their traversals. This ensures that

any in-process modifications do not impact the search threads.

The deletion process starts from the root and traverses downward searching for the key to be deleted. A proactive node fill process is used that will continue downward (no changes), borrow a key, or merge nodes as required. The merge operation combines two sibling nodes that both contain the minimum number of keys allowed (***degree***-1/2) and moves a key down one level from the parent to the child. To ensure that this is possible and does not violate the basic invariants, the child node must have one key more than the minimum. During the traversal, if a node is found to have the minimum number of keys allowed, and its right or left sibling nodes, if they exist, also have the minimum number of allowed keys, only then are the two nodes merged into a single node. If a sibling has excess keys (i.e., greater than the minimum allowed), a key can be borrowed from a left or right sibling. The borrow operation will ensure that the child on the direct traversal path has at least the minimum number of keys plus one which may be required for a subsequent merge operation one level down. The borrow operation moves a key from the sibling (left or right) up to the parent and a key from the parent is moved to the target child node. The associated pointers are shifted as part of this operation. A merge or borrow operation may move a key one level (from child to parent and/or parent to child). While node locks are required on the parent and child, the nodes above the parent node are released for other threads to potentially acquire. This allows other threads to proceed as long as they are on different paths.

If the key to be deleted is found in a leaf node, the key is deleted and the process terminates. The proactive node fill process ensures that the leaf node will have the minimum plus one number of keys so that deleting a key will not violate the B-Tree invariants.

If the key to be deleted is in an internal node, it must be replaced. Simply removing the key would result in a subtree being disconnected from the data structure. The key must be replaced with either the first key from the left-most leaf node, the predecessor node, or the last key from the right-most leaf node, the successor node. During the successor or predecessor operations, all nodes between the current parent/child and the final successor/predecessor node must be locked. This ensures that other threads cannot make changes that would alter the selection of the

successor/predecessor key value. This additional lock acquire and lock release does create additional overhead when required. These operations are described in detail in the following sections.

5.2 Fine-Grained B-Tree Configuration

This section summarizes the fine-grained B-Tree node configuration with regard to the specific structure fields and the sentinel root node.

5.2.1 Fine-Grained B-Tree Node Definition

The specific node definition includes fields for the count, leaf status (true/false), a pointer to an array of keys, and a pointer to an array of pointers (for the child pointers). In addition to the fields of the the coarse-grained node definition (Figure 3, Coarse-Grained B-Tree Node Configuration), a mutex for the specific node is included. This is the locking mechanism for the individual node. Based on the required changes, additional nodes may need to be locked.

```
struct treeNode
{
      myType             *keys;      // key values
      treeNode<myType>   **ptrs;     // child ptrs
      int                cnt;        // current number of keys
      bool               leaf;       // true if leaf, else false
      mutex              nodeLock;   // lock for node
};
```

Figure 15: Fine-Grained B-Tree Node Configuration

The ***myType*** is the passed type for the polymorphic implementation.

In addition to the node structure, class global variables include the declared degree and root pointer. There are no global mutexes used.

5.2.2 Sentinel Node

A sentinel node is used in this implementation. In the context of a fine-grained implementation,

this eliminates some potential complications associated with the insertion algorithm. By using a sentinel node, the simultaneously executing threads can directly compete for the root node lock. If the root did not exist, all threads would be required to check for, and if necessary, create the root node. Once the root is created, such a check could not be skipped which would add extra overhead to the insertion algorithm. The sentinel node is used to store key values so no additional space is required for a non-empty B-Tree.

5.3 Insertion Algorithm

The fine-grained B-Tree insertion algorithm uses a node-by-node locking and unlocking downward traversal algorithm with proactive splitting. All keys are added to a leaf node. During the traversal, nodes discovered containing the maximum allowed keys (based on the instantiated degree) are pro-actively split into two nodes. This ensures that the node above a leaf node is not full. The term full used to to to denote when a node has the maximum allowed number of keys based on the degree. If the insertion into a leaf node fills that leaf node, it would be split by a subsequent insertion operation. The split node operation will bring a key value up one level in the B-Tree. The proactive splitting will ensure that there is always key space in a parent node for a key from a child node that might be required to be moved up one level (from child to parent).

The initial step is locking the root node. The thread that successfully obtains the root lock will initially check if a root split operation is required. If the root is full, the root split will be performed. The specific details of the root split algorithm are detailed in Section 5.3.2, Split Root Node Operation. At this point, all other threads are blocked on the root node.

As the thread continues its downward traversal, the root lock is released allowing another thread to obtain the root lock and proceed. The node-by-node locking blocks other threads on the same traversal path from overtaking any preceding threads. Once past the root node, if the subsequent thread follows a different traversal path, it will not be blocked by the previous thread. Since the two threads are on different paths, both threads can simultaneously modify the different sections of the data structure. This may apply even for two threads on the same path if the initial thread's intended modifications are further down in the B-Tree than the subsequent thread. Again, the

data structure modifications can occur simultaneously. The simultaneous modifications in different sections of the data structure is the key advantage of fine-grained locking.

Simultaneous operations are not always possible. For example, if two threads start and a second thread's intended modifications are beyond the first thread's modifications, the second will be blocked until the first thread's operations are fully completed.

The fine-grained B-Tree insertion algorithm is shown below.

```
1.  void bTree<myType>::insert(myType key, int th)  {
2.          treeNode<myType>    *curr, *prev, *old, *currChld;
3.          int         idx, nIdx;
4.          bool        insertDone = false;
5.
6.          root->nodeLock.lock();
7.          curr = root;
8.
9.          while (!insertDone) {
10.                 // if root full, must split
11.                 if (root→cnt == degree-1)
12.                         splitRootNode(root);
13.
14.                 //  traverse downward to find applicable leaf node,
15.                 //          split full nodes along the way
16.
17.                 prev = NULL;
18.                 while (!curr→leaf) {
19.
20.                         // find next ptr based on key
21.                         idx = 0;
22.                         while ((idx < curr→cnt) && (key > curr→keys[idx]))
23.                                 idx++;
24.
25.                         curr→ptrs[idx]→nodeLock.lock();
26.                         currChld = curr→ptrs[idx];
27.
28.                         // if that child is full, split it.
29.                         if (currChld != NULL && currChld→cnt == (degree-1)) {
```

```
30.                                 splitChildNode(curr, currChld);
31.
32.                                 // re-find where key goes (due to split)
33.                                 nIdx = 0;
34.                                 while ((nIdx < curr→cnt) && (key > curr→keys[nIdx]))
35.                                         nIdx++;
36.
37.                                 if (idx != nIdx) {
38.                                         curr→ptrs[nIdx]→nodeLock.lock();
39.                                         curr→ptrs[idx]→nodeLock.unlock();
40.                                         idx = nIdx;
41.                                 }
42.                         }
43.
44.                         old = prev;
45.                         prev = curr;
46.
47.                         curr = curr→ptrs[idx];
48.                         if (old != NULL)
49.                                 old→nodeLock.unlock();
50.                 }
51.
52.                 //  insert key into non-full leaf node, slide existing keys
53.                 idx = curr→cnt - 1;
54.                 while (idx >= 0 && curr→keys[idx] > key) {
55.                         curr→keys[idx+1] = curr→keys[idx];
56.                         idx--;
57.                 }
58.
59.                 // insert the new key at appropriate location
60.                 curr→keys[idx+1] = key;
61.                 curr→cnt = curr→cnt + 1;
62.
63.                 insertDone = true;
64.                 if (prev != NULL)
65.                         prev→nodeLock.unlock();
66.                 curr→nodeLock.unlock();
67.         }  // end main while loop
68. }
```

Algorithm 12: Fine-Grained B-Tree Insertion Algorithm

Due to the node changes during the insert operation, the data structure may not be in a valid, traversable state until the applicable insertion is fully completed. As such, other threads, including search threads, are blocked from overtaking this thread.

Each of the major operations of the insertion is explained in the following subsections.

5.3.1 Leaf Node Insertion

For insertion into a non-full leaf node, the keys may need to be shifted in order to place the new key into the appropriate position. The leaf will have room for the new key due to the proactive splitting since the node would already have been split if it was full. The process is fully described in Section 4.3.1, Leaf Node Insertion.

In this algorithm, the parent node and the current leaf node are locked during the traversal. Once the key is inserted into its appropriate location, the locks are released (Fine-Grained B-Tree Insertion Algorithm, lines 64 and 66).

5.3.2 Split Root Operation

When the root node has the maximum number of allowable keys, it will be split. Special care must be taken with the root node since another thread may already be locked on the root node. Since the current thread holds the lock for the root, that node must be maintained as the root and the new nodes be created as children to the current root. The B-Tree will grow in height by adding a level between the current root and the original set of child nodes. In accordance with the B-Tree invariants, a single key is allowed to be in the root node. When needed, the fine-grained split root operation is initiated (Fine-Grained B-Tree Insertion Algorithm, line 11).

The fine-grained split root operation algorithm is shown below.

```
1.  void bTree<myType>::splitRootNode(treeNode<myType> *currRoot)  {
2.          int       siz;
3.          treeNode<myType>      *lftSib=NULL, *rhtSib=NULL;
4.
5.          // create new sibling nodes
```

```
6.          lftSib = makeNewNode();
7.          lftSib→leaf = currRoot→leaf;
8.          rhtSib = makeNewNode();
9.          rhtSib→leaf = currRoot→leaf;
10.
11.         // set size based on degree
12.         siz = (degree-1) / 2;
13.
14.         // copy the first (degree-1)/2 keys of current root left sibling
15.         for (int j = 0; j < siz; j++)
16.                 lftSib→keys[j] = currRoot→keys[j];
17.
18.         // copy the first degree/2 ptrs of current root over to left sibling
19.         if (!currRoot→leaf)
20.                 for (int j=0; j < degree/2; j++)
21.                         lftSib→ptrs[j] = currRoot→ptrs[j];
22.
23.         // copy the last (degree-1)/2 keys of current root to right sibling
24.         for (int j = 0; j < siz; j++)
25.                 rhtSib→keys[j] = currRoot→keys[j+siz+1];
26.
27.         // copy the last degree/2 ptrs of current root over to right sibling
28.         if (!currRoot→leaf)
29.                 for (int j=0; j < degree/2; j++)
30.                         rhtSib→ptrs[j] = currRoot→ptrs[j+degree/2];
31.
32.         // set sibling sizes
33.         lftSib→cnt = siz;
34.         rhtSib→cnt = siz;
35.
36.         // update root, shift key
37.         currRoot→leaf = false;
38.         currRoot→cnt = 1;
39.         currRoot→keys[0] = currRoot→keys[siz];
40.         for (int i=1; i<degree-1; i++)
41.                 currRoot→keys[i] = 0;
42.
43.         // set root pointers
44.         currRoot→ptrs[0] = lftSib;
45.         currRoot→ptrs[1] = rhtSib;
46.         for (int i=2; i<degree; i++)
```

```
47.                 currRoot→ptrs[i] = NULL;
48.  }
```

Algorithm 13: Fine-Grained Root Split Algorithm

The split root operation is only initiated after the lock is obtained and as such, no additional locking is required. If another thread currently holds the root lock, it will not be released until that thread has proceeded in its traversal and no longer needs the root.

The initial configuration for a full root of degree 6 is shown for reference.

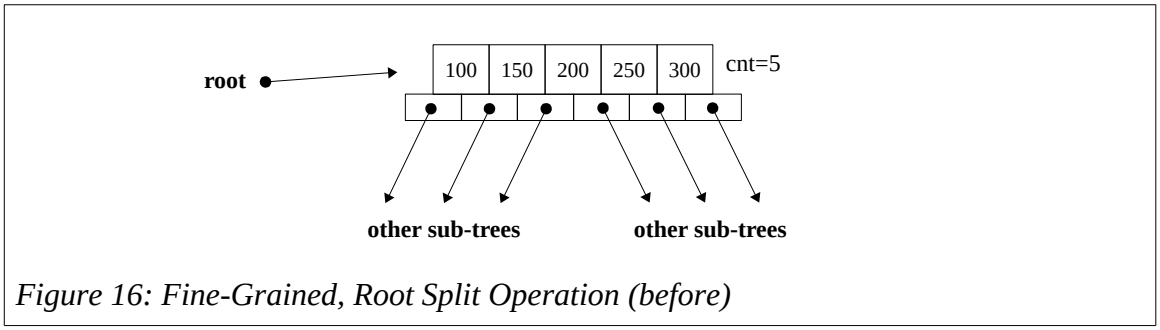


Figure 16: Fine-Grained, Root Split Operation (before)

Once the root lock is obtained, if the root is full, it will be split as shown in the following figure.

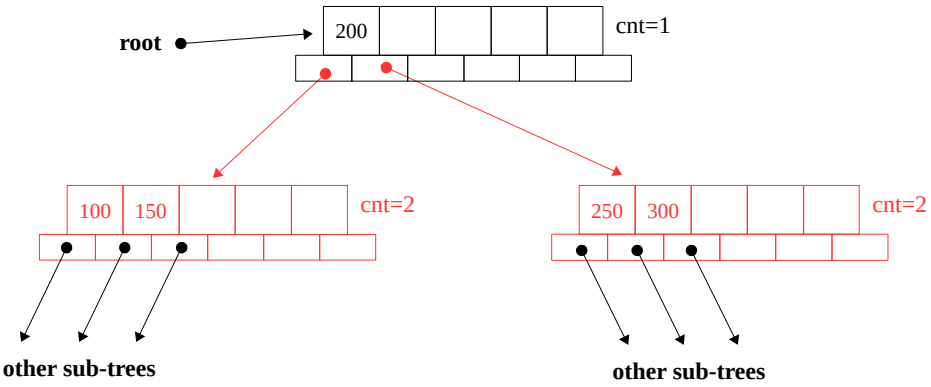


Figure 17: Fine-Grained, Root Split Operation (after)

The root lock ensures that if other threads are attempting to obtain a root lock, they are blocked until all potential changes are completed. If this locking does not occur, it would be possible for other threads to simultaneously alter the root, possibly corrupting the B-Tree.

5.3.3 Split Child Operation

A split operation for a full non-root node is referred to as split-child node operation. Checking for full non-root nodes is done only after the root has been checked. If the root was full, it will have been split and the algorithm proceeds with the assurance of a non-full root. Starting from the non-full root, only the child node directly in the downward traversal path is checked. Other child nodes not in the traversal path are not checked. If that child node is full, the split child node operation is performed which will move up a key from the child to the parent. This is why the parent node, even if it is the root node, is not allowed to be full. Other nodes, not in the current traversal path, are not checked since they are not directly impacted or altered.

