A Practical Implementation of Parallel Ordered Maps and Sets with just Join

Bachelor Thesis of

Daniel Ferizovic

At the Department of Informatics
Institute for Theoretical Computer Science

Reviewer:	Prof. Dr.rer.nat. Peter Sanders
Second reviewer:	Prof. Dr. Dorothea Wagner
Advisor:	Prof. Dr.rer.nat. Peter Sanders
Second advisor:	Prof. Guy Blelloch

Duration: December 15th, 2015 – March 10th, 2016

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, March 14, 2016

.....

(**Daniel Ferizovic**)

Contents

1. Introduction

Modern programming languages nowdays offer support for a variety of data types. Maps, also refered as dictionaries or associative arrays are one of them. Some languages have them built in as part of the language, while other offer support through libraries. Along with maps, libraries often provide implementations of the set data type. Since sets can be viewed as a special kind of an associative container (with unit value associated with each key), they are often implemented as maps. For this reason, from now on we will direct our focus on maps, but note that most things we say also apply for sets. The map data type usually comes in two distinct flavours, it can be either ordered or unordered. An ordered map has a total ordering imposed on the key set of its entries. This can be very useful in practice, since it allows various types of queries based on the key ordering. For example functions like range, rank, select, next/previous-key are usually supported by ordered maps. They are often implemented using some kind of balanced binary search tree as their underlying data structure. This makes most of the operations offered by ordered maps take $O(\log n)$ time. Unordered maps on the other hand have no ordering imposed on their key set. They are usually implemented using hash tables, such that operations like insert and remove take expected $O(1)$ time. The faster time for insertion and removal is favourable for unordered maps, but it often pays off only for a large number of such operations.

In this paper we describe and implement a parallel library for ordered maps and sets. We compare four different balanced binary search trees (BBST) to see which achieves the best performance. This includes AVL trees, weight-balanced trees, red-black trees and treaps. All core library functions are implemented around a single function called `join3`. It takes two balanced binary search trees T_1 and T_2, and an additional key k as arguments, and returns a balanced binary search tree T such that $K(T) = K(T_1) \cup \{k\} \cup K(T_2)$, where $K()$ stands for the key set of a tree. For each BBST we only have to provide an implementation of `join3` in order to gain support for all other library functions. The main reason to implement the interface around `join3` wasn't solely the minimal effort for supporting multiple BBSTs. Aggregate set functions, such as union, intersect and difference can be highly paralellized for ordered trees if implemented using `join3`. The algorithms we use for union, intersect and difference achieve a work of $O(n \log(\frac{m}{n} + 1))$ and a depth of $O(\log m \log n)$, where m is the size of the larger tree. We note that this work is optimal in the comparsion model. These set operations are usually implemented by finger tracing through two sorted sequences simultaneously. This approach is fast in practice and takes $O(n + m)$ time. However, this complexity is worse than ours when it

comes to merging smaller with larger trees. Our experiments (see Chapter 8) also confirm that. Another issue with this approach is that it returns a sorted sequence and not a tree. We note that those issues only occur in the naive sequential implementation. A lot of work has been done in parallelizing bulk operations for certain search trees. Bulk insertions, for example, can be viewed as the union of one tree with another, since it achieves the same effect. To see which of these approaches is more efficient we also compare ourself to state of the art parallel search trees (see Chapter 4). Among other functions that we parallelized are `build`, `filter` and `forall`. In order for our library to be useful in practice we made all operations persistent. That is, taking the union of two maps does not side effect the input maps. Persistence can be useful in many applications. For example, if we have two tables in our database, we probably don't want them to be completly destroyed after taking a single intersection or union of them.

In Chapter 4 we talk more about related research and the work which has been done in this area. Chapter 5 describes the `join3` operation and how it can be realized for the different types of balanced search trees. We also talk about how other operations can be implemented by using only `join3`. Chapter 6 describes the implementation details of our library. We talk about the memory management and how exactly we achieved persistance for the data structures. An overview of our library interface can be found in Chapter 7. and at last, in Chapter 8 we show the experimental results we have performed with our library.

2. Acknowledgements

I would like to thank Prof. Guy Blelloch from the Carnegie Mellon University for giving me useful advice and guidance during my work on this thesis. Thanks also goes to Yihan Sun for providing proofs for the running times of several algorithms.

3. Preliminaries

3.1. Binary Search Tree

A binary search tree (BST) is a rooted tree in which every node has at most two children, refered as left and right child. A node without a left and right child is called leaf node. We refer with the left (right) subtree of a node to the rooted tree starting at its left (right) child. Each node of a binary search tree has an unique key $k \in \Omega$ associated with it. We require Ω to be a totally ordered set. Additionaly, each node is able to store some extra information, refered as value.

From now, if not otherwise stated, we will use the large capital letter T to denote a BST and lowercase letters $u, v \in T$, to refer to nodes. Let $v \in T$ be a node of T. We introduce the following notation:

- $v.left$ represents the left child (or subtree) of v.
- $v.right$ represents the right child (or subtree) of v.
- $v.key$ represents the key associated with v.

In general, if we intruduce a property *prop* to nodes we will refer to it as $v.prop$.

Every binary search tree has to satisfy the so called *BST-property*. It states that for every node $v \in T$ the following holds:

$$\forall u \in v.left : v.key \geq u.key \ \wedge \ \forall u \in v.right : v.key \leq u.key$$

The *height* of a binary search tree is defined as the length of the longest path from the root to any other node in T. Similary, the height of a node $v \in T$ is defined to be equal to the height of the tree rooted at v. For convinience we introduce the following notation for a BST:

- $T.root$ represents the root of T.
- $T.height$ represents the height of T.
- $T.prop$ will be used as a short form for $T.root.prop$.
- *K(T)* is the set of all keys present in T.
- *V(T)* is the set of all values present in T.

3.2. Balanced Binary Search Trees

Most operations on a BST are in worst case proportional to the longest path in the tree. To reduce the cost of such operations we want to have BSTs with a height as small as possible. The best we can hope for a BST with n nodes is a height of $\lceil \log(n) \rceil$.

A *balancing scheme* is a set of restrictions (or invariants) which ensures that the height of a tree is within a constant of its optimal height. We define a balanced binary search tree (BBST) as a BST which is able to maintain a balacing scheme. That is, any modification of the tree should not lead to a violation of the balancing scheme. Most BBSTs achieve this through the use of tree rotations (see section 3.3.1). In the following we introduce four different balanced search trees: AVL trees, weight-balanced trees (or BB[α] trees), red-black trees and treaps.

3.2.1. AVL Tree

An AVL tree is a BBST in which the height of the left and right subtree of each node differ by at most one. As shown in the original paper [AVL62] this property represents a balancing scheme. It is also shown how the balance can be manitained across insert and delete operations on the tree.

3.2.2. Weight-Balanced Tree

Weight-balanced trees were introduced the first time by Nievergelt and Reingold [NR72]. Unlike AVL or red-black trees, they don't maintain balance with a height constraint, but with a restriction on the number of nodes in each subtree. We define a weight function on the nodes of a weight-balanced tree tree as follows:

- $w(\varnothing) = 1$
- $\forall v \in T : w(v) = w(v.left) + w(v.right)$

We say that a weight-balanced tree is of bounded balance α if the following holds:

$$\forall v \in T : \alpha \leq \frac{w(v.left)}{w(v)} \leq 1 - \alpha$$

It has been proven by [BM80] that the bounded balance criterion for $\frac{2}{11} < \alpha \leq 1 - \frac{1}{2}\sqrt{2}$ is sufficient for the tree to be balanced. Futhermore, it has been shown that single and double rotations can reestablish the balance in the case of insert and delete operations.

3.2.3. Red-Black Tree

A red-black tree is a BBST in which we distinguish between two types of nodes, red and black nodes. We define the *black height* at each node to be equal to the maximum number of black nodes on a path from that node to a leaf node. The balancing scheme of a red-black tree contains the following properties:

- The root of the tree is black
- Every red node in the tree has only black children
- The black height of the left and right subtree at each node is the same

Red-black trees were introduced the first time in [GS78]. Just like the other trees, they maintain their balance with the use of tree rotations.

3.2.4. Treap

Treaps are randomized binary search trees [SA96]. Each node has an number assigned to it, which we call priority. The nodes of a treap must statisfy the heap property:

$$\forall v \in T : v.priority \geq v.left.priority \wedge v.priority \geq v.right.priority$$

If the priorities are assigned uniformly at random, it can be shown that such a tree has an expected height of $O(\log n)$.

3.3. Basic Operations on BSTs

There are three basic operations that a BST is required to support: `find`, `insert` and `delete`. They are are implemented using a tree traversal on the key order. We describe the algorithm for `find` below:

```
1 Procedure find(key)
2     return search(root, key)
3
4 Procedure search(curr, key)
5     if curr ==⊥: then False;
6     else if curr.key < key: then search(curr.right, key)
7     else if curr.key > key: then search(curr.left, key)
8     else True;
```

Algorithm 3.1: Finding a key in a BST

Insert is implemented by finding the proper position in the tree where the key needs to be placed, similar to the find algorithm. If the key is already present we do nothing, otherwise we create a new node.

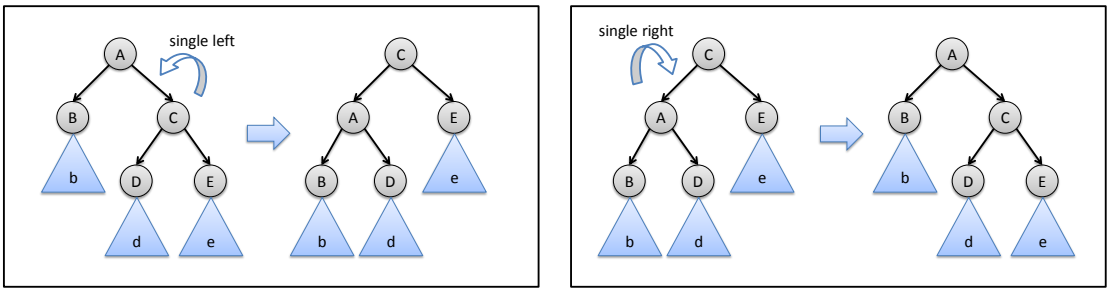
Delete is a bit more complicated, since we can't just remove a node from the middle of a tree. The problem is that we don't have a place to reattach the subtrees from the deleted node. We observe the following cases:

- The node has no childrern. The node can be immediately removed.
- The node has one child. We delete the node and assign the child to the parent of the removed node.
- The node has two childrern. We find the in-order successor of the node which should be removed and swap the keys of the two nodes. We then procceed by deleting the successor node at the lower level until we reach one of the first two base cases.

