Master Thesis

Thread-safe tree-based collections in C#

Authors: Amund Ranheim Lome and Flaviu Muresan
Supervisor: Peter Sestoft, PhD

IT University of Copenhagen
Copenhagen, Denmark

A thesis submitted in fulfillment of the requirements for the degree of Master of Science.

May 2016

Abstract

We describe and implement two different concurrent dictionaries based on balanced trees. The first dictionary is a concurrent lock-free red-black tree implemented using compare-and-swap. The second is a concurrent chromatic tree using the LLX, SCX and VLX custom-defined synchronization primitives and an atomic subgraph replacement template. These primivites are multi-word extensions of the LL, SC and VL primities and are implemented using single-word compare-and-swap. We then extend the chromatic tree with the basic sorted dictionary operations *successor*, *predecessor*, *findMin*, *findMax* and *containsAll*, as well as two types of range queries - *snapshot query* and *dynamic query* - and three enumerators. We also introduce a benchmarking framework for concurrent dictionaries and present the relative performance results of the data structures considered in this paper.

Contents

List of Figures

List of Tables

1 Introduction

The purpose of this paper is to analyze and describe different implementations of concurrent balanced trees and propose a candidate to be included in a future version of a thread-safe C5 Collection library. Further, the selected data structure is extended with basic sorted dictionary operations and *range querying* functionality similar to that offered by the sequential TreeDictionary in C5.

1.1 Target Group

The target group consists of readers with an interest in the fields of concurrent programming and concurrent data structures, with a focus on implementations in C#. The paper assumes that the readers possess a solid understanding of balanced tree structures and are familiar with related concepts such as insertion, deletion, rebalancing, range queries, ordered iterators, etc.

The paper also serves as documentation for anyone interested in extending the work presented here.

1.2 Background and Motivation

With the breakdown of Dennard scaling in 2005-2007, increases in CPU performance has mainly been accomplished by adding more cores to each CPU rather than through an increase of the single-core processing speed. This means that existing sequential dictionary implementations can not exploit the full capabilities of the modern multi-core CPUs. The increasing amount of software relying on parallel computing therefore calls for thread-safe data structures to ensure their correct functioning in a concurrent setting.

The C5 Generic Collection Library [1] includes a sorted tree-based dictionary, but it is not thread-safe. C# includes a ConcurrentDictionary [2], but it is not sorted. Java includes a ConcurrentSkipListMap [3], but its worst case running time for insertion, deletion and finding an element is $O(n)$ rather than the $O(\log n)$ running time possible for a tree-based dictionary. These observations indicate that there is still a need for a reliable, provably correct concurrent tree dictionary in C#.

1.3 Structure

Related work on the topic is provided in Section 2. Sections 3 and 4 describe two different approaches to designing a concurrent balanced tree and present both a theoretical discussion and analysis as well as initial performance results for each approach. After a comparison of the two approaches, one of the two data structures is chosen to be extended. In section 5 the functionality extensions added to the proposed data structure is introduced.

In Section 6, two more data structures are briefly described. These data structures are not central to this paper, but they were considered relevant for inclusion in the performance tests. Section 7 introduces the benchmarking framework used for performance testing, a detailed overview of the results obtained for all data structures presented in this paper, and our interpretation of these results.

2 Related Work

The two most common approaches used to design dictionaries (both sequential and concurrent) use a hash table or a tree as their underlying data structure. The first approach uses hash functions for mapping keys to values and are usually employed when constant-time lookup is needed and the keys do not need to be sorted in a specific order. The latter uses node-based tree structures, maintain the keys sorted and offer functionality for range querying and ordered iteration. This paper focuses solely on tree structures implementing concurrent sorted dictionaries that provide specific operations as those mentioned above.

Ellen et al. [4] describe an implementation of a concurrent non-blocking binary search tree. Structured as a leaf-oriented binary search tree, it uses a concurrency control mechanism based on single-word compare-and-swap as well as a cooperative technique similar to Barnes[5]. Along with the implementation, the authors also provide a full proof of correctness. In a later publication[6], the same authors introduce concurrent *k-ary* search trees as a generalization of the earlier binary trees, while still following the same approach and a similar proof of correctness. However, all these are non-balanced trees resulting in an $O(n)$ worst-case time complexity for their operations.

To accomplish an improved time complexity of $O(\log n)$, balanced trees were considered in several research publications. A concurrent CAS-based *red-black tree* is introduced by Kim et al. [7]. While the paper presents some parts of pseudocode, it does not provide a working implementation, nor any benchmarking results. The authors do provide a sketched proof of correctness for the algorithm. Besa [8] describes an alternative to this approach consisting of a lock-based red-black tree using a number of new rebalancing operations. Both algorithms require that a process executing an *insert* or *delete* operation must continue by rebalancing the tree. A new technique suggested by Hanke[9] decouples the actual operation from the rebalancing procedure. Thus, operations are broken down into smaller suboperations that can be more easily interleaved, thereby increasing the performance under high contention. The tree by Hanke still maintains all the constraints of a red-black tree.

A new type of tree called *chromatic tree* is first introduced by Nurmi and Soisalon-Soininen[10]. Here, the constraints are relaxed compared to the previous trees - the colors are replaced by weights which makes the number of allowed violations on a path configurable. Brown et al. [11] provide a concurrent algorithm and implementation of this tree using two new primitives derived as extensions from LL/SC as well as the rebalancing steps introduced by Boyar and Larsen[12]. They give a complete and easy to understand proof of correctness for both the algorithm and the new primitives. Later[11], they introduce a framework capable of converting any sequential down-tree to a concurrent one using the same primitives.

3 Lock-free red-black tree using CAS

The fact that updates to a red-black tree only require structural changes locally in small confined areas of the tree makes this data structure a good candidate for representing a concurrent tree dictionary. A natural first intuition when designing such a lock-free red-black tree would be to prevent concurrent access in the localized areas required by these updates. Previous attempts in this direction were made by Ma [13], who describes an algorithm using DCAS (double compare-and-swap) and TCAS (triple compare-and-swap). Since these synchronization primitives are not available on most processors, further research was conducted to eliminate them and only keep the widely available single-word compare-and-swap (CAS). The algorithm introduced by Kim et al. [7] that will be presented in the sections below describes such a technique.

We consider it to be of interest for multiple reasons. From a general perspective, this algorithm's approach could be adapted to design other lock-free multi-linked data structures, also besides the particular case of trees, and it represents perhaps the most intuitive way of using CAS to accomplish this. The analysis of the algorithm also gives a deep insight into the complexity and issues that arise when designing both this and other comparable concurrent structures.

3.1 The Algorithm

The algorithms introduced by Ma [13] and Kim [7] are both based on the sequential red-black tree described by Cormen et al. [14]. In addition, they introduce the concept of a *local area*. A local area represents the vicinity of a node where an update occurs, and consists of all the nodes that must not change while the local operation is in progress. The size of the local area will therefore vary depending on the type of operation. In order to handle the concurrency control within and among the vicinities, Kim et al. [7] add two additional fields to each node, a *flag* field and an *intention marker* field. The following sections will describe these concepts in more detail and describe their necessity in a concurrent context.

3.1.1 Local area and flags

A more formal definition of the local area would be that it comprises of all the nearby nodes that a process has exclusive modification rights to while performing an operation.

The local area for insertion depicted in Figure 1 is determined by the union of all accessed nodes in the three insertion cases defined by Cormen et al. The figure includes both the nodes that are accessed during restructuring and recoloring, as well as those that are needed to determine which insertion case the process is in. Thus, when inserting or fixing up at node x, a process may need to hold some or all of the nodes in the local area to safely update the tree.

Similarly, Figure 2 illustrates the local area required by the *delete* operation.

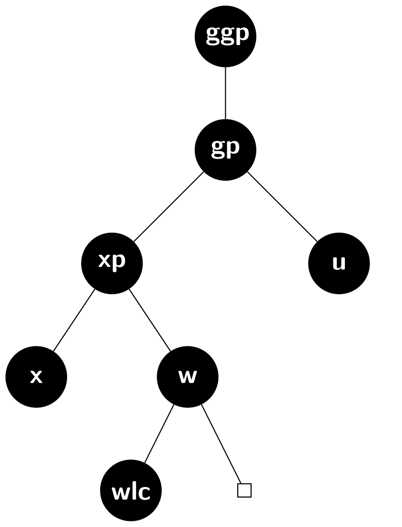


Figure 1: Local area for insertion case originating at x

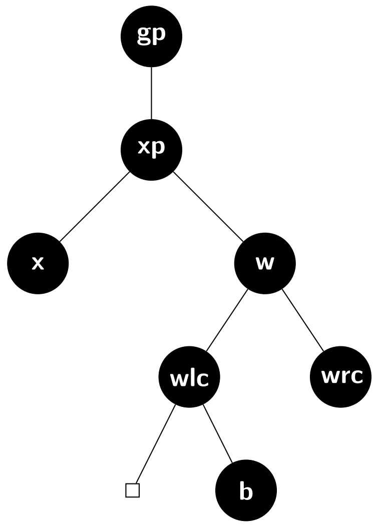


Figure 2: Local area for deletion case originating at x.

To safely acquire the necessary nodes, a process must set its flag on all the nodes contained in the local area using CAS. Setting the flag signals to other processes that the node is currently held by the flagging process and cannot be modified by any other concurrent processes. However, one of the implications of only using single-word CAS is that flags cannot be set on all the local area's nodes atomically. Instead, a process will have to optimistically start setting its flags on the nodes, and in case of failure release all its currently held flags and retry from the start. After all flags for the current local area are acquired and the rebalancing operation has succeeded, the process might have to move its local area upwards in the tree if it has created a new violation above its current position. While attempting to get the new local area, the process continues to hold the flags it acquired in its previous local area. This approach is prone to *deadlock* if two processes are moving up into overlapping local areas. An example of this is depicted in Figure 3, where process P_1 has performed insertion case 1 and P_2 has performed deletion case 2 [15]. To move its local area up, P_1 must acquire the flag on $P_2.xp$ which will be the new x's uncle. Similarly, P_2 must acquire the flag on $P_1.gp$ which will be the new x's sibling's left child. Since both of these nodes are held by the other process and will not be released until the local area is successfully moved up, this results in a deadlock. In order to address this problem, Kim et al. [7] introduce a mechanism relying on *intention markers* to ensure a safe distance between concurrent processes.

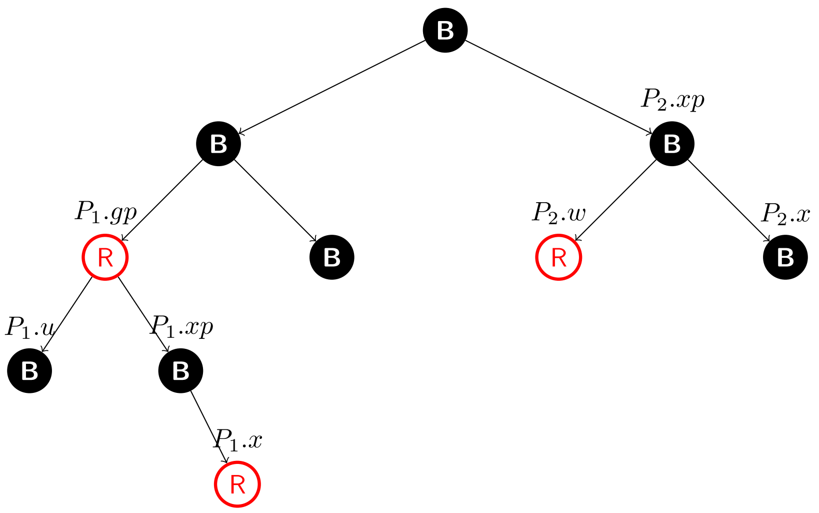


Figure 3: Local area deadlock

3.1.2 Intention markers

Each node has an *intention marker* which can be set by a process to indicate its intention to move its local area to that node. By analyzing all possible scenarios for local area acquisition, the paper proves that taking four markers above the current local area is both sufficient and necessary to ensure the required distance between two processes. However, multiple concurrent rotations executed by *insert* and *delete* operations might result in a "double marker problem", as referred to by Kim et al. [7]. A double marker problem occurs when two markers are simultaneously set on a single node, thus violating the constraint that only one process can have its marker on a node at any given point. This new problem is solved by introducing two rules that must be satisfied when acquiring a marker, the *spacing rule* and the *move-up rule*.

3.1.3 Spacing rule and Move-Up rule

The role of the *Spacing rule* is to prevent two processes from acquiring local areas too close to each other. In other words, it ensures that between any two concurrent processes operating in the tree there will be at least one node whose marker and flag fields are not set. The rule's constraints require that a marker can only be placed on a node if

1. The node's own marker and flag is not set
2. The node's parent's marker and flag is not set
3. The node's sibling's marker and flag is not set

While the Spacing rule will eliminate the double marker problem in an *insert*-only tree, it is not sufficient to ensure correctness in a tree where *insert* and *delete* operations are being executed concurrently. Due to the effects of rotations higher up in the tree, two processes moving up could be brought too close to each other by a third process operating above. In this case, the *Move-Up rule* will only allow one of the two processes to continue moving up, while the second process will have to wait. This rule is especially challenging from a technical point of view, as it requires inter-process communication.

3.2 Implementation

Kim et al. [7] introduced the algorithm described in this section along with an incomplete pseudocode closely resembling C syntax. Based on this pseudocode and the description of the algorithm, we completed it to a working, object-oriented implementation which we present below.

We chose to model a node by defining a `Node` class as shown in Listing 1. Besides the key and value, a node holds both child and parent references, a boolean value indicating the color and two integers representing the flag and marker.

Listing 1: **Node** class

```
class Node<K, V> {
    K key;
    V value;
    bool red;
    Node childLeft, childRight, parent;
    int flag;
    int marker;
}
```

As this implementation of a red-black tree is based on the algorithm introduced by Cormen et al. [14], it also requires a special type of node referred to as NIL to represent external nodes. While the original algorithm suggests using a single sentinel NIL, our implementation closely follows Kim et al. by using new instances for all the occurrences of this special node. These instances are created using the `Nil` subclass of `Node` presented in Listing 2.

Listing 2: **Nil** class

```
private class Nil : Node {

    public Nil() {
        SetBlack();
        Parent = this;
        ChildLeft = this;
        ChildRight = this;
        Flag = 0;
}
```

This approach is intended to eliminate some of the contention on the NIL node as well as improve handling of a number of special cases where NIL nodes are part of the local area. One special note here is that the NIL in Cormen et al. is both the parent of the root and the children of leaf nodes, meaning it might be flagged both by a process reaching the leaf as well as by a process starting at at the root. Using multiple NIL nodes avoids this problem.

To maintain a close correspondence between the concepts used by the algorithm and the implementation, we represent a local area as shown in Listing 3.

