MSE | MASTER OF SCIENCE IN ENGINEERING

Binary Trees Implementations Comparison for Multicore Programming

Technical report

Department: Information and Communications Technology
Field: Computer Science
Keywords: Concurrency, Binary Tree, Performance, Algorithms, Multicore
Author: Baptiste Wicht
Professor: Claude Évéquoz
Project Period: February 20, 2012 - June 06, 2012
Date: June 7, 2012

Abstract

Although there are a lot of optimized concurrent algorithms for lists, queues, stacks, hash tables and other common data structures, there are much fewer optimized concurrent Binary Search Trees implementations.

This paper compares several concurrent implementations of Binary Search Tree. The implementations are also compared to a concurrent skip list, which is the general way to implement a concurrent set or map.

All the data structures are implemented in C++ using Hazard Pointers to manage memory. The different implementations are compared using several measures to determine the best performing structures in different situations.

The benchmark shows that some data Binary Search Tree implementations are performing much better than the concurrent skip list base implementation. The optimistic AVL Tree that has been tested is shown to be performing very well.

Baptiste Wicht
Computer Science Master degree
Switzerland HES-SO University of applied science

Contents

1 Introduction **4**

2 Analysis **5**

2.1 Binary Search Tree 5
2.2 Skip List 6
2.3 Non-blocking Binary Search Tree 7
2.4 Optimistic AVL Tree 8
2.5 Lock-Free Multiway Search Tree 10
2.6 Self Adjusting Search Tree 12
2.7 The ABA problem 13
2.8 Hazard Pointers 13

3 Implementation **15**

3.1 Hashing 15
3.2 Threads 15
3.3 Random numbers 16
3.4 Hazard Pointers 17
3.5 Skip List 19
3.6 Non-blocking Binary Search Tree 21
3.7 Optimistic AVL Tree 22
3.8 Lock-Free Multiway Search Tree 24
3.9 Self-Adjusting Search Tree 26

4 Test procedures **29**

5 Benchmarks **32**

5.1 Configuration 33
5.2 Random distribution 33
5.3 Skewed distribution 37
5.4 Other scenarios 40

5.5 Memory consumption 49

6 Problems 52

6.1 Lack of complete code in the papers 52

6.2 Differences between Java and C++ 52

6.3 Slow Zipf distribution 53

7 Tools 54

8 Final comparisons 55

8.1 Skip List 55

8.2 Non Blocking Binary Search Tree 55

8.3 Optimistic AVL Tree 56

8.4 Lock-Free Multiway Search Tree 56

8.5 Self Adjusting Search Tree 57

9 Conclusion 58

9.1 Future work 59

9.2 What I learned 62

A Code 63

B Content of the archive 63

References 64

Index 66

Glossary 67

Listings 68

List of figures 69

1 Introduction

The Binary Search Tree (BST) is a fundamental data structure, but relatively less work has been performed providing non-blocking implementation of it. Generally, Binary Search Trees are implemented using concurrent skip list. These last years, some implementations of concurrent BST have been proposed, but they were rarely compared one to another.

The goal of this project is to test and compare several proposed concurrent BST versions. This paper does not propose any new data structure.

Moreover, all the proposed implementations are coded in a programming language with a garbage collector which simplifies a lot memory reclamation to the system (Java for all the studied versions). For this project, they will be implemented in C++ with a strong memory management methodology, the Hazard Pointers.

The compared versions are the following:

- A concurrent skip list as a base for comparison. The chosen implementation is the one described by Herlihy and Shavit [Herlihy2008].
- A non-blocking BST described, proposed by Faith Ellen, from the University of York [Ellen2010].
- An optimistic AVL Tree, proposed by Nathan G. Bronson, from the University of Stanford [Bronson2010]
- A lock-free Multiway Search Tree, proposed by Michael Spiegel, from the University of Virginia [Spiegel2010]
- A Concurrent Self-Adjusting Search Tree, proposed by Yehuda Afek, from the Tel Aviv University [Afek2012]

Each data structure is detailed in the Section 2. All the versions have been adapted to be implemented in C++. The details of the implementation of each data structure are detailed in the Section 3.

Each implementation is tested both in single-threaded and multi-threaded environment. Then, they are all compared on different scenarios: random distribution, skewed distribution, search performances, build performances and removal performances. The memory consumption of each data structure is also compared. The results are presented in Section 5

Each data structure is then compared to see their advantages and weaknesses regarding their implementation and performances in Section 8.

2 Analysis

This section describes the different data structures compared in this document. It also describes the main concepts of what this document is about.

2.1 Binary Search Tree

A Binary Search Tree (BST) is node-based data structure with three following main properties:

1. The left subtree of a node contains only nodes with smaller keys than the node's key.
2. The right subtree of a node contains only nodes with greater keys than the node's key.
3. The left and right subtree are both Binary Search Trees.

With these properties, we can define that the data structure keeps the order of the elements in the natural ordering of the keys. Because of that property, a BST is often used when it is necessary to keep the elements sorted when a new element is inserted or when one is removed from the structure.

This structure is used to implement sets, multisets and maps (associate containers). In this comparisons, the different structures will be used to implement a set. In a set, there are no value associated with a key and a key can be present only once in the set. Moreover, not all the structures that are compared are real BST. Some of them are not even trees, but all of them will be implemented in terms of a set.

Each implementation will support insertion, removal and search of value of an arbitrary type. Each tree can support value of any type, T referring to the type of value contained in a BST. The value is transformed to a key using a hash function (See Section 3.1 for more details).

Each implementation will support the three following operations:

- *bool add(T value)*: Insert the value in the tree at the correct position. If the value already exists in the tree, the function returns *false*, otherwise returns *true* if the insertion succeeded.
- *bool remove(T value)*: Remove the node associated with the given value. If the value does not exist in the tree, the function return *false*, otherwise returns *true* if the removal succeeded.

- *bool contains(T value)*: Returns *true* if the tree contains a node with the given value, otherwise returns *false*.

All the operations have a time complexity depending on the height of the node. In the average case, the time complexity depends on the maximal height of the tree. If the tree is perfectly balanced, the height of the tree is logarithmically related to the number of nodes. In that case, the time complexity of each operation is $O(logn)$. In the worst case (the height is the number of nodes), the time complexity is $O(n)$. From that property, it is clear that an important feature of a Binary Search Tree implementation is to keep the tree as much balanced as possible. In a single-threaded environment, there are a lot of different balancing schemes that can be used (RedBlackTree[RedBlackTree] for examples). In the case of a multi-threaded environment, it is much harder because rebalancing the tree often means to modify a lot of nodes which can causes a lot of contention when several threads are working on the tree.

The difficulty keeping a tree balanced in a multi-threaded environment is the main reason why there are only few algorithms for such structures. The naive solution consisting in locking each node needed in an algorithm only leads to low performance implementations. Moreover, it is not even always possible. Indeed, some rebalancing operations need to modify a number of nodes relative to the height of the tree and it is not conceivable to lock a whole path in the tree for a rebalancing.

2.2 Skip List

A Skip List is a common way to implement a concurrent Binary Search Tree, but it is not a tree. A Skip List is a probabilistic data structure. The balancing is only done based on a random number generator. Due to this random balancing, the worst-case performances of its operations is very bad. But it is very unlikely that an input sequence produces the worst-case performance.

Balancing a data structure using probabilistic algorithm is much easier than maintaining the balance. As there is no global rebalancing needed, it is easier to develop an efficient concurrent skip list implementation. Moreover, that are no rebalancing overhead when the insertion pattern is bad.

As an example, in the Java library, there are no concurrent search tree. Instead, there is a highly tuned implementation of a concurrent skip list. Doug Lea, the developer of this Skip List, said [DougLeaQuote] "Given the use of

tree-like index nodes, you might wonder why this doesn't use some kind of search tree instead, which would support somewhat faster search operations. The reason is that there are no known efficient lock-free insertion and deletion algorithms for search trees". The other data structures presented in this document will show that creating a balanced concurrent tree leads to much more complex implementation than this skip list.

There are several possible variations of Skip List. The implementation studied in this paper is the one proposed in [Herlihy2008].

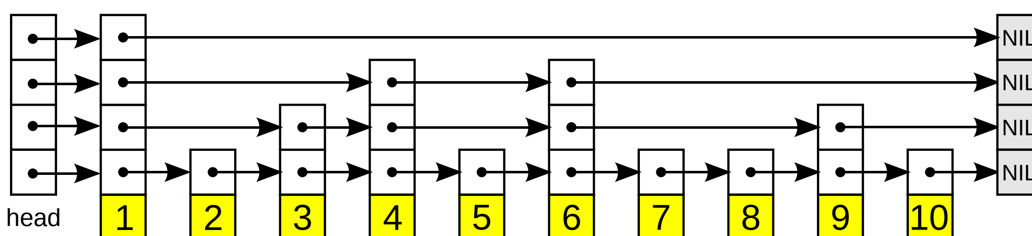

Figure 1: Example of Skip List

Source: Wikipedia

An example of Skip List is presented in Figure 1.

A Skip List is a collection of sorted linked lists. The nodes are ordered by a key. Each node is placed at a specific level chosen randomly at insertion time. The level of the node determines how many forward pointers the node has. The bottom level list contains all the nodes. The higher level lists contain shortcuts to elements of lower lists.

2.3 Non-blocking Binary Search Tree

Faith Ellen proposed a complete implementation of a non-blocking Binary Search Tree in [Ellen2010]. This is the first complete implementation proposed. This implementation does not have any special needs like Transactional Memory, so it can be implemented in any programming language and over any platform. This implementation only uses reads, writes and Compare-And-Swap (CAS) operations. It is really a BST (it verifies all its properties and its general data structure). This structure does not perform any form of balancing.

This implementation is non-blocking and demonstrated to be linearizable as well.

The BST is `leaf-oriented`. Every internal node has exactly two children. All keys are stored in the leaves of the tree. This is done to make the removal of a node much easier to implement as a leaf node has no children and so removing it only means to remove the link from its parent. The downside of this technique is that it leads to a higher memory consumption.

In this implementation, nodes maintain child pointers, but no reference to the parent.

The threads are helping each other to finish their operations. When an operation indicates that it wants to change a child pointer of x, it stores a record about the operation to x and it also flags x with a state indicating what operation it is. It is a form of locking, the child pointers cannot be modified before the operation has been completed the node marked as clean. With that record, another thread can perform the operation itself. To minimize contention, a thread helps another thread only if the other operation prevents its own progress. A contains operation does not help any other thread.

The data structure is presented in Figure 2.

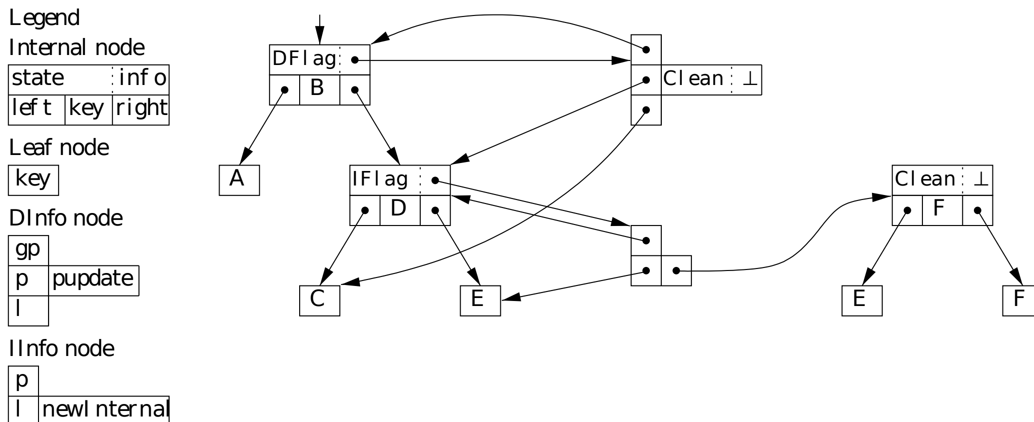

Figure 2: Data structure of the Non-Blocking Binary Search Tree

Source: [Ellen2010]

2.4 Optimistic AVL Tree

Nathan G. Bronson proposed an implementation of a concurrent relaxed balance AVL Tree [Bronson2010].

An AVL Tree[Sedgewick1984] is a self-balancing binary search tree that has the given property: the difference between the height of the left and right

subtrees of a node is never more than one. When this property is violated after an operation on the tree, it is rebalanced again.

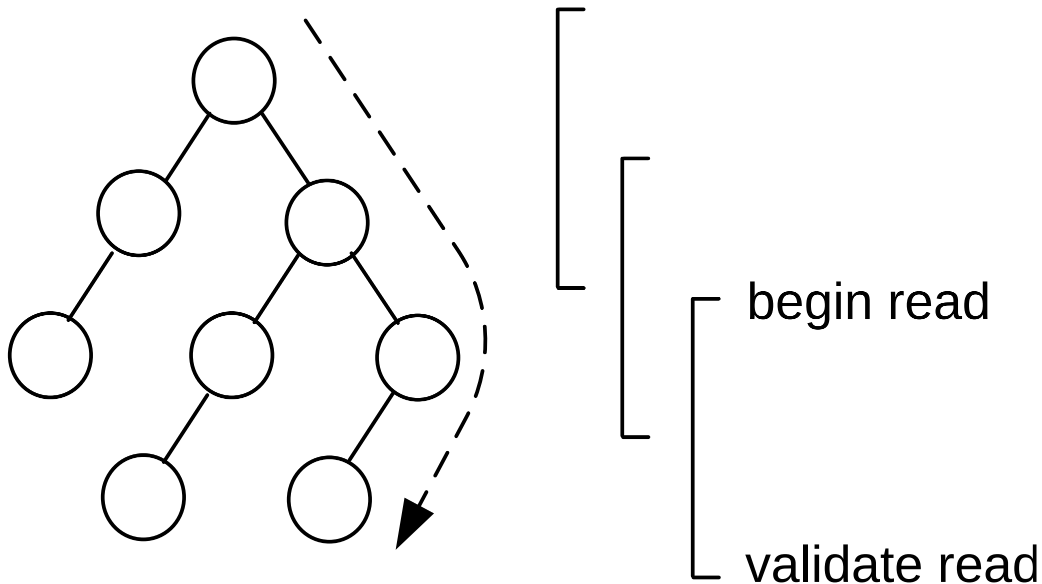

Figure 3: Validation of the AVL Tree

Source: [Bronson2010]

Figure 3 shows the hand-over-hand optimistic validation of the structure.