The fine-grained split child node algorithm is shown below.

```
1.  void bTree<myType>::splitChildNode(treeNode<myType> *parnt,
2.                                     treeNode<myType> *chld)  {
3.          int     siz, idx;
```

```
4.          treeNode<myType>      *sib = NULL;
5.
6.          // create a new sibling node
7.          sib = makeNewNode();
8.          sib→leaf = chld→leaf;
9.
10.         // set size based on degree
11.         siz = (degree-1) / 2;
12.         sib→cnt = siz;
13.
14.         // copy the last (degree-1)/2 keys of child to sibling
15.         for (int j=0; j < siz; j++)
16.                 sib→keys[j] = chld→keys[j+siz+1];
17.
18.         // copy the last degree/2 ptrs of child over to sibling
19.         if (!chld→leaf) {
20.                 for (int j=0; j < degree/2; j++) {
21.                         sib→ptrs[j] = chld→ptrs[j+degree/2];
22.                         chld→ptrs[j+degree/2] = NULL;
23.                 }
24.         }
25.
26.         // reduce the number of keys in child
27.         chld→cnt = chld→cnt / 2;
28.
29.         // find where new key is going in parent
30.         idx = 0;
31.         while ((idx < parnt→cnt) && (parnt→keys[idx] < chld→keys[siz]))
32.                 idx++;
33.
34.         // slide parent child ptrs over to make room for new child
35.         for (int j = parnt→cnt; j >= idx+1; j--)
36.                 parnt→ptrs[j+1] = parnt→ptrs[j];
37.
38.         // add new child to parent
39.         parnt→ptrs[idx+1] = sib;
40.
41.         // slide parent child keys over to make room for new key
42.         for (int j = parnt→cnt-1; j >= idx; j--)
43.                 parnt→keys[j+1] = parnt→keys[j];
44.
```

```
45.         // copy middle key of child to parent
46.         parnt→keys[idx] = chld→keys[siz];
47.
48.         // increment count of keys in this node
49.         parnt→cnt += 1;
50.  }
```

Algorithm 14: Fine-Grained Split Child Node Algorithm

The basic algorithm contains only minor updates from the algorithm and process presented in Section 4.3.3, Split Child Node Operation. The data structure changes are only allowed after the parent and child node locks are acquired, thus blocking other threads from potentially interfering with the operation.

5.4 Search Algorithm

During the insertion or deletion operations, the data structure may not be in a valid, traversalable state due to the in-process node configuration changes. As such, the fine-grained search requires acquisition and release of the node-locks as the search traversal proceeds.

The fine-grained search algorithm is shown below.

```
1.   bool bTree<myType>::search(myType key) const  {
2.          treeNode<myType>      *curr, *prev;
3.          int       i;
4.
5.          root→nodeLock.lock();
6.          curr = root;
7.          if (curr == NULL)
8.                 return   false;
9.
10.         while (curr != NULL) {
11.
12.                // find the first key greater than or equal to key
13.                i = 0;
```

```
14.                 while ( (i < curr→cnt) && (key > curr→keys[i]) )
15.                         i++;
16.
17.                 // see if key is found
18.                 if (i < curr→cnt && curr→keys[i] == key) {
19.                         curr→nodeLock.unlock();
20.                         return   true;
21.                 }
22.
23.                 prev = curr;
24.                 curr = curr→ptrs[i];
25.                 curr→nodeLock.lock();
26.                 prev→nodeLock.unlock();
27.         }
28.         return   false;
29. }
```

Algorithm 15: Fine-Grained B-Tree Search Algorithm

If a search thread is following the same traversal as another thread performing a search, deletion, or insertion operation, the search thread will be blocked from overtaking. If at some point the search thread traversal path differs from a preceding thread, it will no longer be blocked. There is overhead associated with the lock acquire and lock release operations. In no or low contention environments this reduces performance as compared to a coarse-gained approach.

5.5 Deletion Algorithm

The deletion operation will remove a key from the B-Tree. The traversal is initiated from the root. Like the insertion, a proactive approach is used. Specifically, as the thread traversal proceeds, any node on the direct traversal path with a key count at the minimum (***degree***/2-1), will be adjusted or merged as appropriate. The adjustment may borrow a key from an adjacent node. If both the right and left siblings also have the minimum number of keys, a merge operation is performed. Multiple delete operations will gradually contract the width and height of the B-Tree. Since the root is not subject to the minimum key requirement, as keys in the root are removed, the B-Tree will shrink in height.

Deletion of a key in a leaf node can be performed directly with no impacts on other nodes. The parent and leaf nodes are locked during the traversal and no additional locking is required.

Deletion of a key in an internal node may require more extensive modifications to other, possibly non-locked nodes. Simply removing the key directly would invalidate or lose some of the child pointers. A replacement must be identified. This process is fully explained in Section 4.5.2, Internal Node Deletion Operation.

The fine-grained deletion algorithm is shown below.

```
1.  void bTree<myType>::remove(myType key)  {
2.          treeNode<myType>     *curr, *prev=NULL, *old=NULL, *tmpChld;
3.          int       idx, pred, succ, t = degree / 2;
4.          bool      wasLastChild;
5.
6.          if (root == NULL) {
7.                  cout << "Tree is empty." << endl;
8.                  return;
9.          }
10.
11.         root->nodeLock.lock();
12.         prev = NULL;
13.         curr = root;
14.         while (curr != NULL) {
15.
16.                 // find appropriate index for key
17.                 idx=0;
18.                 while ((idx < curr→cnt) && (curr→keys[idx] < key))
19.                         idx++;
20.
21.                 // does this node contain the key?
22.                 if ((idx < curr→cnt) && (curr→keys[idx] == key)) {
23.                         if (curr→leaf) {
24.                                 // leaf node, just remove key, slide keys backwards
25.                                 for (int i=idx+1; i < curr→cnt; i++)
26.                                         curr→keys[i-1] = curr→keys[i];
27.                                 curr→cnt--;
28.                                 if (prev != NULL)
```

```
29.                                     prev→nodeLock.unlock();
30.                             curr→nodeLock.unlock();
31.                             curr = NULL;
32.                     } else {
33.                             // key found, but not leaf
34.                             auto tmp = curr->ptrs[idx];
35.                             auto tmp1 = curr->ptrs[idx+1];
36.
37.                             old = prev;
38.                             prev = curr;
39.                             if (tmp→cnt >= t) {                     // case A
40.                                     pred = getPred(curr, idx);
41.                                     curr→keys[idx] = pred;
42.                                     key = pred;
43.                                     curr = curr→ptrs[idx];
44.                                     curr→nodeLock.lock();
45.                             } else if (tmp1→cnt >= t) {             // case B
46.                                     succ = getSucc(curr, idx);
47.                                     curr→keys[idx] = succ;
48.                                     key = succ;
49.                                     curr = curr→ptrs[idx+1];
50.                                     curr→nodeLock.lock();
51.                             } else {                                // case C
52.                                     tmp→nodeLock.lock();
53.                                     tmp1→nodeLock.lock();
54.                                     merge(curr, idx);
55.                                     curr = curr→ptrs[idx];          // already locked
56.                             }
57.                             if (old != NULL)
58.                                     old→nodeLock.unlock();
59.                     }
60.             } else {
61.                     // key not found yet, keep going down...
62.
63.                     // check for not in tree
64.                     if (curr→leaf) {
65.                             cout << "Key " << key << " does not exist in the tree." << endl;
66.                             if (prev != NULL)
67.                                     prev→nodeLock.unlock();
68.                             curr→nodeLock.unlock();
69.                             return;
```

```
70.                         } else {
71.                                 wasLastChild = ( (idx==curr→cnt)? true : false );
72.
73.                                 // if the child has less than t keys, fill child
74.                                 curr→ptrs[idx]→nodeLock.lock();
75.                                 if (curr→ptrs[idx]→cnt < t) {
76.                                         fill(curr, idx);
77.                                 }
78.
79.                                 if (curr→ptrs[idx] != NULL)
80.                                         curr→ptrs[idx]→nodeLock.unlock();
81.
82.                                 old = prev;                     // keep traversing downward
83.                                 prev = curr;
84.
85.                                 if ( wasLastChild && (idx > curr→cnt) )
86.                                         curr = curr→ptrs[idx-1];
87.                                 else
88.                                         curr = curr→ptrs[idx];
89.
90.                                 curr→nodeLock.lock();
91.                                 if (old != NULL)
92.                                         old→nodeLock.unlock();
93.                         }
94.                 }
95.         } // end main while
96.
97.         // if root has no keys, tree shrinks -> move up info from child
98.         root→nodeLock.lock();
99.
100.        // if root is leaf with no keys, empty tree
101.        if (root→cnt == 0 && root→leaf)
102.                return;
103.
104.        if (root→cnt == 0) {
105.                root→ptrs[0]→nodeLock.lock();
106.
107.                // copy keys up
108.                tmpChld = root→ptrs[0];
109.                for (int i=0; i < tmpChld→cnt; i++)
110.                        root→keys[i] = tmpChld→keys[i];
```

```
111.
112.                    // copy child pointers up
113.                    if (!tmpChld→leaf)
114.                            for(int i=0; i <= tmpChld→cnt; i++)
115.                                    root→ptrs[i] = tmpChld→ptrs[i];
116.                    root→cnt = tmpChld→cnt;
117.                    root→leaf = tmpChld→leaf;
118.
119.                    if (tmpChld→keys != NULL)
120.                            delete [] tmpChld→keys;
121.                    if (tmpChld→ptrs != NULL)
122.                            delete [] tmpChld→ptrs;
123.                    delete   tmpChld;
124.            }
125.            root→nodeLock.unlock();
126.}
```

Algorithm 16: Fine-Grained B-Tree Deletion Algorithm

The leaf node deletion and internal node deletion are addressed in the following sections.

5.5.1 Key Deletion from Leaf Node

Both the parent and leaf nodes are locked during the traversal. Since they are locked, the leaf node deletion can be performed as described in Section 4.5.1, Key Deletion from Leaf Node. Once the leaf-node key deletion is completed, the nodes are unlocked.

5.5.2 Internal Node Deletion Operation

Keys that are found in an internal (non-leaf) node cannot be removed directly since such a removal would orphan the subtree associated with the excess child pointer. A replacement is obtained from either a left-most predecessor or right-most successor node. This creates a temporary duplicate key. The second duplicate key, which will be in a leaf node, must be deleted by the current thread. Other threads must be blocked during such an operation. Since this can only occur in the current subtree, others threads operating in a different part of the data structure can proceed unimpeded.

If a search thread coincidentally is looking for the chosen predecessor or successor key, it will be blocked from overtaking the delete thread. If that search thread was ahead of the deleted thread, the delete thread is blocked from overtaking the search thread which ensures that the search will be able to find the appropriate key without interfering from the delete or other insertion threads.

The predecessor and successor operations are detailed in the following section.

5.5.2.1 Predecessor and Successor Operations

The predecessor and successor operations are similar to the coarse-gained successor and predecessor operations. However, the nodes must be locked and unlocked during the successor and predecessor traversals.

The fine-grained predecessor operation algorithm is shown below.

```
1.  myType bTree<myType>::getPred(treeNode<myType> *nd, int idx)  {
2.          treeNode<myType>      *cur = nd→ptrs[idx], *pre = NULL;
3.
4.          // keep moving to the right-most until leaf
5.          while (!cur→leaf) {
6.                  pre = cur;
7.                  cur = cur→ptrs[cur→cnt];
8.                  cur→nodeLock.lock();
9.                  pre→nodeLock.unlock();
10.         }
11.         cur→nodeLock.unlock();
12.
13.         // return the last key of the leaf
14.         return   cur→keys[cur→cnt-1];
15. }
```

Algorithm 17: Fine-Grained Get Predecessor Algorithm

The fine-grained successor operation algorithm is shown below.

```
1.  myType bTree<myType>::getSucc(treeNode<myType> *nd, int idx)  {
2.          treeNode<myType>     *cur = nd→ptrs[idx+1], *pre = NULL;
3.
4.          // keep moving the left-most until leaf
5.          while (!cur→leaf) {
6.                  pre = cur;
7.                  cur = cur→ptrs[0];
8.                  cur→nodeLock.lock();
9.                  pre→nodeLock.unlock();
10.         }
11.         cur→nodeLock.unlock();
12.
13.         // return the first key of the leaf
14.         return   cur→keys[0];
15. }
```

Algorithm 18: Fine-Grained B-Tree Get Successor Algorithm

Once the locks are acquired, the predecessor operation proceeds as described in Section 4.5.2.1, Predecessor Operation or the successor operation proceeds as described in Section 4.5.2.2, Successor Operation.

The locking during the predecessor or successor traversals is required in order to ensure no interference from other threads traversing the data structure ahead of the current thread. If this is not performed, a race conditions could be created.

For example, a possible race condition may occur between multiple simultaneously executing delete threads. To illustrate, assume that thread, ta_{del}, is performing a delete operation for key k_a and that thread, tb_{del}, performing a delete operation for key k_b. Further assume that k_a, in thread ta_{del} is being deleted and is in an internal node, that the key k_b, in thread tb_{del} is being deleted and is in a leaf node, and that k_b happens to be the successor key value for thread ta_{del}. If thread ta_{del} overtakes tb_{del} due to an inopportune preemption, thread tb_{del} could select k_b as its predecessor or

successor key value. As thread tb_{del} resumes, it will successfully delete k_b from the leaf. As thread ta_{del} proceeds, it will replace k_a with k_b successfully deleting k_a and continue in an attempt to delete is predecessor or successor key value, k_b which has already been deleted generating an error. While this error could be ignored, the key k_b which was successfully deleted by tb_{del} is still in the data structure.

The potential two delete thread race condition is summarized in the following figure.

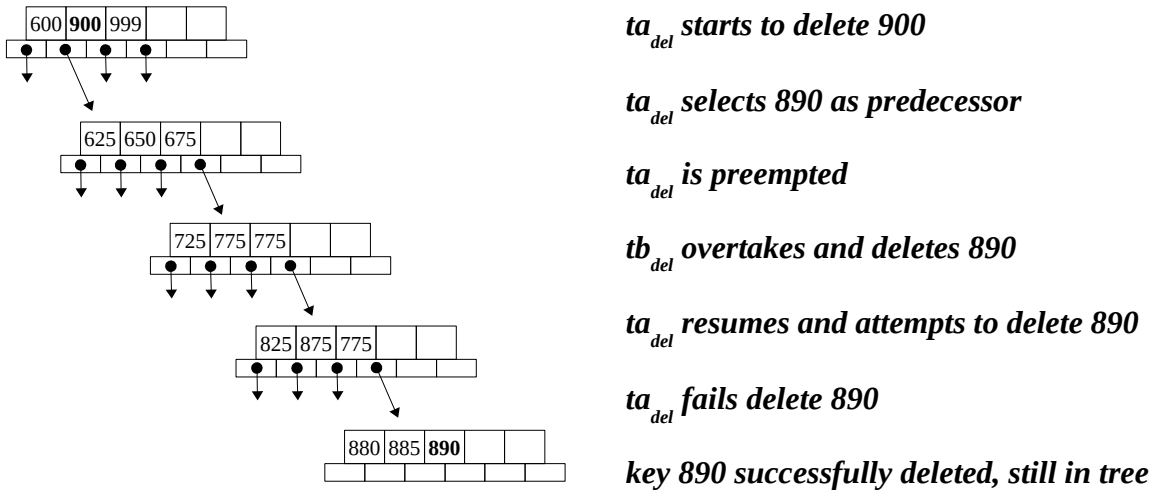


Figure 18: Fine-Grained Two Delete Race Condition Example, Predecessor

While the predecessor example is shown, a similar race condition could occur with the get successor function if the lock is not performed.