Listing 3: **LocalArea** class for insertion

```
class LocalArea {
    Node self, parent, uncle, gp, ggp, gggp;
    Node firstMarked, secondMarked, thirdMarked, fourthMarked;
}
```

The `Node`, `Nil` and `LocalArea` classes form the basis of the red-black tree representation and help in supporting all its operations. The following section will closely describe how they are employed to *insert* or *find* an element in the tree.

3.2.1 Insert operation

The red-black tree *insert* operation consists of two main subroutines. The first is identical to a classical binary search tree insert. But as this operation by itself could potentially leave the tree unbalanced, it is followed by one or several rebalancing operations performed by the second routine.

```
Input: key, value
begin
    newNode ← Initialize the node to be inserted;
    repeat AcquireFlag(root) until true;
    if root has changed then
        ReleaseFlag(root) ;
        Restart()
    end
    parent ← NIL;
    node ← root;
    while not found insertion point do
        ▷ Assert we hold flags on both current parent (if not NIL) and
         current node;
        if node.key = key then
            Update();
            ReleaseFlags(node, parent) ;
            return
        else
            ReleaseFlag(parent);
            parent ← node;
        end
        ▷ Move CurrentNode down;
        if not AcquireFlag(node) then
            ReleaseFlag(parent);
            Restart();
        end
    end
    ▷ Assert CurrentNode is flagged;
    newNode.parent ← parent ;
    if parent is NIL then
        ▷ Inserting root ;
        Assert NIL is flagged ;
        root = node ;
        ▷ Set the root black ;
        ReleaseFlags(root, NIL) ;
        return
    else
        if not SetupLocalAreaInsert(parent) then
            ReleaseFlags(node, parent);
            Restart();
        end
        Execute insertion by setting the references
    end
    FixupInsert(newNode);
end
```
Algorithm 1: Lock-free implementation of the *insert* operation

The pseudocode above presents the concurrent version of the first subroutine, which inserts a given key-value pair as a new node in the tree. It starts by creating a new `Node` object holding the given key and value. The correct insertion point is then found by using a “hand-over-hand” flagging approach with two node references, *parent* and *node*, which are continuously reassigned while traversing the tree downwards. The process begins by acquiring the flag on the root node and sets it as the first *node* before continuing downwards. All flag acquisitions are done using the `AcquireFlag()` method, which takes a node as parameter and returns a boolean value depending on whether or not the operation succeeded. Every iteration will attempt to acquire the flag on one of the current node’s children, determined by comparing its key to that of the node to be inserted. In case of failure to acquire the necessary new flag, the flags on the parent and the current node are released by the `ReleaseFlags()` meyhod and the entire operation restarts. Otherwise, a leaf representing the insertion point has been found and its entire local area must be flagged for insertion. This is accomplished by calling the `SetupLocalAreaInsert` method shown in Algorithm 2.

```
Input: $node$
begin
    $localArea \leftarrow$ empty local area;
    $localArea.parent \longleftarrow node$;
    $grandParent \leftarrow node.parent$;
    if not $AcquireFlag(grandParent)$ then
        return false
    end
    $localArea.gp \leftarrow grandParent$;
    if $grandParent$ has changed then
        $ReleaseFlagsLocalArea(localArea)$;
        return false
    end
    $uncle \leftarrow GetSibling(node)$;
    if not $AcquireFlag(uncle)$ then
        $ReleaseFlagsLocalArea(localArea)$;
        return false
    end
    $LocalArea.uncle \leftarrow uncle$;
    if $uncle$ has changed then
        $ReleaseFlagsLocalArea(localArea)$;
        return false
    end
    if not $GetFlagsAndMarkersAbove(grandParent)$ then
        $ReleaseFlagsLocalArea(localArea)$;
        return false
    end
    return true
end
```

Algorithm 2: Setup local area for insert

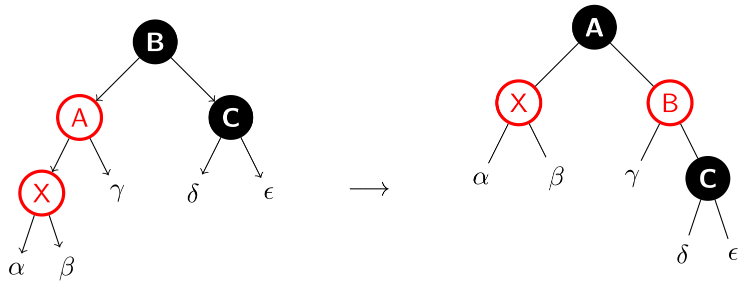
The `SetupLocalAreaInsert` method is called with one argument representing the insertion point, i.e. the newly inserted node's parent. When entering the method, it is assumed that both the inserted node and its parent are already flagged by the current process. It will then attempt to acquire flags on the rest of the inserted node's local area. As a general pattern used throughout the algorithm to acquire flags on local area nodes, three steps are involved: start by obtaining a reference to the node, then acquire the flag and finally verify that the node is still in the same relative position to the starting node. A failure in any of the last two steps results in releasing all the flags acquired so far and restarting the entire operation (in this case *insert*). If all the necessary flags are acquired, the operation continues by calling `GetFlagsAndMarkersAbove` with the grandparent of the inserted node as argument. This method then attempts to acquire flags on the four nodes above the one received as argument, and then acquire markers on the next four nodes above while ensuring that the *spacing rule* is satisfied.

If the entire local area and the necessary markers are successfully acquired, the process continues by initiating the *fix-up* procedure. This second part of the *insert* operation is responsible for fixing potential red-black violations and thereby rebalance the tree. According the Cormen et al. [14] there are six cases of violations, each of them requiring either a recoloring or one/two rotations to eliminate or push the conflict upwards. Before applying any of these rebalancing operations, a process must first flag the nodes in the entire local area of the violation. It is assumed at the start of the *fix-up* procedure that the process already holds the flags on the the local area of the insertion point. As it might have to continue fixing violations upwards from its starting position, it must also be able to *move* its local area up accordingly. This is done by the `MoveInserterUp` method, which also uses the same `GetFlagsAndMarkersAbove` method to acquire the necessary flags and markers.

The actual implementation of `GetFlagsAndMarkersAbove` is at this point not novel and its process of acquiring flags and markers uses the same three-step pattern as shown above in `SetupLocalAreaInsert`. Still, there are some important observations regarding the method. After acquiring flags on the first four nodes above the argument, the next four must be marked in accordance with the spacing rule. The rule uses flags in order to ensure that the nodes being verified do not change during the process of setting markers, which leads to a large number of flags that must be acquired at various points in time in the method execution. For each attempt to move a local area up, there is a total of 12 nodes directly above each other that must be flagged at one point or another in order to succeed. Any failure in acquiring a flag triggers the restart of the operation. Under high contention and a relatively small initial size of the tree, an operation using this method can be expected to require a significant number of retries to complete. This is discussed further in Section 3.3.

3.2.2 Find operation

The paper from Kim et al. does not describe an implementation of the *find* operation. As we considered it to be relevant for benchmarking the data structure, we decided to add one based on the tree traversal to the insertion point employed in *insert*. Our implementation uses the same “hand-over-hand” flagging approach to find a node containing the given key and return its value. The figure below shows why this kind of safe traversal is needed and why a sequential traversal is not enough to ensure the operation’s correctness.



Process P_1 starts a search for the key of A and arrives at node B.

Process P_2 immediately executes a right rotation on B. A is no longer on the search path of P_1.

Table 1: Timeline of events for inconsistent *find*

T_1	P_1 arrives at node B
T_2	P_2 executes a right rotation on B.
T_3	P_1 compares B's key to its argument A and decides to continue left
T_4	P_1 continues in γ
...	...
T_n	P_1 does not find A and returns false despite A being in the tree

The timeline above shows a sequence of events that causes an incorrect result to be returned. A right rotate operation by P_2 interferes with the search of P_1, causing A to be placed above P_1's current position on the search path. Ensuring P_1 acquires the flag on B prevents this problem from occurring as node A cannot be moved without owning the flag on its parent.

The pseudocode for the *find* algorithm is almost identical to the part of *insert* in Listing 1 that finds the correct insertion point, and it is therefore not repeated here.

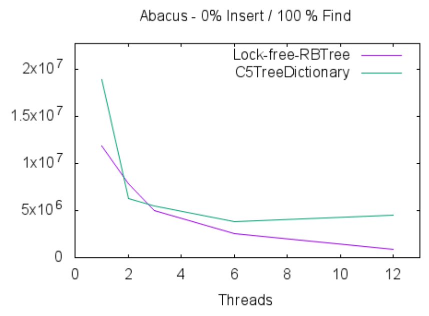
3.2.3 Delete operation

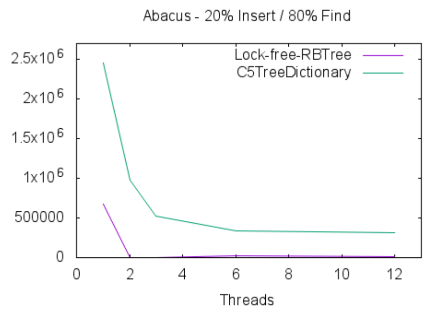
We chose not to implement the *delete* operation due to the two reasons outlined below. First, the *delete* algorithm has an extremely high level of complexity, making implementing or reasoning about its correctness particularly difficult. While the *insert* operation can be implemented using the flags, markers and *Spacing rule* already presented, the deletion operation also requires the *Move-Up rule*. Kim et al. [7] describe the main parts of the *delete* operation both theoretically and in pseudocode. However, its most complex part, the inter-process communication and the data structure necessary for it to work, is mostly assumed rather than described, the authors merely stating in a footnote that it is "subtle but do-able". Second, we considered that the poor benchmarking results shown in Section 3.3 for the *insert* and *find* operations were already conclusive, especially considering that any additional *Move-Up* rule implementation could only decrease the overall performance.

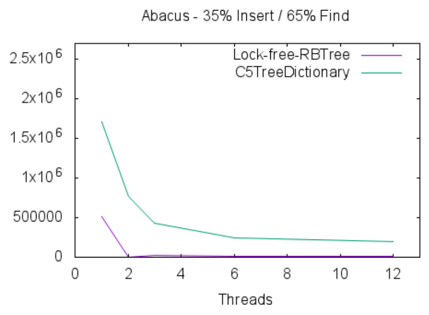
3.3 Experiments

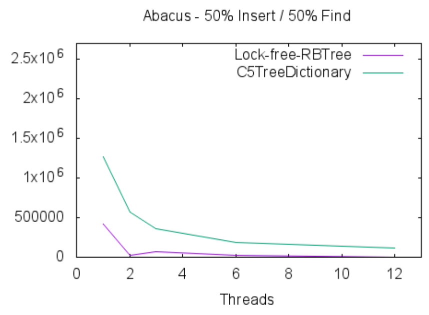
To evaluate the performance of the *insert* and *find* operations, we performed benchmark tests using four different *insert/find* ratios. The tests were performed alongside the synchronized C5 TreeDictionary wrapper class detailed in Section 6.1 for comparison. A comprehensive description of the benchmarking framework and test hardware can be found in Section 7.

All benchmark tests were performed on a prefilled tree of size 50,000, with random keys in the range of 0 to 100,000. Each thread works on a queue of 100,000 elements chosen randomly within the same range (duplicates are allowed), calling *insert* or *find* randomly based on the given ratios. Please note that the ranges on the y-axis differ between the graphs. This was necessary to make the results for the lock-free red-black tree visible.









When only performing *find* operations as shown in the top left corner, the lock-free red-black tree is outperformed by the synchronized C5 Tree Dictionary for almost all number of threads. For a single thread, the cost of performing hand-over-hand flagging is shown to be considerably higher than the cost of taking the uncontested lock for the wrapper class. As additional threads are added, it is clear that the lock-free red-black tree scales worse than the baseline synchronized data structure, which is not encouraging.

When *insert* operations are included, the sequential gap between the two data structures widens. And as soon as multiple threads concurrently access the lock-free data structure, its throughput falls sharply and the test is barely able to make progress.

It is worthwhile to note that for a complete binary tree (which is the best case minimum height of a red-black tree) consisting of 50,000 to 100,000 nodes, the height of the tree will be between 15 and 16 nodes as it is given by $h = \lfloor \log_2(n) \rfloor$ where n is the number of nodes. Even allowing for a doubling of the height in the worst case for a red-black tree ($h \leq 2\log_2(n+1)$), it becomes clear that flagging 12 nodes above the fixup point, as shown in section 3.2.1, will quickly compete with other processes for the nodes close to the top of the tree, including the root itself. Attempts at a benchmark run with a key range of 0 to 10,000 and a tree pre-filled to 5,000 nodes was unable to complete, as the lack of progress meant the benchmark took too long to finish. An increase of the tree size to 500,000 would lessen the contention somewhat, but the effect is unlikely to be very significant as the height only increases by around 3.3 each time the size is multiplied by 10.

Table 2: Node height of Red-Black Tree

Size	Best case	Worst case
5,000	12	24
50,000	15	31
500,000	18	37

3.4 Conclusion

In this section, we presented a first approach to designing a concurrent lock-free balanced tree structure. Theoretically, the algorithm has a relatively simple intuition: based on the well-known Cormen et al. [14] implementation of a red-black tree, it uses compare-and-swap to set flags and markers and ensure that a process has exclusive access to the nodes it requires to perform an update. In practice, it proved to be extremely difficult to implement and even more difficult to reason about its correctness.

One of the sources of the complexity is the large number of references that must be updated at each rebalancing step. Then, the fact that nodes are flagged both upwards and downwards leads to the potential for deadlock, which is solved by introducing the markers and thereby even more complexity.

A discussion with Dr. Helen Cameron and Dr. Peter Graham (co-authors of the paper introducing the algorithm) revealed that no additional research has been conducted in this direction since the publication of the second paper on the algorithm in 2011. Further, they mention that a complete working implementation was never developed. Given the difficulties experienced while implementing the algorithm by both us and the original authors as well as the poor performance results achieved by the *insert* and *find* operations, we do not consider this approach a promising candidate for further research.

4 Chromatic tree

Nurmi and Soisalon-Soininen [10] introduced a new type of binary tree which they call a *chromatic tree.* This data structure can be regarded as a generalization of the red-black tree with less strict constraints, thus being more contention-friendly.

Unlike the red-black tree, in which every node is either red or black, a chromatic tree introduces the concept of *node weight.* Following this principle, each node has an associated non-negative integer as its weight instead of a color. There is a direct correspondence between the weights in the chromatic tree and the colors in the classical red-black tree as the red and black colors are represented by the weights of 0 and 1 respectively. A chromatic tree might also allow weight values greater than 1. As for the red-black tree, there are specific constraints to ensure that the tree remains balanced.