We note that `insert` and `delete` as described above are quite general. Applying them to any of the mentioned BBSTs will most likely lead to a violation of the defining properties. For example, the height of the subtree in which the node is inserted might increase by one. This could violate the balancing scheme of AVL trees. Similary, the key inserted into a treap following only the key order might violate the heap property. This is usually solved by applying one or more tree rotations.

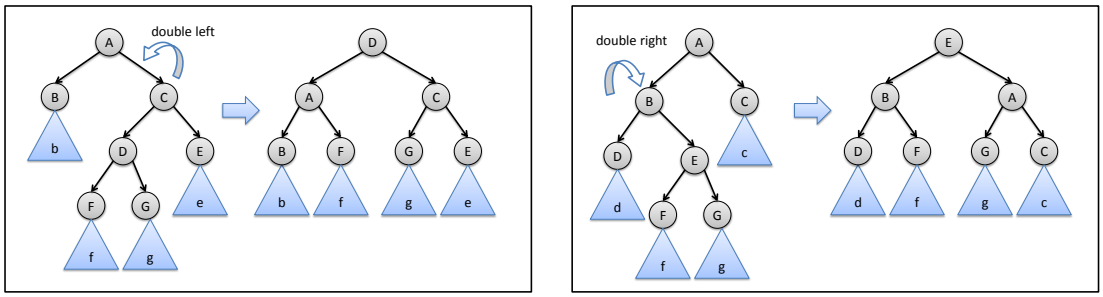
3.3.1. Tree rotations

Tree rotations are often performed in conjunction with other operations. Either to simplify them or to fix the balance condition which might have been violated. There are two types of rotations, single (Figure 3.1) and double (Figure 3.2) rotations. It's easy to see that a double rotation consist of two single rotations, i.e. in Figure 3.2 (a) we first perform a single right rotation on node C, and then a single left rotation on node A.



(a) Single Left Rotation (b) Single Right Rotation

Figure 3.1.: Single rotations on a binary search tree



(a) Double Left Rotation (b) Double Right Rotation

Figure 3.2.: Double rotations on a binary search tree

3.4. Ordered Set

An ordered set is an abstract data structure which contains each element at most once. The elements of an ordered set are totally ordered. They are often used for testing set membership of elements, while being able to add new and remove existing elements. If implemented using a balanced binary search tree, most operations on them can be performed in $O(\log n)$ time.

3.5. Ordered Map

An ordered map is an abstract data structure which contains tuples as elements. The first member of an tuple, refered as key, imposes a total ordering on the elements of a map. The second member represents a value which is associated with the given key. The problem of associating information with a given identifier occurs very often. Just like sets, they can be implemented using a balanced binary search tree. Operations like insert, delte and find all take $O(\log n)$ time.

3.6. Lockfree Data Structures

In a multithreaded environment we often encounter the problem of accessing shared data from multiple threads simultaneously. This usually gives raise to race conditions. A common approach to get rid of them is to serialize the access on shared data by the use of mutexes or locks. While this approach is popular because of its simplicity, it has its drawbacks. Some of them include deadlocks, convoying, priority inversion, threads being not kill tollerant and more [Mic04b].

Lockfree data structures don't suffer from any of the mentioned drawbacks. They provide thread-safe access without making use of synchronization primitives like mutexes. A lock-free data structure guarantees that at any time at least one thread accessing it is making progress. Lock-based data structures fail to satisfy this requirement. Making a data structure lockfree is however more difficult. The reason for this is that we are restricted to a small set of atomic hardware instructions in order to make the structure thread safe. One of the weidely used atomic instruction for this purpose is the `compare-and-swap` (CAS) primitive. It compares the contents of a memory location to a given value and, only if they are the same, modifies the contents of that memory location. Most hardware nowadays supports a double word `compare-and-swap`. Lockfree algorithms based on the CAS primitive have been developed for many common data structures, including linked lists, queues, deques and stacks.

3.7. Persistence

When we modify a data structure we usually get a new modified version of it, without being able to reflect on its previous state. Such data structures are called ephemeral. All modifications on an ephemeral data structure apply to the latest and only version of it. Persistence allows us to keep track of old versions of a data structure. It allows us to lookup older versions, and in some cases to even modify them. Depending on what kind of updates are supported by the different versions of a data structure, we distinguish between three types of persistence:

Partial persistence allows updates only on the latest version of the data structure (Linear versioning).

Full persistence allows updates on any version of the data structure (Tree versioning).

Confluent persistence allows updates on any version. Additionally it allows melding different versions together. (DAG versioning)

3.8. Garbage Collection

A garbage collector (GC) is usually implemented as part of automatic memory management. It is responsible for reclaiming the memory of objects which are not used by a program anymore. We distinguish between two different types of garbage collectors:

- Reference count GC – keeps a reference count to each object in the program. When the reference count to an object becomes zero, the GC immediately reclaims the object's memory.
- Tracing GC – maintains a set of root objects, which includes all objects referenced by the call stack and global variables. The GC reclaims memory by periodically tracing all objects which are reachable from the root set. Everything that was not reached from the root set is considered garbage.

3.9. Parallel Computation Model

The computation model we use is the parallel random access machine (PRAM). It consist of a collection of processors operating on a shared memory. All comunication between the processors is done via the shared memory. Our PRAM model is concurrent-read/exclusive-write (CREW) PRAM, which means that multiple threads are allowed to read simultaneously from the same memory location, but only one thread is allowed to write a memory

location at the same time. This can be achieved using various synchronisation mechanisms. We avoid the use of locks as much as possible and rely on atomic instructions like `compare-and-swap` and `test-and-set`.

We use the work/depth model to express the running times of our algorithms. Work is the total number of operations performed by an algorithm, while depth represents the longest strongly sequential chain of operations that an algorithm needs to perform. In this sense, work is equivalent to the sequential time of an algorithm, and depth is equal to the running time of an algorithm on a machine with an unlimited number of processors.

4. Related Work

Implementing a join operation for balanced binary search trees has been studied before. Blelloch and Miller [BRM98] describe how join can be implemented for treaps. They introduce parallel algorithms for union, intersect and difference based on the join operation. All of them take $O(n \log(\frac{m}{n} + 1)$ time, which is optimal in the comparsion model. Similary, in his work [Tar82], Tarjan gives an efficient join-algorithm for red black trees.

A lot of reasearch has been done in the field of concurrent and parallel operations on binary search trees. Kung and Lehman gave in their work [KL80] a generic approach on how to implement concurrent safe binary search trees. They study in great detail how to build a concurrent safe system which would support the basic BST operations. In order to achieve this, they make use of locks, node copying and back pointers. Some of the later research on concurrent search trees includes the work of [BCCO10] and [BE13]. Bronson et al. describe a concurrent safe AVL tree which uses hand-over-hand optimistic validation along with a relaxed balancing criteria, which reduces the contention on the tree by postponing rebalancing operations. Among other balanced binary search trees which have been studied for concurency are red-black trees. Some of the later research in this area has been done by Besa [BE13]. He describes a fast implementation of a concurrent red-black tree, which outperforms the previously introduced AVL tree by Bronson.

Besides providing concurrent access, attention has also been given to the parallelization of certain operations. Park and Park [PP01] give a highly theoretical description of parallel red-black trees for the PRAM model, including bulk insertion/removal and tree contruction. However, they do not consider implementation related issues, nor they provide an actual implementation of the trees. Later on, Frias and Singler [FS07] have described and implemented a parallel version of red-black trees. They included their implementation as part of the MCSTL (Multi Core Standard Template Library), which is to our knowledge one of the only libraries providing parallel sets and maps for C++. They parallelize bulk updates by initialy splitting the tree by the update sequence, and then in parallel insert the remaining sequences into the right subtrees. Another interesting library available for C++ is the HPC++ which includes the PSTL (Parallel Standard Template Library) [JGB97]. The PSTL supports maps and sets, however, it is designed for the distributed setting as opposed to shared memory.

Parallelizing certain operations has not only been limited to binary search trees. Erb et al. introduce in their work [EKS14] a parallelized weight-balanced B-tree. They make use of the property that partial reconstruction of weight-balanced B-trees can be done

in constant amortized time. This way they avoid the overhead of frequent rebalancing operations during bulk insertions. Similar work has been done by Akhremtsev et al. [AS15], who studied parallel bulk operations on (a, b)-trees. They parallelize bulk updates by splitting the input tree into p distinct trees, where p denotes the number of processors. Then they insert the corresponding sequence parts into the individual trees, after which all of the trees are joined back together. Based on our experimental results, and those of Akhremtsev, the (a, b)-tree implementaion is the fastest among the parallel search trees we tested.

When it comes to persistence, groundbreaking work has been done by Driscoll, Sarnak, Sleator and Tarjan [DSST89]. They describe general transformations on how to make any pointer based structure fully persistent. They achieve persistence using $O(1)$ additional space per operation with only a constant slowdown. This is however only true for full and partial persistence. Confluent persistence turned out to be more difficult. Based on the prior work of DSST, Fiat and Kaplan [FK01] describe the first general transformation for making any pointer based strcuture confluently persistent. The bounds they achieve are not as good as those of DSST. It is still open if better methods exist in achieving confluent persistence for specific data strucutres, i.e. BSTs. Due to the lack of efficient (and practical) transformations for confluent persistence, a common and simple way to achieve it is by *path copying* [ST86]. There are several benefits in this approach: the data structures are by default safe for concurrency, there is only a constant overhead per access and it is simple to realize in practice. However, it is not the most space efficient solution. Our maps and sets make use of this approach, as we describe in more detail in Chapter 6.