Implementing a full rebalancing is generally a bottleneck in a concurrent tree. Indeed, it must acquires locks for all nodes that will be rebalanced. That is why the author decided to use a relaxed balance AVL Tree instead of a strict one. In a relaxed balance Tree, the condition can be violated by the operations and is not always restored by rebalancing, but may be.

The algorithms are based on Optimistic Concurrency Control mechanisms. Each node is tagged with a version number. To optimize the implementation, this version number contains two information: the state of the node and a counter. The state of the node indicates that an operation is currently modifying the node. Using that scheme, we can know if a node is currently moving down (shrunk) or up (grown) the tree.

Using that version control, when the version number of a node is known, reads to it are protected by its version number. If the version number changed in the meantime, the operation has to be retried. Some operations does ignore some change in version numbers. For example, in the search operation, when the current node is growing before the operation gets to the next one, the

change is ignored. That allows to fasten some of the algorithms doing less retries.

The tree used in this algorithm has the particularity to be a partially external tree. An external tree is a tree in which all the information is only contained in leaves. An external tree has the advantage to make removal easy at the cost of increasing the average path length and the memory footprint. In a partially external tree, routing nodes (nodes with no information) are only inserted during removal operation. Moreover, routing nodes with only one child are removed during rebalancing. This makes removal easier and does not increase too much the average search path length. Another advantage of this technique is that a routing node can be converted to a value node and vice-versa only by changing its value.

The algorithm performs only local rebalancing. This balancing routine performs several operations:

- Recomputes the height of the node from the height of its children
- Unlinks unnecessary routing nodes
- Performs single or double rotations to reduce the imbalance

All these operations are based on the apparent height of a node. This apparent height is stored inside the node and represents the height that was apparent at a previous point in time.

The locks are used to modify the children. To avoid deadlock, a thread that has a lock on a node may only request a lock on its children.

The used data structure is very simple. Only `Node` objects are used. Each node stores its own height rather than the difference of the heights of its children. A node also stores its version number (used for Optimistic Concurrency Control). Finally, a node has reference to its left and right children. For convenience, the tree stores a reference to a `root holder`. The root holder is a Node with no key nor value and whose right child is the root. This has the advantage that it never triggers optimistic retries (as its versions is always zero) and allows all mutable nodes to have a non-null parent.

2.5 Lock-Free Multiway Search Tree

Michel Spiegel proposed an implementation of a Lock-Free Multiway Search Tree. This structure is a variation of a skip tree. A skip tree is a randomized

tree in which element membership is determined at the bottommost level of the tree [Messeguer1997]. It can be seen as a cache-conscious isomorphism of a skip list. Where the skip list stores only one element per node, a skip tree stores several elements in a single node. This is done in order to improve the spatial locality of the data structure. Just like a skip-list, the balancing scheme of this structure is also randomly done.

In a skip tree, all paths are of the same length. The structures use nodes with zero elements to preserve the invariant. The neighboring elements are used as bounds ($[min, max]$) for the child references.

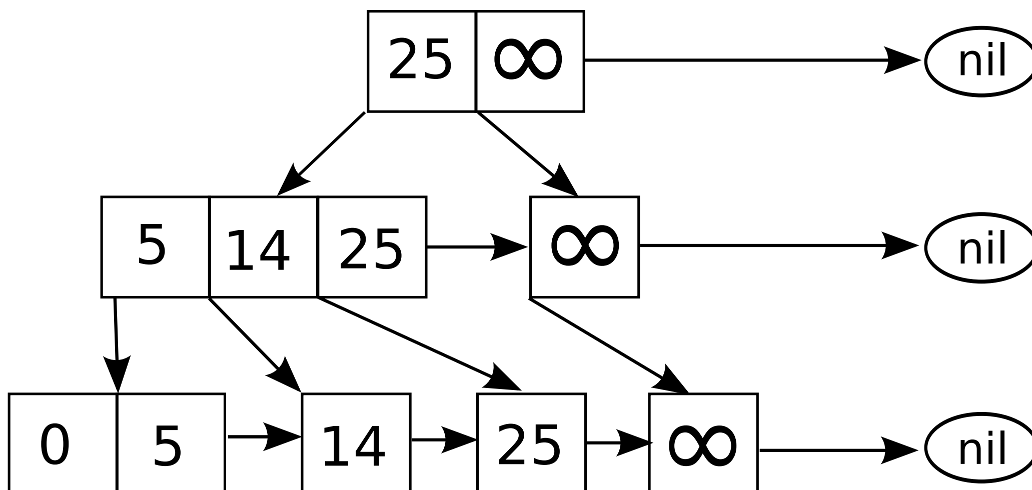

Figure 4: Multiway Search Tree with optimal references

Source: [Spiegel2010]

The Figure 4 shows a skip tree with optimal references.

There are several variations over the original definition of a skip tree:

- The ordering constraint between the nodes is relaxed.
- Optimal paths through the tree can be violated during mutation operations. These paths can eventually be restored using online node compaction.

This data structure supports lock-free add and remove operations and wait-free contains operation.

Another goal of this data structure is to improve the spatial locality of the elements. Indeed, in a skip list when a traversal has to access N elements, it

has to access N nodes. In this structure, there are several elements in each node leading to a better spatial locality.

This tree consists in several linked lists stacked on top of each other. Each node contains some number of elements. The number of elements that is stored inside a node can vary over time. The lowest level of the tree is made of elements with height zero and so is called the leaf level. The other levels are made of routing nodes. A routing node contains some number of references to the elements that are one level below.

When an element is inserted, it is assigned a random height. The heights are distributed according to a geometric distribution. Then, it is inserted at each level up to the chosen height.

During mutation operations, it is possible to introduce so-called empty nodes, nodes with zero elements. It is forbidden to insert new elements in an empty node. They are optimized away using online node compaction. The empty nodes are removed by replacing it by a reference to its immediate successor. Suboptimal references are eliminated by migrating elements from one node to another.

The class structure is a bit complex. Each `Node` consists only of a reference to a `Contents`. This reference is updated only with CAS operations. The `Contents` contains an array of items (the keys) an array of children and a reference to the next node in the linked list. The `HeadNode` contains a single reference to a `Node` object and the tree is made of only an atomic reference to a `HeadNode`.

2.6 Self Adjusting Search Tree

Yehuda Afek proposed a concurrent self-adjusting search tree [Afek2012]. This tree is the only self-adjusting structure tested in this paper. To the best of our knowledge, this is also the only concurrent self-adjusting tree implementation.

A self-adjusting tree adapts its structure to the access pattern. The data-structure will try to keep the average path length as short as possible. It means that this kind of tree is particularly adapted for skewed access patterns.

The implementation proposed by Y. Afek, CBTree (Counter Based Tree) is based on the optimistic BST concurrency control proposed by Bronson (see Section 2.4). The main difference remains in the conditions to perform rebalancing. Another difference is where the nodes are unlinked. In the

AVL Tree, routing nodes with fewer than two children are removed during rebalancing, but this is not the case in this structure. Indeed, routing nodes are never unlinked.

The balancing is made gradually. The CBTree makes at most one local tree rotation on each lookup.

In a conventional splay tree, each time a node is accessed, it is moved to the root. This has the consequence that the root is updated very often making the algorithm not very scalable. That is why this implementation makes more localized changes.

Each node maintains counters to count its accesses. The balancing of the tree is made depending on the counters only, not by comparing the height of the children of a node to make a decision as this is usual. When a lookup detects that rotating a node will decrease the average path length in the tree, rotations are used to move it up in the tree.

The write and read operations to the counters are not protected by locks or other synchronization technique. The reason for that is to avoid contention. A race condition may cause an update to be overwritten, but this is not an issue as the only result is that the adjusting will not be perfect.

2.7 The ABA problem

Almost all lock-free algorithms are subject to the ABA problem. It occurs on this situation:

1. The thread `T1` reads `A` from a shared location
2. The thread `T2` changes the shared location to `B` and then back to `A`
3. The thread `T1` use a CAS instruction to change the value of the shared location. As the value is still `A`, the CAS succeeds.

In this case, as another thread as changed the value of the shared location, the Compare-And-Swap operation should not have succeeded.

Even if this problem is not directly related to memory reclamation, some of the solutions to the memory reclamation problem (Garbage Collector or Hazard Pointers, see Section 2.8) also fix the ABA problem.

2.8 Hazard Pointers

Although lock free structures offer better performance than blocking structures, they are making more complex the memory management. When several threads are manipulating the same dynamically allocated objects, it is very hard to reclaim this memory safely. Even if it is acceptable to use these objects without reclaiming memory for experiment, it is a major obstacle for use in practice.

Hazard Pointers [Michael04] is a memory management methodology that allows memory reclamation to the system. Moreover, it also solves the ABA Problem (see Section 2.7). The operations on Hazard Pointers are wait-free. In this section, the term `Hazard Manager` will be used to refer to the Hazard Pointers manager unit. Another advantage of this technique is that it needs less dynamic allocations, which is very good because in most of systems, dynamic memory allocation is a very costly operation.

The principle is very simple. When a thread need a new object, it asks the `Hazard Manager`. When an object is not used any more in the lock-free data structure, it is released to the `Hazard Manager`. When a node is released, the `Hazard Manager` stores it for later. There are two ways to store released nodes:

1. In a queue local to the thread that has released the node. The advantage of this technique is that there are no concurrent accesses to the queue. So, there is no need for a concurrent queue structure. The disadvantage is that, if all the objects are added by one thread and removed by another one, they will never be reused.
2. In a global queue. This time, the queue needs to be thread-safe.

When a thread ask the `Hazard Manager` for a new object, if there is a free object already allocated, it is given to the thread. An object is free when no thread has a reference on it. This imply that all the threads are publishing references to the dynamically allocated objects.

3 Implementation

This section describes the implementation of the different versions.

All the implementations have been done in C++. As some of the recent features of C++11 (lambda expressions for example) have been used, this implementation requires at least GCC 4.6 to be compiled.

For each of the data structure, a template parameter T defines the type of values stored in the tree. The key is stored with an *unsigned long* type. To configure the number of threads supported by the hazard pointers, a second template parameter *Threads* indicates the number of threads supported by the tree.

The implementation does not use an abstract class to define the features of each tree implementation. Instead, the trees are accessed via template functions to allow polymorphic access to each data structures. Each data structure has a constructor that initializes everything that is necessary for the structure and a destructor that releases all the memory allocated.

3.1 Hashing

This implementation contains hashing functions for `int` and `long` types. The values are mapped to `unsigned long` key. It is necessary that the mapping functions has no collision. If two values have the same key, they cannot be stored twice in the tree.

It has been chosen to use hashing to easily support any type of data in the tree. Another solution would have been to enforce a concept to the type of data. The concept for the type T would have been to be comparable and copyable. Another advantage is that we only have to store an integer key instead of storing full objects or structures. In the case of the implementation of a map, the key could have been stored hashed, but the value would have been stored directly.

3.2 Threads

This implementation uses the C++11 threads library. This library is still young since it only been released with the C++11 version of the C++ language.

Several reasons have conducted this choice:

- It is very simple to use.
- It is part of the standard library. There is no need for a heavy installation.
- It is portable.
- It contains all the functions needed for the different data structures (threads, locking, thread local storage, yielding).

As the C++11 standard is not yet fully implemented in every compiler, a recent version of GCC is necessary to compile the implementation presented here.

In this implementation, the thread id cannot be predicted and the numbers are very big. That is why we stored our own thread id in the thread local storage of each thread. Our thread id starts from 0 and is incremented by one for each thread. We needed this tid model for the Hazard Pointers implementation. The C++11 standard introduces a `thread_local` keyword to declare thread local variables. Sadly, no compilers implement this new keyword at the time of writing. That is why, this implementation use the `__thread` keyword to implement thread local storage. This keyword is GCC specific and so our implementation can only be compiled using GCC.

For all the implementation that use Compare-And-Swap operations, the `__sync_bool_compare_and_swap` GCC Builtin has been used. This function is automatically compiled in the best version possible for the target architecture. In the test machine, it uses only processor operations, so this is very powerful. On architecture that lacks support for CAS at the processor level, this is implemented in software and so is slower. It would have been possible to use the CAS operations of the new C++11 library, but it would have been necessary to store the pointers as atomic reference (`std::atomic⟨T⟩`).

3.3 Random numbers

Some of the data structures need generating random numbers as does the benchmark. All the random numbers used in this implementation are generated using a Mersenne Twister algorithm [Matsumoto1998] on 64-bit word length. The chosen engine is the one implemented in the C++11 random library.

The Mersenne Twister is a pseudorandom number generator developed by Makoto Matsumoto. This generator is based on a matrix linear recurrence

over a finite binary field. It is a very fast generator that produces high-quality pseudorandom numbers. On a 64-bit word length, this algorithm generates numbers with an almost uniform distribution in the range $[0, 2^64 - 1]$.

This generator has a very long period of $2^19937 - 1$. Moreover, the state of the generators is good: it is necessary to observe 624 numbers to predict the future iterates.

3.4 Hazard Pointers

In this implementation, the released nodes are stored in a local queue. The number of threads has to be specified at compile-time because the local queues and reference are statically allocated in an array.

The implementation uses the special thread local variable that indicates the thread number to identify uniquely each thread.

In our implementation, the Hazard Pointers are managed using a single class: `HazardManager`. Each data structure uses one or more instances of this class to manage its memory. This class is configurable by four template parameters:

- `typename Node`: The type of node that has to be managed.
- `int Threads`: The number of threads. This number can be greater than the real number of threads, but cannot be smaller.
- `int Size`: The number of reference each thread can publish. By default, this parameter is set to 2.
- `int Prefill`: The number of nodes to create at the instantiation of the `HazardManager`. By default, this parameter is set to 50.