- A red node should not have a red parent. A violation of this constraint is called a *red-red violation* and, in the case of the chromatic tree, it occurs when both the child and the parent have the weight 0.
- A node should have a weight of either 0 or 1. A violation of this constraint is called a *overweight violation.*

To increase the efficiency of the data structure, particularly under high contention, a specified number of violations are allowed to exist on a path from the root to a leaf node while still considering the tree to be balanced. The main benefit of this approach is that certain violations might be fixed by later updates without the need for any explicit fix-up. For example, an *insert* operation could fix an overweight violation caused by an earlier *deletion.* As the fix-up operations might potentially be very expensive and touch a large part of the tree all the way up to the root, this relaxation might result in a considerable performance gain.

A very elegant and efficient approach to designing a concurrent chromatic tree was introduced by Brown et al. [16]. The following subsection will detail the main underlying concepts, which also represent the foundation of the implementation presented in Section 4.2.

4.1 The Algorithm

Brown et al. [16] describe a general technique for designing non-blocking down-tree structures which are also provably correct. Their technique is based on the possibility to atomically update parts of the tree in a non-blocking way. However, it is only applicable to structures following the model outlined below.

The nodes of the tree must be represented as data records containing both *mutable* and *immutable* fields. As a consequence, some fields of this structure will be impossible to change, and any process attempting to update these values will be forced to reconstruct and then replace the node. For the chromatic tree currently under discussion, the mutable fields consist of the left and right child references. Both the *key* and *value* fields are immutable. As the tree must be a *down-tree*, the nodes do not hold a parent reference. This means that the structure can be regarded as a directed acyclic graph of indegree one, which is a property of the tree that will be used later.

4.1.1 LLX, SCX and VLX synchronization primitives

In order to design their above-mentioned general technique for constructing concurrent data structures, Brown et al. [17] first introduce three synchronization primitives. These primitives encapsulate most of the concurrency control in a concrete implementation together with most of the formal proof of correctness. Having these provably correct primitives makes reasoning about any data structure implementation that uses them (or extending such a data structure) a considerably easier task.

The three primitives - LLX, SCX and VLX - are multi-word extensions of the LL/SC primitives (load-link and store-conditional) and VL (validate). When applied on a data record, a full LLX returns a snapshot of all its mutable fields. A *weaker* form of the LLX need not return a complete snapshot, but may instead return a single value representing the SCX record stored in a specially designated field, effectively linking the record to a later SCX. This is sufficient when modifications to all other mutable fields are exclusively done through an SCX invocation on the returned SCX record. In both cases, the interface of the primitive has the form *LLX(rec)* where *rec* is a data record. The SCX primitive has the interface *SCX(V, R, field, new)*, where the first parameter V is a set of data records which must not have changed in order for the SCX to succeed. The next parameter R represents a subset of V containing data records which are to be *finalized* (i.e. marked as no longer being part of the tree) before successfully completing the SCX operation. The third parameter *field* references a mutable field held by a data record which is in V but not R. Upon completion of the SCX, the value held in *new* will be commited to the reference indicated by *field*.

The last primitive presented by Brown et al. [17] is the VLX, whose semantics follow a similar transformation pattern from single-word VL to a multi-word variant VLX as the aforementioned LL and SC primitives. *VLX(V)* accepts a set V of data records as an argument and only succeeds if none of the data records in the set has changed since the linked LLX.

4.1.2 Tree update template

The tree update template is a technique for atomically replacing an existing subgraph with a new one in any directed acyclic graph of indegree one.

One of the most important implications of the indegree-one property is that any subgraph can be replaced by a single update of the corresponding child reference in its parent. For this update not to remove the subgraph would imply that another node in the tree also held a reference to at least one of the subgraph's nodes, thus violating the requirement of indegree-one. This therefore serves as a proof by contradiction. Having to update only one reference means that the entire operation can easily be executed atomically, which significantly simplifies its proof of correctness.

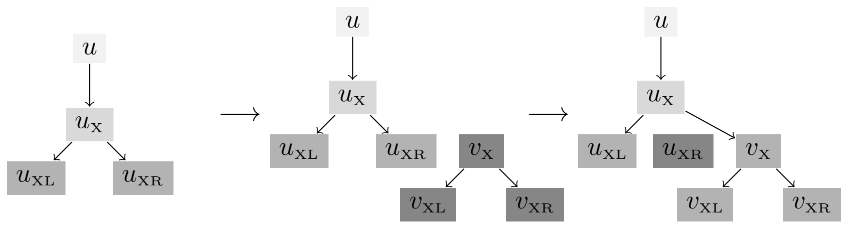


Figure 4: Subgraph replacement

The figure above shows an example where the subgraph $S = \{u_{\mathrm{XR}}\}$ is replaced with subgraph $S' = \{v_{\mathrm{X}}, v_{\mathrm{XL}}, v_{\mathrm{XR}}\}$. Notice that because the operation is replacing S, it must ensure at commit time that none of the nodes in S have changed. Further, as the actual update occurs at node u_{X}, it must also be validated before committing. This requirement ensures that no potential concurrent update to any of the nodes in S or the node u_{X} is lost, thus effectively preventing the *lost update* problem. The mechanism to accomplish this is first employing the LLX primitive to link these nodes to a subsequent SCX. As a consequence, the SCX will have to validate these nodes before committing, which leads to the conclusion that the parameter V required by SCX is $S \cup \{u_{\mathrm{X}}\}$. The next parameter R holds the set of nodes that must be finalized, and by analyzing the figure it is clear that $R = \{u_{\mathrm{XR}}\}$. Notice that $R \subset V$ and that upon successful completion of the tree update, the u_{XR} node will no longer be part of the tree. The third parameter is the *field* being updated which in this case is the *childRight* field of u_{X}. Finally, *new* must be the root of the new subgraph S', which is v_{X}.

SCX argument	In example graph
V	$\{u_{\mathrm{X}}, u_{\mathrm{XR}}\}$
R	$\{u_{\mathrm{XR}}\}$
field	$u_{\mathrm{X}}.\mathit{childRight}$
new	$\{v_{\mathrm{X}}, v_{\mathrm{XL}}, v_{\mathrm{XR}}\}$

Table 3: SCX arguments in the tree update template

This illustrates that the tree update template can be successfully implemented using the LLX and SCX primitives.

4.2 Implementation

Based on the template described above, Brown et al. [16] introduced a Java implementation of a concurrent chromatic tree. We adapted this implementation to C# including the necessary design modifications required by the new platform.

A brief overview of the tree's exposed interface is presented below.

Listing 4: **ConcurrentChromaticTreeMap** class

```
public class ConcurrentChromaticTreeMap<K, V> :
                  ISortedDictionary<K, V> {
   public void Insert(K key, V value);
   public bool Find(K findKey, out V value);
   public bool Delete(K deleteKey, out V val);
   public int Count();
}
```

The next subsections describe the implementation of the primitives in detail, followed by general aspects regarding the tree design and finally the implementation of its operations.

4.2.1 Nodes and primitives

As already suggested in the previous sections, the concurrency control in the data structure takes place at the *node level*. The fact that the node is the basic structural unit of the tree makes it a good candidate for implementing the Data-record specifications described in Section 4.1. A node is modelled by a `Node` class as shown in Listing 5

Listing 5: **Node** class

```
class Node
{
      public readonly NodeKey key;
      public readonly V value;
      public readonly int weight;
      public volatile Operation op;
      public volatile bool finalized;
      public volatile Node left, right;
}
```

A first observation is that a node no longer holds a reference to its parent, since the data structure is a down-tree. Unlike in the previously analyzed red-black trees, the nodes do not have an associated color. Instead, the *chromatic* property of the tree is reflected in the `weight` attribute. As required by the tree update template, the nodes have both mutable and immutable fields, implemented as *volatile* and *readonly* attributes respectively. Further, the attribute `finalized` holds a boolean value indicating whether the current data record has been *finalized.*

The previous sections also specify that each node must hold an SCX record, which stands at the core of the concurrency handling in the data structure. This requirement is captured in the `op` attribute, of type `Operation`.

An *operation* represents the record of an SCX operation, containing all the arguments required by the SCX primitive. Closely following the design of Brown et al. [16], the SCX record is implemented as an `Operation` class.

Listing 6: **Operation** class (SCX record)

```
class Operation
{
	public enum OperationState {
		STATE_INPROGRESS,
		STATE_ABORTED,
		STATE_COMMITTED
	};

	public volatile Node subtree;
	public volatile Node[] nodes;
	public volatile Operation[] ops;
	public volatile OperationState state;
	public volatile bool allFrozen;
}
```

There is a direct correspondence between the interface of the SCX primitive, *SCX(V, R, field, new)*, and the Operation class definition above. The parameter V is implemented as the array of nodes stored in `nodes`. In order to identify the next parameter of the SCX, R, a subtle observation is required regarding the tree update template and subgraph replacement. As shown previously, R is defined as the subset of V that is removed from the tree. By analyzing the tree update template, it becomes clear that in all instances there is only a single node in V that is not to be finalized, and that node is the parent of the subgraph to be replaced. This means that defining R as a new attribute is not necessary. Instead, by convention, the first element of the `nodes` array contains the parent of the subgraph, while the rest of the array consists of all the nodes that will form R, thus being finalized if the operation succeeds. Further, notice that the child to be replaced is always an element of R by being the root of the replaced subgraph. This leads to an extension of the convention, in that the second element of `nodes` is represented by this child node. Hence, explicitly passing the *field* reference as an argument is not necessary either. To briefly summarize, this convention exploits the fact that $R \subset V$ and $|V - R| = 1$ to merge all three arguments into a single one represented by the `nodes` array. The last argument for the SCX is the *new* reference which is the root of the new subgraph. This reveales that the new subgraph is entirely created before the SCX is invoked. The table below summarizes the mapping between the SCX parameters as defined in the interface and the `Node` objects passed as arguments in the concrete implementation.

Table 4: SCX arguments implementation

V	nodes[0] ... nodes[nodes.length-1]
R	nodes[1] ... nodes[nodes.length-1]
field	nodes[0].left **or** nodes[0].right
new	subtree to be added

The `ops` array contains the SCX records of all the nodes in V as returned by the last LLX on them. To validate that none of the nodes in `nodes` has changed, the SCX primitive attempts to update their `op` field to reference its own SCX record. This is done using a sequence of compare-and-swap operations, implemented in C# using `Interlocked.CompareExchange`. Each of these compare-and-swap invocations use the corresponding SCX record in `ops` to compare against, only updating if the values are the same. Once an SCX has successfully updated a node's `op` field, that node is considered to be *frozen* to that specific SCX. If all nodes in V are successfully updated, the `allFrozen` flag is set to `true`.

Freezing a node might at first seem similar to locking a node. However, in contrast to locking, freezing is part of a collaborative mechanism designed such that processes can help rather than compete against each other. Before detailing the mechanism, it is important to mention that each SCX record also holds a `state` attribute, which could be either *InProgress*, *Committed* or *Aborted*, as shown in Listing 6. Initially, every new SCX record has the state *InProgress*, which only ever changes to one of the remaining two states upon the operation's completion, as shown in Figure 5.

When a node is frozen, the SCX record referred by its `op` field has the state *InProgress*. An important observation is that an SCX record holds all the information necessary for any process to complete the operation. This means that if the process that initiated the operation fails, another process can complete it at a later point.

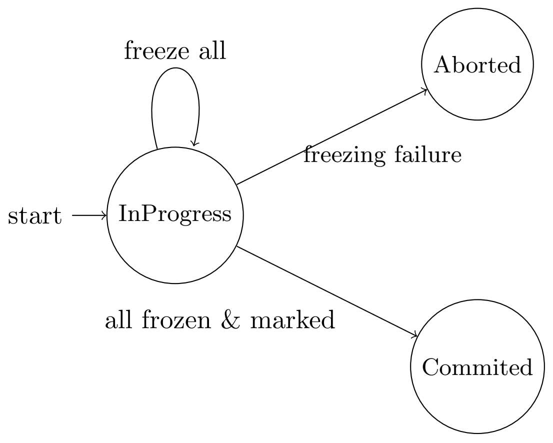


Figure 5: State Diagram for Operation States

This collaborative mechanism is accomplished through the LLX invocation. If the SCX record held in the node's `op` field is either committed or aborted, the LLX simply returns it. However, if the operation's state is *InProgress*, the process will attempt to complete the operation using the information stored within it before continuing.

4.2.2 Node keys and comparator

As expected, since each node in a tree data structure must contain both a key and a value, the `Node` class defines an attribute for each, as shown in Listing 5. The second of these - `value` - is declared as an attribute of type `V`. The key is represented as an attribute of type `NodeKey`. The definition of this class is presented in Listing 7.

Listing 7: **NodeKey** class

```
public class NodeKey : IComparable<NodeKey> {
   public K KeyValue { get; }
   private IComparer<K> comparer;

   public NodeKey(K key, IComparer<K> comparer) {
      this.KeyValue = key;
      this.comparer = comparer;
   }

   public int CompareTo(NodeKey other) {
      return comparer.Compare(this.KeyValue, other.KeyValue);
   }
}
```

The `NodeKey` is a simple class that acts as a wrapper for the actual key of type `K`. The original Java implementation by Brown et al. [16] does not use this approach, instead representing the key as an attribute of type `K` directly on the node. The rationale behind our decision to introduce `NodeKey` in our implementation is mostly related to the way C# handles boxing of primitive types.

The issue addressed by the `NodeKey` class originates in the internal structure of the chromatic tree as described by Brown et al. [16] While most of the nodes hold actual data and contain both a key and a value, other nodes have a special structural meaning and are regarded as *dummy* or *sentinel* nodes. Both are initialized with their `key` reference set to `null`. This property is later used in several tree operations and plays an essential role in ensuring their correct behaviour. The detection of these nodes is based on the assumption that `null` is not a valid key for the public operations such as adding key-value pairs, finding keys and values, etc. That implies that no other nodes except the *dummy* or *sentinel* nodes will have `null` keys. However, the key's type `K` is generic, which means that it could either be a *reference type* or a *primitive type*. The first case presents no problems since it would allow the `key` to be dereferenced in order to build the dummy nodes. But the second case, where `K` is a primitive type, would require an *auto-boxing* mechanism to ensure this dereferecing capability automatically. Such an auto-boxing mechanism should be able to *box* or *wrap* a primitive value (e.g. of type `int`) in an object (e.g. of type `Integer`) such that it can be treated as a reference. While Java offers this mechanism, C# requires value types to be manually boxed, making it very inconvenient if the type is unknown. In our implementation, the `NodeKey` class is responsible for boxing the keys of type `K`. By defining the `key` attribute on the node as being of type `NodeKey`, we ensure that it can still be dereferenced such that the dummy nodes can be created.