5. The Join Operation

As stated earlier, common operations such as insert, delete and lookup are usually implemented using a tree traversal on the key order. In this chapter we introduce the join operation and argue that it is sufficient to implement all other set operations. This makes it possible to provide a variety of generic operations on many different balanced search trees by implementing just join for each tree. One big advantage of using join is that operations such as union, intersect and difference can be efficiently parallelized if implemented with join.

We desribe two variations of the join primitive, $join2$ and $join3$.

- $join3(T_L, k, T_R)$ takes two BBSTs and one key as an argument, such that $K(T_L) < k < K(T_R)$. It returns a BBST T, such that $K(T) = K(T_L) \cup \{k\} \cup K(T_R)$. If we wouldn't care about working with balanced trees, $join3$ could simply create a new node with key k and attach T_L and T_R to the left and right of it. This obivously, will not work in the case of BBSTs. We will show how $join3$ can be implemented for AVL trees, red-black trees, treaps and weight-balanced trees.
- $join2(T_L, T_R)$ is similar to $join3$ except that it does not take a middle key as an argument. It accepts two BBSTs and returns a BBST T such that $K(T) = K(T_L) \cup K(T_R)$. $join2$ can be implemented using $join3$ as described below.

```
1 Procedure join2(T_L, T_R)
2     if T_L ==⊥ return T_R;
3     if T_R ==⊥ return T_L;
4
5     return join3(join2(T_R.left, T_L), T_R.key, T_R.right)
```

Algorithm 5.1: join2

Theorem 5.0.1 *The work of $join2$ is $O(T_L.height + T_R.height)$ for all of the below mentioned implementations of $join3$.*

Proof See [SFB16].

5.1. AVL Trees

For AVL trees *join*3 is performed as follows: first we check if T_L and T_R differ in height by at most 1. If that is the case, we create a new node with the key k and assign T_L and T_R to be its left and right subtree respectively.

If not, let us assume WLOG that $T_L.height > T_R.height + 1$. We traverse along the right spine of T_L as long as $T'_L.height > T_R.height + 1$, where T'_L is the current subtree at the right spine of T_L. Since T_L is a valid AVL tree, at each step the height of T'_L decreases by 1 or 2. This guarantees that the moment we stop the height constraint $|T'_L.height - T_R.height| \leq 1$ is satisfied. We then create a new node with key k and assign T'_L and T_R as its left and right subtree. The parent of T'_L becomes the parent of k.

We note that this might increase the height of the subtrees along the right spine of T_L. If violations of the height constraint occured, we can fix them by doing single and double rotations from bottom up. The algorithm is described below. For convinience we just show the rebalacing for the case when $T_L.height \geq T_R.height + 1$, the other case is performed analoguosly. The rebalancing rules we use are the same as for insertion – in both cases the height of a subtree increases by at most 1.

```
1  Procedure isSingleLeft(T)
2      return T.left.height < T.right.height
3
4  Procedure rebalanceRight(T)
5      if |T_L.height − T_R.height| ≤ 1
6          return T
7      else if isSingleLeft(T.right)
8          return singleLeftRotation(T)
9      else
10         return doubleLeftRotation(T)
11
12 Procedure join3(T_L, k, T_R)
13     if |T_L.height − T_R.height| ≤ 1
14         return new Node(T_L, k, T_R)
15
16     if T_L.height > T_R.height
17         T_L.right = join3(T_L.right, k, T_R)
18         return rebalanceRight(T_L)
19     else
20         T_R.left = join3(T_L, k, T_R.left);
21         return rebalanceLeft(T_R)
```

Algorithm 5.2: join3 for AVL trees

Theorem 5.1.1 *The work of join*3 *for AVL trees, as described in 5.2, is* $O(|T_L.height - T_R.height|)$.

5.2. Red-Black Tree

The *join*3 operation for red-black trees is performed similar to the case of AVL trees. If T_L and T_R already have the same black height we simply attach them as the left and right subtree of a new node with key k. Since the root of a red-black tree has to be black, the newly created node is colored black.

If T_L and T_R are not of the same black height, we assume WLOG that T_L has a larger black height. We descend along the right spine of T_L until we reach a subtree that has the

same black height as T_R. If the node we reach this way is colored red, we keep descending until we hit the first black node. Since at each step the black height either decreases by one (current node was black) or stays the same (current node was red), this will take $O(T_L.height - T_R.height)$. This is due the fact that the number of red nodes can be at most twice the number of black nodes on any path in the tree.

After we find the right subtree T'_L, we merge it with T_R by attaching both of them to a new red node with key k. This might lead to problems, since the parent of the new node might be red as well. This can be easily fixed by repainting the parent node to black and increasing its black height by one. This, however, will lead to inconsistent black heights in the tree, since the black parent will now have the same height as the black grandparent. This is fixed by applying a single rotation as show in algorithm 5.3. Since those two cases might occur multiple times along the way back up, we keep applying those steps when necessary. It is easy to see that the result of these steps will be a valid red-black tree. We describe the algorithm below. The code for the case when T_R has a larger black height than T_L is left out, but it should be clear that this case works analogously.

```
1  Procedure rebalanceRight(T)
2      if color(T) == RED ∧ color(T.right) == RED
3          return Black(T);
4
5      if color(T) == BLACK ∧ color(T.right) == BLACK ∧ T.height == T.right.height
6          T' = rotateSingleLeft(T)
7          T'.color = red
8          T'.left.color = black
9          return T'
10
11     return T
12
13 Procedure joinLeft(T_L, k, T_R)
14     if T_L.height == T_R.height ∧ color(T_L) == BLACK
15         return Red(new Node(T_L, k, T_R))
16
17     T_L.right = joinLeft(T_L.right, k, T_R)
18     return rebalanceRight(T)
19
20 Procedure join3(T_L, k, T_R)
21     if T_L.height ≥ T_R.height
22         return Black(joinLeft(T_L, k, T_R))
23     else
24         return Black(joinRight(T_L, k, T_R))
```

Algorithm 5.3: Join3 for red-blak trees. For simplicity we assume that the black heights change appropriately with any color changes or rotations.

Theorem 5.2.1 *The work of join3 for red-black trees, as described in 5.3, is* $O(|T_L.height - T_R.height|)$.

5.3. Weight-Balanced Tree

We first check if the two trees in question T_L and T_R are weight-balanced in respect to each other. If so, we can simply attach them as the left and right subtree of a newly created node with key k. Otherwise let us assume WLOG that $|T_L| > |T_R|$. We descend along the right spine of T_L until we reach a subtree T'_L that is weight-balanced to T_R. It can be shown that this case has to occur, i.e. it is not possible that we suddenly get a subtree that is so much smaller than T_R that we have an imbalance in the opposite direction. We note

that the restriction of the weight parameter α guaranntes this. At each step the weight of the current subtree decreases at least α-times. It is easy to show that $\left\lceil log_\alpha \frac{|T_L|}{|T_R|}) \right\rceil$ steps are needed until we find the right subtree. In [BM80] it has been shown that a single or double rotation is always sufficient to fix an imbalance after inserting or deleting a node. The case of inserting a whole tree was not considered. However, as shown in [SFB16], the same rebalancing rules still hold true. The algorithm for joining two weight-balanced trees is shown in 5.4.

```
1  Procedure singleLeft(T)
2      return (1 − α)T.left.weight ≤ (1 − 2α)T.weight
3
4  Procedure rebalanceRight(T)
5      if weightBalanced(T.left, T.right)
6          return T
7      else if singleLeft(T.right)
8          return singleLeftRotation(T)
9      else
10         return doubleLeftRotation(T)
11
12 Procedure join3(T_L, k, T_R)
13     if weightBalanced(T_L, T_R)
14         return new Node(T_L, k, T_R)
15
16     if T_L.weight > T_R.weight
17         T_L.right = join3(T_L.right, k, T_R)
18         return rebalanceRight(T_L)
19     else
20         T_R.left = join3(T_L, k, T_R.left);
21         return rebalanceLeft(T_R)
```

Algorithm 5.4: join3 for weight-balanced trees

Theorem 5.3.1 *The work of join3 for weight-balanced trees is* $O(\log_\alpha \frac{|T_L|}{|T_R|})$*, where* $|T_L| \geq |T_R|$.

5.4. Treap

Treaps have the simplest algorithm for $join3$. This is because no rebalancing or fixing is required after we find the right place to join the trees. We first create a new node v with key k and assign it a random priority. If this priority is larger than those of T_L and T_R we simply attach T_L and T_R as the left and right subtree of the newly created node. If not, we take the tree with the root of higher priority and descend to its left or right subtree. We repeat this until both trees in question have a root with priority less than $v.priority$. When this is the case we can attach the trees to the left and right subtree of v. The new parent of v must have a priority higher than v. This is due to the fact that at each step at least one node of T_L and T_R had a priority higher than v, and we always chose to descend along the node with higher priority. The argument applies inductively back up the tree. This means that this approach will lead to a valid treap. We note that the structure of a treap only depends on the priorities assigned to each node, that is, inserting every node of the result tree, one by one, into a new empty treap would lead to the exact same structure. We present the pseudocode below:

```
1  Procedure nodeJoin(T_L, u, T_R)
2      if v.priority > T_L.priority ∧ v.priority > T_R.priority
3          v.left = T_L
4          v.right = T_R
5          return v
6      else if T_L.priority > T_R.priority
7          T_L.right = nodeJoin(T_L.right, v, T_R)
8          return T_L
9      else
10         T_R.left = nodeJoin(T_L, v, T_R.left)
11         return T_R
12
13 Procedure join3(T_L, k, T_R)
14     v = new Node(k)
15     u.priority = random()
16     return nodeJoin(T_L, u, T_R)
```

Algorithm 5.5: join3 for treaps

Theorem 5.4.1 *The work of join3 for treaps, as described in 5.5, is* $O(T_L.height + T_R.height)$.

5.5. Operations using Join

In this section we will show how to implement other set operations using only the $join3$ primitive. This means that all operations we are going to describe can be generically used for all BBSTs we have mentioned. Split is probably the most important operation of them, since it is used as a subroutine in many other set operations.