If some specialized blocking for only delete threads was implemented, it would not be sufficient to solve the race condition issue. Specifically, without the predecessor and successor locking, there could be another possible race condition between multiple simultaneously executing insert and delete threads. To illustrate, assume thread t_{del} is performing a delete operation for key k_a where k_a is found in an internal node, thus requiring a predecessor or successor key value to be obtained. If thread t_{ins} is inserting key k_b into the leaf node where either the predecessor or

successor key will be obtained and t_{ins} overtakes t_{del} after the selection of the predecessor or successor value but before the key replacement, an error may occur. Specifically, for the predecessor case if k_b is greater than k_a, or for the successor case, if k_b is less than k_a, it will violate the B-Tree invariants and corrupt the data structure.

The potential insert and delete thread race condition is summarized in the following figures.

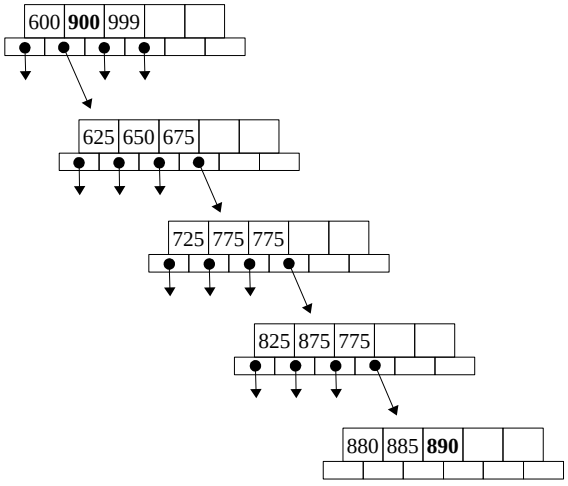


Figure 19: Fine-Grained Insert/Delete Race Condition Example, Predecessor (before)

Assume t_{del} is deleting key 900 and that thread t_{ins} is inserting key 895. As each thread starts executing, the applicable subtree could be configured as shown. As t_{del} executes, it selects the predecessor key value of 890 from the right-most leaf node in that subtree (as shown). If thread t_{ins} is allowed to overtake thread t_{del} passing the node with the 900 key and does so after thread t_{del} has selected the predecessor key of 890, the 890 will be used to overwrite the deleted key of 900. As thread t_{ins} completes, the key 895 will be added to the leaf node that the predecessor key was just obtained from. The resulting subtree would be configured as shown.

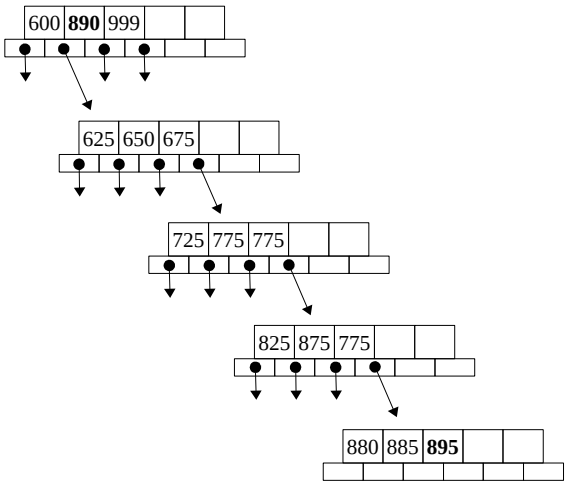


Figure 20: Fine-Grained Insert/Delete Race Condition Example, Predecessor (after)

Since the newly inserted key value of 895 is greater than the 890 it is a violation of the B-Tree invariants. A search traversal for the key 895 would would follow the right child key (from top node shown) and thus not be found.

Locking the nodes during the predecessor or successor traversal solves these types of race conditions. The additional lock acquire and lock release overhead does degrade performance especially in a heavy delete load environment.

5.5.3 Merge Operation

The merge operation will combine two nodes with the minimum number of keys (***degree***/2-1) into a single node. The merge is only initiated after the applicable node locks have been acquired and the locks are not released until the merge operation, if initiated, is completed. The merge may be called directly from the remove (Fine-Grained B-Tree Deletion Algorithm, line 54) or indirectly during a node fill operation (Fine-Grained B-Tree Node Fill Algorithm, line 21 or line 24). In either case, all locking is performed prior to the merge call. Once the locking is complete, the fine-grained merge operation is the same as the coarse-gained merge operation from Section 4.5.3, Merge Operation.

5.5.4 Node Fill Operation

The fine-grained node fill operation is similar to the coarse-gained node fill operation from Section 4.5.4, Node Fill Operation. During the traversal, if a node on the direct traversal path has the minimum number of keys (Fine-Grained B-Tree Deletion Algorithm, line 75), it will be adjusted pro-actively by adding a key or merging nodes. In order to accomplish this, the node fill operation might borrow a key from either a left or a right sibling node. If either a merge or a borrow operation occurs, some additional locking is required. If a merge or a borrow operation is performed, the parent node and applicable sibling nodes must be updated accordingly. This requires that an additional lock be acquired for either the left or the right sibling node to ensure that the changes can be performed without any potential interfere with other active threads.

The fine-grained node fill algorithm is shown below.

```
1.  void bTree<myType>::fill(treeNode<myType> *nd, int idx)  {
2.          treeNode<myType>    *prevSib = NULL, *nextSib = NULL;
3.          int         t = degree / 2;
4.
5.          if (idx > 0)
6.                  prevSib = nd→ptrs[idx-1];
7.          if (idx < nd→cnt)
8.                  nextSib = nd→ptrs[idx+1];
9.
10.         if (idx != 0 && prevSib→cnt >= t) {
11.                 prevSib→nodeLock.lock();
12.                 borrowFromPrev(nd, idx);
13.                 prevSib→nodeLock.unlock();
14.         } else if (idx != nd→cnt && nextSib→cnt>=t) {
15.                 nextSib→nodeLock.lock();
16.                 borrowFromNext(nd, idx);
17.                 nextSib→nodeLock.unlock();
18.         } else {
19.                 if (idx != nd→cnt) {
20.                         nextSib→nodeLock.lock();
21.                         merge(nd, idx);
22.                 } else {
23.                         prevSib→nodeLock.lock();
```

```
24.                         merge(nd, idx-1);
25.                         prevSib→nodeLock.unlock();
26.                     }
27.             }
28.             return;
29.  }
```

Algorithm 19: Fine-Grained B-Tree Node Fill Algorithm

Should another simultaneous active thread be traversing the same path, that thread would require a lock on the parent node before checking any of the child nodes. The consistent ordering of lock acquires ensures that a deadlock does not occur due to the additional locking associated with child sibling nodes.

The locking will block a search thread from traversing any path, even an unrelated path, in the subtree being updated. Potentially, this could degrade performance of a search operation in the impacted subtree. Other insert, search, or delete operations in other subtrees will not be impacted.

5.5.4.1 Borrow Key Operations

The fine-grained borrow key operations are the same as the coarse-gained locking algorithms from Section 4.5.4.1, Borrow Key Operations. This includes both the predecessor node (Algorithm 10, Coarse-Grained B-Tree Borrow from Previous Algorithm) and the successor node (Algorithm 11, Coarse-Grained B-Tree Borrow from Next Algorithm) borrow operations. All applicable locking is performed prior to the the borrow key operations.

5.6 Correctness

This section addresses correctness issues for the fine-grained algorithm.

5.6.1 Deadlock Freedom

The basic required structure of the B-Tree supports freedom from deadlock. The node locks are always acquired using a parent then child, root-to-leaf process. All threads compete for a lock on

the single, shared root node. From the root, a thread can only attempt to acquire node locks on child nodes (i.e., nodes that are on the next lower level in the B-Tree). Update operations require successful acquisition of the parent node lock first and only then can the thread attempt acquisition of the applicable child node. Once the parent node lock is acquired the lock-coupling approach ensures that subsequent threads will not be allowed to overtake. If an insert thread inserting $\boldsymbol{key_0}$ acquires the initial root lock prior to a search for $\boldsymbol{key_0}$, the inability to overtake ensures that the search will succeed even in the event of inopportune preemption's. Conversely, if a search thread searching for $\boldsymbol{key_0}$ acquires the initial root lock prior to an insert for $\boldsymbol{key_0}$, the inability to overtake ensures that the search will fail regardless of the preemption pattern.

At some point, if the thread eventually takes a different path, a different child will be selected. Since the parent lock must be acquired prior to the child, that different path cannot be selected until the parent lock is released. The locking also ensures that a thread cannot overtake a previous thread until the traversal paths diverge. There is no circumstance where the insert or delete algorithms traverse up the B-Tree. The algorithms are deadlock free.

5.6.2 Starvation Freedom

When there is heavy contention multiple threads will be competing for the root node lock. The selection of the thread that is granted the lock is chosen arbitrary. This allows the possibility that the under a persistent heavy load, a specific thread may be repeatedly not be chosen indefinitely. When the load subsides, the thread will be eventually be selected.

5.6.3 Linearizability

The linearizability [31] of the algorithm is based on the linearization point and the lock-coupling approach for both the insertion and deletion algorithms. The linearization point for the insertion algorithm is when the root lock is acquired (Fine-Grained B-Tree Insertion Algorithm, line 6). The linearization point for the search algorithm is when the root lock is acquired (Fine-Grained B-Tree Search Algorithm, line 5). The linearization point for the deletion algorithm is when the root lock is acquired (Fine-Grained B-Tree Insertion Algorithm, line 11). Simultaneously executing threads will contend on the linearlization point for the applicable algorithm.

6.0 Chapter 6,

Lock-Free B-Tree Algorithms

6.0 Chapter 6, Lock-Free B-Tree Algorithms

This chapter presents the lock-free approach as applied to the B-Tree. Multi-threaded concurrent access to mutable shared critical sections must be synchronized. The typical synchronization methods use either coarse-grained or fine-grained mutual exclusion locking. Both these approaches have been explored in the previous chapters.

These lock-based approaches suffer significant performance degradation when faced with the possible delay (e.g., preemption) of a thread that is currently holding a lock. While the thread with the lock is delayed, other active threads that request access to the locked critical section are blocked from making progress until the lock is released by the delayed thread.

In general, lock-free programming is a way to safely share changing data between multiple threads without the cost of acquiring and releasing locks.

6.1 Main Ideas

This section presents a conceptual overview of the B-Tree lock-free algorithmic approach. Changes to the data structure are performed using the CAS operation which will make a change atomically visible to the data structure (as described in Section 2.3, Lock-Free Approach).

The lock-free insertion algorithm initiates at the root and traverses downward. During the traversal, if a child node is found to have the maximum possible number of keys (i.e., ***degree***-1 keys), referred to as full, it will be split. In order to accomplish the split in a lock-free manner, a

copy of the parent and child nodes are made and the applicable changes are made to the copies. Once the changes are completed, an attempt is made to include the new nodes into the B-Tree by using a CAS operation on the pointer in the node above the parent (i.e., the previous node). If another thread is performing an operation that impacts the same parent/child nodes, they will both attempt a CAS operation on the same previous node. Only one thread will succeed and others will fail. The failed threads will delete the local copies and re-attempt the applicable operations re-starting from the root. This approach is also used for a leaf key insertion operation. Copies of the parent and child nodes are made, the changes are performed on the copies (e.g., insertion of the key into the leaf node), and a CAS operation is attempted on the previous node. If the CAS operation succeeds, the operation is complete. The CAS operation will only fail if another thread succeeds. A thread that fails the CAS operation will re-start from the root trying again until it succeeds. A node split operation will move a key from the child to the parent node. In this manner, keys are gradually moved up the B-Tree. A key is only moved up one level per CAS operation. This limits the scope of allowed changes. As such, insertion related movement of keys up in the tree is very gradual.

During an insertion operation, when a CAS operation fails, the newly created nodes were never available to other threads and can be immediately deleted. When a CAS operation succeeds, the nodes that have been removed from the tree may be referenced by another thread that may have been preempted while referencing the newly deleted node (before it was removed from the B-Tree). This possibility is accommodated by queuing the nodes that are being removed from the B-Tree for deletion which maintains the node for a short period of time after the intended deletion ensuring preempted threads will not encounter a dangling pointer. This approach also addresses the ABA problem (as outlined in Section 2.3.3, ABA Problem).

When an node split operation is identified, the applicable nodes are immediately marked as being modified (using the ‘marked’ field in B-Tree Node Definition) before any changes are performed. Another thread working at this level will re-mark the impacted nodes, possibly redundantly. This marking allows threads operating at the next level down to recognize in-process changes at the next higher level and thus not interfere. This helps reduce conflicts and

increase performance by not performing CAS operations that will fail.

In the case of the search algorithm, since no locking is used, the algorithm can proceed from the root node downward until either the key is found or a NULL is found at a leaf node (i.e., key not found). No additional overhead is required for lock operations. As such, the search is as efficient as possible. However, if a key search for $\boldsymbol{key_0}$ is initiated after a key insertion for $\boldsymbol{key_0}$, it is possible that the search operation may overtake the insertion operation and report that $\boldsymbol{key_0}$ is not found. A subsequent search for $\boldsymbol{key_0}$ would succeed.

The delete operation presents some specialized challenges. Specifically, during the downward traversal, a proactive merge of nodes is performed when necessary. A merge operation can be performed when two sibling child nodes each containing the minimum number of keys (***degree***/2-1) are found in the traversal path. The merge operation will move a key one level down in the B-Tree. Since a merge operation will move a key down a level, the parent node must contain at least one more than the minimum number of keys. The root is excluded from this requirement which will allow the root to be fully depleted of keys (one key at a time). When this occurs, the root will be removed and the tree will shrink in height. From the root, the applicable child node is selected based on the key value. If that child does not have at least one more than the minimum number of keys, it must be addressed. This is accomplished by first checking the key counts for the left and right child sibling nodes. If they have more than the minimum number of keys, a merge operation is not possible and a key will be borrowed or shifted from one of the siblings. If the siblings also have the minimum number of keys, a merge is performed. The key borrow operation will move a key from a left or right sibling into the parent node and a key from the parent is moved into the applicable child node. If two or more threads simultaneously attempt similar conflicting operations on the same set of child nodes, only one would succeed. However, if multiple threads attempt such an operation in rapid succession, a conflict can arise. In such a case, a key could be moved into and then out of the applicable child node and a subsequent merge operation on the next level down would violate the B-Tree invariants potentially corrupting the data structure. Additionally, an internal node delete key operation requires a predecessor or successor key from the applicable leaf node (Section

5.5.2.1, Predecessor and Successor Operations). Relocating the key across multiple levels in the B-Tree cannot be done in a single CAS operation since there is no single linearization point. Successive CAS operations create a possible race condition. This type of conflict between simultaneously executing delete operations only occurs during heavy contention. The potential inconsistent multi-level movement of keys right, left, up, and down presents significant challenge for the lock-free approach. To circumvent this problem, the fine-grained deletion operation is used which employs localized fine-grained locking. The lock-free node definition (Section 6.2.1, Lock-Free B-Tree Node Definition) uses atomic fields which ensure that the fine-grained deletion operations do not conflict with the lock-free insertion algorithm, though this approach does hamper performance especially during heavy deletion loads.