One of the effects of the key being wrapped in `NodeKey` is that all key comparisons are now comparing wrapper objects instead of actual key values. Thus, the `NodeKey` class must implement the `IComparable<NodeKey>` interface, which allows overriding the default comparison behaviour by implementing the `CompareTo` method. Further, it must accept an `IComparer<K>` object as argument in the constructor. This *comparer* will be used by the `CompareTo` implementation to compare the boxed key values and return the correct result. If a *comparer* was not explicitly used to create the tree in the tree constructor, a default one is created as shown below.

Listing 8: Comparer initialization in the tree constructor

```
if (cmp != null) {
   comparer = cmp;
} else {
   if (!typeof(IComparable<K>).IsAssignableFrom(typeof(K))) {
      throw new Exception("Type " + typeof(K).Name + " does
            not implement IComparable.");
    }
    comparer = Comparer<K>.Default;
}
```

4.2.3 Dictionary operations

With the primitives in place it is possible to update local areas in the tree, and the dictionary operations presented in Listing 4 can be implemented. The general pattern of updating the local areas involves specialized methods creating SCX records for each update. An example is shown in Listing 9 below.

```
Input: $key$, $value$
begin
    repeat
        Find insertion point $l$ with parent $p$ and count violations on path
         in $v$;
        if $l.key = key$ then
            $op \leftarrow CreateReplaceOp(l, p)$;
        else
            $op \leftarrow CreateInsertOp(l, p)$;
        end
        $success \leftarrow SCX(op)$;
    until $success$;
    if $v \geq MAX\_ALLOWED$ then
        $Fixup(key)$;
    end
end
```

Algorithm 3: `Insert` method

The `Insert` method accepts a key and a value as arguments and subsequently tries to find the correct insertion point in the tree. While traversing the tree to the insertion point, the number of violations encountered is calculated to later determine whether a rebalancing procedure is required. Having found the insertion point, the `Insert` method checks if the key already exists in the tree, i.e. if the insertion point found has the same key as the one received as argument. If so, a `CreateReplaceOp` SCX record containing all the information needed to replace the old leaf node with a new node, containing the same key but the new value, is created. Otherwise, if the node does not already exist, a `CreateInsertOp` SCX record is created containing the necessary information. These node replacements are necessary due to both the key and the value fields being immutable. The SCX record is then passed to the SCX primitive which will attempt to commit the update. In case of failure, the entire operation restarts.

Notice how the general design pattern mentioned above is exemplified by the two routines - `CreateReplaceOp` and `CreateInsertionOp`. Both of them encapsulate all the logic related to that specific update, namely the concrete instances of *V*, *R*, *field* and *new*. After creating and returning them as an SCX record, a later SCX can simply commit it in a general way, without having knowledge of the type of update performed.

The `CreateReplaceOp` method is presented in the listing below. Initially, the method receives four arguments - the insertion node's parent, the insertion node itself, the new key and the new value. The call to `weakLLX` assigns the SCX record of `p` (the parent of the node to be replaced) to `ops[0]` as well as `p` itself to `nodes[0]`. After confirming that `p` is still the parent of the node to be replaced, the new node is built containing the key and value received as arguments. Finally, a new SCX record is created and returned containing all the information needed by a later SCX to perform the update.

Listing 9: Replace operation method

```
private Operation createReplaceOp(Node p, Node l, NodeKey key, V
    value) {
    Operation[] ops = new Operation[] { null };
    Node[] nodes = new Node[] { null, l };
    if (!weakLLX(p, 0, ops, nodes)) return null;
    if (l != p.left && l != p.right) return null;
    // Build new sub-tree
    Node subtree = new Node(key, value, l.weight, l.left,
        l.right, dummy);
    return new Operation(nodes, ops, subtree);
}
```

The `Delete` method is designed in a very similar way, the difference being that it creates a *delete* SCX record (instead of *insert* or *replace*) which it will then try to commit. Both `Insert` and `Delete` will finish by checking whether the number of violations on the path to the affected node is greater than the maximum allowed, in which case a *fix-up* method is called. The `Fixup` method is responsible for rebalancing the path if it encounters more than the allowed number of violations on the path to the given key, and is able to handle all 22 different update cases required by the algorithm (described in [12]). For each of the 22 cases there is a specialized method creating the right SCX record to be used.

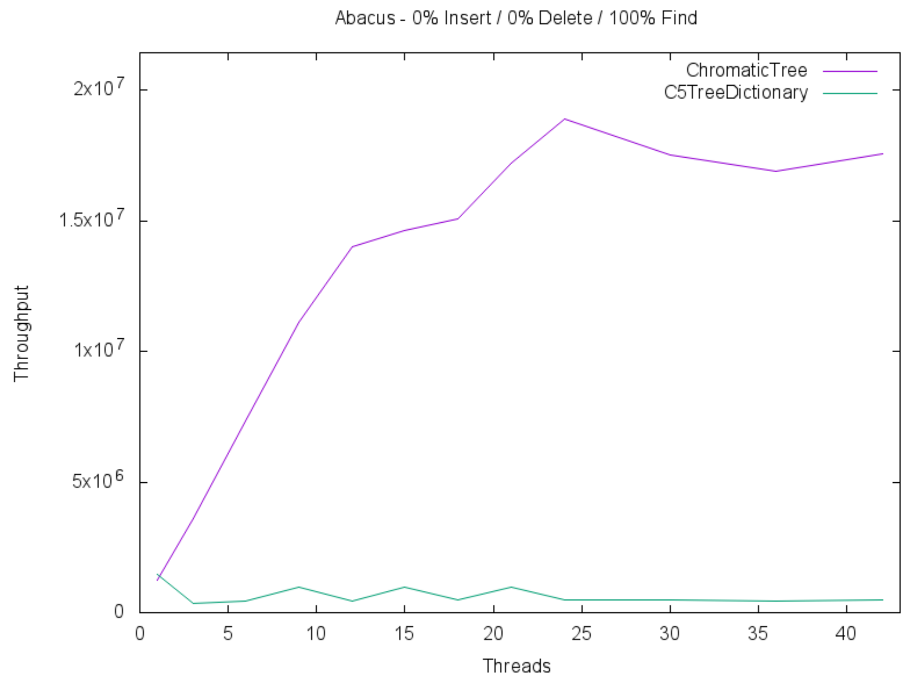
The `Find` method is completely sequential, without any mechanism for concurrency control. The implementation itself is nothing more than the well-known binary search tree *search* operation and it is therefore not presented here. However, the less trivial part is the intuition regarding how this method can be implemented sequentially, yet still be linearizable and thread-safe. First, Brown et al. [16] use three lemmas to prove that the `Find` operation always finds a leaf which, if it contains the search key, is the correct return result. The same lemmas are then used to prove that if the leaf found does not contain the search key, then there is no node in the tree with that key, thus the `Find` can safely return `null`. Both these assumptions are provably correct at the *linearization point* of the `Find` operation, i.e. at the time when the leaf reached was on the search path of the given key. Once the leaf containing the search key is found, it can safely be returned since both its key and value are immutable. As all updates in the tree occur atomically, values cannot be returned in an inconsistent state due to interference with updates by another process. For further information regarding the lemmas and their proof see [16].

The `Count` method is also implemented sequentially, but unlike the `Find` method it does not have a linearization point. The implication is that even though a result n is returned, there is no guarantee that there were n elements in the tree at any given point between the entry and exit point of the method. Generally, it is less complex to ensure a linearization point for a method that returns information about a single node, as previously illustrated with `Find`. Any method that returns information about several nodes require at least one additional step to achieve a consistent and linearizable result, through using the primitives shown in Section 5. In the particular case of the `Count` method, ensuring a linearization point would decrease its performance so much that it would become unusable in practice.

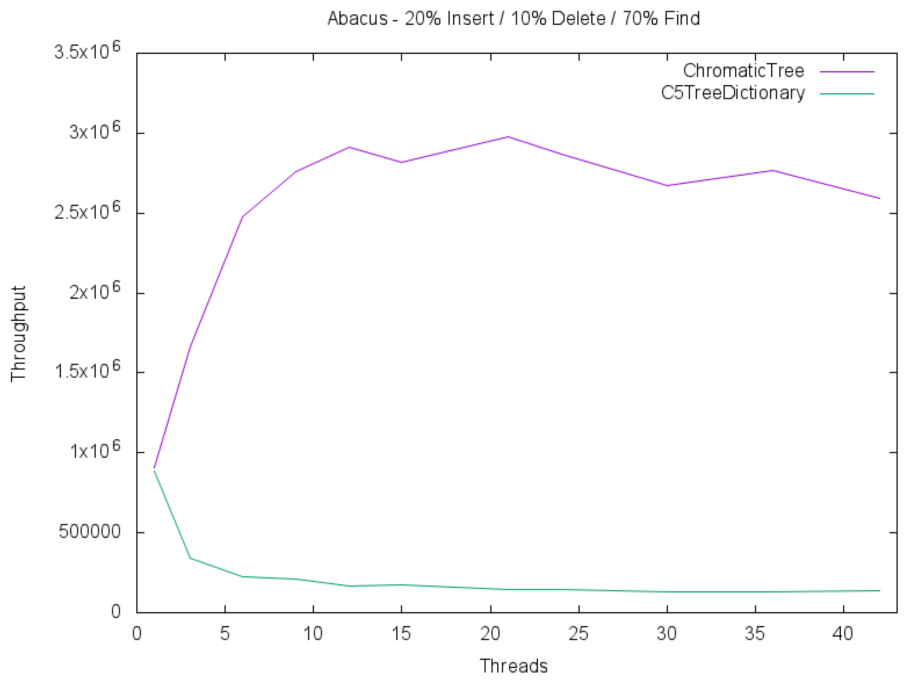
4.3 Experiments

We ran benchmark tests for four different ratios of *insert*, *delete* and *find* operations. As for the *lock-free red-black tree*, we tested the chromatic tree alongside the synchronized wrapper class containing the C5 TreeDictionary. Each thread worked on a queue of 100,000 elements chosen randomly from the key range 0 to 100,000, with each thread performing the defined ratio of operations. The trees were prefilled based on the formula shown in Algorithm 9.

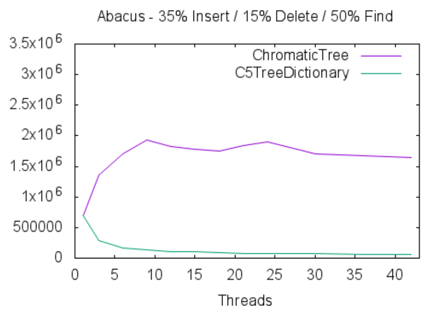
For *find* operations only, the chromatic tree is slightly slower than C5's TreeDictionary for a single thread but scales considerably better. From 1 thread to the 12 threads that can be run on the first CPU, the chromatic tree records a throughput improvement of 11.3x. As the workload moves onto the second 12-core CPU the increase in throughput decreases, but a doubling of threads from 12 to 24 still result in a 1.4x increase in throughput.

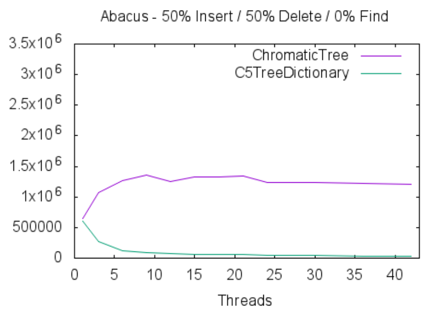


Running the benchmark with 20% *insert*, 10% *delete* and 70% *find* operations introduce concurrent rebalancing operations on the trees. The chromatic tree now records a 3.2x improvement from 1 to 12 threads, and the increase from 12 to 24 threads becomes negligible.



Both the 35/15/50 and 50/50/0 ratios reach a maximum throughput already at 9 threads, though the drop to 12 threads is quite low. It nearly reaches the same throughput for 24 threads as for 9 threads, before falling off as the number of threads overtake the physical CPU cores on the machine. This can be seen as a continuation of the trend visible between the two first benchmark runs, where the throughput improvement from 1 to 12 threads decreases as a larger portion of the operations might require re-balancing, and the increase in throughput from 12 to 24 threads falls until it becomes slightly negative. Compared to the monotonically decreasing throughput of the baseline synchronized C5 TreeDictionary, the scalability and performance of the chromatic tree is a significant improvement.





4.4 Conclusion

The approach presented in this section employs different techniques that solve or eliminate several shortcomings of the previously discussed lock-free red-black tree algorithm. The down-tree structure allows atomic subgraph replacement, which significantly simplifies the proof of correctness. The nodes have immutable fields and are always replaced rather than modified which permits sequential implementations for operations like *find* or range queries.

Another major benefit of this implementation is represented by the primitives it introduces. The complete proof of correctness formulated by the authors for these primitives greatly increases the confidence in any extensions that use them.

The data structure and all the mechanisms supporting it and ensuring concurrency control are relatively new and under active development. The paper introducing the data structure [16] was presented at the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, one of the leading conferences in the field on parallel programming.

Provably correct primitives, good scalability and active development all suggest that the chromatic tree by Brown et al. is a good candidate for a concurrent version of C5 SortedDictionary.

5 Extensions to the chromatic tree

Based on the paper from Brown et al. [16] introducing the concurrent chromatic tree, as well as the two previous papers describing the LLX, SCX and VLX primitives [17] and range queries in k-ary search trees [18], we present an implementation of range queries for the concurrent chromatic tree. Our implementation employs the same primitives and techniques for concurrency control used by the other operations of the chromatic tree. Additionally, we introduce a *validate* primitive used to efficiently provide a linearization point for query operations. The range query implementation is also provably correct, which follows from the already existing proof of correctness given by Brown et al. [16] [17] for the chromatic tree and tree update template. Further, we introduce three different *enumerators* that work both on the tree itself as well as on ranges returned by queries. Along with the implementation, we describe some theoretical aspects behind the design decisions and the choice of functionality to be provided by the new operations.

5.1 Validate primitive

The primitives LLX and VLX from Brown et al. [17] provide the mechanism for validating data records and ensure the existence of a linearization point for the operations using them. However, as the requirements for the query operations are less strict than what is provided by a full VLX, we introduce a *validate* primitive.

When validating a node using VLX, a process starts by using LLX to acquire the node's SCX record. Subsequently both the node and its SCX record are passed as arguments to VLX, which will verify that the node's SCX record has not changed since the LLX invocation. For a collection of nodes that needs to be validated, a process will have to construct an equally sized collection containing the corresponding SCX records for each node. Each SCX record is acquired by calling LLX on the node at the point it is first encountered by the process. Both collections will then be used by sequential VLX invocations to validate all the nodes. This is normally required in the query operations that return sets of nodes, as they need to provide a linearization point and ensure that the data returned is consistent. This linearization point is given by the start of the sequential VLX validations.