5.5.1. Split

Split is a function which takes a BST T and a key k as arguments and returns two BSTs, one containing all nodes with keys less than k, and one containing all nodes with keys greater than k. Fromally speaking, $split(T, k)$ is a function which returns a triple $(T_L, flag, T_R)$, such that $K(T_L) < k < K(T_R)$. The trees T_L and T_R contain all nodes from T except the node with key k, if it was present at all. The return value of $flag$ is a boolean value indicating whether a node with key k was present in the tree or not. *split* works recursively by making usage of $join3$. We do a search for the key on the tree. Lets assume the key is in the left subtree of T, and $(T_L, flag, T_R)$ is the result of splitting the left subtree of T. A split of the whole tree according to the key k would simply be $(T_L, flag, join3(T_R, T.key, T.right)$. We give an example of the pseudocode below.

```
1  Procedure split(T, k)
2      if T.key == k
3          return (T.left, true, T.right)
4      else if T.key > k
5          (T_L, r, T_R) = split(T.left, k)
6          return (T_L, r, join3(T_R, T.key, T.right)
7      else
8          (T_L, r, T_R) = split(T.right, k)
9          return (join3(T.left, T.key, T_L), r, T_R)
```

Algorithm 5.6: split

Theorem 5.5.1 *The work of split as described in 5.6 is* $O(T.height)$ *for all mentioned BBSTs.*

Proof See [SFB16].

5.5.2. Union

Union is one of the set operations which can be paralelized using $join3$. $union(T_1, T_2)$ takes two BSTs T_1 and T_2 and returns a new BST T such that $K(T) = K(T_1) \cup K(T_2)$. We note that this is different than $join2$ since no restriction on the key sets are made. In the case that the same key is present in both trees we need a policy to decide which node to keep, since in the case of a map we have an additional value associated with the key. For sets it is irrelevant which node we take. Our implementation uses by default the values of the left tree, but it makes it possible to pass a binary operator which calculates a new value for the given key in the result tree. The binary operator takes two value types as arguments and returns a new value type.

Union works as follows: we first split T_2 by $T_1.key$ and obtain the trees (T_L, T_R) such that $K(T_L) < T_1.key < K(T_R)$. We now compute in parallel $T'_L = union(T_1.left, T_L)$ and $T'_R = union(T_1.right, T_R)$. Since it also holds that $K(T'_L) < T_1.key < K(T'_R)$, we can return the join of T'_L and T'_R using the key of the root of T_1. We give an pseudocode in 5.7.

```
1  Procedure union(T1, T2)
2      if T1 ==⊥
3          return T2
4      if T2 ==⊥
5          return T1
6
7      (TL, , TR) = split(T2, T1.key)
8
9      T'L = spawn union(T1.left, TL)
10     T'R = union(T1.right, TL)
11     sync
12
13     return join3(T'L, T1.key, T'R)
```

Algorithm 5.7: Union of binary search trees

Theorem 5.5.2 *The work of union as listed in 5.7 is* $O(n \log(\frac{m}{n} + 1))$ *where* m *is the size of the larger tree and* n *is the size of the smaller tree. The depth of union is* $O(\log m \log n)$.

Proof See [SFB16]

5.5.3. Intersect

Intersect can also be parallelized using $join3$. $intersect(T_1, T_2)$ takes two BSTs T_1 and T_2 and returns a new BST T such that $K(T) = K(T_1) \cap K(T_2)$. Loosely speaking, it returns the intersection of its two input trees. Just like union it needs a policy for selecting which node should go into the result tree. We use the same strategies as in union. The algorithm first splits T_2 by the root of T_1, obtaining the tripple $(T_L, flag, T_R)$. Since we know that $K(T_L) < T_1.key < K(T_R)$ it holds that $K(T_1.left) \cap K(T_R) = \varnothing$, as well as $K(T_1.right) \cap K(T_L) = \varnothing$. This makes it possible to compute the intersection of T_1 and T_2 by computing $T'_L = intersect(T_1.left, T_L)$ and $T'_R = intersect(T_1.right, T_R)$. This step is done in parallel. If $flag$ was true we know that $T_1.key$ was also in the other tree, so we should include it in the result. This is done by returning $join3(T'_L, T_1.keyT'_R)$. Otherwise we return $join2(T'_L, T'_R)$. We state the algorithm in 5.8.

```
1  Procedure intersect(T1, T2)
2      if T1 ==⊥
3          return ⊥
4      if T2 ==⊥
5          return ⊥
6
7      (TL, r, TR) = split(T2, T1.key)
8
9      T'L = spawn intersect(T1.left, TL)
10     T'R = intersect(T1.right, TL)
11     sync
12
13     if r == true
14         return join3(T'L, T1.key, T'R)
15     else
16         return join2(T'L, T'R)
```

Algorithm 5.8: Intersection of binary search trees

Theorem 5.5.3 *The work of intersect as listed in 5.8 is* $O(n \log(\frac{m}{n} + 1))$ *where* m *is the size of the larger tree and* n *is the size of the smaller tree. The depth of intersect is* $O(\log m \log n)$.

Proof See [SFB16].

5.5.4. Difference

The last binary operator we are going to describe for BSTs is difference. The algorithm is similar to those of union and intersect. $difference(T_1, T_2)$ is taking two BSTs T_1 and T_2 as its arguments and returns a new BST T such that $K(T) = K(T_1) \setminus K(T_2)$. Since we are only keeping those nodes in T_1 which have a key that is not present T_2, there is no ambiguity which node to keep. The algorithm is almost the same as for intersect. The pseudocode is given in 5.9.

```
1  Procedure difference(T1, T2)
2      if T1 ==⊥
3          return ⊥
4      if T2 ==⊥
5          return T1
6
7      (TL, r, TR) = split(T2, T1.key)
8
9      T'L = spawn difference(T1.left, TL)
10     T'R = difference(T1.right, TL)
11     sync
12
13     if r == true
14         return join2(T'L, T'R)
15     else
16         return join3(T'L, T1.key, T'R)
```

Algorithm 5.9: Difference of Binary search trees

Theorem 5.5.4 *The work of* $difference$ *as listed in 5.9 is* $O(n \log(\frac{m}{n} + 1))$ *where* m *is the size of the larger tree and* n *is the size of the smaller tree. The depth of* $difference$ *is* $O(\log m \log n)$.

Proof See [SFB16].

5.5.5. Insert

Insertion can also be realized with the use of $join3$. If we want to insert a new key k into T we first split T into T_L and T_R by k, and then join them back using k as the middle key. The pseudocode is given in 5.10.

```
1 Procedure insert(T, k)
2     (T_L, flag, T_R) = split(T, k).
3     return join3(T_L, k, T_R).
```

Algorithm 5.10: Insertion into Binary search trees

5.5.6. Delete

Removing an entry with a given key k from a BST is done similary to inserting. We first split the tree T into T_L and T_R by k, but instead of calling $join3$ we call $join2$ in order to merge the trees back.

```
1 Procedure delete(T, k)
2     (T_L, flag, T_R) = split(T, k).
3     return join2(T_L, T_R).
```

Algorithm 5.11: Deletion from binary search trees

5.5.7. Range

Range accepts two keys as its arguments k_{low} and k_{high} and returns a new tree containing all nodes with keys between k_{low} and k_{high}. Depending on the implementation k_{low} and k_{high} can be included or not. For clearity, the pseudo code we describe in 5.12 does not include the boundary keys.

```
1 Procedure range(T, k_low, k_high)
2     (T_L, flag, T_R) = split(T, k_low).
3     (T'_L, flag', T'_R) = split(T_R, k_high)
4
5     return T'_L
```

Algorithm 5.12: Range on Binary search trees

5.5.8. Filter

Many functional languages provide a filter function for their data structures. In the case of BSTs, it is taking a BST T and a boolean function $f : K(T) \times V(T) \longrightarrow \{true, false\}$ as its input, and it returns a new BST T' with nodes that have key-value pairs satisfying the condition of f. In 5.13 we describe how to implement filter for BSTs. We note that we consider the general case here, that is, those of ordered maps.

```
1  Procedure filter(T, f)
2      if T ==⊥
3          return ⊥
4
5      T_L = spawn filter(T.left, f)
6      T_R = filter(T.right, f).
7      sync
8
9      if f(T.key, T.value)
10         return join3(T_L, (T.key, T.value), T_R)
11     else
12         return join2(T_L, T_R)
```

Algorithm 5.13: Filter for binary search trees

Theorem 5.5.5 *The work of filter as listed in 5.13 is $O(n)$ where n is the number of nodes in the tree. The depth of filter is $O((\log n)^2)$.*

5.5.9. Build

Building a BST from an array of keys can be achieved in different ways. One way to build a BST is to insert each key one by one into the tree, yielding a complexity of $O(n \log n)$. Another way to build a BST is to make use of union, as described in 5.14. We note that this recursive algorithm even works if the array of keys is not sorted. In the general case the running time of 5.14 is also $O(n \log n)$, but if the array of keys is already sorted the running time becomes $O(n)$. If we assume that the input array is already sorted, we can replace the *union* call with a *join*3, by always taking the middle key of the current array as the middle key for *join*3. This also leads to a complexity of $O(n)$, however with a lower constant. We give an exaple of the build variant for presorted keys in 5.15.

```
1  Procedure build({k_1, ..., k_n})
2      if n == 0
3          return ⊥
4      if n == 1
5          return new Node(k)
6
7      T_L = spawn build({k_1, ..., k_⌊n/2⌋})
8      T_R = build({k_⌊n/2⌋+1, ..., k_n})
9      sync
10
11     return union(T_L, T_R)
```

Algorithm 5.14: Building binary search trees

Theorem 5.5.6 *The work of build as listed in 5.14 is $O(n \log n)$ where n is the number of keys to build a tree from. The depth of build is $O((\log n)^3)$.*

```
1  Procedure buildSorted({k_1, ..., k_n})
2      if n == 0
3          return ⊥
4      if n == 1
5          return new Node(k)
6
7      m = ⌊n/2⌋
8      T_L = spawn buildSorted({k_1, ..., k_{m-1}}
9      T_R = buildSorted({k_{m+1}, ..., k_n})
10     sync
11
12     return join3(T_L, k_m, T_R)
```

Algorithm 5.15: Building binary search trees from a presorted sequence

Theorem 5.5.7 *The work of buildSorted as listed in 5.15 is $O(n)$ where n is the number of keys to build a tree from. The depth of buildSorted is $O(\log n)$.*

6. Implementation Details

6.1. Our library

We designed and implemented a library for ordered sets and maps around the join operation. The implementation was done in C++. For each BBST we only had to implement the *join*3 primitive in order to support the other operations mentioned in Chapter 5. In our experiments we compared the performance of the different trees with each other. The AVL tree turned out to be the best choice, but only by a small amount. We decided to fix the AVL tree for our containers, since allowing the user to provide a tree as additional template argument would lead to less readable code. A more complete overview of the library interface is described in Chapter 7, while in the following we talk more about implementation relevant details. Some features worth noting are that our library is persistent, parallel and concurrent.

