CHALMERS

A Study of Concurrent Data Structures

Master of Science Thesis

in the Programme Networks and Distributed Systems

BO LI

Department of Computer Science and Engineering

CHALMERS UNIVERSITY OF TECHNOLOGY

Göteborg, Sweden, May 2013

A Study of Concurrent Data Structures

Bo Li

Supervisor & Examiner: Philippas Tsigas

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

[Cover:
]

Department of Computer Science and Engineering
Göteborg, Sweden, May 2013

ACKNOWLEDGEMENTS

I am very grateful for all the support and advices I have received. This project could not have been realized were it not for the help and support of a great many people.

First of all I would express my thanks to Professor Philippas Tsigas and the Department of Computer Science of Engineering, particularly my project advisor, Bapi Chatterjee for his guidance throughout the project and his valuable feedbacks and support.

I would especially like to thank my wife Stella Zhang, who helped me out of many difficulties in life and provided me with warm encouragement.

Finally I would like to dedicate this project to my family who support and understand me in my overseas study.

ABSTRACT

This Master thesis studies four concurrent data structures but emphasizes on two concurrent tree data structures, in particular concurrent search trees. We have studied two concurrent search trees - concurrent AVL tree and concurrent counting-based tree (CBTree) and two concurrent queues - lock-free concurrent queue and two-lock concurrent queue. We implemented two variants of concurrent CBTree as well as the two concurrent queues.

The optimistic concurrency control mechanism used in the concurrent tree data structures is called hand-over-hand optimistic validation.

We further evaluated the implementations of the data structures coded in Java. The evaluations we done on an Intel workstation with Linux platform running 24 hardware threads.

Furthermore, the advantages and drawbacks of these data structures are analyzed. Our study shows that CBTree should be implemented with some special mechanisms to achieve a better performance. Thus, this thesis achieves to present important explorations towards better implementation of concurrent search trees.

Keywords: *Concurrent data structure*, *Concurrent Queue*, *Concurrent AVL tree*, *Optimistic hand-over-hand validation*, *CBTree*, *Single adjuster*

CONTENTS

1. INTRODUCTION

1. INTRODUCTION

With multicore computers being widely available and unicore computers disappearing, multithreaded programs are not just an option but the necessity. When we try to use multithreading in conventional data structure, it essentially leads us to concurrency among threads and hence the data structures using multithreading and synchronization among threads become concurrent data structures.

However, the concurrency makes the algorithm of data structure more complex and at times less efficient. With multicore processors being norm of the day, even hand held devices like smart phones being shipped with multicore processors, it is being increasing necessary to design and improve efficiency of algorithms of concurrent data structures.

The objective of this master project is to study and implement concurrent data structures and the comparison of the performance of them in order to indicate the advantages and drawbacks. Prior to the discussion of data structures, the basic theory and knowledge of concurrent programing have been studied and presented in Chapter 2. We have studied concurrent blocking and lock-free queues and this thesis carries a brief report of that in Chapter 3, with details in the report of the minor thesis project finished earlier. This thesis emphasizes on two useful concurrent search trees - (a) concurrent AVL tree [1] and (b) concurrent CBTree [2].

Concurrent AVL tree [1] is one of the concurrent data structures studied in this thesis. It uses an Optimistic Concurrency Control (OCC) scheme that uses version numbers in order to avoid the conflicts among threads that make structural changes on the tree. In the formation of this tree structure, hand-over-hand optimistic validation is introduced for correctly searching, and partially external trees for deleting internal nodes as well. All these schemes are used for managing concurrency in a relaxed balanced AVL tree (Chapter 4).

Afek's counting-based Tree, CBTree for short [2], is another tree data structure studied in this thesis. CBTree is a self-adjusting binary search tree which is derived from Sleator and Tarjan's seminal splay tree [3] . Similar to splay tree, CBTree moves accessed nodes towards the root of the tree, but with different set of rules for adjusting the structure. It gives a property of "weight" to each node, which indicates the number of accesses to the node's subtree, and the rotation of CBTree based on some algorithmic calculations of "weight" values

between several nodes. In addition, CBTree uses the same mechanism as concurrent AVL tree to manage concurrency. An alternative rule for rotations has also been described and implemented in this thesis with experiments. The description of these algorithms is presented in Chapter 4.