However, the same linearization point and consistency guarantees can be ensured without the need of building an additional set of SCX records. For this purpose we introduce the *validate* primitive, which for a set of nodes sequentially verifies that none of them are *finalized*.

```
Input: nodes
begin
    foreach node in nodes do
        if node.marked then return false;
    end
    return true
end
```

Algorithm 3: `Validate`

This primitive does not require a set of SCX records, which makes the operations using it more memory-efficient. The fact that it does not need to invoke LLX while building the node set also contributes to an increase in performance. Further, it is important to bear in mind that an SCX operation first replaces a node's SCX record with its own before finalizing it. Only after this is done, when all nodes are finalized and their SCX records updated, is the node actually removed from the tree via a subgraph replacement. The implication is that a node is valid for reading (though not for updating) at any point before the replacement occurs, even though its SCX record has already been updated. While this makes the *validate* primitive slightly less strict than LLX-VLX, it does not affect the consistency guarantees or linearization point, which remains at the start of the validation.

But in its current form, the implementation of the validate primitive hides a subtle flaw. Should a process crash right after finalizing the node but before replacing it in the tree, that node will remain finalized but continue to be part of the tree. Until another process calls LLX on the node and helps its SCX operation to complete, the node will continue to be returned by the query (as it is still in the tree) and continue to cause to the validate function to fail (as it is finalized). Since this might cause the *validate* to enter an infinite loop of retries, the inconsistency must be handled in the *validate* itself by attempting to help the SCX operation complete when finalized nodes are encountered.

```
Input: nodes
begin
    foreach node in nodes do
        if node.marked then
            SCX(node);
            return false
        end
    end
    return true
end
```

Algorithm 4: `Validate` with SCX

The listing above shows the final and correct pseudocode of the *validate* primitive, as it is used by most of the extensions presented in the following sections.

5.2 Basic SortedDictionary functionality

We begin describing the tree extensions by introducing a number of basic *Sorted-Dictionary* operations.

The *FindMin* operation is, as the name suggests, responsible for returning the element in the tree with the lowest key. Conversely, *FindMax* will return the element with the highest key.

```
begin
    $l \leftarrow root$;
    while $l$ not $\perp$ do
        $l \leftarrow l.left$ ;
    end
    return $l$
end
```

Algorithm 5: `FindMin` method

The listing above shows the pseudocode for the *FindMin* operation. Just like *Find*, its implementation is completely sequential. The process starting at the root always follows the leftmost path in order to reach the leaf with the lowest key.

Even though this operation is not described by Brown et al. [16], its proof of correctness would be very similar to the one presented for *find*. The intuition is that the updates in the tree always occur atomically and that subgraphs are replaced rather than modified. Thus, Brown et al. [16] prove that, in the case of *Find*, the operation is still linearizable even if the search traverses nodes that are no longer in the tree. This can also be used for proving *FindMin* correct and arguing that the operation has a linearization point between the entry and exit points of the method.

FindMax has a very similar implementation, the only difference being that the process starting at the root always chooses the rightmost path. The same arguments presented above can be used to reason about its correctness.

The next two operations specific to sorted data structures are *successor* and *predecessor*. Brown et al. [16] describe these operations at a theoretical level and offer the corresponding pseudocode. However, they are not part of their Java implementation of the chromatic tree.

```
Input: key
begin
    $l \leftarrow root$;
    $S \leftarrow \emptyset$;
    $lastLeft \leftarrow root$;
    while $l$ not leaf do
        if $key < l.key$ then
            $S \leftarrow \{l\}$;
            $lastLeft \leftarrow l$;
            $l \leftarrow l.left$;
        else
            append $l$ to $S$;
            $l \leftarrow l.right$;
        end
    end
    if $lastLeft = root$ then return $\bot$;
    if $key < l.key$ then return $\langle l.key, l.value \rangle$;
    $s \leftarrow lastLeft.right$;
    while $s$ not leaf do
        append $s$ to $S$;
        $s \leftarrow s.left$;
    end
    if $validate(S)$ then
        return $\langle s.key, s.value \rangle$
    else
        Restart;
    end
end
```

Algorithm 6: `Successor` method

As shown above, the *successor* operation starts at the root and traverses the tree downwards attempting to find a leaf node containing the given key. While traversing, a list of nodes is gradually built from the last node where the search chose the left path, adding all nodes on that path down to the current position. Thus, the list must always be reinitialized when a left path is chosen. If the search ends at a leaf that has a higher key than the one received as argument, it will return its key-value pair and the operation will have the same linearization guarantees as the `Find` method. Otherwise, the given key has been found and the operation must find its successor node. To do so, it returns to the last node where the search went left (the first in the list) and starts from the right child instead. It then follows all the left child references from that node downwards, such that that the leaf found at the end will contain the lowest key higher than the key given as argument. While traversing, the search appends all the nodes visited on the path to the existing list, effectively creating a complete list of all the nodes between the initially found node and its successor. When both leaf nodes have been found, the path between them is validated to ensure that the returned result is consistent. The validation of this path also provides the linearization point of the *successor* operation.

Brown et al. [16] suggest using LLX on all the nodes on the search path, and then VLX to accomplish the validation. However, our implementation uses the *validate* primitive described in the previous section, as it is more efficient and offers the same guarantees.

The *predecessor* operation is also part of our implementation, but it is not described here since it is simply a mirrored version of *successor*.

Perhaps unexpectedly, the *successor* method in a down-tree is not less efficient than in a tree where the nodes hold parent references. Using the approach of storing a reference to the *lowest common ancestor* of the starting node and its successor as above, the same number of nodes are visited in order to get from the first to the latter in both cases (the depth d_s of the successor relative to the lowest common ancestor). Thus, the asymptotic upper bound is given by $O(\log n)$ for both type of trees.

The last basic operation presented here is *ContainsAll*, which accepts a list of arguments representing keys and determines whether they all are contained in the tree.

```
Input: keys
begin
    $S \leftarrow \emptyset$;
    foreach key in keys do
        $node \leftarrow find(key)$;
        if $node = \perp$ then return false;
        append node to $S$;
    end
    if $validate(S)$ then
        return true;
    else
        Restart;
    end
end
```

Algorithm 7: `ContainsAll` method

The implementation of this method relies on the *Find* functionality. That is, for all the keys received as arguments, the operation tries to find the node holding that specific key and append it to a list. If one of the keys is not found (i.e. there is no corresponding node), the method returns `false`. Otherwise, the list of nodes must be validated using the *validate* primitive. A failure in validation causes the operation to restart, as it might be caused by a node whose key remains in the tree. A successful validation on the other side confirms that all the keys were in the tree at the linearization point, which is the entry point of the *validate* primitive.

5.3 Range Queries

Balanced binary search trees are acknowledged to allow efficient range querying due to the way they are structured internally. We extended the chromatic tree described by Brown et al. [16] with range querying functionality, provided through the methods listed below.

Listing 10: Range query methods

```
IRange<K, V> SnapshotRangeFromTo(K lowerBound, K upperBound);
IRange<K, V> DynamicRangeFromTo(K lowerBound, K upperBound);

IRange<K, V> SnapshotRangeFrom(K bot);
IRange<K, V> DynamicRangeFrom(K bot);

IRange<K, V> SnapshotRangeTo(K top);
IRange<K, V> DynamicRangeTo(K top);

IRange<K, V> SnapshotRangeAll();
IRange<K, V> DynamicRangeAll();
```

A range is modelled as a class implementing the `IRange<K, V>` interface and all methods querying for ranges must return an object of this type. As shown above, there are four groups consisting of two methods each. Both methods in a group return the range in the same interval, but provide different consistency guarantees. The four groups can be regarded as querying for ranges in double-bounded, left-bounded, right-bounded and unbounded intervals respectively. A bound that is not specified indicates that the search will continue in that direction until there are no more elements to be included.

Every class implementing `IRange<K, V>` must provide an implementation for the `Count` method, which returns the number of elements in the range. Further, the range must be iterable, i.e. implement the `IEnumerable<KeyValuePair<K,V>>` interface.

Listing 11: `IRange<K, V>` interface

```
public interface IRange<K, V> : IEnumerable<KeyValuePair<K, V>>
{
    Int32 Count();
}
```

We identified and implemented two kinds of ranges - *snapshot ranges* and *dynamic ranges*. The following sections will describe each of them and discuss the different guarantees and functionality they offer.

5.3.1 Snapshot ranges

The first kind represents the *snapshot* of a range in the tree at a given point in time. The data it contains is ensured to be consistent, i.e. all the key-value pairs did exist in the tree simultaneously in that specific range. The snapshot range is modelled by the class `SnapshotRange` implementing the `IRange<K, V>` interface.

Listing 12: `SnapshotRange` implementation

```
public class SnapshotRange : IRange<K, V> {
    private readonly ConcurrentChromaticTreeMap<K, V> tree;
    private readonly List<Node> snapshot;

    public SnapshotRange(ConcurrentChromaticTreeMap<K, V> tree,
        List<Node> snapshot) {
        this.tree = tree;
        this.snapshot = snapshot;
    }

    public int Count() {
        return snapshot.Count;
    }

    public IEnumerator<SCG.KeyValuePair<K, V>> GetEnumerator() {
        ...
    }
}
```

The attribute `snapshot` contains the actual nodes in the range and it is received as an argument in the constructor. It is then never reassigned, which means that the snapshot range is *static* - once it is created, the range does not change, regardless of the changes occurring in the tree. The `Count` method is a requirement of the `IRange` interface, and its implementation simply uses the list's `Count` attribute to return the size of the range. The second requirement of the interface is that the range must provide an `IEnumerator`, which is implemented in the method `GetEnumerator` described in Section 5.4.

The previous section introduced four types of range queries corresponding to different types of intervals. All return a range as an object of type `SnapshotRange`. But creating this object requires that the snapshot is provided. To do so, each of the four query methods use an internal function named *RangeQueryInternal(min, max)*. The parameters of this function represent the lower and the upper bounds of the range. If any of these bounds is omitted, the interval is considered to be unbounded at that specific end. The return value is a list containing all nodes in the range.

```
Input: $min, max$
begin
    $S \leftarrow$ empty stack;
    $R \leftarrow \emptyset$;
    $node \leftarrow root$;
    push $node$ to $S$;
    while $S.count > 0$ do
        $node \leftarrow$ pop from $S$;
        if $node$ is leaf then
            if ($min = \perp$ or $node.key \geq min$) and ($max = \perp$
              or $node.key \leq max$) then
                add $node$ to $R$;
            end
        else
            if $max = \perp$ or $node.key \leq max$ then
                push $node.right$ to $S$;
            end
            if $min = \perp$ or $node.key \geq min$ then
                push $node.left$ to $S$;
            end
        end
    end
    if $validate(R)$ then
        return $R$
    else
        return false
    end
end
```

Algorithm 8: `RangeQueryInternal` method

The function uses a *depth-first traversal* in the tree to find all nodes in the specified interval. This traversal is executed from left to right to ensure that the nodes will be encountered and appended to the snapshot list R in the right order. Before returning the result, the snapshot must be validated to ensure its consistency, which is done using the *validate* primitive.

Using `RangeQueryInternal`, all the query methods can easily be implemented, as exemplified in Listing 13 for `SnapshotRangeFromTo` and `SnapshotRangeFrom`. Notice that the first method creates two `NodeKey` objects to be passed as arguments, while the latter passes only one valid NodeKey followed by `null`, indicating that the range is not bound to the right.

Listing 13: `SnapshotRange` implementation

```
public IRange<K, V> SnapshotRangeFromTo(K low, K high) {
   List<Node> snapshot;
   while (!RangeQueryInternal(new NodeKey(low, comparator), new
       NodeKey(high, comparator), out snapshot)) {}
   return new SnapshotRange(this, snapshot);
}
public IRange<K, V> SnapshotRangeFrom(K low) {
   List<Node> snapshot;
   while (!RangeQueryInternal(new NodeKey(low, comparator),
       null, out snapshot))
   {}
   return new SnapshotRange(this, snapshot);
}
```

The linearization point for this type of range query is given by the *validate* invocation in `RangeQueryInternal`. Once the snapshot structure is constructed and stored in the range object, all operations executed on it can be considered linearizable at that same point. Such operations might include *max*, *min*, *count*, etc. Another observation is that all these operations use an underlying data structure that is final (the snapshot list), which means that their results would be consistent over multiple invocations.

The space complexity of this range query is $O(k)$ where k is the number of nodes in the range. This bound is given by the size of the snapshot list which needs to be validated and then stored for later access. The time complexity is $O(k + \log n)$ given by the added complexities of finding the range subtree and then traversing it to the leaves.

5.3.2 Dynamic ranges

This kind of range is regarded as being *dynamic* due to the fact that it will reflect changes to the tree occurring after its creation, and it does not store a snapshot of the nodes it contains. Instead, it uses the two range bounds received as arguments to provide the functionality required by the `IRange<K, V>` interface.

Listing 14: `DynamicRange` implementation

```
public class DynamicRange : IRange<K, V> {
   private readonly ConcurrentChromaticTreeMap<K, V> tree;
   private readonly NodeKey low, high;

   public DynamicRange(ConcurrentChromaticTreeMap<K, V> tree,
       NodeKey low, NodeKey high) {
       this.tree = tree;
       this.low = low;
```

```
        this.high = high;
    }

    public int Count() {
            return tree.Count (tree.root, low, high);
    }

    public IEnumerator<SCG.KeyValuePair<K, V>> GetEnumerator() {
        ...
    }
}
```

A structure containing all the nodes in the range is never constructed or validated, which means that this type of range is not linearizable and does not offer any consistency guarantees.

The implementation of the `Count` method uses a sequential (not thread-safe) *count* operation provided by the tree, which returns the number of keys in a specified range. Note that the result is not cached, and the operation is re-executed each time the `Count` method is invoked. In contrast to the snapshot ranges, operations executed on a `DynamicRange` object are not guaranteed to be consistent over multiple invocations and the same operation might return different results.

5.4 Enumerators

In the previous section, we introduced two different kinds of ranges obtainable by querying the tree. Perhaps the most important operation supported by these ranges is *iteration*. For this purpose, we introduce three *enumerators* and describe their behaviour and the rationale behind them.

5.4.1 Snapshot enumerators

The *snapshot enumerators* iterate through the snapshot of a range and are therefore employed by the *snapshot ranges*. We implemented two such enumerators - *fail-fast* and *fail-safe*. This distinction is inspired by Java's homonymous concepts of iterators, even though they are slightly modified in this paper and adjusted to the chromatic tree implementation.