The concurrency of our library is a side effect of the persistence and differs from concurrent access in the traditional sense. Inserting n elements concurrently into an empty map produces in our case n distinct maps, each with one element. As opposed to common concurrenct search trees, where all elements are inserted into the same map. For parallelism we use Intel's `Cilk-Plus` extension for C++, which makes it possible to express dynamic nested parallelism for shared memory. To achieve confluently persistent data structures we use a *path copying* approach, which is also used in many functional languages. We note that having full persistence is not enough in our case, since set operations like union effectively meld different data structure versions together. In conjunction with path copying we use a referene counting scheme. Each node stores a reference count which denotes to how many objects it belongs to. This can be useful in multiple ways as we will see later. A problem which arises is that of allocating new nodes and deleting nodes which do not belong to a tree anymore. This is because our aggregate set operations allocate and free memory from multiple threads simultaneously. The conventional memory allocation mechanisms of C++ (`new` and `delete`) were not designed for this setting and scale very poorly. The reason for this is that they lock the heap on each call. We tested several scalable memory allocators (`hoard` and `Intel's tbb`) which were available, but did not get too good results either. For this reason we implemented our own shared memory allocator with garbage collection. In the next section we will describe how we made our data structures persistent. Following that, in section 6.3, we talk more about the memory management of our library.

6.2. Persistence

What we want from persistence is to retain BSTs across various operations on them. That is, inserting elements into an existing tree T should produce a new tree T' but without destroying the old tree T. Similary, taking the union of two trees should produce a new tree without having any visible effect on the input trees. One way to achieve persistence is to copy the entire input trees before an operation and apply the operation on the copies. This however can be more costly than the operation being performed. Inserting into a tree wouldn not take $O(\log n)$ time but $O(n)$.

One way to improve the idea of copying the whole input is to only create copies along the modification path(s) of an operation. This approach is usually called *path copying*. By doing so we only copy those nodes which would have otherwise been modified by an operation. Nodes which are adjacent to the modification path will be pointed to by their old parent nodes as well as the newly created copy. This leads to the fact that large parts of the input tree(s) are going to be shared with the created result tree. Figure 6.1 illustrates an example of path copying in the case of insert. Since nodes can be shared across multiple trees, we keep track of a reference count at each node. This is necessary, since our garbage collection needs to know when it is safe to reclaim a node. This is not the only benefit, however. Depending on the reference count of a node we distinguish between 3 actions:

- $Reference\ count = 0$ – node does not belong to any tree; it is safe to reclaim it by the garbage collection. As the node is recollected, the reference counts of its children are decreased recursively.
- $Reference\ count = 1$ – node belongs to only one tree. If we detect this during an operation we can safely use in-place updates instead of creating copies – destructive operations.
- $Reference\ count \geq 2$ – node belongs to more than one tree. If we detect such a node during an operation we have to create a copy of it, otherwise we would be also modifying a tree which was not meant to be part of the operation. The copy initialy has a reference count of 1 (it belongs to the result tree). The reference counts of its children are increased accordingly. This way we ensure that once we detect a shared node, the whole subtree must also be shared.

Making any tree operation persistent becomes very simple. The only thing we have to do before we start modifying a tree is to increase the reference count of its root. This will trigger the path copying right from the root of the tree and copy all nodes which are affected by the operation. The reference counts also give us the freedom to actively choose whether we want to use persistence or in-place (destructive) updates. Not increasing the reference count before an operation would modify the tree destructively as much as possible. That is, until it detects a node which is shared with another tree. From that point on a path copying would be performed. Being able to perform both persistent and destructive operations is useful, since the later one are more performant. To see how costly the path copying really is, we performed an experiment with both versions. The results can be found in Chapter 8. Our library makes it possibe to choose between persistance and destructive updates. It provides an `final` wrapper for this case, which we describe in Chapter 7.

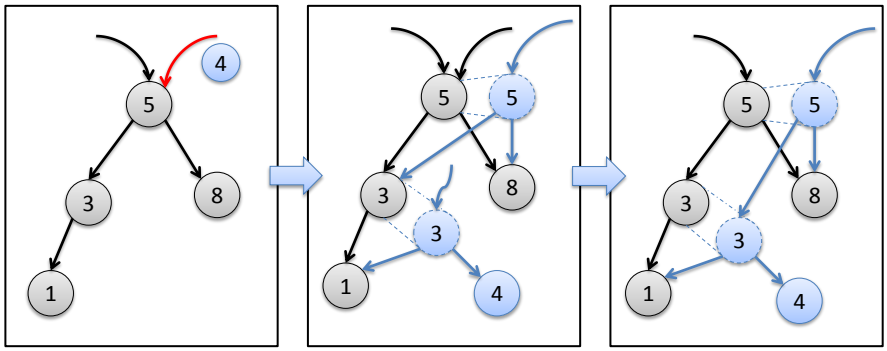


Figure 6.1.: The figure shows the insertion of a node with key 4 into a tree which is shared among two sets. Since the root has a reference count of 2 the copying of nodes is propagated along the insertion path. On the last figure we fix the reference counts of the nodes which have been copied and adjust the pointers accordingly.

6.3. Memory management

We implemented a concurrent reference counting memory management for our library. We decided to use a refernce counting mechanism because it is simple to realize in a multithreaded environment. Maybe even more relevant was the fact that we already kept track of the reference count at each node in order to support destructive operations. One benefit over reference tracing is that nodes can be reclaimed as soon as they are not referenced anymore. This can also be a drawback, since operations can get more costly due to the work of the GC. On the other hand we do not need a "*stop the world*" phase to reclaim nodes, which would be expensive if triggered in the middle of an operation. A disadvantage of reference counting is that it suffers from poor locailty. This is because the memory map of nodes gets more and more scattered as nodes get recollected and allocated again. To improve the cache performance one could occasionally copy the global memory pool to another memory location. This is however left as a posibility for future work.

6.3.1. Reference count collection

Each node keeps track of the number of references pointed to it. Once the reference count drops to zero, we reclaim the node and add it to the pool of free nodes. The thread which recollects a node recursively decrements the reference counts of its children (if present). This is done in parallel using a nested fork-join. Before we recollect a node we copy all the fields we may possibly access after it has been reclaimed. To be concurrent safe all modifications of the reference counts are performed using a `compare-and-swap` primitive.

The memory allocator is concurrent, parallel and lockfree[1]. We maintain a memory pool of nodes, of which all free nodes are assigned to several *free stacks* of equal size. The stacks are implemented as linked lists. Our implementation uses a default stack size of 2^{16}. Initialy when constructing a memory pool all nodes are "free", so we place all the nodes into free stacks. We do this construction step in parallel. For each thread we keep a *local stack* of free nodes. If a thread allocates a new node, it takes it from its own local stack, likewise, if a thread recollects a node it places it back on its own local stack. This way we greatly reduce the contention on the shared memory pool. Beside the local stacks, we maintain a *global stack* of free stacks. The global stack implementation is lockfree.

[1]This is only partially true. The construction of the shared memory pool still needs to be performed under a lock

We use a double-word-compare-and-swap (16B) primitive on the head-pointer and its version number in order to avoid common concurrency pitfalls such as the ABA problem and memory corrucption. We also tested a version with hazard pointers, as described in [Mic04a]. Both of them compared equaly good.

Let s denote the maximum size of a free stack. If the local stack of a thread gets empty it pulls a new stack of nodes from the global stack. However, if the local stack reaches the size of $2s$, we cut the stack in two and place one half back into the global stack. This way we ensure that after we acess the global stack by a thread, at least s allocations or recollections need to be done till the next access. If the global stack runs out of memory, the first thread detecting it will initiate the construction of a new memory pool. New free stacks will be constructed in parallel and placed into the global stack. This is the only time a thread can actually block during an operation. However, as other threads repeatedly pull for new stacks while the global stack is being refilled, the allocator provides reasonable scalability even in this scenario.

7. Library Interface

Our library offers two kinds of ordered maps and one ordered set, all of which are persistent and safe for concurency:

- `tree_map<K, V>`
- `tree_set<K>`
- `augmented_map<K, V, Op>`

where `K` denotes the key type, `V` the value type for maps, and `Op` a binary operator taking two value types, and returning a value type. The generic types were expressed using C++'s templates. The augmented map differs from the common ordered map by accepting an additional binary operator. Each node of an augmented map stores the application of the binary operator on the values of the tree rooted at itself. This can be an useful feature in many applications. In table 7.1 we list the core functions supported by all of our containers along with their cost.

Function	**Work, Depth**
`insert`, `delete`, `find`	$\log n$
`union`, `intersect`, `difference`	$n \log(\frac{m}{n} + 1)$, $\log^2 n$
`forall*`	n, $\log n$
`acumulate*`	$\log n$
`filter`	n, $\log^2 n$
`build`	$n \log n$, $\log^3 n$
`split`, `range`	$\log n$
`next`, `previous`, `first`, `last`	$\log n$
`rank`, `select`	$\log n$

Table 7.1.: The core functions in our map and set library and their asymptotic costs in big-O notation. `acumulate` is only supported by augmented maps, while `forall` only by tree maps.