6.2 Lock-Free B-Tree Configuration

This section summarizes the lock-free B-Tree node configuration with regard to the specific structure fields.

6.2.1 Lock-Free B-Tree Node Definition

The specific node definition includes fields for the count, leaf status (true/false), a pointer to an array of keys, and a pointer to an array of pointers (for the child pointers). A mutex for the specific node is included which is used during the deletion operation. The keys and child pointer arrays are marked as atomic which is required for the CAS operation.

```
struct treeNode
{
      atomic<myType>                *keys;      // key values
      atomic<treeNode<myType> **>   ptrs;       // child ptrs
      atmoic<int>       cnt;        // current number of keys
      atmoic<bool>      leaf;       // true if leaf, else false
      atmoic<bool>      marked;     // node is being modified
      mutex             nodeLock;   // lock for node (delete)
};
```

Figure 21: Lock-Free B-Tree Node Configuration

The ***myType*** is the passed type for the polymorphic implementation. In addition to the node structure, class global variables include the declared degree and root pointer. There are no global mutexes used.

6.3 Insertion Algorithm

The insertion initiates from the root and traverses down using the applicable child pointer based on the new key value (i.e., key to be inserted). Duplicate keys are checked for prior to the insertion, however it would be possible to verify as an initial step in the insertion algorithm if desired (based on implementation design parameters).

The proactive node splitting process starts at the root. If the root is full, it will be split. The root split operation (detailed in Section 6.3.1, Split Operation, Root Node) will add a new level to the B-Tree and assign a new root. While configuration of the root is altered, none of the key values nor child nodes are changed. The old root is queued for deletion which ensures that should a preempted thread awake with a reference to the old root, it does not find a dangling pointer.

While similar, a root split operation and an internal node split operation do differ and are split into different functions.

The lock-free B-Tree insertion algorithm is presented below.

```
1.  void bTreeInsert(key) {
2.
3.          treeNode<myType>    *currPtr=NULL, *prevPtr=NULL, *newRoot=NULL
4.          treeNode<myType>    *currChild=NULL, *newCurrChild=NULL
5.          treeNode<myType>    *newCurrPtr=NULL, *newLeafPtr=NULL
6.          treeNode<myType>    *sib1=NULL, *sib2=NULL
7.          int     currIndex=0, prevIdx=0
8.          bool    insertDone
9.
10.         insertDone = false
11.         while not insertDone {
12.
13.                 // if root is NULL, create new root
```

```
14.     if (root == NULL)
15.         // create and populate newRoot
16.         newRoot = makeNewNode()
17.         CAS in newRoot
18.             // fail => delete newRoot, continue from top
19.             // success => enqueue released node for deletion,  exit function
20.
21.     // if root is full, tree grows in height (figures 23 and 24)
22.     if (root→cnt == (degree-1))
23.         // split root into two nodes, move middle element up to new root
24.         newRoot = splitRootNode(root)
25.         CAS in newRoot
26.             // fail => delete new nodes
27.             //      continue from top (while insert not done)
28.             // success => enqueue released node for deletion
29.
30.     // traverse downward to find applicable leaf node, split full nodes along the way
31.     currPtr = root
32.     prevPtr = NULL  and  prevIdx = -1
33.
34.     while currPtr is not a leaf node {
35.
36.         // find/set  currIndex  for appropriate child based on key
37.         currChild = currPtr[currIndex]
38.
39.         if  currChild  node is full         // must split child node (figure 25)
40.
41.             // set flags for impacted nodes to indicate updates in process
42.             set  currPtr→marked  and  currChild→marked
43.
44.             // split child node (figure 26)
45.             // creates new current and child nodes
46.             newCurrPtr = splitChildNode(currPtr, currChild, sib1, sib2)
47.
48.             // attempt to cut-in split node into previous level (figure 27)
49.             //      if at root, must update at root
50.             if (prevPtr == NULL)            // at root?
51.                 CAS in new  newCurrNode  into root
52.                     // fail => delete nodes newCurrPtr, newChild,
53.                     //          sb1, and sb2
54.                     //      continue from top (while insert not done)
```

```
55.                                         // success => enqueue old nodes for deletion
56.                         else            // not at root
57.                             // note, if marked, abandon changes, continue from top
58.                             if prevPtr→marked ||
59.                                CAS in new newCurrPtr into prevPtr[prexIdx]
60.                                     // fail => delete newCurrPtr and newChild
61.                                     //         continue from top
62.                                     // success => enqueue related nodes for deletion
63.                             // re-find currIndex for appropriate child based on key
64.
65.                     // move down tree to next level, track previous level
66.                     prevPtr = currPtr
67.                     prevIdx = currIndex
68.                     set currNode to appropriate child based on currIndex
69.
70.             } // end while
71.
72.             // insert key into non-full leaf (figure 28)
73.             newLeafPtr = insertNonFull(key)
74.
75.             // attempt to cut-in split node into previous level (figure 29)
76.             //      if at root, must update at root
77.             if (prevPtr == NULL)                        // at root?
78.                     CAS in newLeafPtr into root
79.                         fail => delete newLeafPtr
80.                                 continue from top (while insert not done)
81.                         success => set insertDone=true
82.             else
83.                     // note, if marked, abandon changes, continue from top
84.                     if prevPtr→marked ||
85.                        not CAS in newLeafPtr into prevPtr at prevIdx
86.                         // fail => delete newLeafPtr
87.                         //         continue from top (while insert not done)
88.                         // success => enqueue released nodes for deletion
89.                                     set insertDone=true
90.
91.         } // end while not insertDone
92.
93. } // end insert
```

Algorithm 20: Lock-Free B-Tree Insertion Algorithm

Each of the major operations is explained in the following sections.

6.3.1 Split Operation, Root Node

A B-Tree only grows from the root, not the leaf nodes. This only occurs when the root node becomes full (i.e., has the maximum number of key values). One of the initial steps in the algorithm is to check if the root node is full, and if so, call the function to split the root node.

The algorithm for handling a root split operation is shown below.

```
1.  treeNode<myType> *bTree<myType>::splitRootNode(treeNode<myType> *currRoot,
2.                                      treeNode<myType> *sib1, treeNode<myType> *sib2)  {
3.          int                         siz=0;
4.          treeNode<myType>    *newRoot = NULL;
5.
6.          // create new sibling nodes
7.          sib1 = makeNewNode();
8.          sib1->leaf = currRoot→leaf;
9.          sib2 = makeNewNode();
10.         sib2->leaf = currRoot→leaf;
11.
12.         // set size based on degree
13.         siz = (degree-1) / 2;
14.
15.         // copy the first (degree-1)/2 keys of current root to sibling 1
16.         for (int j = 0; j < siz; j++)
17.                 sib1→keys[j] = currRoot→keys[j];
18.
19.         // copy the last (degree-1)/2 keys of current root to sibling 2
20.         for (int j = 0; j < siz; j++)
21.                 sib2→keys[j] = currRoot→keys[j+siz+1];
22.
23.         // copy the first degree/2 ptrs of current root over to sibling 1
24.         if (!currRoot→leaf)
25.                 for (int j=0; j < degree/2; j++)
26.                         sib1→ptrs[j] = currRoot→ptrs[j];
27.
28.         // copy the last degree/2 ptrs of current root over to sibling 2
29.         if (!currRoot→leaf)
```

```
30.                     for (int j=0; j < degree/2; j++)
31.                             sib2→ptrs[j] = currRoot→ptrs[j+degree/2];
32.
33.             // set sibling sizes
34.             sib1→cnt = siz;
35.             sib2→cnt = siz;
36.
37.             // create and populate new root
38.             newRoot = makeNewNode();
39.             newRoot→leaf = false;
40.             newRoot→ptrs[0] = sib1;
41.             newRoot→ptrs[1] = sib2;
42.             newRoot→keys[0] = currRoot→keys[siz];
43.             newRoot→cnt = 1;
44.
45.             return   newRoot;
46.  }
```

Algorithm 21: Lock-Free B-Tree Split Root Algorithm

The following figure shows a full root of degree 6 which must be split.

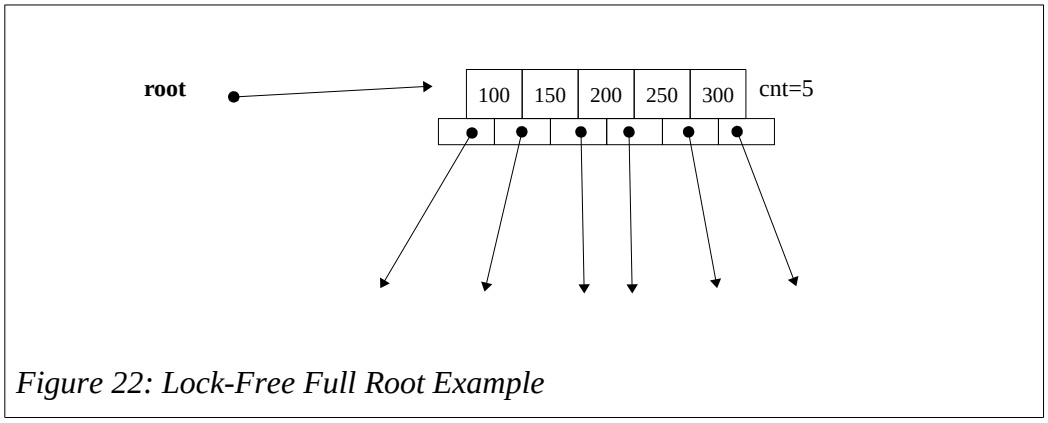


Figure 22: Lock-Free Full Root Example

In order to split the root in a lock-free context, a new set of nodes (root, left, and right) must be

created and populated. The new root will contain the middle key, the new left node will contain the left most keys and associated child pointers, and the new right node will contain the right most keys and the associated child pointers.

The following diagram shows the set-up.

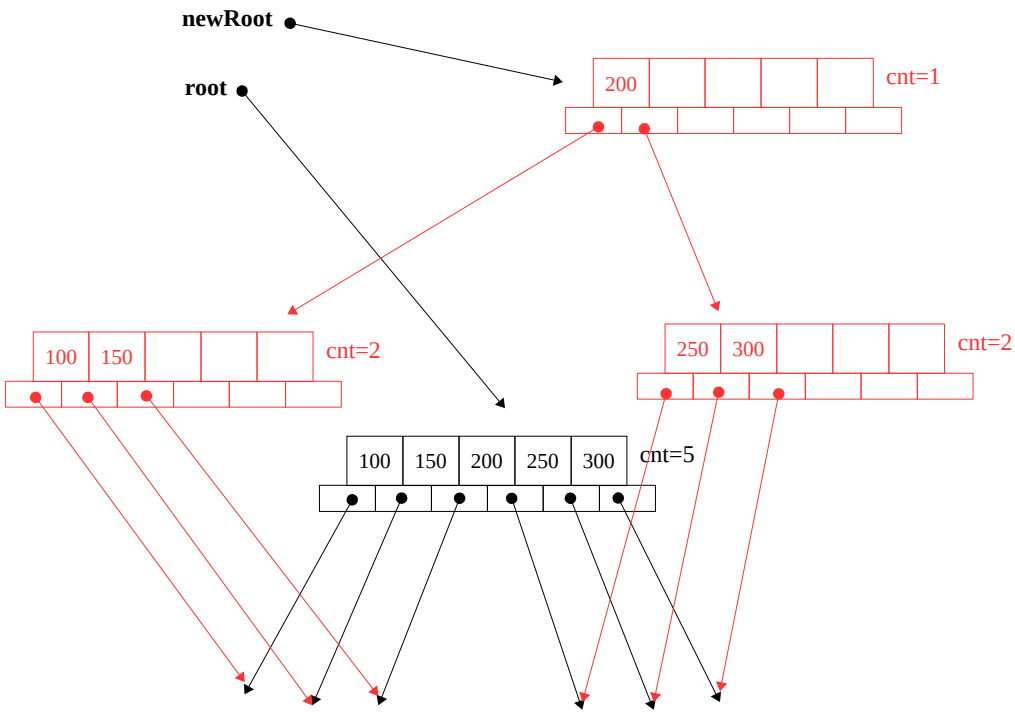


Figure 23: Lock-Free Split Root, Initial Configuration

Once the set-up is complete, the CAS operation will be performed on the root node. Specifically, the current value of the root node obtained at the beginning of the loop prior to the set-up operations (expected value) is compared to the current value of the root.

If the expected value does not match the current value, another thread has already performed and completed the split-root operation. If this occurs, the newly created nodes (new root, new left,

and new right) can be immediately deleted and the algorithm re-started from the beginning of the loop. Since the root is now split, the operation will continue its downward traversal.

If the expected value does match the current value, the CAS will change the root pointer to the new root node thus updating the data structure atomically. A successful CAS operation is shown as follows.

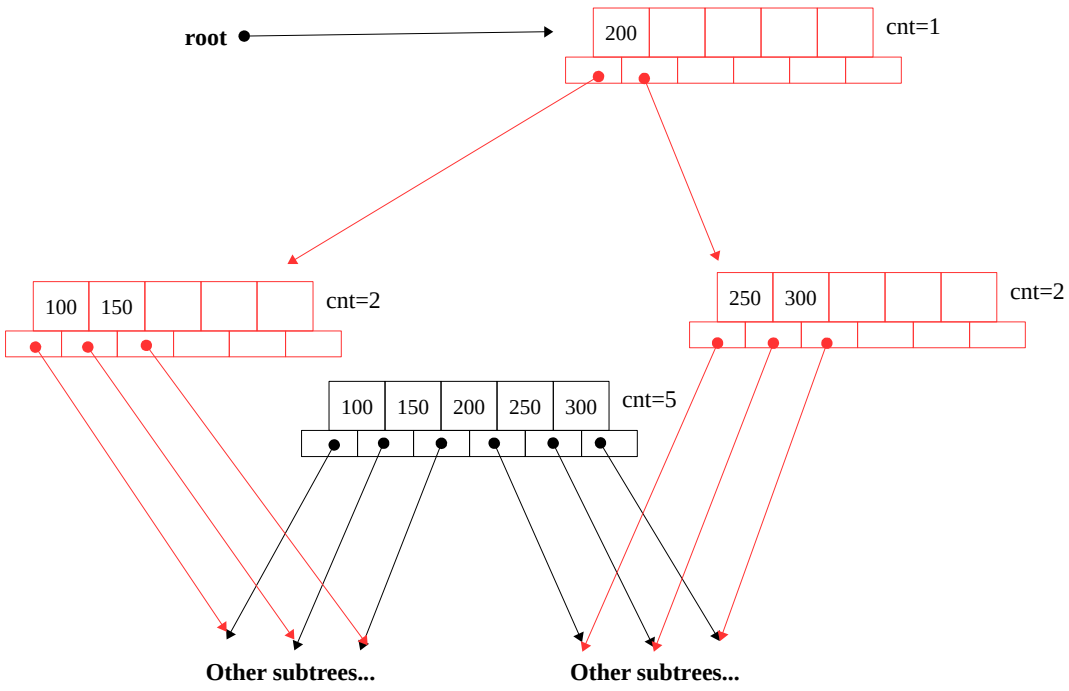


Figure 24: Lock-Free Split Root, Final Configuration

Since the CAS operation is atomic, the switch between the old and newly configured data structure appears instantly to all new threads. After a successful CAS operation, the old root is queued for deletion. This ensures that any preempted threads awaking to the updated structure will not crash based on a deleted node.