Fail-fast. The *fail-fast* enumerator does not accept structural changes to occur affecting the nodes it has not yet visited. Any such change will cause the enumerator to throw an `InvalidOperationException` with the message "Collection was modified". This behaviour makes it somewhat similar to the C# enumerators provided by e.g. `List(T).GetEnumerator()` as well as Java's iterators for e.g. `HashMap` which throw a `ConcurrentModificationException`. The main difference is that both of the above-mentioned iterators fail if a modification occurs in any part of the data structure. The enumerator we propose only fails if a node it reaches is *finalized*, while being completely oblivious to all other nodes - including those already visited. Note that the snapshot the enumerator is iterating through contains references to nodes that might still be in the tree. If a node is no longer part of the tree, it must have been finalized, causing the enumerator to fail. From this point of view the enumerator operates indirectly on the data structure, which represents another similarity with Java's fail-fast iterators.

The Java implementation of this type of iterator relies on an internal *mods* flag that each collection must hold. This flag is then updated by every operation performing a structural change in the collection, indicating to the iterator that it should fail when attempting to move forward. However, the access to the flag is not synchronized, hence the iterator cannot offer any guarantees that a modification will be detected and reacted upon. The Oracle documentation states that the iterator acts correctly "on a best-effort basis" [19] which implies that it cannot be relied upon for the program's correctness. Taking a lock before every access to this flag would sequentialize accesses to it and therefore reduce scalability, which means that this approach is unlikely to be beneficial in a concurrent data structure. The enumerator we present does not rely on such a mechanism and has less strict constraints regarding the allowed modifications in the tree. In exchange, it offers strong guarantees regarding its behaviour and is therefore more reliable in a concurrent context.

Fail-safe. The *fail-safe* enumerator iterates through a snapshot of the range and is completely oblivious to any modifications in the data structure. This can easily be achieved by ignoring the *finalized* state of any node in the snapshot and continue the iteration independently of it. The snapshot is not implemented as a copy of the nodes in the data structure. However, the fact that the node's fields that are relevant in this case (key and value) are immutable, allows us to claim that, conceptually, this enumerator is equivalent to one that would iterate through a *snapshot of clones*. This makes the enumerator similar to those defined by C# in e.g. `ConcurrentQueue` or Java in `ConcurrentHashMap` and `CopyOnWriteArrayList`. An important difference is that our enumerator might avoid the overhead introduced by maintaining a snapshot of clones in memory. In the best-case scenario, none of the nodes in the snapshot have been finalized thus the snapshot still references the nodes in the tree instead of nodes that have been replaced.

Both the fail-fast and the fail-safe enumerators are implemented using the `SnapshotEnumerator` class, which in turn, implements the `IEnumerable<KeyValuePair<K, V>>` interface.

Listing 15: `SnapshotEnumerator` class

```
private class SnapshotEnumerator : IEnumerator<KeyValuePair<K,
    V>> {
```

```
    private ConcurrentChromaticTreeMap<K, V> tree;
    private int currentIndex = 0;
    private List<Node> snapshot;
    private EnumeratorType enumeratorType;

    public SnapshotEnumerator(ConcurrentChromaticTreeMap<K, V>
        tree, List<Node> snapshot, EnumeratorType enumeratorType)
    {...}

    public KeyValuePair<K, V> Current {...}
    public void Dispose() {...}

    public bool MoveNext() {
        if (currentIndex < snapshot.Count) {
            Node current = snapshot[currentIndex];
            if (enumeratorType == EnumeratorType.FAIL_FAST &&
                current.marked == true) {
                 throw new InvalidOperationException("Collection
                     was modified");
            }
            currentIndex++;
            return true;
        } else { // There are no more elements in the snapshot
            return false;
        }
    }

    public void Reset() {...}
}
```

As shown above, the `SnapshotEnumerator` is instantiated with an `EnumeratorType` which is then used in `MoveNext` to react to concurrent modifications accordingly. Thus, the fail-fast and fail-safe enumerators must simply provide the correct type when initializing `SnapshotEnumerator`.

Listing 16: Snapshot enumerators

```
private class FailFastEnumerator : SnapshotEnumerator {
    public FailFastEnumerator(ConcurrentChromaticTreeMap<K, V>
        tree, List<Node> snapshot) :
    base(tree, snapshot, EnumeratorType.FAIL_FAST) { }
}

private class FailSafeEnumerator : SnapshotEnumerator {
    public FailSafeEnumerator(ConcurrentChromaticTreeMap<K, V>
        tree, List<Node> snapshot) :
    base(tree, snapshot, EnumeratorType.FAIL_SAFE) { }
}
```

Even though *reverse snapshot enumerators* are not currently included as part of this extension, they can easily be implemented by passing the reverted snapshot list as argument for the enumerator's constructor. This approach maintains the same correctness and linearization guarantees.

5.4.2 Dynamic enumerator

The *dynamic enumerator* starts by finding the lower bound of the range and uses the *successor* operation to perform the iteration. It does not require a pre-constructed snapshot, hence it can be used by the *dynamic ranges*. This approach does not ensure a linearization point for the entire iteration. In exchange, concurrent modifications occurring in the tree after the enumerator was created might still be visible. A similar type of enumerator is used by C# in `ConcurrentDictionary<TKey, TValue>`.

The dynamic enumerator is implemented by the `DynamicEnumerator` class shown in Listing 17.

Listing 17: `DynamicEnumerator` class

```
private class DynamicEnumerator : IEnumerator<KeyValuePair<K,
    V>> {

    private ConcurrentChromaticTreeMap<K, V> tree;
    private Node current;
    private NodeKey low, high;

    public DynamicEnumerator(ConcurrentChromaticTreeMap<K, V>
        tree, NodeKey low, NodeKey high)
    {...}

    public KeyValuePair<K, V> Current {...}
    public void Dispose() {...}

    public bool MoveNext() {
        if (high != null && current != null &&
            current.key.CompareTo(high) >= 0) {
             return false;
        }
        if (current == null) {
            if (low == null) {
                tree.FindMin(out current);
            } else {
                if (!tree.Find(low.KeyValue, out current))
                    tree.Successor(low, out current);
            }
        } else {
            tree.Successor(current.key, out current);
        }
        if (current != null) {
            return high == null || current.key.CompareTo(high) <=
                0;
        }
        return false;
    }

    public void Reset() {...}
}
```

The *dynamic enumerator* is guaranteed to make progress even if concurrent updates are made within its range. Unlike the *snapshot enumerators* which are guaranteed to terminate, infinite insertions to the right of an unbounded *dynamic enumerator* would cause it never to terminate. This behavior is consistent with the semantics of the enumerator, as the current position would always have a successor.

As for the snapshot enumerators, a *reverse dynamic enumerator* could easily be implemented. In this case, the iteration would start from the upper bound of the range and use *predecessor* to move to the next element. While the reverse snapshot enumerator did not require any change to the existing implementation, the reverse dynamic enumerator would use a mirrored version of the code presented above.

6 Other algorithms and implementations

In addition to the data structures presented in Sections 3 and 4, we also include benchmark tests for two other implementations in Section 7. This section will present these additional data structures.

6.1 Synchronized C5 TreeDictionary

The `TreeDictionary` is the C5 implementation of a tree-based sorted dictionary. This data structure uses a red-black tree to provide the expected functionality, and its implementation is purely sequential. We considered a lock-based synchronized version of this data structure to be of great interest as part of the benchmarking experiments for several reasons. First, it serves as a baseline for interpreting the results of all the other data structures. The expectation is that an algorithm using a much more complex concurrency control technique (e.g. non-blocking CAS, finer-grained locks, etc.) should outperform this simple synchronized version by scaling significantly better. Second, we regarded this data structure as a good indicator reflecting the correctness of the benchmarking framework. Obtaining the expected results would serve as a confirmation regarding this aspect.

For this purpose, we introduced `SynchronizedTreeDictionary`, a wrapper class for the C5 `TreeDictionary`. The class contains an attribute `guard` of type `Object` that acts as a *monitor* for all the dictionary methods. Since this is a very basic implementation, we expect the throughput for a single thread to be higher than the other data structures', but then drop as the number of threads increases.

6.2 Concurrent red-black tree

Besa and Eterovic [8] introduced a concurrent version of a red-black tree that uses optimistic concurrency control techniques and new rebalancing steps. Even though this implementation uses locks to prevent interferences between the processes, the fact that they are *fine-grained* allows for increased performance under high contention as well as good scalability. It is important to mention here that unlike the direction set in this field by Nurmi [10] and Hanke [9], this tree is not *relaxed*, i.e. the rebalancing procedure is not decoupled from the actual insert or delete operation. Instead, a process is said to *own* any potential violation that it migh have introduced and thus be responsible for fixing it.

The algorithm is somewhat similar to that shown in Section 3 since they both define a concept of *local area*, ensuring exclusive access to its nodes for the current process and progressively moving it upwards in order to resolve violations. One of the main differences however, is that here the nodes are always locked in a top-down manner. This observation is of great importance since this approach eliminates the possibility of *deadlock* and make the use of any additional mechanisms (such as markers) unnecessary. Second, Besa and Eterovic only lock nodes as high as the grandparent of the node owning the imbalance. The implication is that the local area never grows more in height upwards, thereby increasing the overall performance.

One drawback of this algorithm is that locking the nodes does not allow for a collaborative approach such as the one used by Brown [16] for the chromatic tree. The fact that a process owns its violation means that only that specific process may fix it.

7 Experiments

7.1 Benchmarking framework

We introduce a benchmarking framework that allows comparing performance results for multiple data structures. All implementations included in the tests must provide the functionality required by the `IBenchmarkable<K, V>` interface, shown in Listing 18 below. Preparing a data structure for testing involves creating a wrapper class that implements the interface. These wrapper classes then perform all the requested operations on the underlying data structure they contain.

Listing 18: **IBenchmarkable** interface

```
public interface IBenchmarkable<K, V> where K : IComparable<K> {
       void Insert(K key, V value);
       bool Delete(K key);
       V Find(K key);
}
```

This kind of wrapper class was initially required to synchronize the C5 TreeDictionary and make it eligible for benchmark testing. Since this approach allowed additional data structures to be added with little effort, the wrapper class was made a requirement for all the data structures included in the tests. As any sequential data structure needed a synchronizing wrapper class, the results also became more comparable by ensuring that all data structures have the same overhead, small as it may be.

The benchmarking framework accepts a `BenchmarkConfig` object as its only argument. This configuration object is responsible for providing all the information needed for a given benchmarking test. In addition to the parameters presented in the listing below, the object also contains settings specifying which data structures to include in the tests.

Listing 19: **BenchmarkConfig** class

```
class BenchmarkConfig {
    int[] ThreadsToRun;
    int[] ElementsToTest;

    int[] PercentageInsert;
    int[] PercentageDelete;

    int StartRangeRandom;
    int EndRangeRandom;

    int WarmupRuns;
    int MinimumRuns;
    int SecondsPerTest;
    bool PrefillTree;
}
```

Every combination of ThreadsToRun, ElementsToTest and *insert/delete/find* ratio (contained in PercentageInsert and PercentageDelete) is then run for each data structure to be tested.

If the Prefill flag is set, the tree is prefilled to its steady-state reached at the size given by:

$$s = \frac{(EndRangeRandom - StartRangeRandom) * PercentageInsert}{PercentageInsert + PercentageDelete}$$

Algorithm 9: Formula for Steady State

The only exception is represented by the configuration with 100% *find* operations for which the steady state is instead set to half of ElementsToTest.

A queue of random keys is generated for each of the threads in ThreadsToRun, making the data structure under test the only resource shared between the threads. The threads are then created, started and finally set to wait at a barrier for the main thread to finish its work and reach the barrier itself. This ensures that the random number generator does not affect the test results [20]. It is also important to mention here that every thread is created individually, without reliance on any threadpool or alike. When all worker threads are ready, the main thread reaches the same barrier and starts its timer. This ensures that only work on the data structure itself takes place while the timer is running [20]. The main thread is immediately set to wait at another barrier for all the worker threads to complete, and stops the timer when all threads are done with their work. Timing is made in 10 nanosecond increments through the ElapsedTicks method found in the Stopwatch class and the throughput is calculated as *operations/time*.

Each data structure first completes WarmupRuns runs, attempting to force any Just-In-Time compilation to be completed before the measured runs. An average is then taken over at least MinimumRuns performed after WarmupRuns was completed. Additional runs are performed if the test has not yet run for SecondsPerTest. This dual reliance on SecondsPerTest and MinimumRuns is included to ensure sufficient runs for a representative average. If each test completes very rapidly, the deviation is expected to be higher than for longer runs, and it is therefore necessary to ensure short tests are executed a sufficient number of times.

For each combination of `ThreadsToRun`, `ElementsToTest` and *insert/delete/find* ratio, a *.dat* file named following the pattern *TimeStamp-ElementsToTest-PercentageInsert-PercentageGet* is generated. This file contains the *throughput* result for every number of threads found in `ThreadsToRun` for each data structure. Every combination also automatically outputs a *gnuplot* file for use with its corresponding *.dat* file. Finally, a shell script is outputted, which runs each *gnuplot* file with its matching *.dat* file to generate the graph it contains.

7.2 Test environments

All the tests presented in the following section, as well as those shown in the *Experiments* sections of the algorithms were conducted in one or both of the two environments below.

▷ Slim node on Abacus, DeIC National HPC Centre, SDU
Two 12-Core Intel E5-2680v3 CPUs
64 GB RAM
OS: CentOS 7, Unix 3.10.0.123
.NET version: 4.5 (4.0.30319.17020)

▷ HP Z620 Workstation at ITU
6-Core Intel E5-2620v2 CPU
16 GB RAM
OS: Microsoft Windows NT 6.1.7601 Service Pack 1
.NET version: 4.6 (4.0.30319.42000)

The first environment is a slim node that is part of the Abacus supercomputer hosted at the DeIC National HPC Centre. The operating system is Unix CentOS 7 running the open-source implementation of Microsoft's .NET framework provided by the Mono Project. The second environment is a 6-core machine hosted at the IT University of Copenhagen. The operating system is Windows 7, running the .NET Framework version 4.6 provided by Microsoft. The majority of the benchmarking tests were conducted on the Abacus node since we considered that the server's architecture would provide more relevant results with respect to scalability, given the 24 physical cores. The second environment is included because it runs Microsoft's own implementation of the .NET Framework.

7.3 Included data structures

The following data structures were included in the benchmarking tests:

1. **C5 TreeDictionary** presented in Section 6.1.
2. **Concurrent red-black tree** by Besa et al. presented in Section 6.2.

3. **Chromatic tree** presented in Section 4, including the `NodeKey` class and the extensions from Section 5.
4. **Original Chromatic tree** based on the Java implementation by Brown et al. without the `NodeKey` class or extensions.