7.1. Method Summary

We will not go in much detail explaining each function in table 7.1. Most of them were already introduced in Chapter 5 in the setting of BSTs. All functions except `union`, `intersect` and `difference` (and `final`) are member functions. Functions always return their result, however for the different containers some of the functions return a different result type. For example, map insertion returns a new map, but the set insertion returns a new set. In the the following we describe some of the functions from table 7.1, along with functions we did not mention. For clarity the descripton will refer to the map variants of functions.

`assigment operator (m1 = m2)` – is performed in $O(1)$ time. The reference count of the root of $m2$ is increased by one. A new pointer to it is passed to $m1$, but prior to that we invoke `clear` (see below) on $m1$.

`m.content(out_iterator)` – accepts an output iterator and appends all map entries to it. The resulting sequence is sorted in an ascending key order.

`m.forall(f)` – takes a function $f : V \to V_{new}$, and returns a new map where each value v of the old map is replaced by $f(v)$. We note that V_{new} can also be a different data type from V. Since sets do not have values stored in their nodes, we do not provide a `forall` method for sets. We also choose not to add it to the interface of augmented maps. The reason for this is the additional binary operator of augmented maps. Changing the value type of a map by `forall` would make the old binary operator meaningles.

`m.acumulate(k)` – only supported by augmented maps. Takes a key k and returns the application of the map specific binary operator on all values whose entries have a key less or equal than k.

`m.filter(f)` – takes a boolean function $f : K \times V \to \{true, false\}$ and returns a new map with all entries satisfying f. The key-value pairs are expressed using the std::pair type.

`m.split(k)` – takes a key k and returns a pair of maps. The first map contains all entries with keys comparing less than k, while the second map contains all entries with keys greater than k. The result is returned as a std::pair of maps.

`m.range(l, r)` – takes two keys l and r and returns a new map containing all entries with keys in the interval $[l, r]$.

`final(m)` – can be used as a wrapper for maps. Functions accepting a map wrapped with final will operate destructively on the map.

`map_union(m1, m2)`, `map_intersect(m1, m2)`, `map_difference(m1, m2)` – take two maps as arguments and return a new map which represents the union, intersection and difference respectively. Can be used in conjuction with the `final` wrapper. Wrapping a map argument with `final` tells the function to perform the operation destructively. We achieve this by not increasing the reference count before the operation starts. For example, `m = map_union(final(a), b)` will destroy the first map, but the second map will remain unaffected by the operation.

`m.clear()` – empties the map. All nodes owned solely by the map are recollected by garbage collecting threads. The collection is done in parallel using a nested fork-join.

`tree_map<K, V>::init()` – used to initialize the memory allocator. Needs to be called once at the program start before any other operation has been performed.

`tree_map<K, V>::reserve(n)` – accepts a number n and allocates space for n additional nodes. The allocation is concurrenct safe and done in parallel. The function call is also non-blocking – the allocation is performed asynchronously to the program flow. Unlike `init`, calling this function can also be ommited. If the memory allocator runs out of memory it will allocate a predefined amount of space automatically.

`tree_map<K, V>::finish()` – counterpart of `init`. Destroys the memory allocator and returns all the reserved memory to the operating system.

One thing worth noting is that the memory allocator is not bound to individual objects, as it's commonly implemented. It's bound to a whole class instance. This is due to the fact that nodes are shared among multiple maps and sets. For this reason `init`, `reserve` and `finish` are implemented as static methods.

The `maybe<T>` type

Some of our functions may or may not return a valid result. In the case of maps, `find` accepts a key k and only if k is present in the map it returns the value associated with it. But what should `find` return if no such key was present? For this reason we implemented a generic maybe type which wraps its result. The maybe object is convertible to a boolean, and as such it indicates whether it contains a valid value or not. To access a value wrapped inside a maybe we use the star operator (*).

8. Evaluation

To test our library we performed several experiments, both sequential and parallel. This includes comparing various functions and the join based BBST implementations with each other. We also compare our library to the STL implemenations of ordered sets and maps, as well as their generic implementation of `set_union`. Since the STL only offers sequential implementations, we could not use it for any parallel experiments. To see how well our library performs against other parallelized search trees, we compared it against several available implementations, including: MCSTL's red black trees [FS07], weight-balanced B-trees [EKS14] and (a, b)-trees [AS15].

8.1. Setting

For our experiments we use a 64-core machine with 4 x AMD Opteron(tm) Processor 6278 (16 cores, 2.4GHz, 1600MHz bus and 16MB L3 cache). The library was compiled using the `g++` 4.8 version with the `Cilk-Plus` extension for nested paralleism. The only compilation flag we used was the `-O2` flag. For testing purposes we choose our `tree_map`, but we note that our `tree_set` and `augmented_map` would lead to similar results. To construct the maps in our experiments we generate multiple sets of key-value pairs varying in size from 10^4 to 10^8. All keys and values are 32-bit unsigned integers choosen uniformly at random. In a single data set the keys are required to be unique, whereas the same value may occur multiple times. In all of our experiments, for `union`, `intersect` and `difference`, n will denote the size of the larger tree, while m will denote the size of the smaller tree.

8.2. Comparing different trees

To compare the various BBSTs we choose `union` as the representative operation. Other operations would lead to similar results since all operations except `join3` are generic across the trees. The experiments we performed showed that the trees are competitive with each other. This may be due to the fact that the cost of cache misses dominates over the other operations we perform. Despite the similarities, the AVL tree gave the overall best results. It is about 15% better than the red black tree, which had the slowest times in our experiments. Table 8.1 shows the timings of all our BBSTs across different thread counts for $n = m = 10^8$, while table 8.2 shows the results for $n = 10^8$ and $m = 10^7$. It is evident that our union algorithm performs work proportional to the smaller input tree.

Tree	Number of Threads						
	1	2	4	8	16	32	64
AVL	34.69	17.75	8.86	4.51	2.29	1.17	0.80
WB	35.37	18.30	9.40	4.79	2.41	1.23	0.87
Treap	37.35	19.40	10.03	5.08	2.57	1.30	0.86
RB	40.91	21.02	10.85	5.54	2.79	1.41	0.97

Table 8.1.: Timings for `union` across different trees ($n = m = 10^8$).

Tree	Number of Threads						
	1	2	4	8	16	32	64
AVL	3.37	1.75	0.91	0.46	0.23	0.12	0.085
WB	3.60	1.85	0.94	0.49	0.24	0.12	0.089
Treap	3.71	1.95	1.01	0.50	0.25	0.13	0.088
RB	4.46	2.19	1.14	0.55	0.28	0.14	0.1

Table 8.2.: Timings for `union` across different trees ($n = 10^8, m = 10^7$).

In Figure 8.1 we show the speedup of `union` across different trees for $n = m = 10^8$. We get a 40-fold speedup on our machine. For smaller input trees we get less speed up, which is due to the lack of paralelism. In the case of $n = m = 10^5$ we get a speedup of 25. Figure 8.2 illustrates the timings of `union` as a function of size where $n = m = 10^i$.

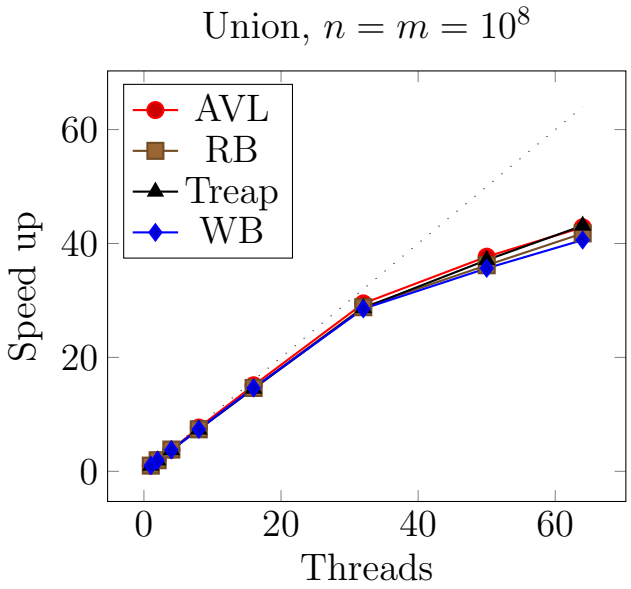


Figure 8.1.: Speed up for `union` across different trees ($n = m = 10^8$).

8.3. Comparing functions

We use the AVL tree as the representative tree to compare different functions with each other. As one can see from the previous experiments, choosing any other tree would not have a significant impact on our timings. In Figure 8.3 we give a comparison of

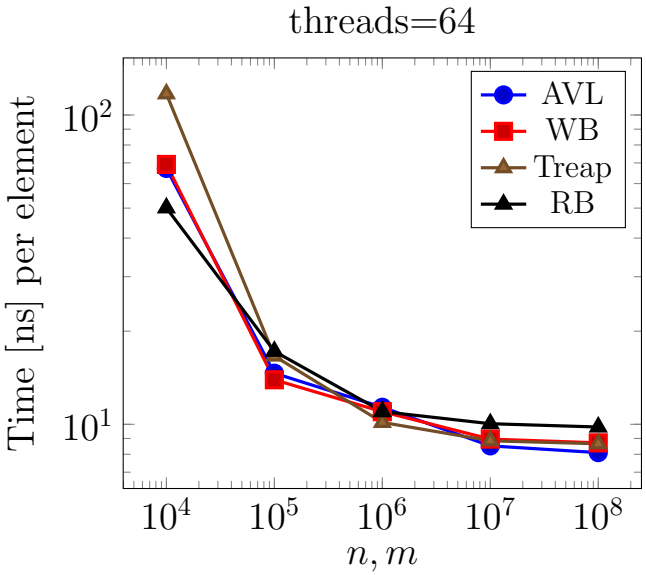


Figure 8.2.: Timings of `union` across different trees as a function of size $n = m = 10^i, i \in \{4..8\}$.

various functions, including `union`, `intersect`, `difference` and `filter`. For the first three operations the size of the larger tree is fixed, while the size of the smaller tree is varied from 10^4 to 10^8. Since `filter` only operates on a single tree, it uses the smaller tree as its input. For the filter condition we choose to keep all entries with an even value. This way every node has an equal probability of making it into the result set. The first three operations compare very similar, while filter is a bit faster. This is due to the less work it has to perform. In another experiment we tested different approaches of building a map from a sequence of random key-value pairs. The first approach is making use of the union-based `build` as introduced in Chapter 5. The second approach is first presorting its input using the parallel sort implementation of the STL. In contrast to the `union`-based build we use `join3` to merge the subtrees together as we go back up the recursion tree. This is more efficient than calling `union`, which effectively also ends up calling `join3`. Both approaches take $O(n \log n)$ time. The dominant cost for the second appraoch is the presort, while the actual build takes $O(n)$ time. The comparison of different `build` methods is given in Figure 8.3. It shows that from an input size of 10^5 the union-based `build` compares worse against the join-based. We think that the worse time for smaller sizes is because the parallel sort of STL is not well suited for smaller input sizes. In our library we make use of the join-based variant. We argued in the section about persistance that we support both persisting and non-persisting (destructive) operations. We expect that a destructive operation is less costly than its persisting counterpart. The reason for this is that destructive operations omit the cost of copying nodes entirely. To see how much of the cost falls to persistance we performed an experiment with `union`. The code for both variants is the same, the only difference is that the destructive `union` takes its arguements wrapped in a `final` statement. Figure 8.3 gives a comparison of a persisting and a destructive `union`. The results show that the destructive version is about 40% faster across all input sizes. This supports the fact that destructive operations can be useful in practice.