6.3.2 Split Operation, Child Node

One of the more complex steps in the algorithm is to check if a child node is full, and if so, split the child node into two nodes. The following figure shows a full child node, in context, which must be split. Local pointers to the previous node and current node are shown for reference. In this example, during the traversal the applicable child node is determined to be full and must be split. The child node split will divide the full node into two nodes and move one key value up into the parent node.

The algorithm for splitting a full child node is shown below.

```
1.  treeNode<myType> *bTree<myType>::splitChildNode(treeNode<myType> *parent,
2.                                                  treeNode<myType> *child,
3.                                                  treeNode<myType> *sib1,
4.                                                  treeNode<myType> *sib2)  {
5.          int                     siz, idx;
6.          treeNode<myType>        *newParent = NULL;
7.
8.          newParent = cloneNode(parent);
9.
10.         // create new sibling nodes
11.         sib1 = makeNewNode();
12.         sib1→leaf = child→leaf;
13.
14.         sib2 = makeNewNode();
15.         sib2→leaf = child→leaf;
16.
17.         // set size based on degree
18.         siz = (degree-1) / 2;
19.
20.         // copy the first (degree-1)/2 keys of child to sibling 1
21.         for (int j = 0; j < siz; j++)
22.                 sib1→keys[j] = child→keys[j];
23.
24.         // copy the last (degree-1)/2 keys of child to sibling 2
25.         for (int j = 0; j < siz; j++)
26.                 sib2→keys[j] = child→keys[j+siz+1];
27.
```

```
28.
29.         // copy the first degree/2 ptrs of child over to sibling 1
30.         if (!child→leaf)
31.                 for (int j=0; j < degree/2; j++)
32.                         sib1→ptrs[j] = child→ptrs[j];
33.
34.         // set sibling size
35.         sib1→cnt = siz;
36.
37.         // copy the last degree/2 ptrs of child over to sibling 2
38.         if (!child→leaf)
39.                 for (int j=0; j < degree/2; j++)
40.                         sib2→ptrs[j] = child→ptrs[j+degree/2];
41.
42.         // set sibling size
43.         sib2→cnt = siz;
44.
45.         // find where new key is going in new parent
46.         idx = 0;
47.         while ((idx < newParent→cnt) && (newParent→keys[idx] < child→keys[siz]))
48.                 idx++;
49.
50.         // slide new parent child ptrs over to make room for new child
51.         for (int j = newParent→cnt; j >= idx+1; j--)
52.                 newParent→ptrs[j+1] = newParent→ptrs[j];
53.
54.         // slide new parent child keys over to make room for new key
55.         for (int j = newParent→cnt-1; j >= idx; j--)
56.                 newParent→keys[j+1] = newParent→keys[j];
57.
58.         // add new siblings to new parent
59.         newParent→ptrs[idx] = sib1;
60.         newParent→ptrs[idx+1] = sib2;
61.
62.         // copy middle key of child to new parent
63.         newParent→keys[idx]  = child→keys[siz];
64.
65.         // increment count of keys in new parent
66.         newParent→cnt += 1;
67.
68.
```

```
69.             return   newParent;
70.  }
```

Algorithm 22: Lock-Free B-Tree Split Child Node Algorithm

Due to the proactive splitting on the downward traversal, the current node and nodes above the current node, including the root, will not be full. For example, the process for a full child node split operation is shown in the following figures.

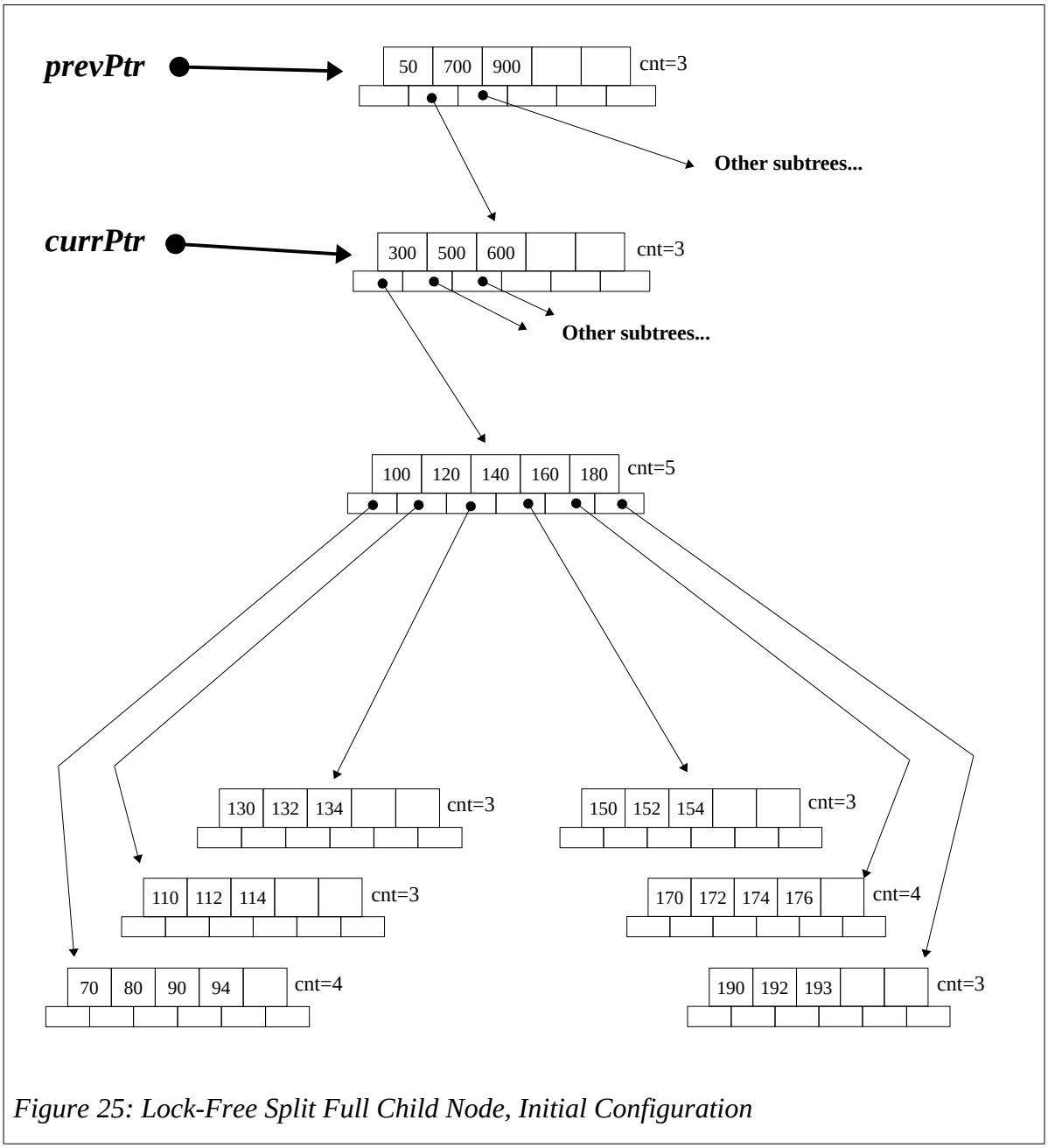


Figure 25: Lock-Free Split Full Child Node, Initial Configuration

For the child node split operation, it does not matter if the lowest level of nodes shown above are leaf nodes or have extensive subtrees as they are not altered in this operation.

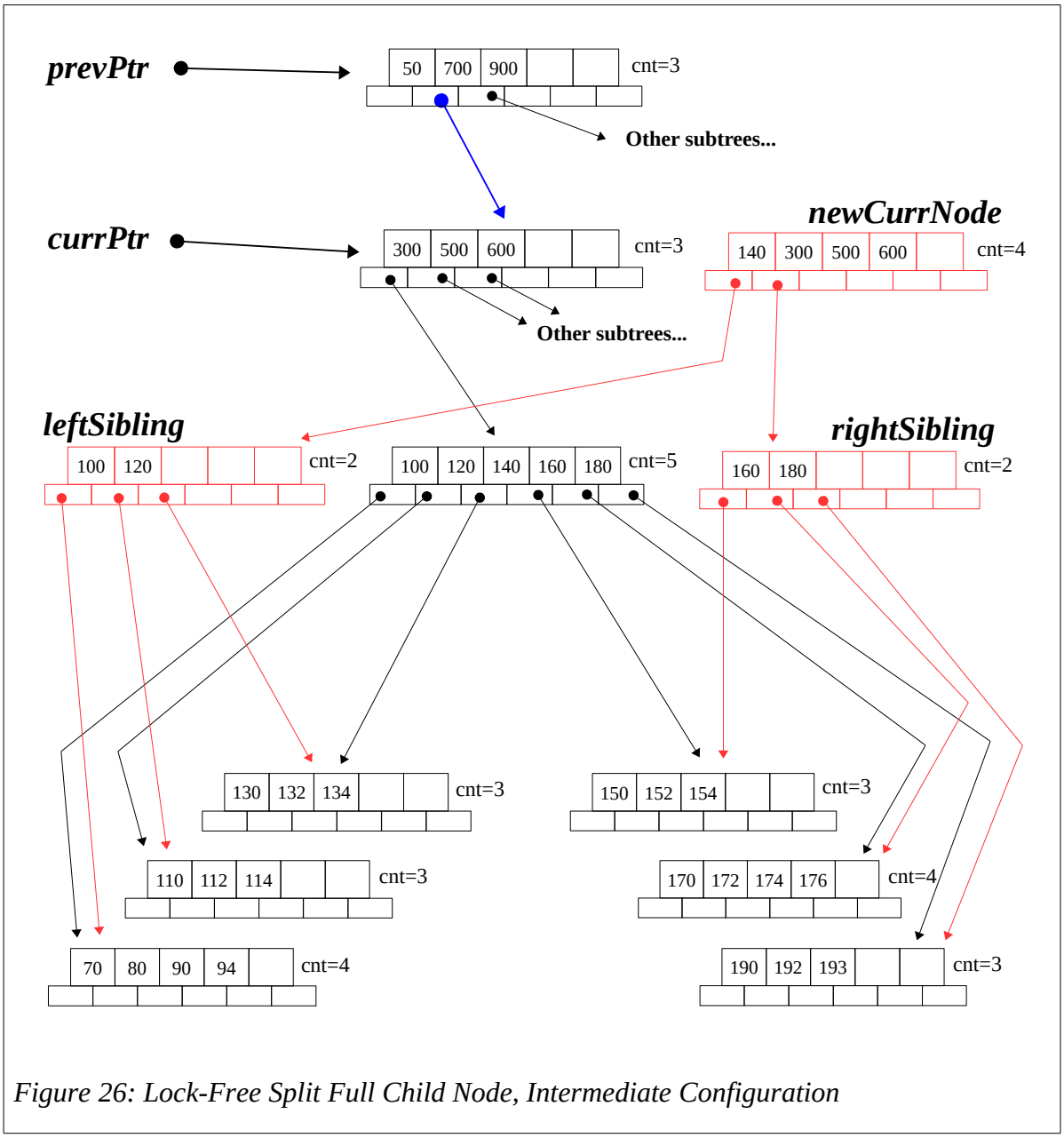


Figure 26: Lock-Free Split Full Child Node, Intermediate Configuration

Once the new nodes have been created and populated, the new structure can be swapped into the data structure by changing the single pointer in the previous node to point to the new current node with a CAS operation. Once this is successfully completed, the new structure is fully available to all threads.

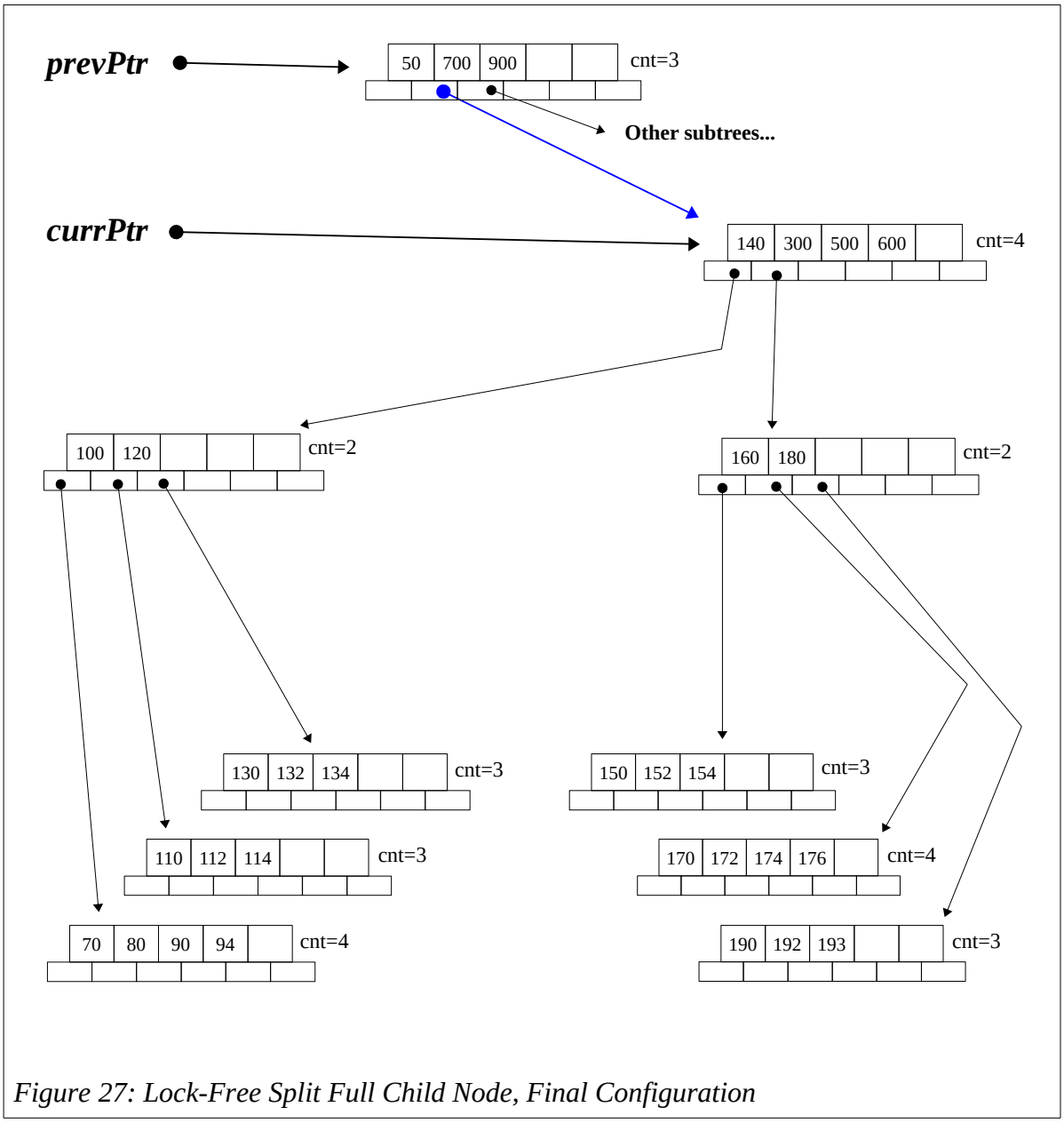


Figure 27: Lock-Free Split Full Child Node, Final Configuration

If the CAS operation fails, the new current node, new left sibling, and new right sibling nodes are deleted and the operation is re-started from the root. An unsuccessful CAS operation will occur only when some other thread has succeeded in performing the same operation, invalidating the need for this operation.