In Chapter 5, experimental analysis of both of these two concurrent tree data structures with operations on random data sets is presented. It shows that under certain conditions, CBTree outperforms the concurrent AVL tree because it has a better path length. In addition, the performance and comparison of the concurrent blocking queue and lock-free queue is presented in this Chapter.

Finally, the conclusion and future work of this thesis is presented in Chapter 6.

2. CONCURRENT PROGRAMING

2. CONCURRENT PROGRAMING

Moore's law has been around for almost 50 years, and it has not become invalid yet. Although a lot of transistors are packed in a chip, their clock speed can not be increased without overheating [4]. Because of that, multiprocessor and multicore architectures are developed in order to speed up modern computers. Today's computers are able to handle various operations at the same time with effective performances. With the rising of these technologies, parallelism of hardware and concurrency of program are developed. Both of them can make threads run simultaneously though safely, either in physical cores or in logical cores.

2.1 SEQUENTIAL AND PARALLEL PROGRAMING

Process, which is also called a CPU activity [5], is a program's basic entity that can be executed in a computer. Moreover, a thread, which is contained inside a process is the smallest sequence of instructions that operating system can execute. A program usually contains several processes and a process usually includes one or more threads. A program in sequential fashion means each of its processes is with a single thread of control, and these processes are executed one after another. This kind of program can only perform one task by CPU at a time. Figure 1 shows that how processes are handled by the time in a sequential programing fashion.

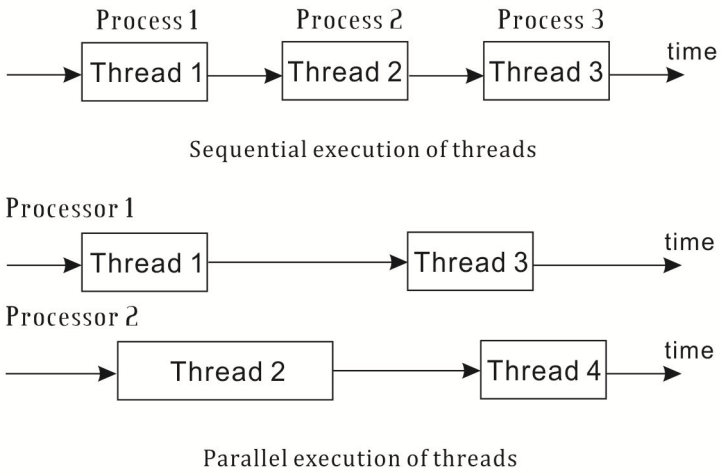


Figure 1. Sequential and parallel programing fashion

Comparing with processes with single thread of control, nowadays almost every process in computers contains multiple threads, and they share the resources that the process can access, like the main memory. On a multiprocessor or a multicore architecture, multiple threads in a process can be executed at the same time, which is called the parallel execution. This type of architecture allows a CPU to handle more than one task of a program at a certain point of time. For example, when a user is browsing a web site, the browser can not only retrieve graphs from the Internet, but also store the data like cookies of this web site at the same time. It significantly improves the efficiency of CPU and the performance of the whole operating system. On the other hand, it also brings the difficulties of programming on such architectures. Wasting of resources could happen when there are inefficient codes in the program. The difference between parallel fashion and sequential fashion of programing are also presented in Figure 1 .

2.2 CONCURRENT PROGRAMING

When multiple software threads running at different hardware, threads (physical/logical) simultaneously try to access same memory location or other shared resources, it leads to concurrency. However, different from a program running in parallel way, concurrency in a single processor system is only "logically" achieved. Single CPU does not really execute multiple threads simultaneously. It just switches its occupation in different threads of the process to make it look like they are executed at the same time (Figure 2). While one of the threads in a process is executing, other threads can be hung up and blocked to access the shared resources until the current thread finishes. This mechanism reveals that the pure concurrent programming pattern is achieved by programing, not hardware. Obviously, running a concurrent program on an architecture of multicore or multiprocessor that has parallelism in hardware gives a better efficiency of CPU and also a better performance of the program.