This class has the following interface:

- `Node* getFreeNode()`: Return a free node that can be used by the calling thread.
- `void releaseNode(Node* node)`: Indicates that the given node is not used any more in the tree. It is possible that others threads still have some references on it.

- `void safe_release_node(Node* node)`: Release the given node, allow duplicates. If the node is already stored in the `LocalQueue` of the thread, this function has no effect. This function should be used wisely. Indeed, it can have a heavy cost because it will have to lookup through the entire `LocalQueue` to find if the node already exists.
- `void publish(Node* node, int i)`: Publish our i^{th} reference to the given node.
- `void release(int i)`: Release the ith reference.
- `void releaseAll()`: Release all references of the thread.

All the functions are depending on calling thread. You do not have to pass the thread id to the function, the function will automatically identify the calling thread and do the action in consequence. The implementation uses the thread local variables explained in Section 3.2

The implementation does not release memory back to the system. Instead, it stores the released node in two queues (two for each thread):

- The local queue: Contains the nodes that have been released by the thread, but that can still be referenced by other threads.
- The free queue: Contains the nodes that are not referenced by any thread.

When a thread ask for a free node, three cases are possible:

- The free queue is not empty: a node is popped from the queue and returned to the thread.
- The free queue is empty and the local queue contains more than R nodes: all the free nodes from the local queue are pushed to the free queue and the first node of the free queue is returned to the thread.
- If none of the previous cases applied, a new node is allocated (via the new operator) and returned to the thread.

In this implementation, $R = (Size+1)*Threads$. This parameter will define when the local queue nodes are pushed to the free queue. Each thread stores also the count of local nodes in simple `unsigned int` counter.

At the instantiation of the class, the free queues are filled with some nodes. The `Prefill` template argument defines how many nodes are automatically allocated by default. This is done to speed up the first uses of the Hazard Pointers to avoid having a lot of allocations one after one after testing if the free queue is empty or not.

When the object gets destructed, it releases all the memory it has allocated and also releases all the nodes still present in both queues for each thread regardless of the references that can still be there.

The queues are implemented using `std::list` instances. The array of pointers is implemented using `std::array` to make sure that the access does not overlap memory. The std::array has a at() function that performs bounds-checking on array accesses. This function has only been used during the debugging and the tests and has been replaced by the normal array accesses without bounds checking for the benchmark.

To fix performance issues, we added access to the content of the queues with two other functions:

- std::list⟨Node*⟩& direct_free(unsigned int t): Return a reference to the `FreeQueue` of the thread *t*.
- std::list⟨Node*⟩& direct_local(unsigned int t): Return a reference to the `LocalQueue` of the thread *t*.

These two methods expose the internal state of the `HazardManager`. The functions are only here for very special cases like the one described in Section 3.8. They must be manipulated with care and never concurrently as the `std::list` is not thread safe.

To ease the debugging of structures using `HazardManager` macros have been used to activate deep debugging of the Hazard Pointers. When the macro `DEBUG` is defined, all the accesses to the different are made using the `at(int)` function. This function throws an error if an out of range access is detected. Moreover, this macro activates also a test to verify that when a node is released it is not already present in the list. This debugging method helped fixing heap corruption problems and multiple releases of nodes. By default, this debugging feature is not activated in the code and the code needs to be recompiled in order to the feature to be activated.

3.5 Skip List

The proposed implementation in Java is using the *AtomicMarkableReference* class. This class allows to atomically manipulate both a reference and a boolean (the mark bit of each address). There is no such class in the C++. As in C++, there are some free bits in addresses (the number of free bit is depending on the architecture), the last bit of each address has been used to store the mark bit. The mark bit is manipulated using bit operations on the addresses:

- $Unmark(address) = address\&(\ 0 - 1)$
- $Mark(address) = address|0x1$
- $isMarked(address) = address\&0x1$

These three operations are made using inline functions. The manipulation of the pointer as an integer is made simply by casting the pointer to a `unsigned long`.

Then, the addresses are manipulated using Compare-And-Swap (CAS) operations.

This implementation has a little limitation. Each node holds an array with the maximum number of forward pointers even if most of them are not used. This makes the algorithm easier, but makes the memory footprint higher because each node will hold the maximum number of pointers. If the numbers of nodes stored in the tree is close to the maximum, this is not a problem, but when the maximum level is set high and there are few nodes, the memory usage will be high.

The level of new elements is computed using a geometric distribution based on a Probability P. It means that there is a probability of P^i that the elements will be at level i.

The memory management has been accomplished using three Hazard Pointers for each thread. References to the nodes that are used in CAS operations are published just after being obtained. It has been very easy to find where were released the nodes as there is only The array of the next elements in the `Node` structure is created at the instantiation time of the `Node` and when the node is obtained from the `HazardManager` it is filled with null pointers.

The Skip List was rather simple to implement in C++. The algorithms are trivial to implement and there was no big challenge adapting the structure to C++. This implementation is also the shortest, less than 300 lines.

3.6 Non-blocking Binary Search Tree

The proposed implementation adds information into a pointer to an Info object. This pair of information has to be stored in a single CAS word in order for the implementation to work. For that, the C++ implementation stores an enumeration value into the two last bits of an Info pointer. So, in the C++ implementation, an Update (typedef to Info*) is equivalent to the structure presented in Figure 1.

Listing 1: Update struct

```
enum UpdateState {
    CLEAN = 0,
    DFLAG = 1,
    IFLAG = 2,
    MARK  = 3
};

struct Update {
    Info* info;
    UpdateState state;
};
```

The management of this information is made using bit operations to retrieve and store the enum information:

- $getState(address) = address\&3$
- $Unmark(address) = address\&(\ 0 - 3)$
- $Mark(address, state) = (address\&(\ 0 - 3))|state$

These three operations are implemented using inline functions. The manipulation of the pointer as an integer is made simply by casting the pointer to a `unsigned long`.

The proposed implementation used two class hierarchies:

- `Node`: implemented by `Leaf` and `Internal`
- `Info`: implemented by `IInfo` and `DInfo`

This works well in the Java implementation, but it is not practical to implement Hazard Pointers. Indeed, a `HazardManager` instance is only able to handle one type of objects. For that, the two hierarchies were merged in two single classes: `Info` and `Node`. `Leaf` and `Internal` are identifiable using a boolean member, **internal**. `DInfo` and `IInfo` are not identifiable, but this is not a problem as no operation needs to know what kind of classes it is. This increases the memory footprint, but makes memory reclamation much easier.

The proposed implementation for this code is using a non-existing operator to create instances of `SearchResult` with only some of the values. In this implementation, the `SearchResult` are created on the stack and passed to the `Search` function that fills the result.

As there are two managed resources (`Info` and `Node`), this implementation uses two `HazardManager` instances. Each HazardManager contains three Hazard Pointers for each thread.

Because of this double hazard management, it was a bit more difficult to add Hazard Pointers to this structure. There are also more locations where the release of nodes must be done. But even with this little difficulty, adding Hazard Pointers was quite straightforward. As the addresses can be marked with some information, it is important to release only unmarked nodes and published references only to unmarked nodes. So each node is unmarked before releasing it to the HazardManager. In the contrary, the references would not match the nodes and some nodes would be reused too soon.

In general, it has been relatively easy to port this structure to C++. The implementation is not very long, a bit more than 400 lines.

3.7 Optimistic AVL Tree

The main problem with the implementation of this version was that the code on the paper was not complete at all. Moreover, the code on the online repository was not the same as the one provided on the paper. That is why the code provided on the paper was not used at all.

The version implemented in this benchmark is the last version proposed on the GitHub repository[BronsonRepository].

The locks are implemented using a *std::mutex* stored in each node. A `std::lock_guard` is used to emulate synchronized-like scopes of Java that are not available in C++.

The Node class has been adapted a little. The `Object` stored in each Node

is replaced by a simple boolean value indicating if the node is a routing node (value = `false`) or a normal node (value = `true`). The structure used in this implementation is presented in the Listing 2.

Listing 2: AVLTree Node structure

```
struct Node {
    int height;
    int key;
    long version;
    bool value;
    Node* parent;
    Node* left;
    Node* right;

    std::mutex lock;
};
```

The implementation uses one `HazardManager` instance to manage memory. Only Node instances are instantiated dynamically. However, this structure needs six Hazard Pointers to publish references.

The problem has been that some functions holding references can be called recursively, but each published reference must be indexed with an integer. This is the case for some advanced rebalancing operations. The solution to this problem has been to keep a thread local counter to know which reference should be published next. The thread local counters are stored into an array of `unsigned int` with a slot for each thread. When the tree gets constructed, the counters are set to 0. Each time a thread published a reference it increments its counter to set the index of the next reference.

A reference to a node is published each time the thread wants to acquire a lock on the node. With that condition, a node cannot be freed as long as a thread holds a lock on it.

A Node is released only after the unlink operation. The operation is also called during rebalancing ensuring that all routing nodes are released.

Even if this implementation is quite long (about 900 lines), it has not been very complicated to adapt it to C++. Most of the code remained the same than the Java implementation.

3.8 Lock-Free Multiway Search Tree

A problem with this implementation was that the code on the paper was not complete. For that, the implementation was based on the code provided on the GitHub repository[SpiegelRepository] by the author (M. Spiegel).

The code was using Java arrays to store the multiple elements of a node, but in C++, it is not possible to know the size of an array. To avoid adding some overhead using `std::vector` or another equivalent, the C++ implementation uses two specialized classes representing an array of children, respectively an array of keys.

The Java code also uses the `System.arrayCopy()` function to copy array. Plain-old pointers loops have been use to replace these calls. The same has been done for the function `Arrays.copyOf()`.

The random level generation is made using a hard-coded random operation (the one provided on the repository). The random seed is generated using a Mersenne Twister algorithm.

The online implementation support fast copy and clone operations. These operations have been removed. The support for these operations (use of proxy object) has also been removed from the implemented operations. All the iteration support has also been removed from this implementation.

The Contents object of a Node was managed with an `AtomicFieldReferenceUpdater` class. In the C++ implementation, this has been replaced with a normal pointer that is written with Compare-And-Swap operations.

In the reference implementation, the add and remove functions return the previous value associated with the given key. This implementation only returns true or false depending on the success of the operation.

The Multiway Search Tree uses special keys: Positive Infinity and Empty Key. For that, the C++ implementation uses a struct representing a key and a special flag indicating if it is a normal key, a positive infinity or an empty key.

The use of Hazard Pointers in this structure was, by far, the most difficult one. Indeed, this structure needs six different classes:

- `HeadNode`: A special class is used to represent the head node. The head node is changed completely by instantiating a new instance and try to change the previous with a Compare-And-Swap.
- `Node`: Each node is managed by an instance of this class. The node

class contains only a Contents pointer that is changed by Compare-And-Swap.

- `Contents`: Contains all the information of a Node: a link to another node, the items of the node and its children.
- `Keys` and `Children`: They are used as dynamically allocated arrays.
- `Search`: Special objects used to perform operations in several parts, especially when inserting a new node to the tree.

All of these classes are allocated dynamically. A good solution would have been to merge `Contents`, `Keys` and `Children`. However, parts of an old `Contents` object is sometimes put in a new `Contents` object before being set in a Node. This problem made impossible to merge them without modifying a lot the implementation, and that is beyond the scope of this project.

Each class uses by the structure is managed using an `HazardManager` instance. Only one reference is necessary for the `HeadNode`. The Search nodes does not need any reference as they are local to a thread. It would be possible to use only new and delete operations for the Search instances, but using `HazardManager` is useful as an object pool to avoid doing too much memory allocation. The other four `HazardManager` instances uses the same number of references: 4 + 8. The first four references are used normally and the other height are used to store references to the objects contained in the array of Search that is created during the insertion of new nodes. The operation add() makes a lookup to get references to each object that needs to be updated (one for each level potentially). Then, in several steps, these objects are modified. The references are stored in an array of `Search`. A reference to each object in the array must be published.

There is no doubt that this management of nodes adds a heavy disadvantage for the performance of this structure. This structure has really only be thought for the Java programming language. It is certainly powerful using a Garbage Collector, but this is apparently not case when managing the references directly in the code.

As there are a lot of functions in this structure, it has been very difficult to publish the references at the correct locations. And also very complicated too to release the nodes at the good place. The release of most of the objects have been easy, but this is not the case with Node. It has been very complicated to find the point where a Node instance is not referenced any more in the structure. Indeed, sometimes references to a node are moved, especially to

make better references, but it is not very clear where they are really removed. It has been found that most of the nodes are removed in the `pushRight()` operation. This operation cleans empty nodes. However, it has not been possible to find the point where the last reference to a node in the tree was removed. For that, each thread keeps track of the pseudo removed nodes in a thread local storage. When the tree gets destructed, the stored nodes for each thread are reduced into one sets containing only unique pointers. This set is then combined with the nodes already in the `HazardManager` and finally the destruction can be done as usual in the `HazardManager` destructor. Not all the nodes are removed automatically by the remove operation in this tree. For that, it is necessary to iterate through the whole tree in the destructor to collect the still existing nodes (and all other objects of the tree) and combine them with the previously collected elements. Again, this adds a big overhead to the structure. This adds also a memory overhead to the structure as more nodes are being stored even when they should be reused. And finally, this adds an indirect overhead because the nodes cannot be reused and this causes more memory allocations than necessary.

It is clear that the implementation that has been done of this structure is not optimal. Nevertheless, it would have taken much more time to modify the structure itself to ease the use of Hazard Pointers.

This implementation is by far the most complex of all, counting more than 1900 lines.

3.9 Self-Adjusting Search Tree

The locks are implemented using a *std::mutex* stored in each node. A `std::lock_guard` is used to emulate synchronized-like scopes.

The implementation has been based on the Java implementation provided directly its author, Yehuda Afek.

The implementation is very similar to the one of the Optimistic AVL Tree, except that it is based on an older code base and that the rebalancing code has been entirely rewritten to be frequency-based instead of height-based.

Again, the class used to store Node has been adapted a little for the need of the implementation. The structure used in the implementation is presented in the Listing 3.