After a successful CAS operation, the old current node root is queued for deletion. This ensures that any preempted threads awaking to the updated structure will not crash due to a non-existent node.

A key advantage of this approach is that if one thread starts the operation and is preempted, then another thread also starts the same operation and is also preempted, another thread may start and complete the operation. A preempted thread will not block other threads.

6.3.3 Leaf Node Insertion

The insertion of a key into a leaf node is fairly straight-forward. Due to the proactive splitting during the downward traversal, a leaf node is guaranteed to be not full. Insertion into a non-full leaf node involves swapping the old leaf for a newly configured leaf. For example, given the initial configuration shown below and assuming that the key value to 75 is to be inserted.

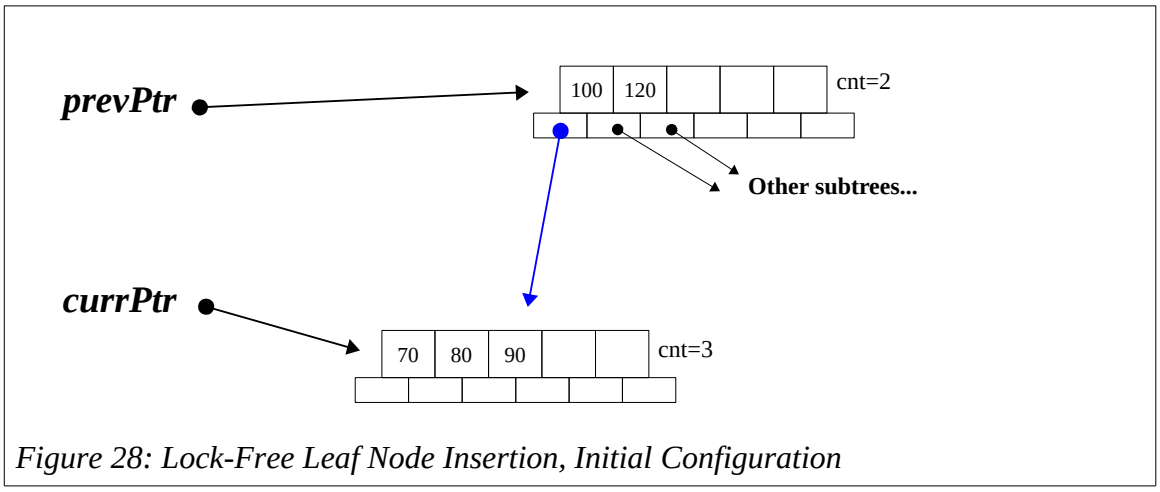


Figure 28: Lock-Free Leaf Node Insertion, Initial Configuration

A new leaf node will be created and populated with the existing keys and the new key value in the appropriate order as shown below.

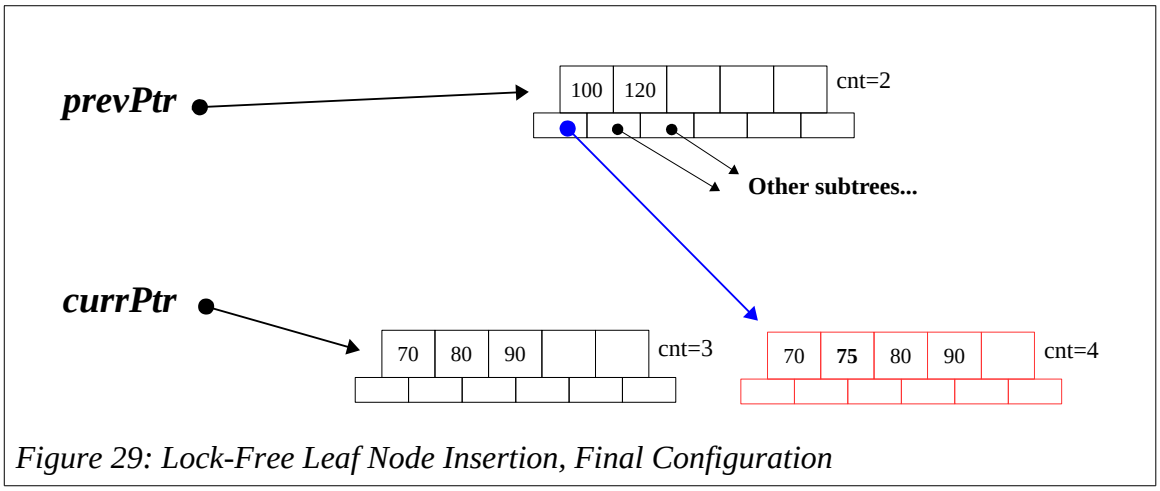


Figure 29: Lock-Free Leaf Node Insertion, Final Configuration

The CAS operation is performed on the previous node. If the CAS operation is unsuccessful, another thread has succeeded in updating the node. Since the node has been changed by another thread, the current changes must be abandoned by deleting the new local leaf node and restarting the traversal from the root. Due to the relatively shallow nature of B-Tree's, restarting from the root is not a significant detriment.

After a successful CAS operation, the new node with the new key value is immediately available in the data structure. The old leaf node is queued for deletion. This will ensure that if another, preempted thread operating on this node, awakes to the updated structure that it will not crash due to a non-existent node.

6.4 Search Algorithm

The lock-free search algorithm starts at the root and follows a basic downward traversal until the key is either found or a leaf node is found without the key (e.g., key not found). The search operation does not alter any nodes and does not create any conflicts with other operations. Any in-process modifications are not visible to the search thread. If another thread is modifying the data structure, the search will only see that modifications after they are fully complete which only occurs after a successful CAS operation.

The search operation is a direct top to bottom traversal.

```
1.  bool search(myType key) const {
2.
3.          curr = root;
4.          if (curr == NULL)
5.                  return  false;
6.
7.          while (curr != NULL) {
8.                  // find/set  idx  for appropriate child based on key
9.                  // see if key is found
10.                 if ( idx < curr→cnt && curr→keys[idx] == key)
11.                         return  true;
12.                 curr = curr→ptrs[idx];
13.         }
14.         return  false;
15. }
```

Algorithm 23: Lock-Free B-Tree Search Algorithm

Due to the lock-free approach, there is no possibility of blocking. However, if an insertion of key, k_1, and a search for key, k_1, are simultaneously initiated, it is possible that the insertion process may be delayed and the key reported as not found. A subsequent search would find the key.

6.5 Deletion Algorithm

For key deletion, during the downward traversal, a proactive merge of nodes is performed which, over-time, gradually contracts the B-Tree size as more and more deletes are performed. The proactive merge process will move a key down in the B-Tree which requires that the parent node contain at least one more than the minimum number of keys. This requirement does not apply to the root node which allows the root to be gradually depleted of keys. When the root contains no keys, it can be removed from the B-Tree thus decreasing the height by one. Starting form the root, the deletion will check the current node, and if the key is not found, select the appropriate

child node to continue the traversal. Before the traversal continues, the child key count is checked to ensure that the child has at least one more than the minimum number of keys. If the child has only the minimum number of keys, it must be addressed. That child's existing sibling nodes key counts are checked and if a sibling has excess keys (i.e., at least one more than the minimum required), a key may be borrowed from a sibling. If the siblings also have the minimum number of keys, a merge operation is performed. The key borrow operation will move a key from a left or right sibling into the parent node and a key from the parent is moved into the applicable child node. This provides the required excess key in the applicable child node. This process is fully detailed in Section 5.55.5, Fine-Grained Deletion Algorithm.

For this type of operation, if a CAS operation is used two or more delete threads could simultaneously attempt conflicting operations on the same set of nodes, the CAS operation will ensure that only one succeeds. A conflicting operation might include moving a key into and then out of a adjacent sibling nodes. If multiple threads attempt conflicting operations in succession, they may both succeed resulting a potential violation of the B-Tree invariants from a subsequent merge operation on that child that moves a key down. A key in a leaf node can be removed directly since there are no child pointers to update. A key in an internal node cannot be deleted without extensive restructuring of the B-Tree or replacement. Restructuring would require rebuilding the subtree rooted that node and is not feasible. A replacement can be obtained from a predecessor or successor key from the applicable leaf node (Section 5.5.2.1, Predecessor and Successor Operations). Since there is no single linearization point, the multi-level relocation of a key across in the B-Tree cannot be done in a single CAS operation. Immediately successive CAS operations create a possible race condition. The current lock-free primitives do not provide the functionality required for this type of operation due to the multi-level movement of keys.

The address the problem, the fine-grained deletion algorithm is used which employs localized fine-grained locking. The deletion algorithm uses the fine grained deletion algorithm detailed in Section 5.5, Deletion Algorithm (Algorithm 16, Fine-Grained B-Tree Deletion Algorithm) which uses fine-grained locking. The lock-free B-Tree Node configuration (Figure 21, Lock-Free B-Tree Node Configuration) includes atomic fields which ensures that the fine-grained locking for

the deletion can work with the lock-free CAS operations. The 'marked' field is used to mark nodes that are being updated to eliminate conflicts with the insertion operation.

Overall, the use of a fine-grained deletion approach hampers the overall performance especially during heavy deletion loads.

6.6 Correctness

This section addresses correctness issues for the lock-free algorithms.

6.6.1 Deadlock Freedom

With regard to deadlock, the lock-free insertion approach neither requires nor performs any locking and is thus free from deadlock. Since no locking is performed multiple simultaneously executing threads could potentially perform operations on the same node. This potential conflict is handled by the CAS instruction as described in Section 2.3.1, Lock-Free Primitive.

The deadlock freedom for the fine-grained deletion is addressed in Section 5.6.1, Deadlock Freedom.

6.6.2 Linearization

The linearization point for the lock free insertion approach is established at the CAS location. The insertion routine has multiple linearization points depending on the specific operation being performed. The details are outlined in the following sections.

Since the search algorithm does not alter the data structure, there is no linearization point.

The linearization point for the fine-grained deletion approach is detailed in Section 5.6.3, Linearizability.

6.6.3 Insertion Algorithm Correctness

The primary initial step for the insertion algorithm is the creation of a root node in an empty tree. With no contention, this is fairly straight-forward. If multiple threads are simultaneously attempting to insert a key into an empty tree, each thread could find the tree empty and attempt to create the new root. Each thread will create a new potential root node, populate it with the

applicable key, and attempt the CAS operation (Lock-Free B-Tree Insertion Algorithm, line 17). One thread will succeed and a new root will be created. The other threads will fail, restart, will find an existing root (i.e., the new one) and continue to insert the key into the appropriate location.

As keys are added, the root node will be filled with keys values and need to be split. Each thread will check for a full root, and if found, attempt to split to the root node. If multiple threads simultaneously attempt to perform the root split operation and attempt the CAS operation (Lock-Free B-Tree Insertion Algorithm, line 25), only one will succeed. The other threads will fail, re-start, and now that the root is split, they will be able to continue.

The next operation is insertion of a key into a leaf node. If multiple threads are attempting to insert different keys into the same leaf node, each will create a copy of the current leaf node and attempt the CAS operation (Lock-Free B-Tree Insertion Algorithm, line 78 and 85). The CAS operation uses the previous node (i.e., node above) as the linearization point. If that node is the root, the root must be updated directly and is the CAS location. Otherwise, the previous node is the CAS location. If multiple threads simultaneously operate on the same leaf node, one thread will succeed and the others will fail. The threads that fail will only fail if another thread succeeds which ensures that progress is made. The threads that fail will re-start the insertion process from the root. Since B-Tree's are generally comparatively shallow, such a re-start is not a performance bottleneck.

The insertion approach includes a proactive node splitting during the traversal. If multiple simultaneous threads attempt to split the same node, each creates a new split node and attempts the CAS operation (Lock-Free B-Tree Insertion Algorithm, line 51 and 59). If that node is the root, the root must be updated directly and is the CAS location for this operation and otherwise, the previous node is the CAS location. One thread will succeed and the others will fail ensuring progress.

6.6.4 Search Algorithm Correctness

The search algorithm performs a direct root to leaf traversal. The search neither blocks nor alters the data structure. However, since there is no locking, a search thread searching for $\boldsymbol{key_n}$ may overtake a thread inserting $\boldsymbol{key_n}$ and report that $\boldsymbol{key_n}$ is not found. Subsequent searches for $\boldsymbol{key_n}$ will find $\boldsymbol{key_n}$.

6.6.5 Deletion Algorithm Correctness

The correctness for the deletion algorithm is outlined in Section 5.6, Correctness.

7.0 Chapter 7,

Empirical Results

7.0 Chapter 7, Empirical Results

This chapter presents the empirical results from the implementations of the algorithms presented in the previous sections. A detailed explanation of the testing methodologies is presented including the testing environment. The performance results are presented comparing the baseline coarse-gained implementation to the fine-grained and lock-free implementations.

7.1 Testing Methodologies

A specialized sequential insertion and deletion methodology was used during the initial testing. This approach is not representative of typical workloads, but helps to identify problems by maximizing the contention. For the primary testing and performance analysis, a more standard methodology was utilized. The primary testing methodology follows the general testing techniques utilized for various concurrent data structures, including skip-lists [32], binary search trees [33], other balanced trees [34][35], and B^{+}-Trees/B*-Trees/B^{link}-Trees [4][5][36]. This involves a random permutation of key values which is more representative of typical expected workloads.

Both the initial and primary testing approaches are outlined in the following sections.

7.1.1 Testing Environment

Testing was performed on various Ubuntu 18.04 LTS systems using the GNU G++ Compiler version 7.3.0. Various systems with both AMD and Intel processors with processors ranging

from 8 to 16 cores, representing commonly available hardware were used. All codes are C++11/17 compliant with no external or specialized libraries. Only basic compiler options were used including **`-Wall`**, **`-pedantic`**, **`-Werr`**, and **`-std=c++11`**. The standard thread library, POSIX pthread, was used (which is fully supported in the C++11/17 standard).

7.1.2 Initial Testing Process

A custom initial testing methodology involves inserting and deleting sequential values across multiple threads. Each thread is assigned a thread number 0, the thread ***threadCount***-1. The key value inserted by each thread starts at an initial value, 0 for simplicity. Each thread inserts a key value of the initial value plus the thread number. The next value inserted by each thread is defined as:

$$keyValue = PreviousValue + threadNumber + STEP$$

The ***STEP*** value must be greater than the number of threads used in the testing process.

The following figure shows the assignment visually based on an initial starting value of 0 and a ***STEP*** value of fifty. The ***STEP*** can be easily adjusted as needed.