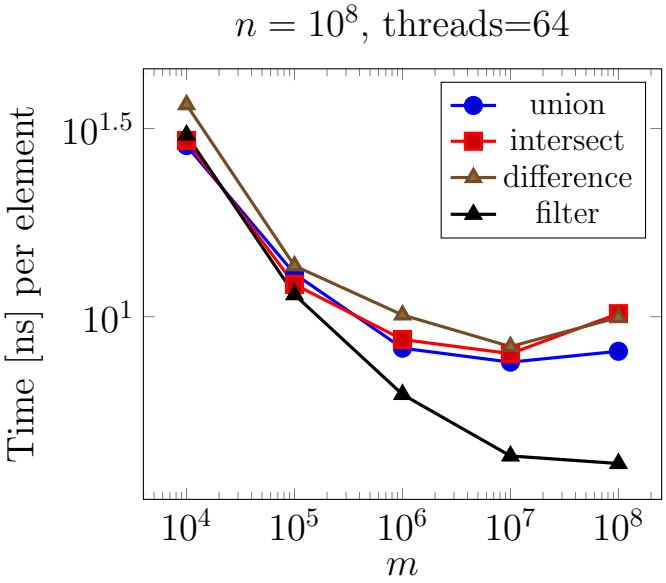


Figure 8.3.: Comapring different functions on AVL trees. ($n = 10^8, m = 10^i$ for $i \in \{4..8\}$).

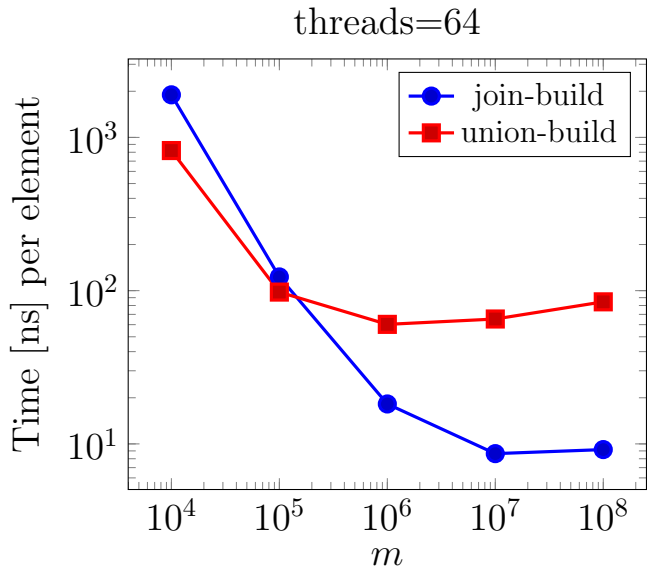


Figure 8.4.: Comparing different versions of building a map.

8.4. Comparing to STL

The STL has implementations of an ordered map and set. Both are implemented using a red-black tree as the underlying structure. Many operations that we offer are not supported by the STL. One of the things we tested is the insertion into a map. We were inserting elements in a random key-order into the maps. Since our implementation of insert is making use of join and split, it is expected to be slower than a direct implementation. To see how big the difference is between a join-based and a conventional implementation, we implemented and tested both version. We illustrate a comparison in Figure 8.6 (a). Our join-based insertion is by a factor of 2 slower than our direct implementation. The STL map was a bit faster than our direct implementation which is most likely due to the fact that we do extra work in order to check for shared nodes. The second experiment we performed with the STL was about taking the union of two containers. The STL

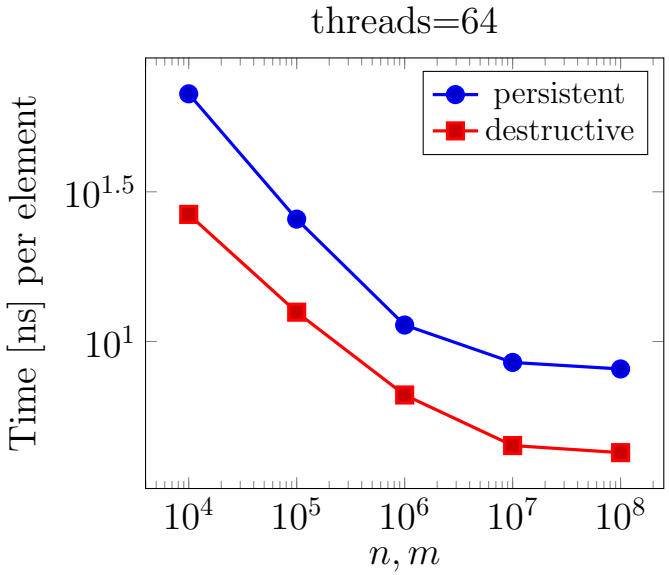
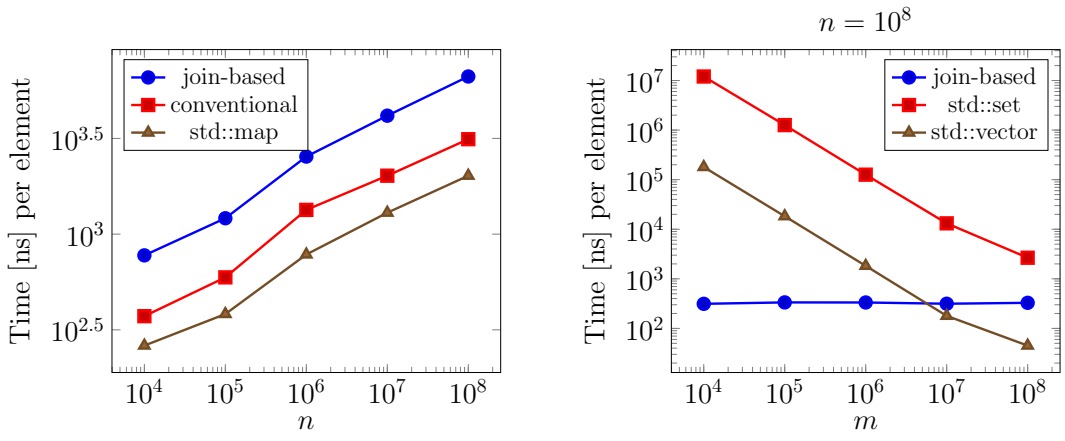


Figure 8.5.: Persistent vs. Destructive `union`.

offers for this the `set_union` function, which takes two sorted containers and returns their union. It is implemented by finger tracing both input sequences simultaneously, yielding a running time of $O(n + m)$ in the worst case. However, this is only true for random-access containers. `set_union` can be also used to compute the union of maps or sets, but it will do so by inserting all the elements into the result container. This will lead to a worst case complexity of $O((n+m)\log(n+m))$. We give a comparison of `set_union` with `std::vector` and `std::map` along with our `union` implementation in Figure 8.6 (b). It is evident that our `union` is a lot faster in merging smaller with larger trees. Our `union` is up to 4 magnitudes faster than the vector-based `set_union` for $n = 10^4$ and $m = 10^8$ The worst case for our `union` algorithm is when both trees are of equal size. At this point we get worse than the linear time `union`.For $n = m = 10^8$ we are about 8 times slower than the vector-based `set_union` and about 8 times faster than map-based `set_union`.



(a) Map insertion (b) Sequential Union

Figure 8.6.: Comparing our library to the STL

8.5. Comparing to parallel implementations

We compare our library against several parallel search trees, which include parallel red-black trees from the Multi Core Standard Template Library (MCSTL), weight balanced B-trees, as described in [EKS14], and parallelized (a, b)-trees [AS15]. To compare the various implementations we use union as the representative operation. All of the mentioned search trees provide a parallel implmenetation of union or offer support for bulk insertions. Since both operations achieve the same effect, we will simply refer to them as union. To get times which would reflect the most accurate performance in practice, we measured the time across a sequence of incremental union operations. That is, we first construct a main tree and then iterate over a sequence of newly constructed trees (bulks) and merge them one by one to the main tree. This scenario is likely one for which the union operation would be commonly used in practice. We report the average time over the sequence of updates. In order not to overuse the main memory when performing this experiment with larger input trees, we restrict the number of iterations to be 100.