Listing 3: CBTree Node structure

```
struct Node {
    int key;
    bool value;

    Node* parent;
    long changeOVL;

    Node* left;
    Node* right;

    int ncnt;
    int rcnt;
    int lcnt;
};
```

The size and the logarithmic size of the tree are stored using `std::atomic<int>` type. These two sizes are manipulated using only atomic operations. The operations used are the one provided by the C++11 standard. The Compare-And-Swap operation that is used is the strong one (the library proposes a weak implementation as well). The difference between strong and weak operation is mostly architecture related. The weak version may fail spuriously even if the current value is the expected one. In some architecture, for example those that implements some kind of relaxed consistency, the strong operation would be much more costly than the weak one. Normally, on amd64 architecture, there are no difference between the two versions. The local sizes are stored in an array with a case for each thread.

The `NEW_LOG_CALCULATION_THRESHOLD` is calculated at the construction time of the tree based on the number of threads.

Like the Optimistic AVL Tree, this structure uses only one `HazardManager`, with five hazard pointers. The same dynamic hazard pointers has been applied (see Section 3.7 for details).

A Node is released only after the unlink operation. However, there is a difference between the two structures: the AVL Tree cleans out all the routing nodes during rebalancing, but the Counter Based Tree does not have this feature. To pally this problem, at the destruction of the tree, the tree is depth-first searched to release all nodes still remaining on the tree.

A reference to a node is published each time the thread wants to acquire a lock on the node. With that condition, a node cannot be freed as long as a

thread holds a lock on it.

4 Test procedures

As all the data structures were entirely implemented in another language than that of the provided source, they were all tested for correctness. The tests consist in two part:

- A first set of tests is applied in a single-threaded environment. This set of tests is more complete than the second one, because it is also the easiest.
- A more limited set of tests is then applied to the data structure in a multi-threaded environment. This second tests is executed several times, each time with a different number of threads.

For the sake of ease, these two tests were not implemented using a unit test framework, but in plain C++ using assertions. Another reason for that choice is that there are no complete unit testing library for multi threaded code. All our assertions are only to verify that boolean functions returns the good value, so a unit testing library will not bring much to these tests.

The single threaded test consists in several steps:

1. Construct an empty tree
2. Test that random numbers are not contained in the tree and that they cannot be removed from the tree
3. Test that all the numbers in $[0, N]$ can be inserted into the tree. Each inserted number must be contained into the tree after the insertion.
4. Test all numbers in $[0, N]$ can be removed from the tree. Each removed number must not be present after the removal.
5. Test that no numbers in $[0, N]$ are present in the tree
6. Insert N random numbers taken in $[0, INT_MAX]$ in the tree
   (a) If the number is already present in the tree, assert that it is not possible to add it again
   (b) Otherwise, insert it and verify that it has been inserted (it is present)
7. Test that numbers not present in the tree cannot be removed

8. Test that all the random numbers inserted at Point 6 can be removed
9. Test that the tree does not contain any number

The multi threaded tests are launched with 2, 3, 4, 6, 8, 12, 16 and 32 threads. Again, there are several steps to the test:

1. Verify that each number in $[tid * N, (tid + 1) * N]$ can be inserted into the tree. Verify that the inserted number has been inserted.
2. Remove all the numbers inserted at Point 1.
3. Verify that all the numbers in $[0, Threads * N]$ are not present in the tree.
4. Each thread is assigned a random fixed point
5. Insert the random fixed point into the tree
6. Do 10'000 times:
   (a) 33% of the time, remove a random value (different from each fixed points) from the tree
   (b) 67% of the time, insert a random value (different from each fixed points) into the tree
   (c) After each operation, verify that the fixed point is still in tree
7. Remove all the inserted values but the fixed points
8. Verify that all the fixed points are still in the tree

Each step is made by each thread.

During the sequential insertion phase of both tests, the number of nodes to insert is divided by 25 if the tree is not balanced. A simple type traits is used to determine at compile-time if a tree is balanced or not.

All the random numbers used during the tests are generated using a Mersenne Twister Engine (See Section 3.3).

All the implementations have also been tested for memory leaks. For that, the tests were run with Valgrind [valgrind]. The default tool of Valgrind (memcheck) has been used. The main reason for that was to check if the Hazard Pointers implementation was correct. As all dynamically objects

are managed using an HazardManager instance, if this class works every object should be deleted when the tree gets destructed. As other test, the benchmark itself has also been tested itself for memory leaks to verify that each node is removed from the different trees.

It should be noted that the full test suite takes a lot of time, especially on not so-recent hardware.

5 Benchmarks

This section describes the performance tests done on the different data structures tested during this project. These tests are made to compare each implementation on different scenarios.

All the tests were performed on a Lenovo Thinkpad W510. The processor is an Intel Core i7 Q820 at 1.73Ghz. The processor has 8MB of cache. It has four cores and Hyper Threading (HT) on each of them. The computer has 10 GB of DDR3 RAM. The computer is running on Gentoo with the 3.3.0 Linux Kernel.

All the code is compiled with GCC 4.7.0 with the following options: -O2, -funroll-loops, -march=native.

The benchmarks are performed in the machine when the least possible other processes are running in the machine and no user is doing something on the computer. Only Gnome Shell and the terminal are running in the background. During the tests, about 9.5 Go of memory is available for the benchmark.

The measures of time have been done with the C++11 chrono library using the `std::high_resolution_clock` clock to get the best precision available. The durations are then computed in milliseconds.

As there are four cores with HT, the best number of threads (the number where the structure is the most performing) should be eight.

The values inserted in the tree are always `unsigned int`.

In the results of the different benchmarks, the following names are used for the structures:

skiplist The Skip List

nbbst The Non-Blocking Binary Search Tree

avltree The Optimistic AVL Tree

lfmst The Lock-Free Multiway Search Tree

cbtree The Counter-Based Tree

5.1 Configuration

Some of the structures are based on different parameters. This section describes each parameter used during the benchmarks.

The Skip List `MAX_LEVEL` has been set to 24. This allows $2^{24} = 16777216$ elements in the structure (chosen because the maximal number of elements will be 10'000'000). Even for tests when the tree is known to contains less elements, the `MAX_LEVEL` has not been adapted. The `P` probability for the geometric distribution has been set to 0.5.

The AVL Tree `SpinCount` parameter has been set to 100.

The Multiway Search Tree `avgLength` parameter is set to 32.

The Counter Based Tree `SpinCount` parameter has been set to 100. The `YieldCount` has been fixed to 1.

5.2 Random distribution

The first benchmark that is performed on the data structures is based on random numbers. Our experiments emulate the methodology used by Herlihy[Herlihy2006].

The benchmark depends on three parameters, the key range, the distribution of operations and the number of threads.

The performances of the binary search tree in concurrent environment depends on the key range applied during the benchmark, so the benchmark has been made on three different key ranges: $[0, 200]$, $[0, 2000]$ and $[0, 20000]$. A last test is also made on a higher ($[0, 2^{31} - 1 = 2147483647]$). A bigger range means less collisions for the insert and remove operations and also means that the tree can grow bigger and the average path length can go longer as well.

Most of the time, the `contains()` operation is the one that is the most used and logically, the most optimized by the data structures, but that can depend on the situation. As it is not possible to test all existing cases, three were chosen to be tested:

1. 50% of add, 50% of remove, 0% of contains
2. 20% of add, 10% of remove, 70% of contains
3. 9% of add, 1% of remove, 90% of contains

Each of these cases have been tested with different number of threads: 1, 2, 3, 4, 8, 16, 32.

Each thread performs one million operations. At each turn, an operation is randomly chosen according to the current distribution. The operation is performed on a value randomly chosen on the current key range.

It has to be taken into account that for each add operation, the key, if inserted (if the insertion indicates success), is inserted into a vector local to the thread. This is done in order to avoid memory leaks in the benchmark. Experiments have shown that this overhead is very small and does not influence the results as the impact is the same for each data structure. The time to remove the elements is not taken into account, neither is the time necessary to create or delete the instance of the tree. The time to create the threads is counted, but should not change the results.

The used comparison metric is the throughput of each data structures, measured in $[operations/ms]$. Each test is executed 12 times and the mean of the 12 trials is taken as the result.

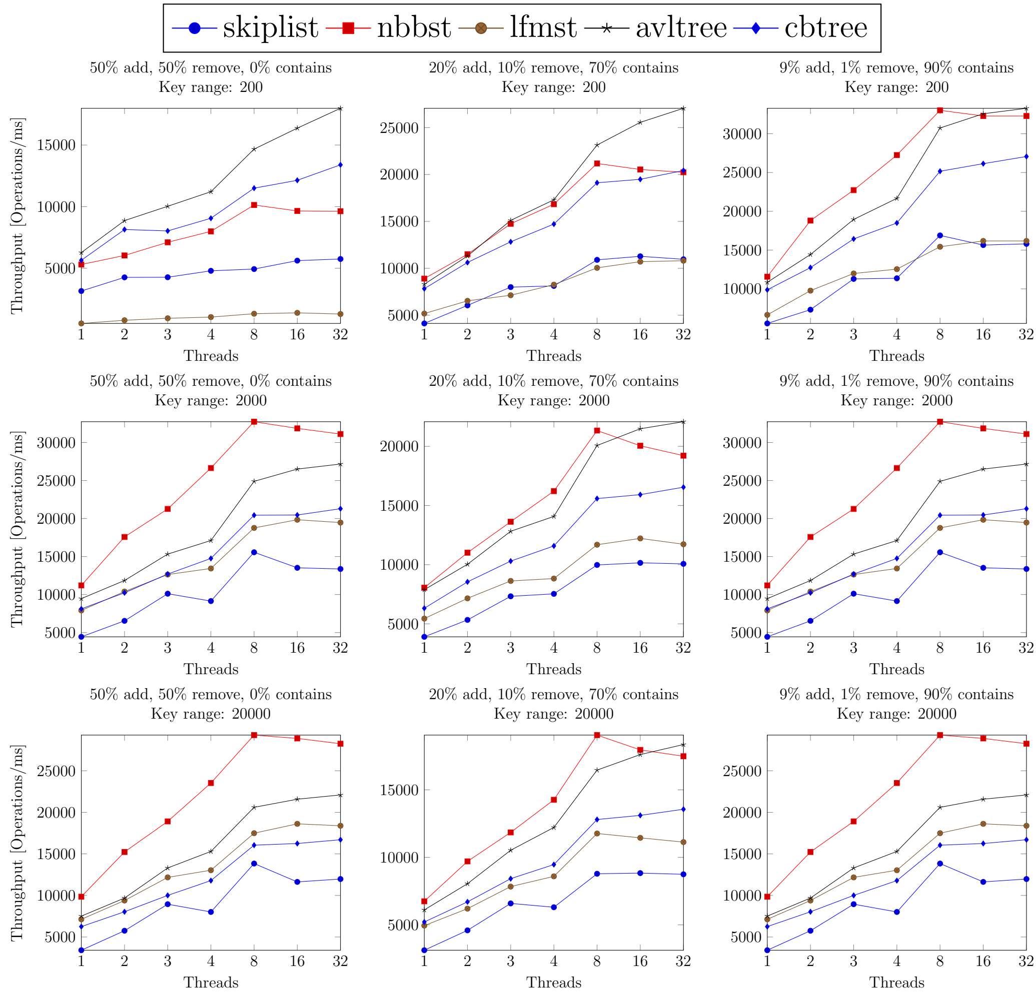

Figure 5: Results of the random distribution benchmark. The higher is the better.

The Figure 5 presents the results that have been obtained for this benchmark.

In this situation, the most efficient structure is the `nbbst`. Except for the smallest range, it is more than 20% more efficient than the second one. It is not a surprising result as this data structure is not balanced and very few operations are necessary for each operation performed by the benchmark. As the used distribution is random, the balancing is already very good, so that the balancing is not really necessary in this case.

At the second position is the `avltree`. On small ranges, it even outperforms the `nbbst`. Its performances are very regular and its scalability very good.

After that, comes the `cbtree`. It outperforms the other two, but not always with a high difference.

The `lfmst` is not performing very well here. Most of the time it is better than the `skiplist` but still very close to it. On high range, it is at the same as the `cbtree`.

The scalability of the different structures is interesting. From one to eight thread, each structure does have about the same scalability, but this is the case for the higher number of threads. Indeed, for number of threads higher than eight, the `nbbst` and the `skiplist` are losing performances while the other are still scaling slowly.

Then, the benchmark has been run again for a higher range.

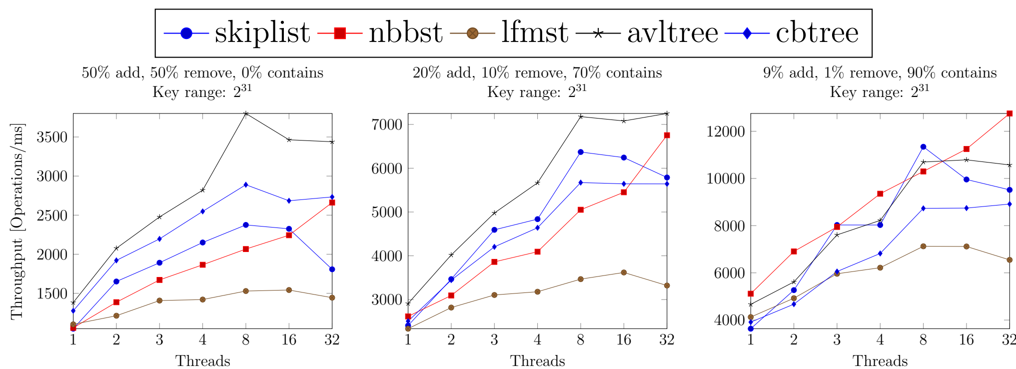

Figure 6: Results of the benchmark on higher range. The higher is the better.

The Figure 6 presents the results that have been obtained for this benchmark on the highest possible range ($[0, 2^{31} - 1]$).

With this range, almost all the insertions should succeed so that the tree would go bigger. It is clear from the results that the throughputs are much lower due to that bigger size.