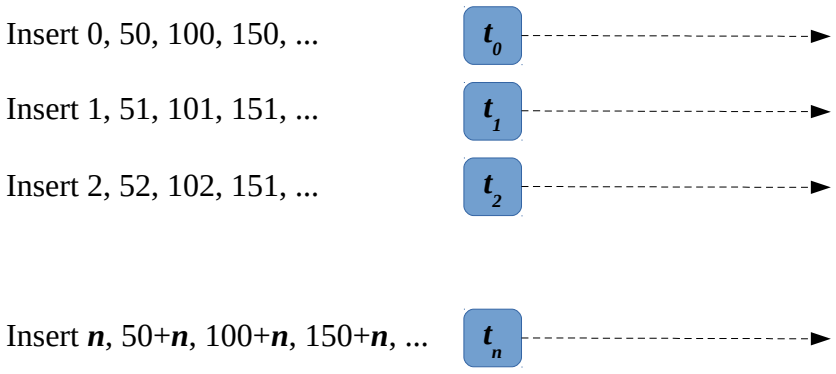


Figure 30: Maximize Thread Overlapping Testing

With a ***STEP*** value of fifty, the number of concurrent threads must be less than fifty.

The key reason for using this testing technique is that threads will generally attempt to perform operations, insert or delete, depending on the specific test scenario, in the same subtrees of the data structure. This maximizes the contention and conflict between simultaneously executing threads providing a more comprehensive test for concurrent threads. As the execution continues and the data structure grows, the contention will decrease. Overall, this testing approach results in significantly more contention than the more typical random processes.

No examples of this testing technique were found in the literature, likely as this approach does not represent a realistic or typical expected load. Additionally, this approach results in a worst-case performance scenario due to the maximization of the concurrency.

7.1.3 Primary Testing Process

The primary testing process uses a random permutation of key values across a defined range. The values were generated using the modern version of Fisher-Yates Algorithm [37] popularized by Donald Knuth [7]. The number of threads, range of keys, and number of keys used are configurable. The total number of keys used for this testing effort was 30,000,000 which ensured a large enough data set to ensure concurrent operations for all simultaneously executing threads.

In order to ensure the appropriate overlapped execution, a three phase execution process was used. In the first or set-up phase, a configurable number of insert threads were initiated as a group. Once complete, phase two used a configurable number of threads with a configurable combination of insert, search, and delete operations initiated as a group. In phase two, the search threads searched for key values inserted during the first phase which ensures that they will be successfully found. The delete threads also deleted key values inserted in phase one, ensuring that the keys will be available for deletion. This process is used in order to ensure that if a search or delete thread cannot find a key, it would signify an error. The insert threads inserted values not already inserted. In this manner, a final verification phase can ensure that all insertions and deletions were correctly performed. The final phase removes the key values inserted in phase

two and verifies the final status of the data structure for validity.

A visual summary of the process is shown below.

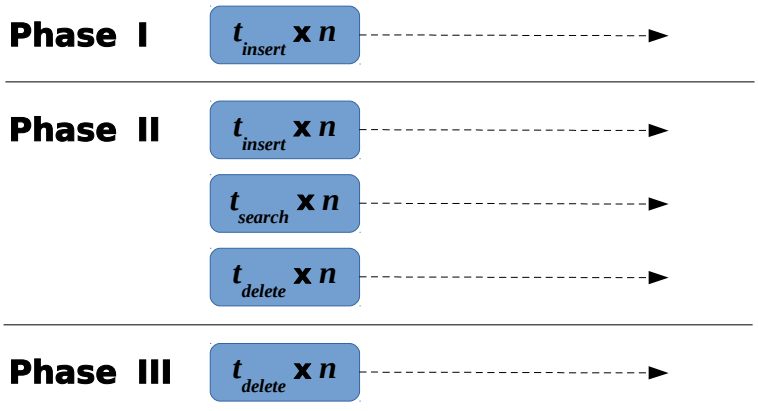


Figure 31: Test Phases Overview

The number of thread performing insert, search, and delete operations is how the load mix is controlled during phase two.

7.2 Performance Comparisons

The coarse-gained implementation was used as a base-line for the comparison to the fine-grained and lock-free implementations. The performance comparisons were done over a series of different concurrency levels (1, 3, 6, 9, and 12 simultaneous threads). Since the degree of a B-Tree data structure is configurable, tests were completed with a mix of different degrees (6, 8, 10, and 12). The performance of a data structure is also impacted by the specific job mix of simultaneous insertion, search, or deletion operations being performed. In general, a search operation will perform better than an insert or delete operation since no changes in the data structure are required. Different environments will generate different workloads. Following processes established in the literature [10][36][38][39], and in order to provide a balanced testing

process, different typical workloads were used. The specific workload used is presented in the corresponding section. To ensure consistent testing, all tests were executed on one machine. The specific machines was an Intel core i7-8700 12 cores running Ubuntu 18.04 LTS. This represents a current machine in the target environment as outlined in Section 1.0, Introduction.

While different machines will generate different timings, the comparative results would be consistent.

The performance comparisons are presented in terms of the relative performance of two implementations processing the same data set in the same concurrency configuration in the same hardware environment. It is the improvement in speed of execution of a set of tasks executed with different implementations. The formula used is the ratio of the baseline coarse-gained implementation to the fine-grained or lock-free implementation [40].

7.2.1 Coarse-Grained vs Fine-Grained

The coarse-gained and fine-gained implementations were executed using both an even job mix and a heavy search load mix. The specific job mixes used are as follows:

- Even Job Mix – Insertion (33.3%), Search (33.3%), and Deletion (33.3%)
- Heavy Search Mix – Insertion (10%), Search (80%), and Deletion (10%)

The even load job mix (33.3%, 33.3%, 33.3%) results are shown in the following figures.

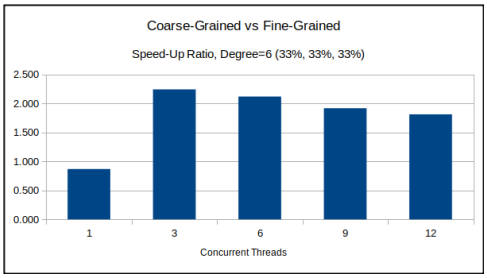


Figure 32: Performance: CG vs FG, Degree 6, Load (33.3%, 33.3%, 33.3%)

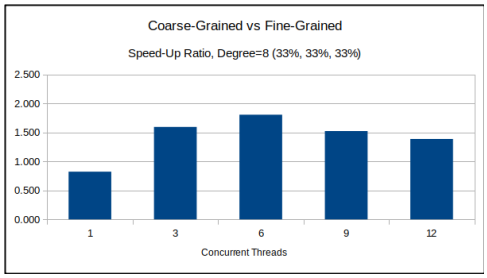


Figure 33: Performance: CG vs FG, Degree 8, Load (33.3%, 33.3%, 33.3%)

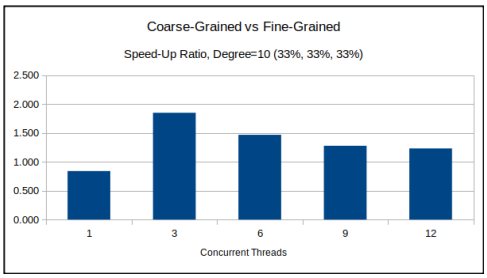


Figure 34: Performance: CG vs FG, Degree 10, Load (33.3%, 33.3%, 33.3%)

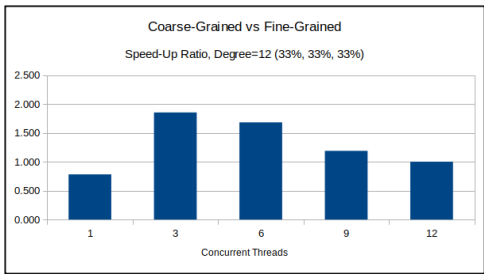


Figure 35: Performance: CG vs FG, Degree 12, Load (33.3%, 33.3%, 33.3%)

The heavy search job mix (10%, 80%, 10%) results are shown in the following figures.

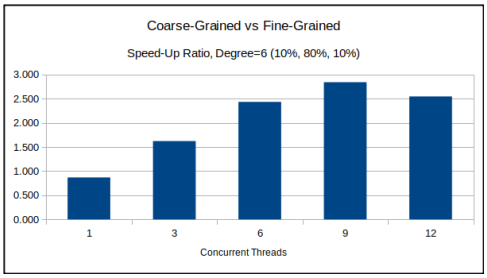


Figure 36: Performance: CG vs FG, Degree 6, Load (10%, 80%, 10%)

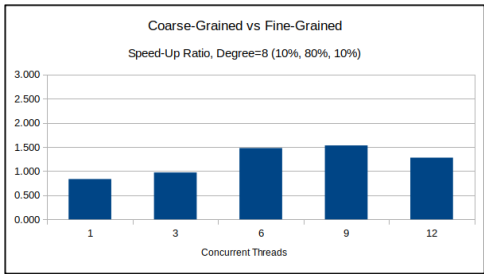


Figure 37: Performance: CG vs FG, Degree 8, Load (10%, 80%, 10%)

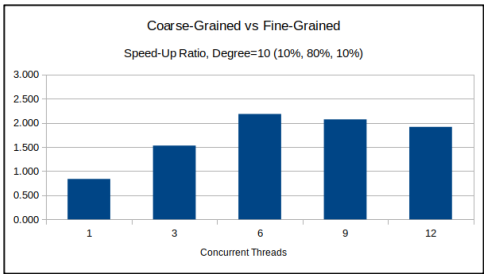


Figure 38: Performance: CG vs FG, Degree 10, Load (10%, 80%, 10%)

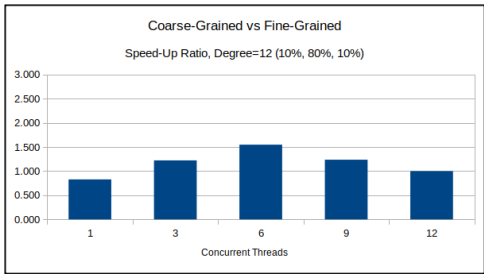


Figure 39: Performance: CG vs FG, Degree 12, Load (10%, 80%, 10%)

The following section presents an analysis of the testing results.

7.2.1.1 Performance Analysis

As can be seen in all tests, for a single threaded environment, the performance of the fine-grained implementation was below the performance of the coarse-gained implementation. This is due to the fine-grained lock acquire and lock release overhead of nodes in all traversals. For the coarse-gained implementation, a single lock was used. With a single thread, there is no opportunity for performance gains associated with concurrent operations.

As the concurrency increased, the speed-up increased and the fine-gained implementation out-performed in the coarse-gained implementation. Based on the workload and job mix, the performance speed-up was between 1.25 and nearly 3 (2.9) times. The heavy search job mix provided the best performance. The lock duration associated with the search is less than the insert and delete operations as the search does not make any changes to the data structure. While still better than the coarse-grained, the fine-grained locking algorithms performance decreased as the B-Tree degree increased. As the degree increased, the number of key values tied up during a lock operation is increased which has a negative impact on the overall amount of concurrent actions possible. This is more impacting when such locking is near the B-Tree root.

7.2.2 Coarse-Grained vs Lock-Free

The coarse-gained and lock-free implementations were executed using both an even job mix and a heavy search load mix. Since the lock-free implementation required a fine-grained delete, a set of tests was performed using only the search and lock-free insertion. An additional set of tests was performed using the lock-free insertion with the fine-grained deletion algorithm.

The specific job mixes are as follows:

- Even Job Mix (without delete) – Insertion (50%) and Search (50%)
- Heavy Search Mix (without delete) – Insertion (20%) and Search (80%)
- Even Job Mix (with delete) – Insertion (33.3%), Search (33.3%), and Deletion (33.3%)
- Heavy Search Mix (with delete) – Insertion (10%), Search (80%), and Deletion (10%)

The even load job mix without delete (50%, 50%) results are shown in the following figures.

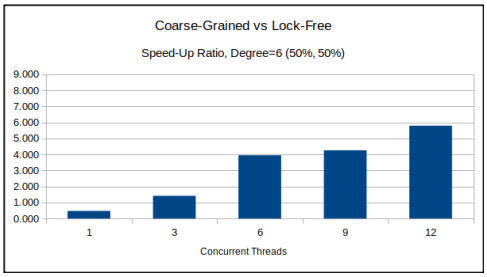


Figure 40: Performance: CG vs LF, Degree 6, Load (50%, 50%)

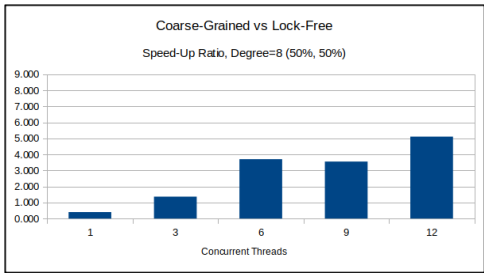


Figure 41: Performance: CG vs LF, Degree 8, Load (50%, 50%)

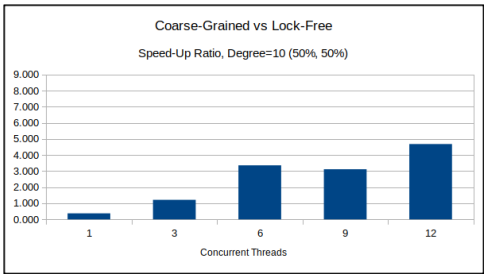


Figure 42: Performance: CG vs LF, Degree 10, Load (50%, 50%)

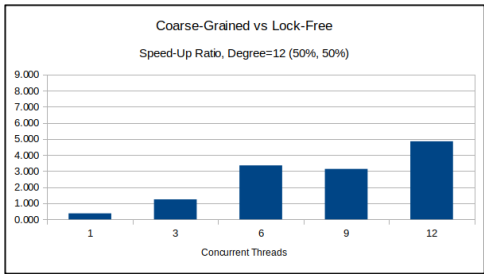


Figure 43: Performance: CC vs LF, Degree 12, Load (50%, 50%)

The heavy search job mix without delete (20%, 80%) results are shown in the following figures.