7.4 Test data

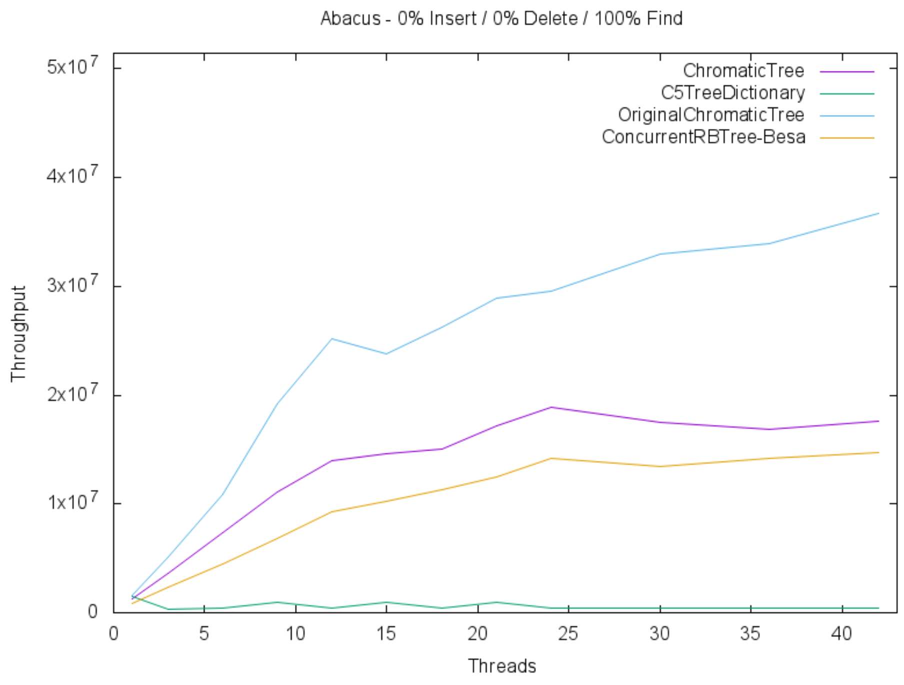
The thread workload in the benchmarking framework consists of queues containing either `Integer` or `String` objects. The `Integer` class shown below is necessary to benchmark the *original chromatic tree* by Brown et al. and the *concurrent red-black tree* by Besa et al. with `int`s as they both require keys to be of reference types.

Listing 20: **Integer** class

```
public class Integer : IComparable<Integer> {
        public int Value { get; set; }
        public Integer(int value) {
                this.Value = value;
        }

        public int CompareTo(Integer other) {
                return Value.CompareTo(other.Value);
        }
}
```

7.5 Results

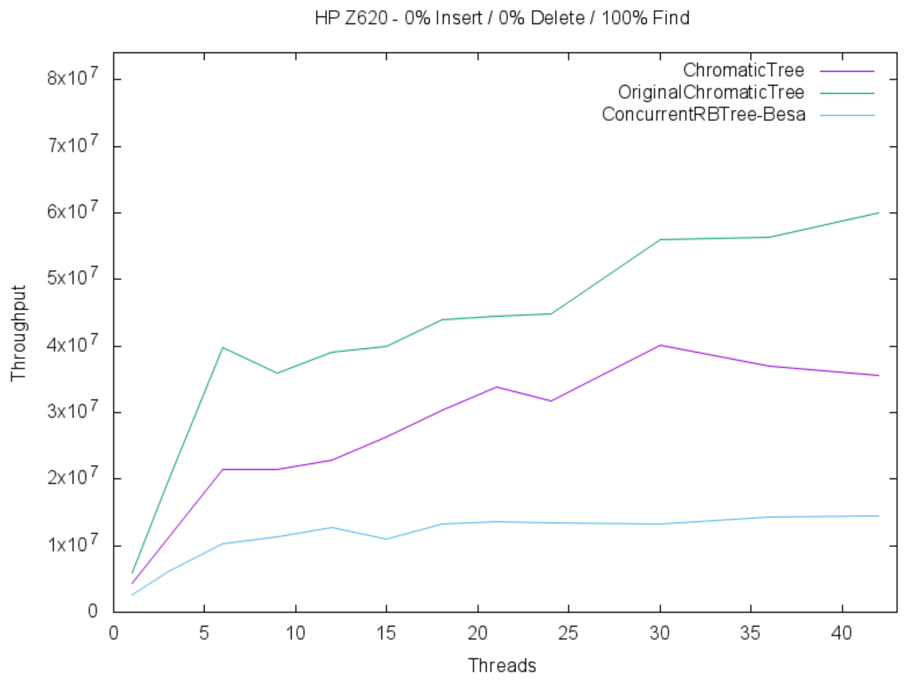


The graph above shows the results obtained by testing the *find* operation using keys in the range 0 - 100,000. As mentioned in the previous section, in this case all data structures are prefilled to a size of half the key range, i.e. 50,000 elements. The tests were conducted on the Abacus slim node.

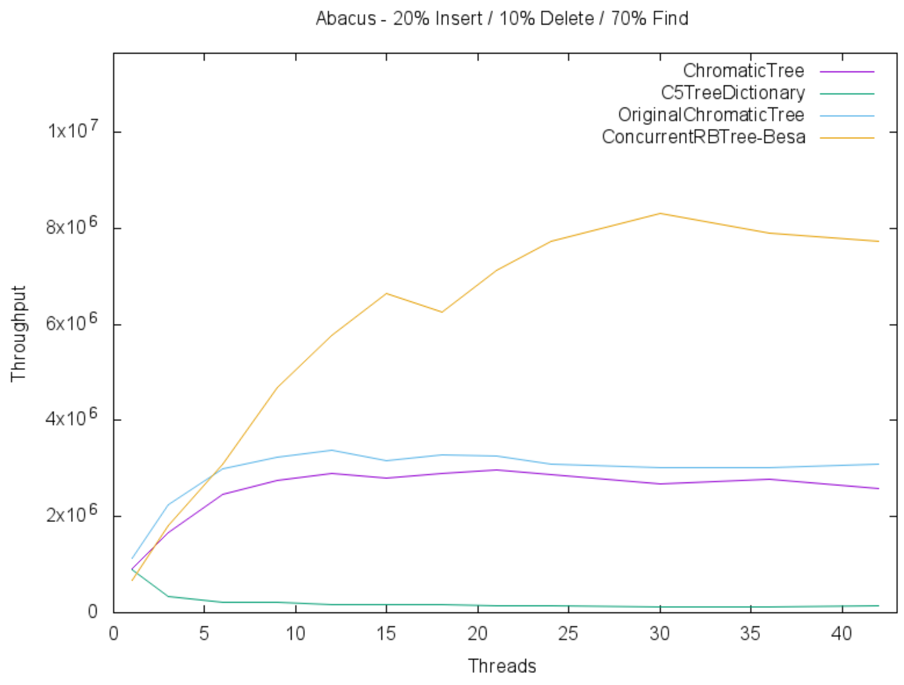
It is worth mentioning that the chromatic tree implementation employs a completely sequential *find* with no concurrency control involved. The implementation by Besa, while still resulting in very little overhead, does require that a field meant for concurrency control, *changeOVL*, is checked. The Java version defines this field as a *volatile long*. In C# however, this has been translated to a regular *long* that must be accessed through the `Interlocked` read and write functionality, which could potentially be one of the reasons causing it to perform worse. As expected, when comparing the original C# version of the chromatic tree with the one using the `NodeKey` class, one can notice a decrease in performance for the latter. This is most likely due to the additional number of steps taken for every key comparison, which involve first obtaining the key values stored in the key wrapper objects before comparing those to get the expected result.

Another observation regarding this graph is that all implementations except the synchronized C5 TreeDictionary scale well.

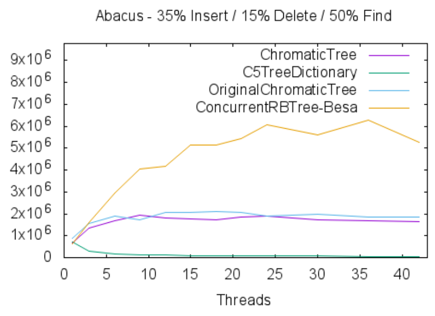
Running the same benchmark test on the Windows 6-Core machine only changes the relative differences between some of the data structures, and retains the ordering found when running on Mono.

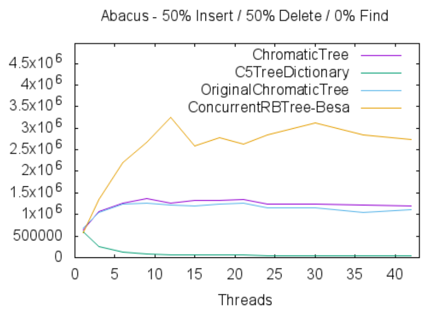


The next graph shows the results of a different test using a 20%/10%/70% ratio for *insert*/*delete*/*find*. The steady state size of the tree is now given by Formula 9 as 66,667.



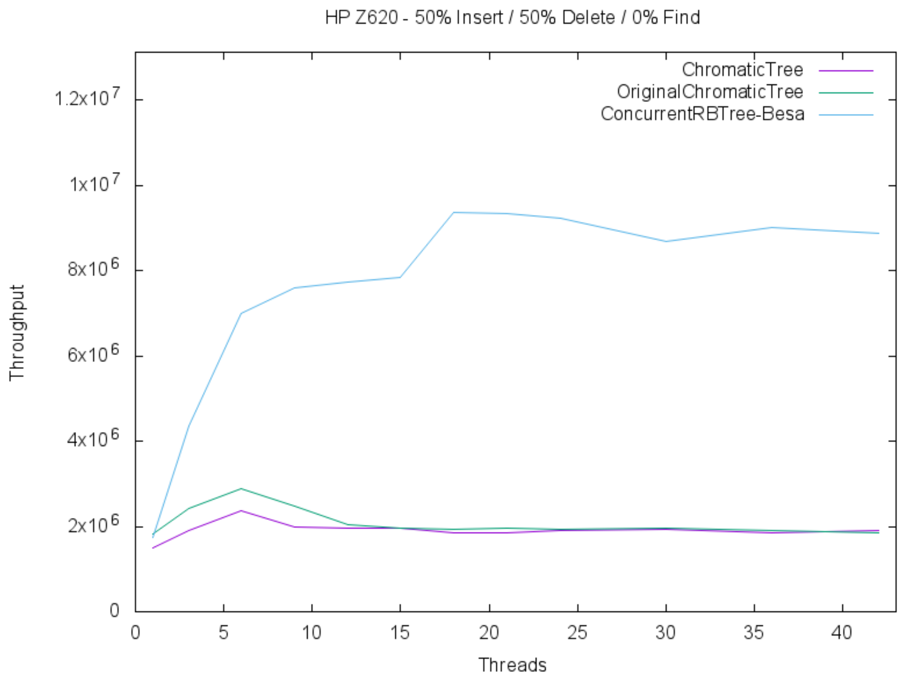
In this case, the implementation by Besa scales considerably better than all the other implementations. Even though the chromatic trees scale similarly up to 6 threads, they are significantly outperformed for any number of threads beyond that. The graphs below show similar results for ratios of 35%/15%/50% and 50%/50%/0%. The main difference between them is that the point corresponding to the number of threads at which the Besa implementation starts scaling better than the rest, shifts slightly to the left as the *insert* and *delete* percentages increase.





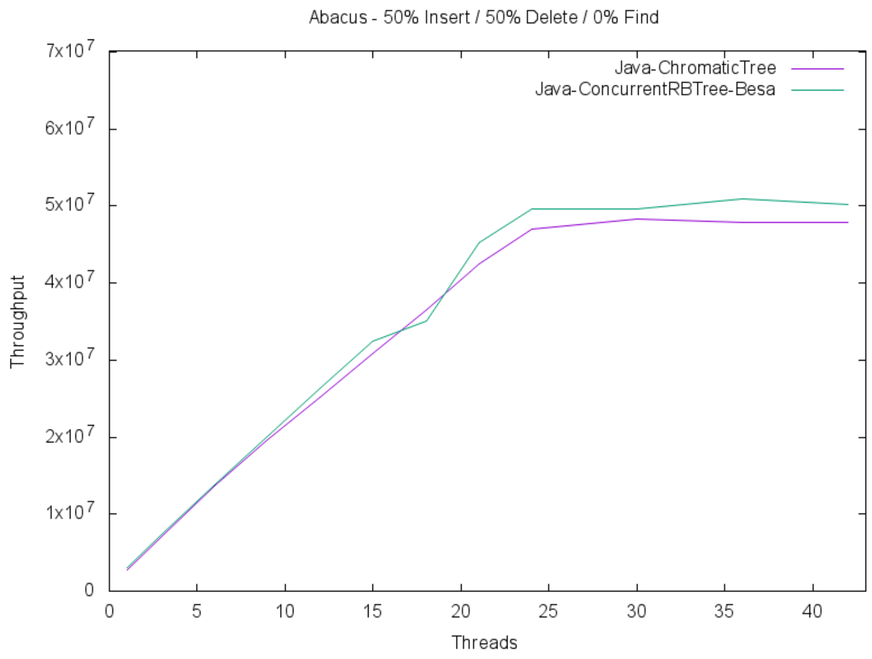
An interesting note is that the Besa implementation has the lowest single-threaded throughput of all tested data structures. This suggests it is the increased scalability of this implementation that makes it perform better in a multi-threaded context.

Running the previous test on the 6-core Windows machine yields the results shown below.



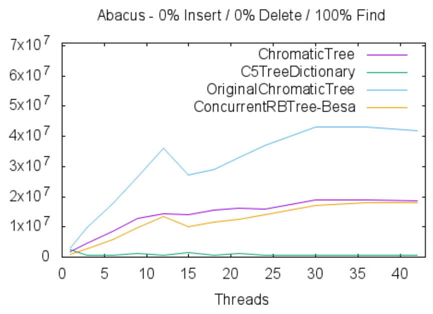
The single-threaded throughput on the Microsoft implementation of .Net is higher than when running the benchmark on Mono. However, the scalability for the chromatic tree implementations are considerably poorer, falling to near sequential speed when the number of threads goes beyond the 6 physical cores of the machine. But the results still show the same performance gap between the Besa implementation and that of the two chromatic tree implementations as found on Mono.

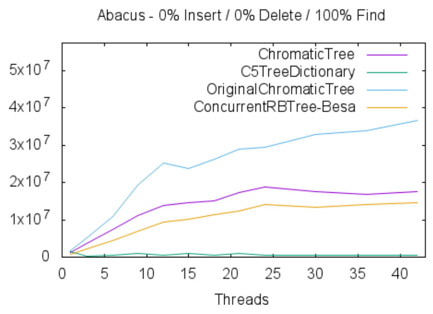
A different result is obtained by testing the original Java implementations of the chromatic tree against Besa's concurrent red-black tree.