The `avltree` outperforms the `nbbst` in this case. An interesting thing is that the `nbbst` scalability is much better here than for the previous results. Indeed, the performances are growing even for number of threads higher than eight. From eight threads, it outperforms the `avltree`.

It is interesting to see that the `skiplist` is performing well here, as well as the `cbtree`.

The `lfmst` obtains very bad results, the contention on insertion is really too high. However, we can see that its performance are growing well when the

insertion rate diminish.

5.3 Skewed distribution

To test if the Counter-Based Tree was performing well when some nodes are more accessed than another, we created a benchmark using a skewed distribution.

We used an access pattern based on a Zipf distribution [zipf1949human]. This distribution generates random number in $[A, B]$ with a greater probability to generate numbers near A. A parameter (*skew*) indicates the number of numbers that should be generated near the beginning in the range. This empirical law is based on the analysis of word frequency in natural language. This law also occurs in other rankings like population or income rankings.

The reasons why this distribution has been chosen is simple, it is the distribution used to prove the usefulness of the Counter-Based Tree by its authors.

The distribution results is generated in a file that is then read at runtime. Section 6.3 gives more information about the implementation of the distribution and the reasons that have made this choice.

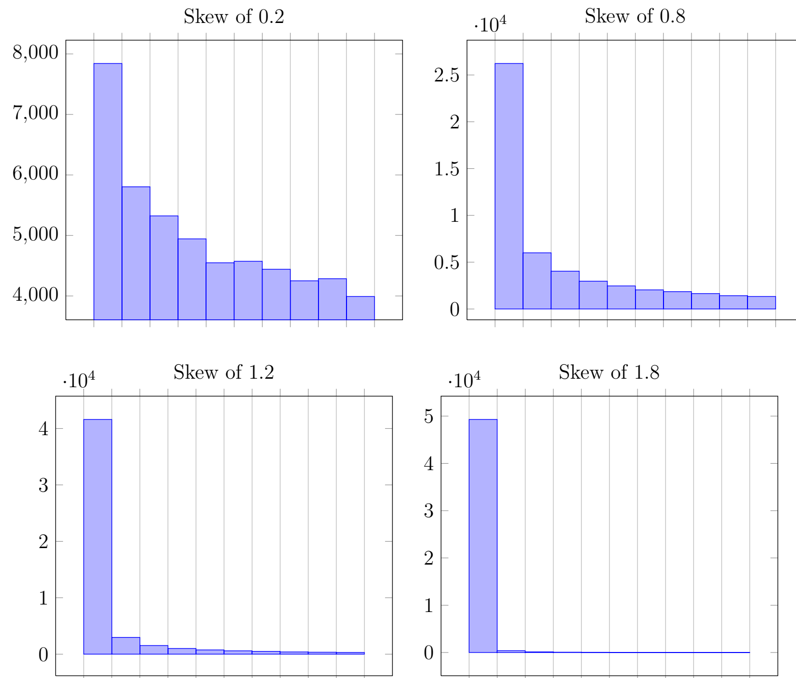

Figure 7: Zipf distributions histograms with different skews

The Figure 7 presents histograms of Zipf distributions with different skews. These histograms have been generated on samples of 50'000 elements in the $[0, 1000]$ range. We see that the higher the skew is, the more elements there is on left side of the range. This is exactly what is expected from a skewed distribution.

In this benchmark, the number of threads is fixed to eight, which is the number of threads at which each implementation is the most performing on the test machine. Instead, the skew parameter of the distribution changes. The test is made for four different key ranges: $[0, 2000]$, $[0, 20000]$ and $[0, 200000]$. It would have been interesting to test higher ranges as well, but the time necessary to generate the distribution sample would have been too high.

At the beginning, each thread performs one million insertions on the tree. On the small ranges, most of these insertions will fail. Therefore, they will behave like a lookup operation. After that, each thread performs another one million operations on the tree. This time, the operation is uniformly randomly chosen with these percentages: 10% add and 90% search.

The values of the Zipf distribution are loaded from a file into memory. Each

file contains one million values for a specific range and for a specific skew value. When a thread needs a random value, it chooses it in the values loaded from the file at a random index (generated with a Mersenne Twister engine).

It has to be taken into account that for each add operation, the key, if inserted (if insertion function indicates success), is inserted into a vector local to the thread. This is done in order to avoid memory leaks in the benchmark. Experiments have shown that this overhead is very small and does not influence the results. The time to remove the elements is not taken into account into the measure.

The used comparison metric is the throughput of each data structures. The throughput is measured in [$operations/ms$]. Each test is executed 5 times and the mean of the 5 trials is used as the result.

The construction and the deletion of the tree object is not counted in any measure, so it doesn't change the results of the benchmark.

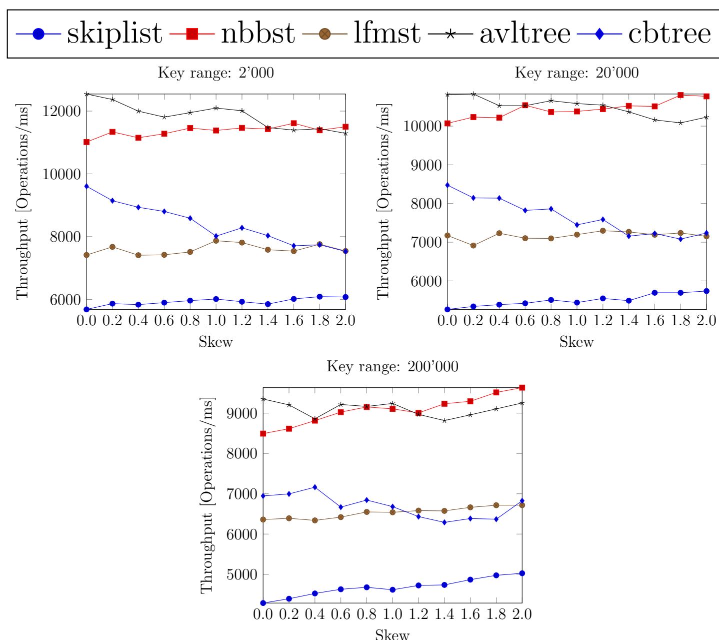

Figure 8: Results of the skewed benchmark

The results of this benchmark are presented in Figure 8.

The results are very surprising. The balancing scheme of the `cbtree` is made for insertion patterns like this one, so it was expected to be the most

efficient under these conditions. The results shown a totally different behavior. Indeed, it is the only structure that suffers from the augmentation of the skew parameter.

The other structures obtain good results and are even gaining performances over the skew growth. This is certainly because the higher the skew is the less elements there is in the tree.

The `cbtree` performances are not very good and are decreasing when the skew increases even if it should the contrary.

No reason has been found to explain these results. It is possible that it comes from the C++ implementation, but that does not seem very likely because the implementation is quite similar to the Optimistic AVL Tree that works very well. It is perhaps of the contention of the insertions under skewed distribution that is higher than otherwise. Another problem is perhaps that the range of the distribution is too small to really test this tree. Due to problems with the generation of Zipf sample (See Section 6.3), it has not been possible to perform tests on higher ranges.

5.4 Other scenarios

Random accesses to a tree is a good way to get performance measurement for the general case, but there are cases when other patterns of access are used. For example, for some people, only the performances of the search operation does matter and for other, only the insertion in existing tree does matter.

In this section, different behaviors are tested on the tree implementations to test how they perform in specific operations.

For these scenarios, only the following numbers of threads are being tested: 1, 2, 3, 4 and 8.

Each test is executed 5 times and the mean of the 5 trials is taken as the result.

5.4.1 Search

This section describes tests that are done to test the ability of each tree to search values. Different tree sizes are tested: 50000, 100000, 500000, 1000000, 5000000 and 10000000 elements. Then, each thread is performing one million search operation. The searched key is randomly chosen through the inserted range. Only contained values are searched. Only the throughput

(in [$operations/ms$]) of the search operations is measured. The time needed to construct the tree and insert the values is not taken into account.

The tests are separated in two tests:

- In the first test, the elements are inserted in random order in the tree
- In the second test, the elements are inserted sequentially

This separation is made to compare the balancing of each tree.

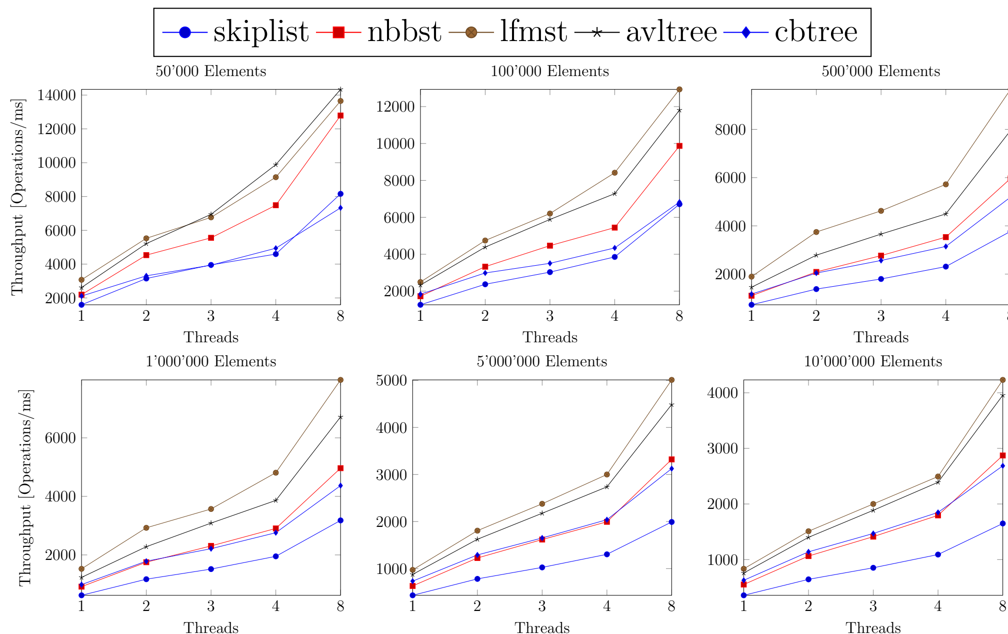

Figure 9: Search performance of the different structures. The elements are inserted in random order. The higher is the better.

Figure 9 shows the search performance of the threads.

The results are very interesting. Indeed, this time, the `lfmst` outperforms all the other structures. It is closely followed by the `avltree`. After that, the `nbbst` and the `cbtree` have quite similar performances.

The worst data structure is here the `skiplist` with performances thrice worse than the `lfmst`.

It is interesting to note that for search operations all data structures scales well.

Then, the same test have been performed with numbers sequentially inserted into the structure. The same number of threads is used. Smaller sizes are tested: 1'000, 5'000 and 10'000 elements. The results are presented in Figure 10.

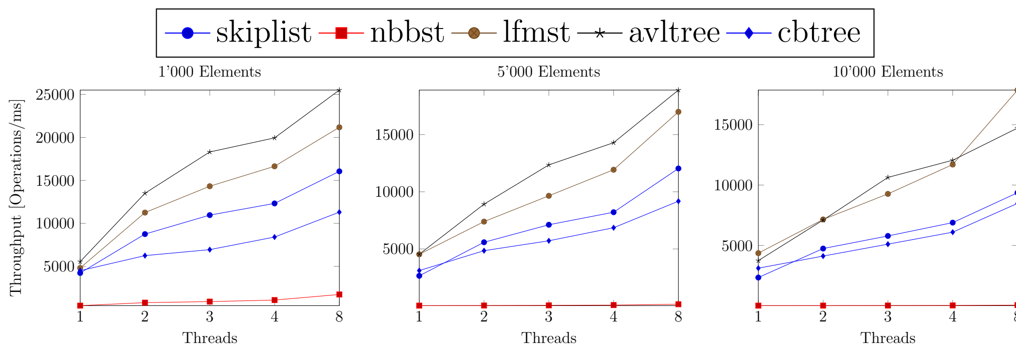

Figure 10: Search performance of the different structures. The elements are inserted in sequential order. The higher is the better.

When the `nbbst` was one of the fastest in the previous results, this time it is, by far, the slowest. Indeed, its performance are orders of magnitude worse than the others. This result was expected as this data structure is not balanced. This is not a problem for random data because randomness is itself a kind of balancing. Sequential insert order is the worst order because it will lead to the highest path length. This can change when working with several threads as the real insert order cannot be guessed.

For the other structures, the best are the `lfmst` and the `avltree`. They are about twice better than the `cbtree` and `skiplist`.

Higher sizes are also tested in the Figure 11. This time, the `nbbst` has been omitted from the benchmarks as it is, by far, too slow to use on trees of these sizes.

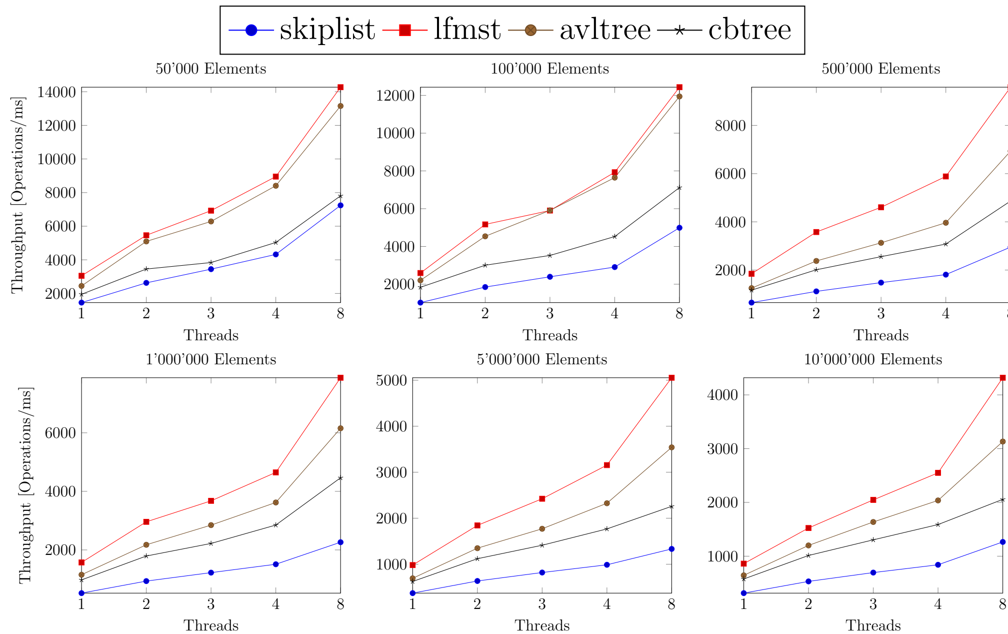

Figure 11: Search performance of the different structures. The elements are inserted in sequential order. The higher is the better.