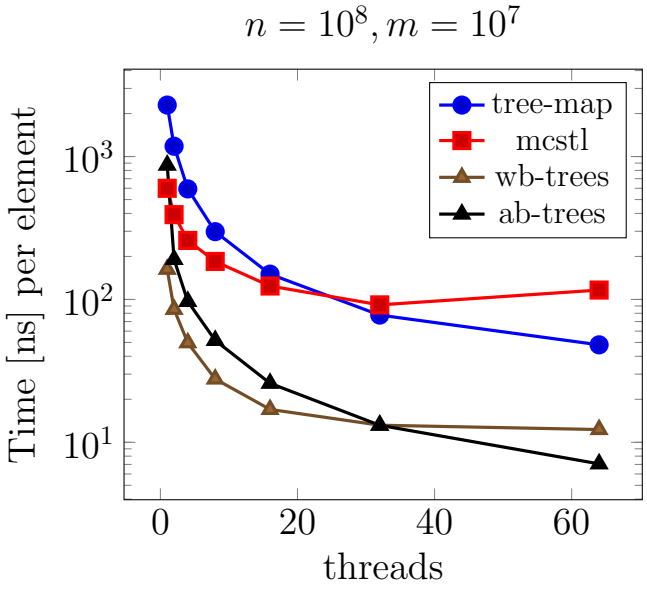


Figure 8.7.: Comparision of `union` on parallel search trees. All times represent the avarage time taken over 100 consecutive union operations.

In Figure 8.7 we give an comparsion on how well the various implementation perform for the case $n = 10^8$ and $m = 10^7$. We note that n here indicates the size of the main tree, while m stands for the bulk size. The times show that our algorithm compares worse than the other search trees, except from the MCSTL. Although it has a slower sequential time, it achieves a far greater speedup. From a thread count of about 20 and onwards our algorithm outperforms the MCSTL by up to a factor of 2 on 64 threads. The B-tree and (a, b)-tree implementations turned out to be superior over the binary search trees. The (a, b)-trees achieved the best times. Compared to our implementation on 64 threads, they are by a factor of 7 faster, while the weight balanced B-trees are faster by a factor of 4. On another figure 8.8 we show the same experiment but across different tree sizes. We fix the main tree size to be 10^8 and set the thread count to 64. The figure illustrates the times as a function of the bulk size, from 10^4 to 10^8. The figure shows that the binary search trees get barely better at processing single elements as the bulk size increases on 64 cores. The (a, b)-trees, on the other hand, get about 10 times faster in processing an element when the bulk size changes from 10^4 to 10^8. Overall our implementation turned out to be competitive with the MCSTL's red-black trees, but not with the other

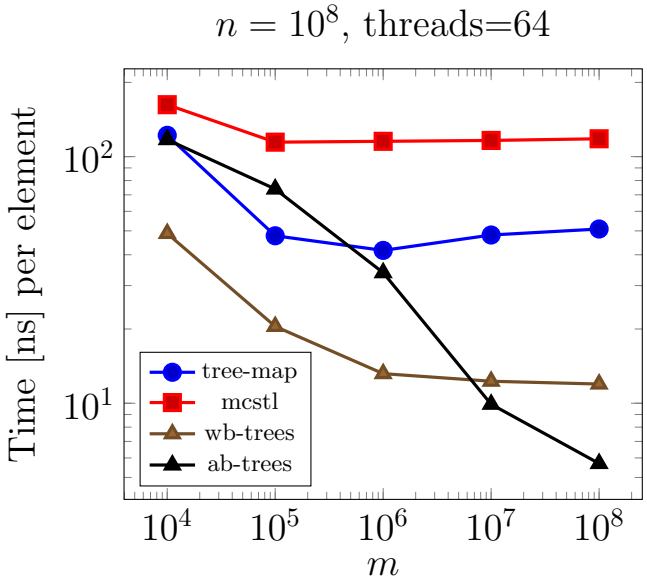


Figure 8.8.: Comparision of `union` on parallel search trees. All times represent the average time taken over 100 consecutive union operations.

implementations. Some reasons might include their high cache efficency, but also overall lower height compared to the binary search trees.

9. Conclusion

We implemented a parallel and persistent C++ library for ordered maps and sets. For the underlying structure we compared the performance of four different balanced binary search trees, including AVL trees, red-black trees, weight-balanced trees and treaps. The implementation of a single tree is fully captured by the join operation. We describe efficient join algorithms for all of our trees. They are work efficient and construct a balanced result tree. With the use of only join we show how to implement many other tree operations.

Based on the experiments we performed our library achieves its best results with the AVL tree. Some reasons for this might be the stricter balacing condition and the simpler implemenation. We demonstrated that the join-based algorithms for union, intersect and difference can be up to several magnitudes faster (on a single core) in merging smaller with larger trees, compared to a naive sequential implementation. However, our implementation did not turn out to be the best option when it comes to the parallel setting. Despite being comparable to the MCSTL, our implementation was outperformed by weight-balanced B-trees and (a, b)-trees. Our parallel algorihms achieve a speedup of up to 44 on 64 cores. On larger input files the speedup is almost linear when using less than 32 threads, beyond we achieve a parallel slowdown. We do not think that the reason for this is the lack of parallelism. It is more likely due to the memory communication, which seems to be a bottleneck. Our implementation was designed to be persistent and concurrent safe. Despite the additional work we do to achieve this, our implementation compared reasonably well to the STL implementation of ordered sets and maps. Persistence can be useful in many applications. We use a version of path copying which makes it possible to use to use persistence at will. We have shown that it can be useful in practice to support destructive operations as well. It would be interesting to see how our implementation would perform with other persistence techniques, which are more memory efficient. Another thing we left as a possibility for future work was the implementation of a tracing garbage collector. It would improve the cache locality and reduce the time overhead of recollecting nodes immediately.

Bibliography

[AS15] Y. Akhremtsev and P. Sanders, "Fast parallel operations on search trees," 2015.

[AVL62] G. M. Adelson-Velsky and E. M. Landis, "An algorithm for the organization of informaiton," *Soviet Mathematics Doklady*, 1962.

[BCCO10] N. Bronson, J. Casper, H. Chafi, and K. Olukotun, "A practical concurrent binary search tree," *PPoPP '10 Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2010.

[BE13] J. Besa and Y. Eterovic, "A concurrent red-black tree," *Journal of Parallel and Distributed Computing*, 2013.

[BM80] N. Blum and K. Mehlhorn, "On the avarage number of rebalacing operations in weight-balanced trees," *Theoretical Computer Science 11*, 1980.

[BRM98] G. Blelloch and M. Reid-Miller, "Fast set operations using treaps," *10th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'98)*, 1998.

[DSST89] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, "Making data structures persistent," *Journal of Computer and System Sciences*, 1989.

[EKS14] S. Erb, M. Kobitzsch, and P. Sanders, "Parallel bi-objective shortest paths using weight-balanced b-trees with bulk updates," *Proceedings of the 13th International Symposium on Experimental Algorithms*, 2014.

[FK01] A. Fiat and H. Kaplan, "Making data structures confluently persistent," *Proceedings of the 12th Annual Symposium on Discrete Algorithms*, 2001.

[FS07] L. Frias and J. Singler, "Parallelization of bulk operations for stl dictionaries," *Proceeding Euro-Par'07 Proceedings of the 2007 conference on Parallel processing*, 2007.

[GS78] L. Guibas and R. Sedgeweick, "A dichromatic framework for balanced trees," *Proceedings of the 19th Annual Conference on Foundations of Computer Science*, 1978.

[JGB97] E. Johnson, D. Gannon, and P. Beckman, "Hpc++: Experiments with the parallel standard template library*," *ICS '97 Proceedings of the 11th international conference on Supercomputing*, 1997.

[KL80] H. T. Kung and P. L. Lehman, "Concurrent manipulation of binary search trees," *ACM Transactions on Database Systems (TODS)*, 1980.

[Mic04a] M. M. Michael, "Hazard pointers: Safe memory reclamation for lock-free objects," *IEEE Transactions on Parallel and Distributed Systems*, 2004.

[Mic04b] ——, "Scalable lock-free dynamic memory allocation," *ACM SIGPLAN 2004, Conference on Programming Language Design and Implementation (PLDI)*, 2004.

[NR72] J. Nievergelt and E. M. Reingold, "Binary search trees of bounded balance," *STOC '72 Proceedings of the fourth annual ACM symposium on Theory of computing*, 1972.

[PP01] H. Park and K. Park, "Parallel algorithms for red–black trees," *Theoretical Computer Science*, 2001.

[SA96] R. Seidel and C. R. Aragon, "Randomized search trees," *Algorithmica*, 1996.

[SFB16] Y. Sun, D. Ferizovic, and G. Blelloch, "Just join for parallel ordered sets and maps," 2016.

[ST86] N. Sarnak and R. E. Tarjan, "Planar point location using persistent search trees," *Communications of the ACM*, 1986.

[Tar82] R. E. Tarjan, "Data structures and network algorithms," *SIAM*, 1982.

Appendix

A. Examples

Set union example

```
#include <iostream>
#include <vector>
#include "tree_set.h"

tree_set<int> read_set() { ... }

int main() {
    tree_set<int>::init();

    // construct sets
    tree_set<int> a = read_set();
    tree_set<int> b = read_set();

    tree_set<int> c = set_union(a, b);

    vector <int> output;

    c.content(std::back_inserter(output));
    for (vector<int>::iterator it = output.begin(); it != output.end(); ++it) {
        std::cout << *it << endl;
    }

    tree_set<int>::finish();
    return 0;
}
```

Filter exaple

```
#include <iostream>
#include <vector>
#include "tree_map.h"

tree_map<int, int> read_map() { ... }

int main() {
    tree_map<int, int>::init();

    // construct map
    tree_map<int, int> m = read_map();

    tree_map<int, int > res = m.filter([&] (pair p) {return (p.second % 2 == 0);});

    vector <int> output;
```

```
16      res.content(std::back_inserter(output));
17      for (vector<int>::iterator it = output.begin(); it != output.end(); ++it) {
18          std::cout << *it << endl;
19      }
20
21      tree_map<int, int>::finish();
22      return 0;
23  }
```

Find example

```
1  #include <iostream>
2  #include "tree_map.h"
3
4  tree_map<int, int> read_map() { ... }
5
6  int main() {
7      tree_map<int, int>::init();
8
9      // construct map
10     tree_map<int, int> m = read_map();
11
12     maybe<int> sol = m.find(42);
13
14     if (sol) {
15         cout << "The solution is: " << *sol << endl;
16     } else {
17         cout << "No solution was found" << endl;
18     }
19
20     tree_map<int, int>::finish();
21     return 0;
22 }
```