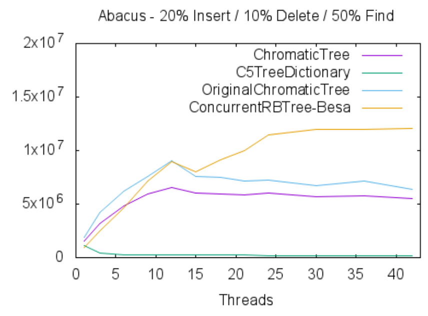


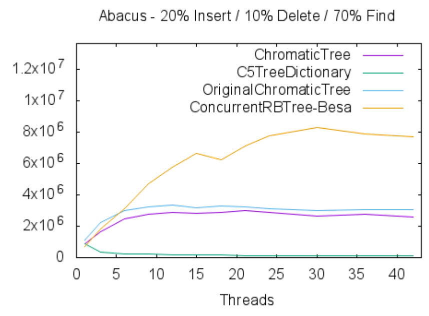
The above results are obtained through Brown's test harness for Java [21] run on the Abacus slim node. The test is run with keys in range 0 to 100,000, 50% *insert*, 50% *find*, and the data structures are prefilled to their steady states. Unlike the results presented so far, this test shows that on JVM both implementations scale similarly up to the number of physical cores and have comparable throughput.

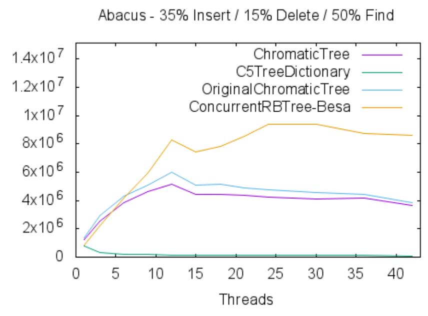
Benchmarking tests were also performed on Mono for 10,000 elements, and the results are shown side by side with those for 100,000 below for the different *insert/delete/find* ratios.

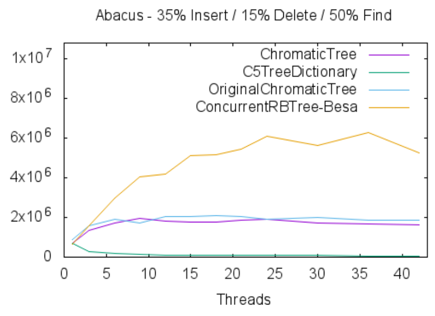


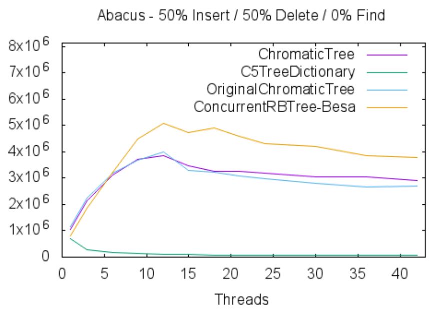


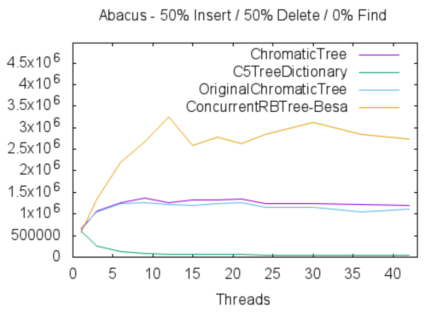






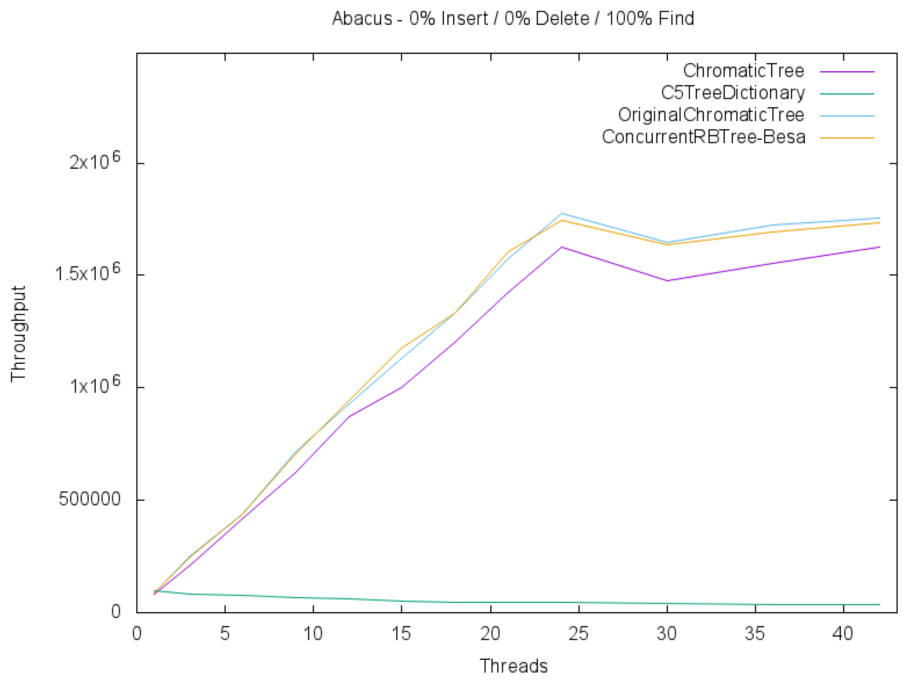




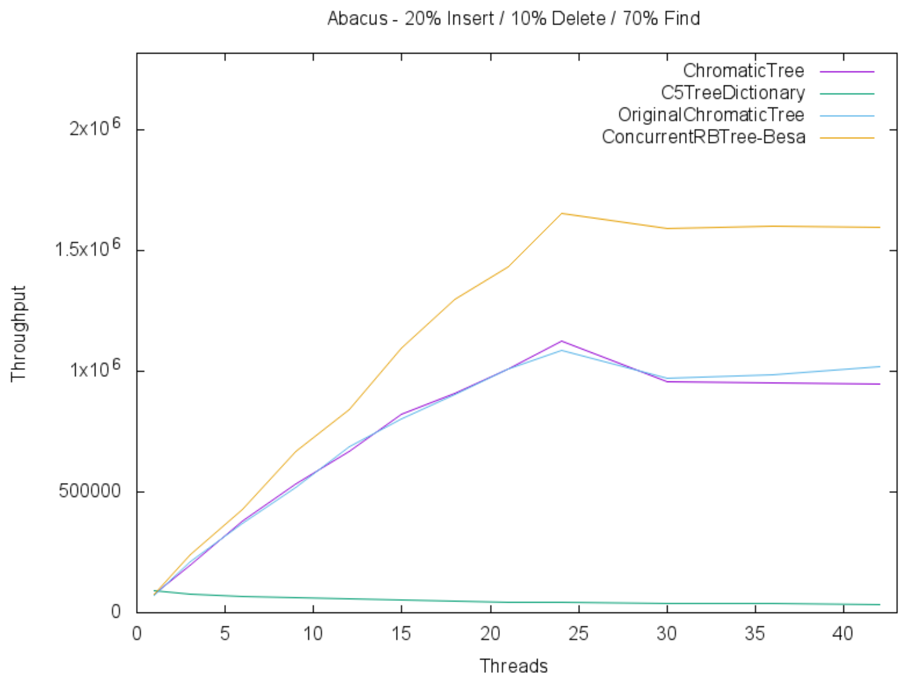


The results for 10,000 elements generally have the same form as those for 100,000 elements. There are some minor variations on when certain result lines cross, but the overall form of the graphs and their results are similar. The 10-fold increase in key-range span between the two test sizes can be said to mainly emphasize the effects already shown.

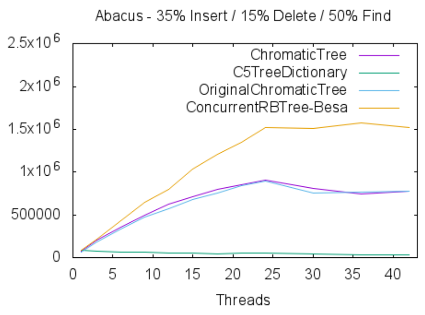
Finally, the benchmarking tests for 100,000 elements were then run with Strings instead of Integers. The strings are simply *ToString()* versions of the random Integers used in the previous tests, meaning they range from “0” to “99,999”.

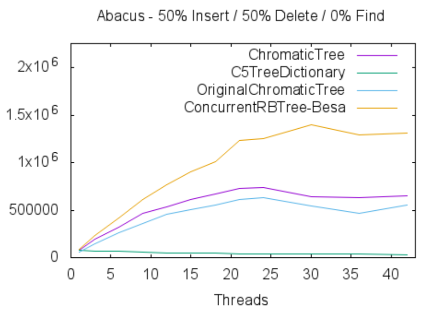


As comparing strings are more computationally expensive than comparing Integers, we expected to see the resulting general performance loss. In addition to that, we also see that the throughput for the different concurrent data structures converges compared to the test with Integer, and that the four now have quite comparable performance.



The implementation from Besa still outperforms the Chromatic tree for 20% insertion and 10% deleting, though by less than for Integer. It is also clear that the scalability for the chromatic tree implementations are considerably better here than for the corresponding Integer test. With strings, the throughput gain from 1 to 24 threads is 14.6x, compared to a mere 3.2x improvement in the test using Integer.





The same scalability improvement is present for the 35%/15%/50% and 50%/50% /0% ratios of *insert*/*delete*/*find* operations. The throughput increases monotonically from 1 thread up to the 24 physical cores on the machine for all concurrent data structures, unlike the Integer test. This improvement might be related to the increased amount of work that takes place without working on the data structure itself, as a larger portion of the total execution time is now spent thread-locally in comparing the strings.

8 Conclusion and future work

In this paper we analyzed several existing approaches and implementations of concurrent balanced trees. In Section 3 we presented a detailed description of a CAS-based lock-free red-black tree and discussed the mechanisms used for concurrency control, its efficiency and the current state of development. Similarly, in Section 4, we presented a concurrent chromatic tree using the custom-defined LLX, SCX and VLX synchronization primitives and a template for atomic subgraph replacement. Based on the theoretical analysis and the initial benchmarking results, we chose to extend the chromatic tree.

Section 5 introduced our extensions to the chromatic tree, consisting of basic sorted dictionary operations, range queries and enumerators. These extensions rely on a new *validate* primitive for efficiently ensuring a linearization point, which we defined, implemented and presented in the same section.

Section 6 briefly introduced two additional data structures included in our benchmark tests. The results of these tests were then presented in detail in Section 7. These results revealed that the chromatic tree performed well and proved to be scalable in a concurrent context on both Mono and .NET platforms.

The good performance results and the strong proof of correctness for the chromatic tree make it a good candidate to support the TreeBag, TreeSet and TreeDictionary collections in a future concurrent version of C5. In addition, its reliance on the provable primitives mentioned above makes it easy to extend with further functionality, as well as reason about the correctness of such additions.

Further work in this direction might include eliminating the need for *NodeKey* by representing the sentinel and dummy nodes differently, develop additional extensions and continue optimizing the performance of the data structure on .NET. Brown et al. has already received interest in implementations of the chromatic tree for other languages than Java, and we consider our current implementation to be both a good foundation for further improvement, as well as a fully functioning C# implementation in its current state.

References

[1] Peter Sestoft. *The C5 Generic Collection Library*. May 2016. URL: `https://www.itu.dk/research/c5/`.

[2] Microsoft. *ConcurrentDictionary Class*. May 2016. URL: `https://msdn.microsoft.com/en-us/library/dd287191(v=vs.110).aspx`.

[3] Oracle. *ConcurrentSkipListMap (Java Platform SE 8)*. May 2016. URL: `https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentSkipListMap.html`.

[4] Faith Ellen et al. "Non-blocking Binary Search Trees". In: *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*. PODC '10. Zurich, Switzerland: ACM, 2010. ISBN: 978-1-60558-888-9.

[5] Greg Barnes. "A Method for Implementing Lock-free Shared-data Structures". In: *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '93. New York, NY, USA: ACM, 1993. ISBN: 0-89791-599-2.

[6] Trevor Brown and Joanna Helga. "Non-blocking K-ary Search Trees". In: *Proceedings of the 15th International Conference on Principles of Distributed Systems*. OPODIS'11. Toulouse, France: Springer-Verlag, 2011. ISBN: 978-3-642-25872-5.

[7] Jong Ho Kim, Helen Cameron, and Peter Graham. "Lock-Free Red-Black Trees Using CAS". 2011.

[8] Juan Besa and Yadran Eterovic. "A Concurrent Red-black Tree". In: *Journal of Parallel and Distributed Computing* 73.4 (Apr. 2013). ISSN: 0743-7315.

[9] Sabine Hanke, Thomas Ottmann, and Eljas Soisalon-Soininen. "Relaxed Balanced Red-Black Trees". In: *Proceedings of the Third Italian Conference on Algorithms and Complexity*. CIAC '97. London, UK, UK: Springer-Verlag, 1997. ISBN: 3-540-62592-5.

[10] Otto Nurmi and Eljas Soisalon-Soininen. "Chromatic Binary Search Trees". In: *Acta Informatica* 33.5 (Aug. 1996). ISSN: 0001-5903.

[11] Trevor Brown, Faith Ellen, and Eric Ruppert. "A General Technique for Non-blocking Trees". In: *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '14. Orlando, Florida, USA: ACM, 2014. ISBN: 978-1-4503-2656-8.

[12] Joan Boyar, Rolf Fagerberg, and Kim S Larsen. "Amortization Results for Chromatic Search Trees, with an Application to Priority Queues". In: *Journal of Computer and System Sciences* 55.3 (Dec. 1997). ISSN: 0022-0000.

[13] Jianwen Ma. *Lock-Free Insertions on Red-Black Trees*. 2003.

[14] Thomas H. Cormen et al. *Introduction to Algorithms*. 2nd. McGraw-Hill Higher Education, 2001. ISBN: 0070131511.

[15] Jong Ho Kim. *High-Concurrency Lock-Free Algorithms for Red-Black Trees*. 2005.

[16] Trevor Brown, Faith Ellen, and Eric Ruppert. “A general technique for non-blocking trees”. 2014.

[17] Trevor Brown, Faith Ellen, and Eric Ruppert. “Pragmatic Primitives for Non-blocking Data Structures”. 2013.

[18] Trevor Brown and Hillel Avni. “Range queries in non-blocking k-ary search trees”. 2012.

[19] Oracle. *HashMap (Java Platform SE 7)*. May 2016. URL: `https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html`.

[20] Brian Goetz et al. *Java Concurrency in Practice*. Addison-Wesley, 2006. Chap. 12. ISBN: 0321349601.

[21] Trevor Brown. *Test harness*. May 2016. URL: `http://www.cs.utoronto.ca/~tabrown/`.