This time, the `lfmst` is performing even better than before compared to the other. With random insertions, its performance were comparable to the `avltree`, but this time, its performance are much better, by about 25%. This definitively proves that the search performances of this structure are very good.

There are no big differences in the other structures performances.

5.4.2 Building the tree

In this section, we tested how each implementation was performing to build a whole tree. The construction of trees of 50'000, 100'000, 500'000, 1'000'000, 5'000'000 and 10'000'000 elements was tested.

Two tests are performed:

- Insert numbers in random order.
- Insert numbers in sequential order.

We tested both cases to differentiate the balancing of each tree. In both cases, the numbers are in the interval $[0, size]$.

The size of the tree is divided between the threads. The numbers to insert are divided between the threads in this way: the xth thread inserts numbers $[x * (size/Threads), (x+1) * (size/Threads)]$, where $Threads$ is the number of threads. The test has been made with 1, 2, 3, 4 and 8 threads.

The measured metric is the time taken to build the tree. The construction time of the Tree object is also taken into account.

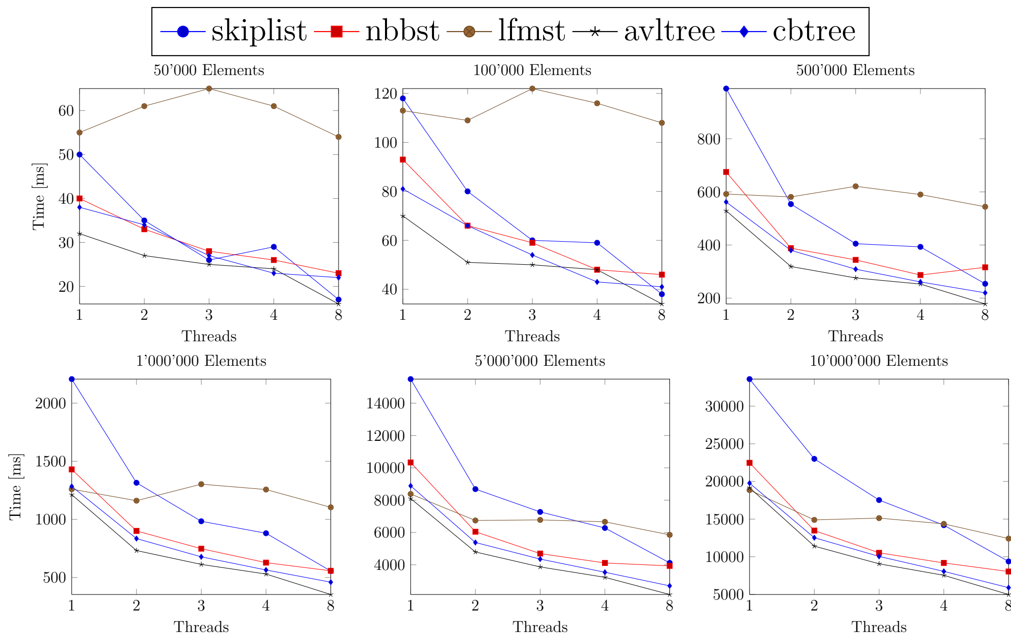

Figure 12: Time to build the different structures. The elements are inserted in random order. The lower is the better.

The first thing that can be observed is that the `lfmst` is not performing well here. Indeed, its scalability is worse than all the others. For high numbers of elements, the `skiplist` is about as bad as the `lfmst`. This two as several times slower than the others.

The three others have good performances and similar scalability, the `avltree` being the fastest.

Then, the same test have been performed with numbers sequentially inserted into the structure. The same number of threads is used. Smaller sizes are tested: 1'000, 5'000 and 10'000 elements. The results are presented in Figure 13.

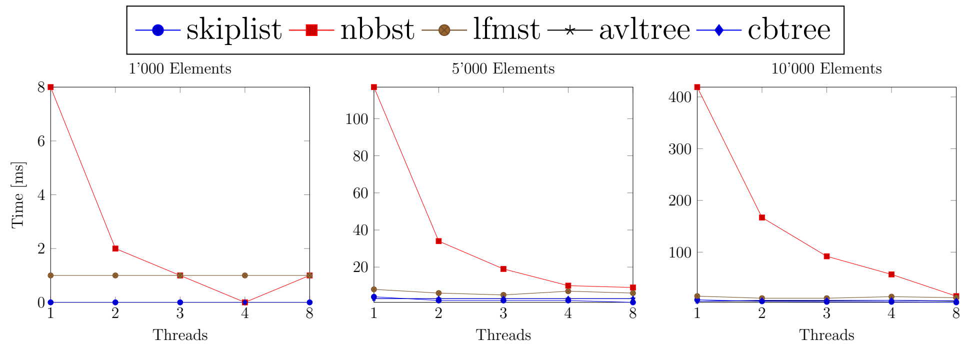

Figure 13: Build performance of the different structures. The elements are inserted in sequential order. The higher is the better.

Just like when testing the search performances of the structure, the `nbbst` is orders of magnitude slower than the other, as it not balanced.

The others were tested on higher sizes to see how they were really performing.

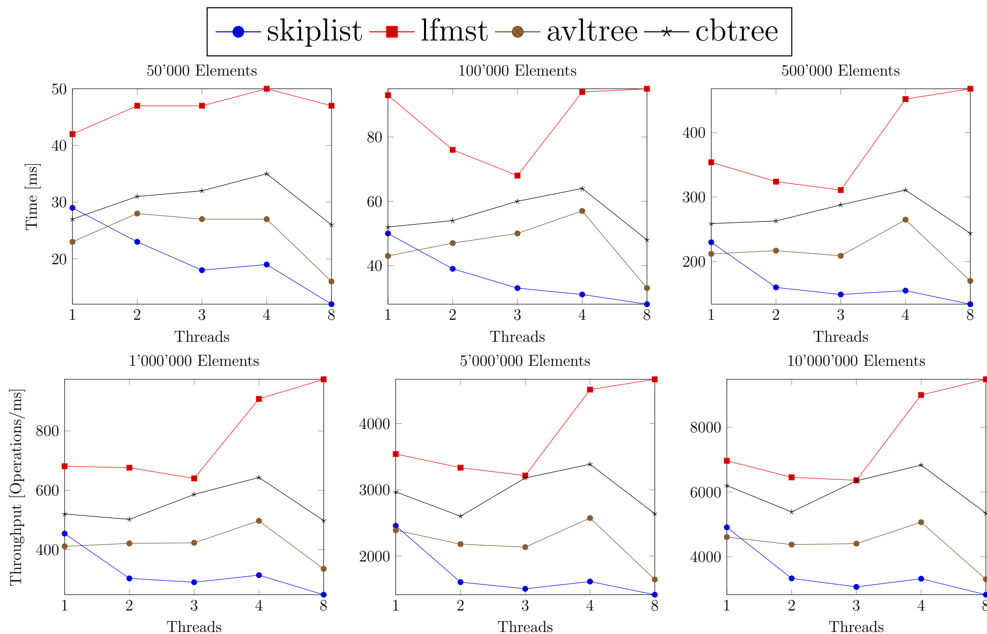

Figure 14: Construction performance of the different structures. The elements are inserted in sequential order. The higher is the better.

Again, the `lfmst` is much slower than the others and appears to have scalability issues with a lot of contention. This time, the `skiplist` performs

very well compared to the other. This is because the other balanced structure have a lot of work to balance themselves in this insertion order. The `skiplist` insertion operations have always the same cost whatever the insertion pattern is.

It is very interesting to note the scalability of the `avltree` and `cbtree`. It is much worse than for a random insertion pattern. It seems that the balancing operations incurs a lot of contention here. It is certainly because the balancing operations need a lot of locks to perform their transformations.

5.4.3 Empty a tree

Finally, we also tested the performances of removing elements from an existing tree. The removal of all elements of trees of 50000, 100000, 500000, 1000000, 5000000 and 10000000 elements was tested.

The tests are separated in two tests:

- In the first test, the elements are inserted in random order in the tree.
- In the second test, the elements are inserted sequentially

In both cases, the elements are removed in the same order as they were inserted.

This separation is made to compare the balancing of each tree.

The size of the tree is divided between the threads. The numbers to remove are divided between the threads in this way: the xth thread removes numbers $[x * (size/Threads), (x + 1) * (size/Threads)]$, where $Threads$ is the number of threads. The test has been made with 1, 2, 3, 4 and 8 threads.

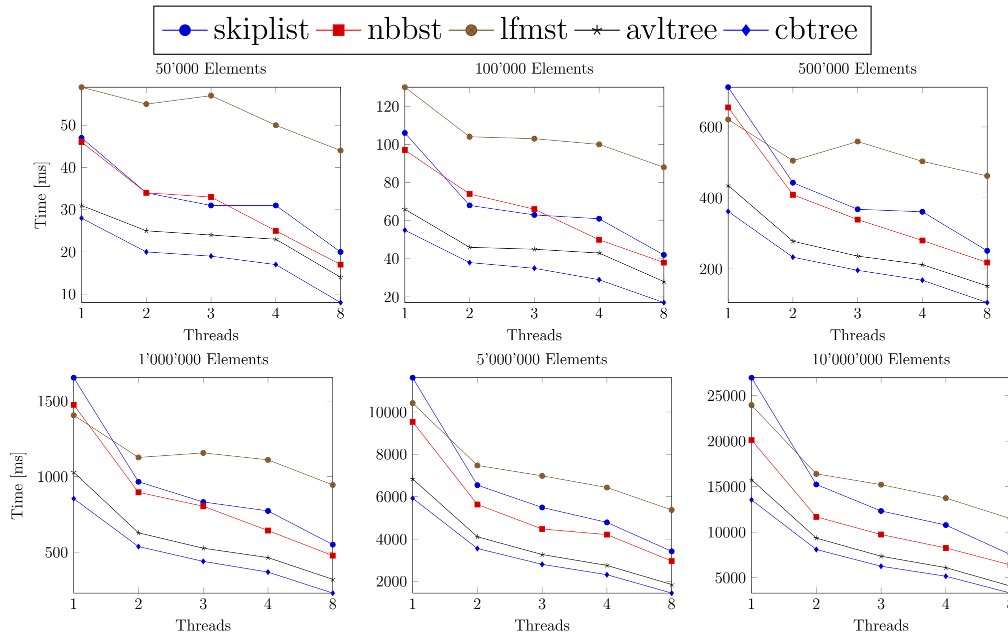

Figure 15: Time to remove elements from the different structures. The elements are inserted and removed in random order. The lower is the better.

It seems that all structures have a similar scalability.

Just like it was the case for insertion, the lfmst is the slowest structure, but this time it is closer than the other and has better scalability, especially for high number of elements.

It is also very interesting to note that the `cbtree` outperforms the `avltree` on removal.

Then, the same test have been performed with numbers sequentially inserted into the structure. The same number of threads is used. Smaller sizes are tested: 1'000, 5'000 and 10'000 elements. The results are presented in Figure 16.

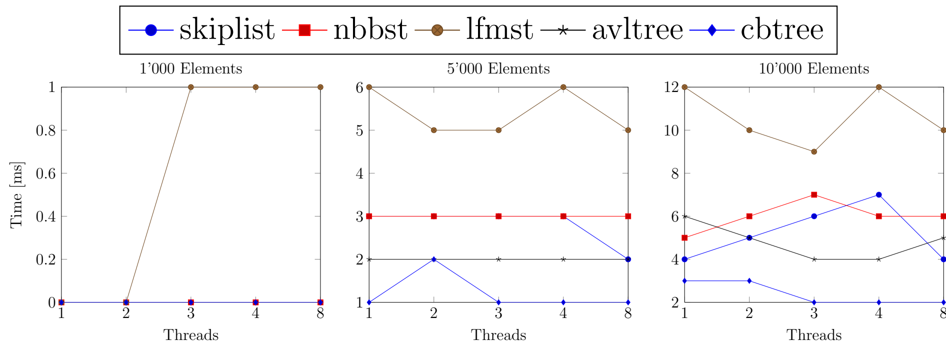

Figure 16: Removal performance of the different structures. The elements are inserted and removed in sequential order. The lower is the better.

Results are not very interesting here. Indeed, the removal is too fast to produce interesting measures on such small sizes. However, the `nbbst` has been removed from the tests with higher size because it would be too long to build the tree. The others were tested on higher sizes to see how they were really performing.