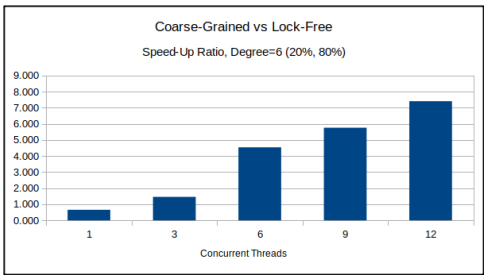


Figure 44: Performance: CG vs LF, Degree 6, Load (20%, 80%)

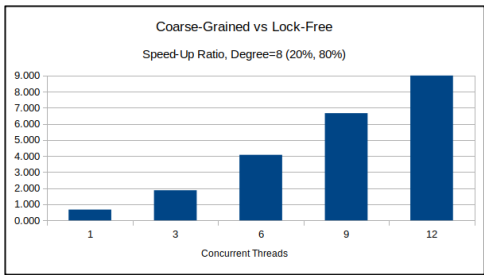


Figure 45: Performance: CG vs LF, Degree 8, Load (20%, 80%)

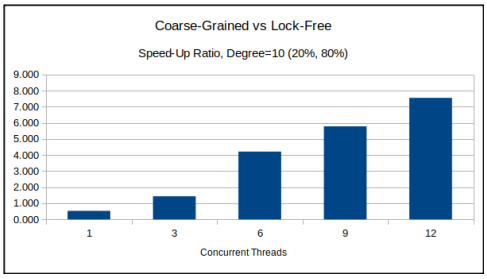


Figure 46: Performance: CG vs LF, Degree 10, Load (20%, 80%)

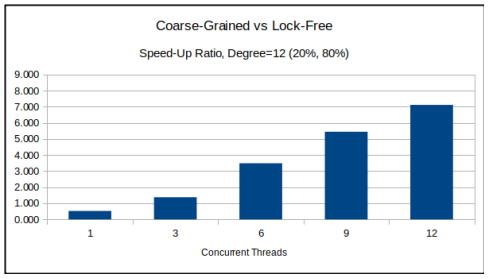


Figure 47: Performance: CG vs LF, Degree 12, Load (20%, 80%)

The even load job mix with delete (33.3%, 33.3%, 33.3%) results are shown in the following figures.

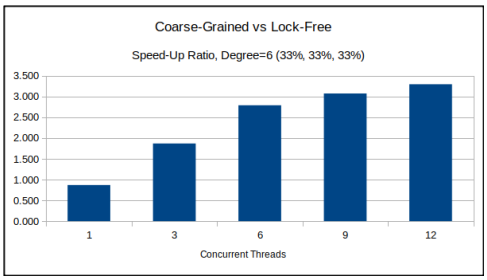


Figure 48: Performance: CG vs LF, Degree 6, Load (33.3%, 33.3%, 33.3%)

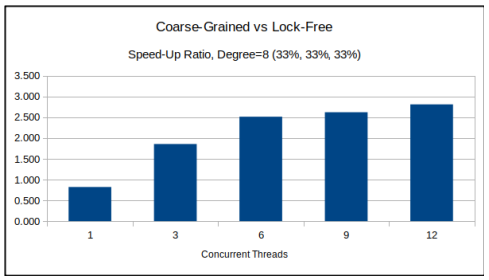


Figure 49: Performance: CG vs LF, Degree 8, Load (33.3%, 33.3%, 33.3%)

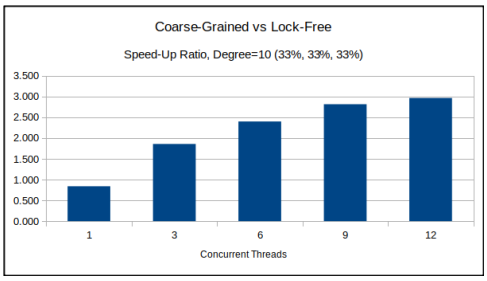


Figure 50: Performance: CG vs LF, Degree 10, Load (33.3%, 33.3%, 33.3%)

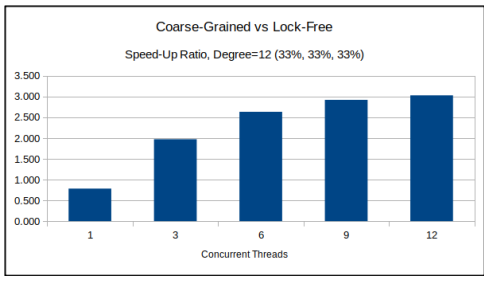


Figure 51: Performance: CG vs LF, Degree 12, Load (33.3%, 33.3%, 33.3%)

The heavy search job mix with delete (10%, 80%, 10%) results are shown in the following figures.

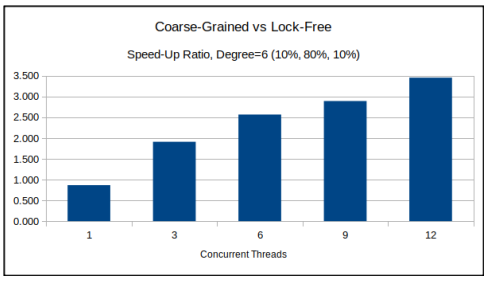


Figure 52: Performance: CG vs LF, Degree 6, Load (10%, 80%, 10%)

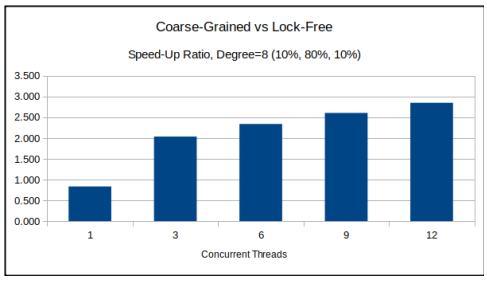


Figure 53: Performance: CG vs LF, Degree 8, Load (10%, 80%, 10%)

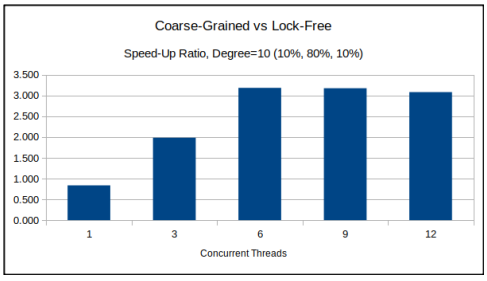


Figure 54: Performance: CG vs LF, Degree 10, Load (10%, 80%, 10%)

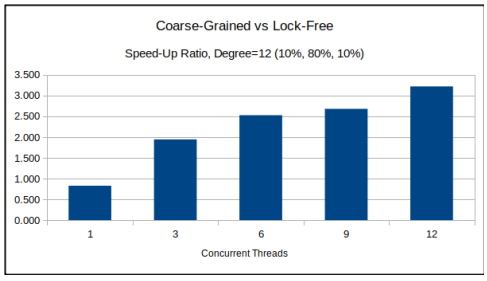


Figure 55: Performance: CG vs LF, Degree 12, Load (10%, 80%, 10%)

The following section presents an analysis of the testing results.

7.2.2.1 Performance Analysis

As can be seen in all tests, the performance in a single threaded environment for the lock-free implementation was below the performance of the coarse-gained implementation. This is due to the added complexity of the lock-free algorithm and overhead of the CAS operations for data structure updates. A single threaded implementation does not allow potential efficiencies from concurrency which is why the coarse-gained implementation outperforms the lock-free in a no or minimal contention environment.

As the concurrency increased, the speed-up increased and the lock-free implementation out-performed the coarse-gained implementation. Based on the no delete operation and different job mixes, the performance speed-up was between 1.25 and over 9 (9.09) times. The heavy search, no delete job mix provided the best performance. The non-blocking nature of the lock-free approach reduces interference with other simultaneously executing threads.

With the delete operation included, the results show that the lock-free with fine-grained locking approach outperforms the all fine-gained approach especially for higher concurrency environments.

8.0 Chapter 8,

Conclusion

8.0 Chapter 8, Conclusion

8.1 Summary

Due to the comparative flat tree structure and sub-logarithmic search time, a B-Tree is an especially ideal data structure for searching large-scale data sinks. The B-Tree data structure is commonly used in databases and file systems and is well suited for use with any large data repository. The inherent complexity of a standard B-Tree makes it a challenge for concurrent implementations. This dissertation examined the concurrency approaches for a standard B-Tree including the coarse grained, fine-grained locking, and the lock-free techniques. A reference implementation for the standard coarse-grained approach was developed and used as a base-line. A complete customized fine-grained algorithm was developed and presented. A lock-free implementation was developed and presented. A comprehensive review of the lock-free issues as applied to the implementation was discussed including how the ABA problem was addressed. A comparison of the trade-offs was presented and discussed. The final part of this effort addresses the specific testing processes which were discussed and presented.

8.2 Future Work

The primary future work should focus on a lock-free deletion algorithm. This should be possible when a practical double compare-and-swap operation [41] becomes available. Furthermore, the existing implementations should be extended in order to add support for other tree operations such as shadowing and cloning [42] and snapshots [43].

The existing work should be expanded to support external data structures in a lock-free efficient manner. It is hoped that this work can be expanded into more general techniques for implementing concurrent approaches for other highly complex data structures. A general process for addressing concurrency for more complex data structures would ease the implementation burden.

Bibliography

[1] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes, *Acta Information*, 173-189, 1971.

[2] Douglas Comer. Ubiquitous B-Tree, *ACM Computing Survey*, 121-137, 1979.

[3] Jim Gray and Andreas Reuter. Transaction Processing: Concepts and Techniques, *Morgan Kaufmann Publishers Inc.*, 1992.

[4] Wedekind, Hartmut. On the selection of access paths in a data base system, *Data Base Management*, 385-397, 1974.

[5] Jaluta, Ibrahim & Sippu, Seppo & Soisalon-soininen, Eljas. Concurrency control and recovery for balanced B-link trees, *The Vldb Journal - VLDB*, 257-277, 2005.

[6] Michael Folk and Bill Zoellick. *File Structures (2nd ed.)*, Addison-Wesley, 1992.

[7] Donald Knuth. *The Art of Computer Programming, Volume 2 (Second ed.)*, MIT Press and McGraw-Hill, 1998.

[8] Michael Bender, Erik Demaine, and Martin Farach-Colton. Cache-Oblivious B-Trees, *SIAM Journal on Computing*, 341-358, 2005.

[9] Alfred Aho, John Hopcroft, and Jeffrey Ullman. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[10] B. Samdi. B-trees in a system with multiple users, *Information Processing Letters*, 107-112, 1976.

[11] Rudolf Bayer and Karl Unterauer. Prefix B-Trees, *ACM Transactions on Database System*, 11-26, 1977.

[12] C. S. Elliot. Concurrent search and insertion in 2–3 trees, *Acta Informatica*, 63-68, 1980.

[13] Philip L. Lehman and S. Bing Yao. Efficient locking for concurrent operations on B-trees, *ACM Transactions on Database Systems*, 650-670, 1981.

[14] K. Fraser. Practical Lock-Freedom, *University of Cambridge, Computer Laboratory*, 2004.

[15] Anastasia Braginsky and Erez Petrank. A Lock-free B+Tree, *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, 58-67, 2012.

[16] Peter S Pacheco, *Parallel Programming with MPI*, Morgan Kaufman Publishers, Inc., 1997.

[17] William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir, *MPI - The Complete Reference, Volume 2*, MIT Press, 1998.

[18] D.M. Tullsen, S.J. Eggers, and H.M. Levy. Simultaneous multi-threading: Maximizing on-chip parallelism, *22nd Annual International Symposium on Computer Architecture*, 392-403, 1995.

[19] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming, Revised First Edition*, Morgan Kaufmann Publishers, 2012.

[20] Wikipedia contributors, Lock (computer science), 2019, https://en.wikipedia.org/w/index.php?title=Lock_(computer_science)

[21] D. Tullsen, J. Lo, S. Eggers, and H. Levy. Supporting Fine-Grained Synchronization of a Simultaneous Multi-threading Processor, *Proceedings of the 5th International Symposium on High Performance Computer Architecture*, 1-12, 1991.

[22] M. Herlihy. Wait-free Syncronization, *ACM Transactions on Programming Languages and Systems*, 124-149, 1991.

[23] Harris, Tim, Larus, James, Rajwar, Ravi. Synthesis Lectures on Computer Architecture, *Transactional Memory, 2nd edition*, 1-263, 2010.

[24] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms (Second ed.)*, MIT Press and McGraw-Hill, 2001.

[25] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-Free Synchronization: Double-Ended Queues as an Example, *23rd International Conference on Distributed Computing Systems*, 522, 2003.

[26] D. Dechev. The ABA problem in multicore data structures with collaborating operations, *7th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom)*, 158-167, 2011.

[27] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. Understanding and Effectively Preventing the ABA Problem in Descriptor-Based Lock-Free Designs, *13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, 2010.

[28] Mark Allen Weiss. *Data Structures & Algorithm Analysis in C++ (4th ed.)*, Pearson Education, 2012.

[29] Anthony Williams. *C++ Concurrency in Action*, Manning Publications Co., 2012.

[30] Marcel Kornacker, and Banks Douglas. High-Concurrency Locking in R-Trees, *VLDB '95 Proceedings of the 21th International Conference on Very Large Data Bases*, 134-145, 1995.

[31] Maurice Herlihy and Jeannette Wing. Linearizability: A Correctness Condition for Concurrent Objects, *ACM Trans. Program. Lang. Syst.*, 463-492, 1990.

[32] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A Provably Correct Scalable Concurrent Skip List, *OPODIS '06: Proceedings of the 10th International Conference On Principles Of Distributed Systems*, 801-810, 2006.

[33] Nathan Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A Practical Concurrent Binary Search Tree, *Proceedings of the 15th ACM SIGPLAN Symposium on Principals and Practice of Parallel Programming*, 257-268, 2010.

[34] Christian S. Jensen, Dan Lin, and Beng Chin Ooi. Query and update efficient B+-tree based indexing of moving objects, *Proceedings of the Thirtieth International Conference on Very Large Data Bases*, 768-779, 2004.

[35] Maya Arbel and Hagit Attiya. Concurrent Updates with RCU: Search Tree as an Example, *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, 196-205, 2014.

[36] Douglas Comer. Ubiquitous B-Tree, *ACM Computing Survey*, 121-137, 1979.

[37] R. Durstenfeld. Algorithm 235: Random permutation, *Communications of the ACM 7*, 420, 1964.

[38] Aravind Natarajan and Neeraj Mittal. Fast Concurrent Lock-free Binary Search Trees, *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 317-328, 2014.

[39] Michael Bender, Jeremy Fineman, Seth Gilbert, and Bradley Kuszmaul. Concurrent Cache-oblivious B-trees, *Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, 228-237, 2005.

[40] John Hennessy and David Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 2012.

[41] Timothy Harris, Keir Fraser, and Ian Pratt. A Practical Multi-Word Compare-and-Swap Operation, *Springer-Verlag*, 265-279, 2002.

[42] Ohad Rodeh. B-Trees, Shadowing, and Clones, *ACM Transactions on Storage (TOS)*, 1-27, 2008.

[43] Aleksandar Prokopec, Nathan Grasso Bronson, Phil Bagwell, and Martin Odersky. Concurrent Tries with Efficient Non-blocking Snapshots, *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 151-160, 2012.

Curriculum Vitae

Graduate College
University of Nevada, Las Vegas

Edward R. Jorgensen II
e-mail: ed.jorgensen1@gmail.com

Degrees:

Master of Science in Computer Science 1992
University of Nevada, Las Vegas

Bachelor of Science in Computer Science 1984
University of Nevada, Las Vegas

Dissertation Title: Coarse-Grained, Fine-Grained, and Lock-Free Concurrency Approaches for Self-Balancing B-Tree

Thesis Examination Committee:

Chairperson, Dr. Ajoy Datta, Ph.D.
Committee Member, Dr. John Minor, Ph.D.
Committee Member, Dr. Laxmi Gewali, Ph.D.
Committee Member, Dr. Sidkazem Taghva, Ph.D.
Graduate Faculty Representative, Dr. Emma Regentiva, Ph.D.