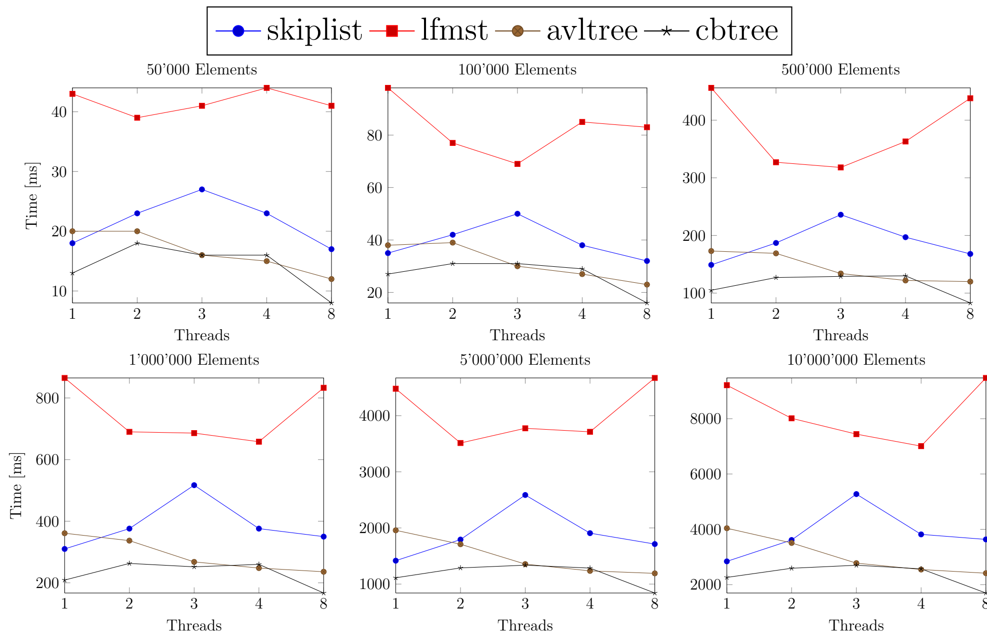

Figure 17: Removal performance of the different structures. The elements are inserted in sequential order. The lower is the better.

Again, the lfmst is not performing very well and appears to have scalability issues. It is four times slower than the best one.

The skiplist list also appears to have scalability issues here for an unknown reason.

The cbtree is performing better than the avltree, but they are close. Again, on these two structures, the scalability is not very good due to the high cost of the rebalancing that as to performing during removing elements from the tree.

5.5 Memory consumption

The memory consumption of each data structure has also been tested. To get the memory usage, Memory Allocation Hooks[GnuHooks] have been used. The C++ standard library allow to place a hook that will be called in place of one of the allocation functions (malloc, free or realloc). In this case, hooks for malloc and free have been used. Every time the malloc hook gets called, it counts the allocated size. When the free hook is called is remove the size of the freed memory from the allocation counter. There are no standard way to get the size of memory pointed by a pointer. For that, the pointers returned by malloc and their size are stored in a map to be retrieved in the free hook. Most of the structures are not freeing any memory before their destruction, but some of them are using allocations. For example, the Multiway Search Tree is using dynamic allocations to store temporary array so that memory should not be counted as it is released.

Using a hook is quite straightforward. At first, the hook has to restore the base hook and then call the real function (malloc or free) and finally, the hook should be restored.

Before the construction of the tree, the allocation counter is set to zero and its value is taken at the end of the construction to see the amount of memory needed by the structure. With this technique it is necessary to allocate the tree itself on the heap to count its size. Indeed, the allocations on the stack do not use malloc.

For each size, every number in $[0, size]$ are inserted in random order into the tree.

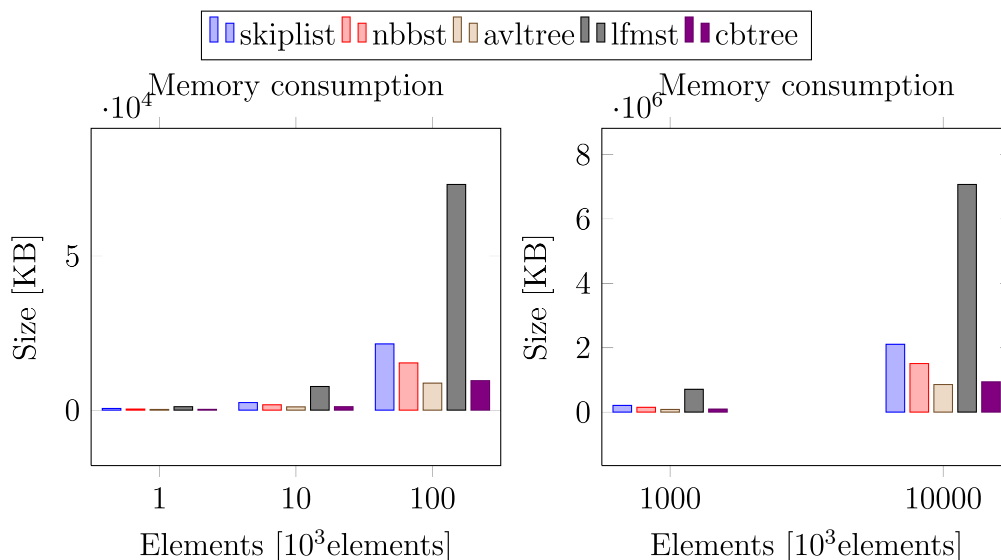

Figure 18: Memory consumption of the different structures on full range $[0, size]$

The `lfmst` very high memory usage is due to several things. First, a lot of nodes are not reused. Indeed, most of them are stored and then released at the end due to some problem with the implementation of Hazard Pointers (See Section 3.8). Moreover, the storage of these nodes does also need memory as well.

The differences are not negligible. The SkipList memory usage is more than twice higher than the usage of the optimistic AVL Tree. This can be explained by the number of links stored to each node that his way higher than in a real tree. Indeed, in the chosen implementation, each node stores the maximum number of forward pointers to simplify the algorithm. This has a huge cost on the memory usage. If only the needed pointers were created, the memory usage would be much smaller.

The difference between the Non-Blocking Binary Search Tree and the Optimistic AVL Tree and the Counter-Based Tree can be explained by the fast that the first is leaf-oriented (all the data are stored in the leaves) and not the others twos. The last two are very similar. The difference is due to the storage of three counters in each node for the Counter-Based Tree compared to only one height counter in the first.

Tests were also made on the whole range of integers ($[0, 2^{32}]$). Random numbers are inserted into the tree until the tree has the target size. This test

is made to test the impact of routing nodes. Indeed, when all the numbers of a given range are inserted, it is highly probable that there will be very few routing nodes necessary. There should be more when the numbers are not sequential.

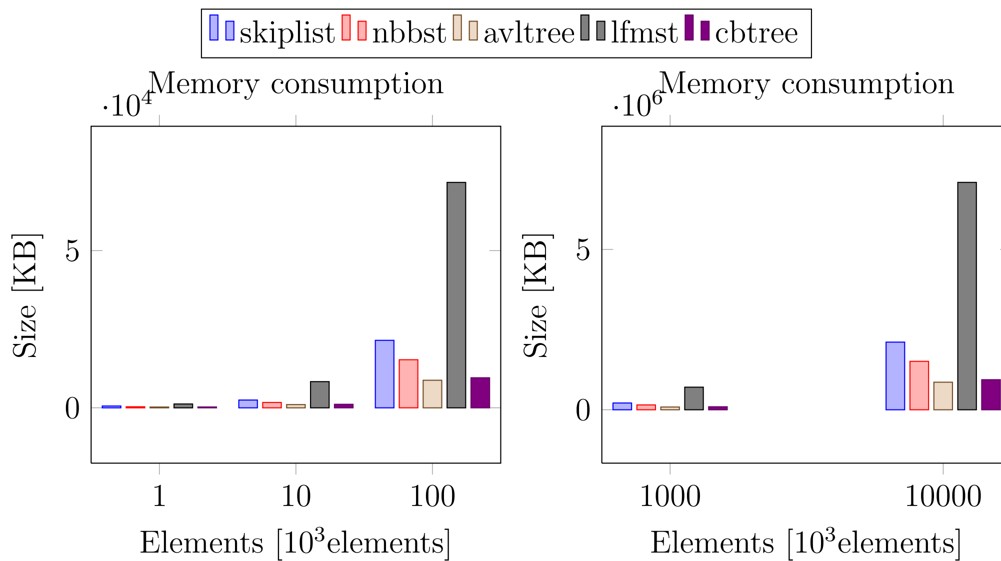

Figure 19: Memory consumption of the different structures on range $[0, 2^{32}]$

The results of this benchmark are presented in Figure 19.

There are no big differences between these results and the previous one. It seems that not too many routing nodes are necessary even when the numbers are distributed in a high range. There are some very small differences in the numbers, but it is negligible.

6 Problems

This section contains the main problems encountered during this project.

6.1 Lack of complete code in the papers

Except for the Non-Blocking Binary Search Tree paper, all the other papers do not contains complete or even working code for an implementation.

For example, none of the papers was containing the code for balancing operations.

The problematic implementations are provided in online source code repository. Nevertheless, the code in the online repository was not the same as the code in the paper, so it was not possible to use parts of the code in the papers.

6.2 Differences between Java and C++

All the implementations compared in this paper are implemented in Java by their authors. The main reason for that is because Java has a Garbage Collector and so, makes memory reclamation trivial to implement. In our implementations, this was solved using Hazard Pointers (see Section 2.8). But using this technique does not make all trivial, it is still up to the developer to indicate where a node is no longer needed and where a reference to a node must be published.

Some structures were using some of the Java Concurrent Library features. By chance, most of them are now available in the C++11 concurrent library. Like that atomic references or the compare-and-set operations. Some of the structures are adding some information to a reference. For that they use an `AtomicMarkableReference` instance. There are no such class in C++. It is possible to make something even more efficient in C++ by using bits of the address itself to store the information. With that technique, it is possible to edit atomically both the reference and the information associated with the reference. It is even possible to store several bits (the available number of bits depends on the addressing mode).

There are another differences that made the transformation harder. First, the Java arrays are storing their length in a field unlike C++ arrays. C++ vector could have been used for that, but it would have added a little overhead, so the length has been stored aside the elements. This has especially been

one of the main challenge for implementing the Multiway Search Tree (see Section 2.5).

6.3 Slow Zipf distribution

The Zipf's power law has been used to implement a skewed distribution. The first implementation that was realized was based on the Zipf distribution by Ken Christensen [ChristensenTools]. The problem with this implementation was that generating a new random number was very slow. Indeed, generating a new number have a complexity of $O(N)$ where N is the range of generated numbers. As each thread generates a lot of these numbers (potentially several million), this was slowing down the benchmark, making the measures useless.

To fix this problem, another implementation was used, the one from mkrn of the Boost Supplement library [BoostSupplement]. This implementation uses a discrete distribution as the probabilities. This implementation has a bad construction complexity $O(N^2)$ but the complexity of generating a new random numbers has a time complexity of $O(1)$. The distribution is generated once for each bench so the construction time is quickly amortized. As it is very fast to generate new random numbers, this has no effect on the quality of the measures.

The implementation has been rewritten to use the random features of C++11 instead of Boost. The unnecessary code of the discrete distribution has been removed.

Even if this last implementation was fast to generate numbers it was impossible to use it on a high range (more than one million), so we switched again to the generator of Ken Christensen. But this time, the results are saved to a file and then loaded in memory at the beginning of the benchmark and then used like this. The generator has been converted to C++ and optimized a bit to fit our specific needs.

7 Tools

Not many tools were used for this project. vim has been used as the main editor to develop the whole project.

The project uses CMake[cmake] to build. CMake is not really a build automation tool, it is a generator for other tools. In this case, it is used as a makefile generator. So, CMake has to be launched first to create the makefiles and then *make* is used to build the project. The advantage is that the CMake configuration file is easier and more expressive than a makefile. Moreover, CMake manages the dependencies between the files, so that the generated makefile is very complete. Finally, it also has a portability advantage as CMake can also generate Microsoft Visual C++ project files, so you can build it on Windows as well.

Although there is no outstanding editor, there are a lot of command-line tools that can really help a C++ developer to produce robust applications. In this project, we have used several of these command-line tools. We used cppcheck[cppcheck] as a static analyzer. This open source application checks your code to find allocation errors, variable issues, members that can be *const* and other defects.

More than being helpful to test our Hazard Pointers implementations, Valgrind[valgrind] has also been used to check for memory problems in the application itself. We have corrected some memory leaks in the application and some problems about uninitialized values.

GCC 4.7.0 has been used to compile each source file of the projects. At the time of writing, this is the last version available. This version has been chosen because it supports most of the C++11 standard that has been used widely in this project. Moreover, this version contains new optimizations that are always interesting.

8 Final comparisons

This section describes the strength and weaknesses of each data structures.

8.1 Skip List

- - Big memory footprint for high number of elements
- - Bad performance
- - Medium scalability
- + Simple balancing scheme, enough to support sequential order
- + Very simple

The skip list is only a derivative mean to implement a concurrent binary search tree and because of that is not very well performing compared to real concurrent BST. However, its performance are not that are and are very stable. It supports bad insertion pattern as well as random pattern.

This structure should only be used when no balanced concurrent BST implement is available. Indeed, the other balanced structures outperforms it in all situations.

8.2 Non Blocking Binary Search Tree

- + Very good performances under random access pattern
- - Not balanced: Not adapted to sequential access pattern or others patterns leading to bad balancing
- - Not optimal memory footprint as the tree is external
- + Very simple

This structure is the only one that is not balanced. So, when the access pattern is known to be completely random this data structure is a very good candidate. Indeed, the operations are very simple and the scalability is very good under this condition, so this tree is one of the best. However, when the access patter is not known or is known to be kind of sequential it a very bad choice as its performances will be very low. For huge trees with worse insertion pattern, it is not usable.

8.3 Optimistic AVL Tree

+ Very high performances
+ Very powerful balancing
+ Very good scalability
+ Low memory footprint

This data structure has very good performances in all scenarios. It is the best candidate for every usage. The results of the benchmarks have shown that its the best under almost situations, except pure search performances where the Lock-Free Multiway outperforms it a by a little margin.

8.4 Lock-Free Multiway Search Tree

- High contention on the insert and remove operations
+ Very good search performances
- Very complex
- Lots of references to manage
- High memory usage for management of nodes

This data structure is not performing very well in this benchmark. Indeed, for most of the benchmarks it is just a bit faster than the skiplist sometimes even slower. This comes from the very high overhead of Hazard Pointers in its case and the different mechanisms that have been used to handle memory correctly. However, it is also the structure that has the best lookup performances of the tested ones. If the insertion phase does not need to be very fast and can be performed with few threads and only the search performance matters, this tree can be a very good choice.

If this structure should be used in practice, it should be better to improve its node management to be more like the others, avoiding the overhead of temporary storage of nodes that are not known to be completely free.

8.5 Self Adjusting Search Tree

+ Low memory footprint
- The frequency balancing scheme does not keep its promises
+ Good scalability
+ Good average performances

This structure should have performed very well under skewed distribution, but it has not kept its promises. However, its performances are comparable to the ones of the AVL Tree, but always about 25% slower. It is generally a better candidate than the other trees, but it has no advantage over the Optimistic AVL Tree and so should never be preferred to it.

9 Conclusion

To conclude, it has been a very interesting and challenging project. It has been more difficult that I thought to adapt some structures with Hazard Pointers. Indeed, when the structure relies heavily on dynamic allocations like the Lock Free Multiway Search Tree and is very complex, it is hard to adapt it. This is especially when the person translating the structure to another language is not the source structure's author.

I have been surprised by the diversity of the different solutions. Indeed, I thought I would have to implement data structures that were all deviations of simple Binary Trees. I did not think there were powerful structures like Multiway Search Trees and Skip Lists that can be used for the same problem without being trees at all. The richness of the balancing schemes are also quite interesting to study. Moreover, I thought it was quite ingenious to use transactional memory techniques like optimistic concurrency control to improve the efficiency on a tree without requiring too many locks.

At first I was thinking that a structure with several locks like the AVL Tree would not have good performances, but I was wrong. Indeed, with the use of Optimistic Concurrency Control, this tree is performing very well in all situations and can easily handle very large trees.

The results have also been interesting to study. Indeed, the results are not always what I expected. For example, after having implemented the Lock-Free Multiway Search Tree and having found that it was very heavy to adapt to Hazard Pointers, I was surprised to see it so powerful for lookup operations. Its random balancing scheme and the use of lots of references made it very convenient to search even for bad insertion pattern.

At first, all the benchmarks I implemented were random-based. With that, the results were showing the Non-Blocking Binary Search Tree in very good position. After quite a while, I thought about balancing and added new benchmarks with worse insertion pattern and immediately saw that this unbalanced structure cannot be used for non-random pattern.

I was very surprised with the results of the Counter-Based Tree. Even if its frequency-based balancing scheme made it slower than the Optimistic AVL Tree for random and sequential patterns, it should have made it the most efficient under skewed distribution. Indeed, even if the difference between the two trees goes smaller when the skew increases, the Counter-Based Tree should have been before the AVL Tree passed a skew of about 1.0. I did not find where that problem does come from. Perhaps it comes from the C++ implementation, but it can also come from the specification itself that is not

as powerful as expected.

I have been disappointed with the adaptation of the Lock-Free Multiway Search Tree to C++ and to Hazard Pointers especially. I am convinced that this structure can perform very well even with Hazard Pointers, but that would require some work on the structure itself, which is out of the scope of this project.

The given implementations are not bug-free. Indeed, during the tests, some bugs are still popping out. I was not able to fix all of them. It is very difficult to track all these bugs. The first reason is that they never arise when the program is compiled in debug mode, as it too slow, only with full optimizations. It is much harder to find a bug once a program has been fully optimized by the compiler than when it is full of debug symbols. The second reason is that some of the versions are very complex and the code base very big. As I do not have a strong understanding of the underlying algorithms, it is quite hard to know where does the bug come from. In my opinion, the bugs come from my translation to C++ and adaptation for the use of Hazard Pointers, but it is also possible that the specification implementation itself is not exempt of bugs. As the bugs are just some possible interleaving issues that arises when several threads are contending for the same resource, it does not change the meaning of the results and thus the conclusions of the benchmark.

9.1 Future work

An interesting test would be to benchmark the different structures using a faster allocator. Indeed, the standard allocation system of the C library (`malloc()`) is not very efficient under multi-threading condition. There exists several thread-safe and very efficient implements of malloc freely available online. It is highly probable that this will reduce the penalty of a structure that made a lot of dynamic memory allocation like the Lock-Free Multiway Search Tree. Moreover, it will also show what structures are the most sensitive to the efficiency of the underlying malloc implementation.

Another valuable test would have been to benchmark the performances of iterating through the whole tree. This would need a lot of work as not all the data structure have functions to do that, but that would be interesting in cases where BST are used only to keep a sorted set of data.

As the skewed benchmark results are not really concluding, it could be interesting to implement a second skewed benchmark using another distribution. It could also be worth trying to investigate the Counter-Based Tree imple-

mentation why it did not perform very well under skewed distribution.

The adaptation of the Lock-Free Multiway Search Tree is not optimal at all. It could be valuable to review its implementation to be more compliant with the others implementation. That would require to rethink some of the algorithms of this structure. This is especially true for the node removal techniques used in this structure. Indeed, this has been the biggest problem of this tree to manage nodes.

If our implementations of the trees presented in this paper are to be released publicly, it would require some work to make them widely usable. Indeed, the implementations have some flaws for general usage:

- The number of threads is fixed at compile-time for each data structure. It is very convenient for a benchmark, but this is not the case for general usage of a data structure. Allowing any number of threads would require to use linked lists to store the Hazard Pointers for each thread and would also require some work to support threads to die (releasing nodes from the linked lists of hazard pointers). The CBTree implementation would also need to be rewritten using linked lists for thread counter or a thread local store for the thread counters.
- The memory is never released back to the system. Again, it is convenient for our use in the benchmark, but in real use, depending on the cases, it can be useful to release the free memory to the system. This change would not be hard to make. Indeed, when the released nodes are searched for free nodes in the HazardManager, it is possible to free the nodes if a limit has been reached. That would make the HazardManager highly configurable.

More than flaws, the implementations are also lacking some useful features:

- They are all implemented in term of sets, but Binary Search Tree are often used as associate containers (maps). This change would incur to store the value associated with each key. As each of the data structures tested in this paper are defined in terms of maps, this change does not come as a big cost.
- It is not configurable whether they can hold duplicates. This change can somehow complicate the code if we want to keep it configurable (a template parameter or a constructor parameter would be the best choice). Even if it is simple to add this possibility to a Skip List it is not the case for each data structure.

- There is no way to iterate through the elements of the tree. That would require some adaptations as the iteration can be done using depth-first or depth-first algorithms. A challenge for adding this feature would be to support removal of nodes through the path that is currently iterated.

- There is no way to clear the entire tree. At the present time, the only way to remove all the elements is to keep a track of the inserted elements to remove them one by one. Once there is a concurrent function to iterate through the tree, it is not difficult to remove all the elements. Even a not thread safe clear function would be highly interesting as it will allow to empty the tree to avoid memory leaks.

- If the tree object gets destructed when some elements are still present in the tree, there is a memory leak. Again, if there is a function to remove all the elements, it could be called from the destructor.

- All the configuration parameters of the data structures are fixed at compile-time. For example, the MAX_LEVEL property of the Skip List is defined as a macro. When the user knows the maximum number of elements that will be used in the tree, it could adjust this parameter to a smaller or higher value than a default. For example, if the user knows that its tree will hold as much ten thousand elements, it can set the parameter to 14 instead of the 24 by default (support more than sixteen millions of elements) resulting in a much smaller memory footprint.

If someone would like to reuse our implementation of Hazard Pointers, there, again, should be some work to perform on it:

- Like the data structures, the number of threads is fixed at compile-time.

- The HazardManager class should accept a number of hazard pointers equals to 0. In this case, it could be used as an object pool.

- Each reference is indexed with an integer. In general, it is the more efficient way to store hazard pointer, but when the data structure becomes complex, it starts to be difficult to set the good index. If a function is recursive, it is not even possible to use fixed index. This can be fixed by using a thread-local index for the current reference. This index is incremented after each publication of a reference and decremented when the reference is released.

9.2 What I learned

This project was a very good opportunity to deeply study Binary Search Tree implementations. Indeed, before this project, I only implemented very simple Binary Search Tree structure with simple balancing scheme. In this paper, the studied trees are very different.

I also learned a lot about concurrent ways to implement a Binary Search Tree. Before this project, I did not know most of the techniques that are used here like Skip List, AVL Tree or Skip Tree. Moreover, it was also a good way to update my knowledge about BST themselves.

Of course, this has been an occasion to study concurrency as well. The best way to learn more about concurrency is always by practicing as this project showed me.

I also learned how to implement Hazard Pointers in practice. In simple data structures like skip list or the Non-Blocking Binary Search Tree, it is a very convenient technique. On some more complex structures like The Multiway Search Tree, it is not very adapted. A data structure should be designed from the beginning to handle memory efficiently without memory leaks.

In this project, a lot of features from C++11 were used. For example, the new chrono library is used to calculate the duration between two points of the code. The new random library is used to generate all the random numbers and distributions used in the benchmark code and in the implementations as well. The thread library is used to manipulate all the threads of the application. More than the new libraries, new language features were also used: the lambda expressions (to make easier the creation of threads), the foreach loop and the initialization list.

A Code

The C++ implementation of the different structures, the Hazard Pointers implementation as well as the benchmark and the tests are available on `https://github.com/wichtounet/btrees/`.

B Content of the archive

In appendices of this document, you will find an archive containing the following elements:

- This document in PDF format: report.pdf
- The logbook in PDF format: logbook.pdf
- *sources*: The sources of every document
  - *report*: The LaTeX sources of this document
  - *logbook*: The LaTeX sources of the logbook
- *btress*: The C++ sources of the application
  - *src*: The C++ source files
  - *include*: The C++ header files

References

[Afek2012] Yehuda Afek. Cbtree: A practical concurrent self-adjusting search tree. 2012.

[BoostSupplement] mrkn. Boost supplement library. http://coderepos.org/. Accessed: 03/04/2012.

[Bronson2010] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. *SIGPLAN Not.*, 45:257–268, January 2010.

[BronsonRepository] nbronson. N. bronson snaptree repository. https://github.com/nbronson/snaptree. Accessed: 30/04/2012.

[ChristensenTools] Ken Christensen. Christensen tools page. http://www.cse.usf.edu/˜christen/tools/toolpage.html. Accessed: 28/03/2012.

[DougLeaQuote] Doug Lea. Concurrentskiplistmap.java. http://www.docjar.com/html/api/java/util/concurrent/ConcurrentSkipListMap.java.html. Accessed: 11/05/2012.

[Ellen2010] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '10, pages 131–140, New York, NY, USA, 2010. ACM.

[GnuHooks] mrkn. Gnu c library memory allocation hooks. http://www.gnu.org/software/libc/manual/html_node/Hooks-for-Malloc.html. Accessed: 03/04/2012.

[Herlihy2006] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A provably correct scalable skiplist (brief announcement). In *Proceedings of the 10th International Conference On Principles Of Distributed Systems*, OPODIS 2006, 2006.

[Herlihy2008] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[Matsumoto1998] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, January 1998.

[Messeguer1997] Xavier Messeguer. Skip trees, an alternative data structure to skip lists in a concurrent approach. *ITA*, 31(3):251–269, 1997.

[Michael04] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004.

[RedBlackTree] Wolfram. Red-black tree. http://mathworld.wolfram.com/Red-BlackTree.html. Accessed: 25/05/2012.

[Sedgewick1984] Robert Sedgewick. *Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984.

[Spiegel2010] Michael Spiegel and Paul F. Reynolds, Jr. Lock-free multiway search trees. In *Proceedings of the 39th Annual International Conference on Parallel Processing (ICPP)*. IEEE Computer Society, 2010.

[SpiegelRepository] nbronson. M. spiegel skiptree repository. https://github.com/mspiegel/lockfreeskiptree/. Accessed: 30/04/2012.

[cmake] Cmake : the cross-platform, open-source build system. http://www.cmake.org.

[cppcheck] cppcheck: A tool for static c/c++ code analysis. http://cppcheck.sourceforge.net.

[valgrind] Valgrind: A suite of tools for debugging and profiling. http://valgrind.org.

[zipf1949human] G.K. Zipf. *Human behavior and the principle of least effort: an introduction to human ecology*. Addison-Wesley Press, 1949.

Index

ABA, 12, 13
AVL Tree, 1, 3, 7, 21, 26, 38, 55–57

balancing, 5, 8, 9, 11, 22, 25, 26, 34, 38, 39, 41, 42, 45, 48, 51, 57

C++, 1, 3, 19–21, 23, 38, 51, 58, 62
C++11, 14, 31, 61
Compare-And-Swap, 6, 12, 15, 19, 20, 23, 24, 26
complexity, 5

GCC, 15, 31, 53
geometric distribution, 11, 19

Hazard Pointers, 1, 3, 12, 13, 16, 19, 21, 23, 25, 53, 61

Java, 3, 5, 18, 21–23, 25, 51

Mersenne Twister, 15, 29

Zipf, 36, 52, 57

Glossary

C++11 The last standard for the C++ programming language. . 14

Garbage Collector A process that keeps tracks of allocated memory and know when the memory can be safely returned to the system. . 12

hash function an algorithm that maps large data sets to smaller data sets of a fixed length. 4

linearizable The results of the operations are the same as is they have been performed sequentially, in the order of their linearization points. 6

non-blocking An operation always completes, starting from any configurations, with any number of failures. 6

Optimistic Concurrency Control Concurrency control method that assumes that multiple transactions can complete without affecting each other. . 8, 9, 57

wait-free Every operation has a bound on the number of steps the algorithm will take before the operation completes. 13

Listings

1 Update struct 21
2 AVLTree Node structure 23
3 CBTree Node structure 27

List of Figures

1 Example of Skip List 7
2 Data structure of the Non-Blocking Binary Search Tree 8
3 Validation of the AVL Tree 9
4 Multiway Search Tree with optimal references 11
5 Results of the random distribution benchmark 35
6 Results of the random benchmark with high range 36
7 Zipf distributions histograms with different skews 38
8 Results of the skewed benchmark 39
9 Results of the search benchmark, random order 41
10 Results of the search benchmark, sequential order 42
11 Results of the search benchmark, random order (Continued) . 43
12 Results of the construction benchmark, random order 44
13 Results of the construction benchmark, sequential order 45
14 Results of the search benchmark, sequential order (Continued) 45
15 Results of the removal benchmark, random order 47
16 Results of the removal benchmark, sequential order 48
17 Results of the removal benchmark, sequential order (Continued) 48
18 Memory consumption 50
19 Memory consumption (Continued) 51