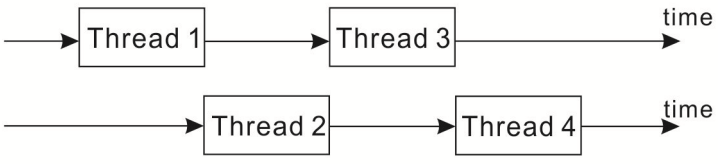


Figure 2. Concurrent execution of threads

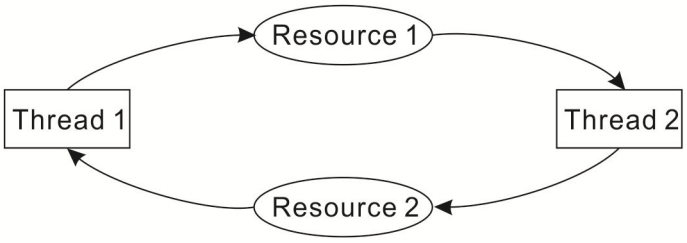
Concurrency leads to the problem of conflict among threads to access shared resources. This conflict needs to be resolved and hence we need a particular synchronization scheme. Following are the widely used ones.

Blocking techniques

In concurrent programing, multiple threads access shared resources. However, it is impossible that when one thread is using the resource, another thread can also access it. A general method to prevent such a situation is blocking other threads when the resource has already been occupied. Mutual exclusion [6] is typically implemented by means of locks that protect critical sections of codes [7], where the “critical sections” indicate that threads need to access a shared resource.

Several primitives of synchronization are built based on mutual exclusion, like semaphore [8] and monitor [9]. In addition, there are various locks techniques, using which simple lock based concurrent data structures are built. Michael and Scott [10] presented a two-lock queue algorithm for concurrent enqueue and dequeue. Bronson et al.’s [1] concurrent AVL tree, also blocks other threads changing the tree structure by using the version numbers mechanism.

When system implements synchronization with blocking, some exceptions might always be there. Deadlock and livelock (Figure 3) are two typical ones. A deadlock happens if one thread (T1) tries to access a resource occupied by another thread (T2), when T2 is also waiting to access another resource that is held by T1. Both of T1 and T2 only release the resource they are occupying when they successfully access the objective resource. It makes T1 and T2 getting stuck so that they can not make any progress. Whereas, a livelock is a mutation of a deadlock. It makes the threads with regard to each other asking for access to a shared resource infinitely; however, they are not stuck by each other. Both of the threads make no progress under such situation. Such failures reduce the reliabilities of a system.



Deadlock: Thread 1 is waiting to access Resource 1, which is occupied by Thread 2, and Thread 2 is waiting to access Resource 2, which is occupied by Thread 1. Both the threads do not release the occupied resource until they get the accesses. None of them make any progresses.

Livelock: both of the threads are asking for the resource but neither of them can achieve it.

Figure 3. Deadlock and livelock

Designers have to pay attention to these exceptions when they are using blocking mechanism to develop a program. It is easy to design, and runs usually well. In that case, non-blocking method is an alternative.

Non-blocking techniques

In order to overcome various problems associated with blocking mechanism, non-blocking techniques are developed. Threads executing a non-blocking algorithm are allowed to make progress instead of stopping them in order to simultaneously access the shared resources. Lock-freedom and wait-freedom are two conditions that guarantee one or more threads to make progress in the operations.

Lock-free mechanism does not block any thread in its operation. Although individual thread may starve, lock-freedom pays attention on the global throughputs. It guarantees at least one non-faulty thread to make progress in a sufficiently long though finite unit of running time.

Compare-and-swap (CAS) (Figure 4), is a synchronization primitive used in building lock-free data structures a lot. There are three parameters in CAS: an address of memory, an expected value and a new value. The expected value is

compared with the current value of the address. If they are equal, then the new value is written to the location [11].

```
CAS(x, old, new) /*compare-and-swap*/
    ⟨ if (x == old) {x ← new; return (true);}
      else return (false); ⟩
```

Figure 4. CAS primitive.

Trieber [12] represents the stack that is implemented by linked list and the using of CAS primitive [13]. There is also a lock-free queue based on CAS introduced by Michael and Scott [10]. This algorithm gives two pointers at the head and the tail of the queue respectively, and keeps a dummy node in front of the queue. It uses CAS to update both of the two pointers. Meanwhile, a helping technique [14] is used for keeping the tail pointer moving to the end of the queue by the remaining threads, which keeps other threads to make system-wide progress. This concurrent queue will be discussed in Chapter 3.

Wait-freedom guarantees progress of every non-faulty thread in a finite number of steps of operations. Consequently, it is difficult to design and implement in practical. Kogan and Petrank [15] firstly presented a practical wait-free queue by expanding Michael and Scott's [10] lock-free queue with CAS primitive. The discussion of wait-freedom is limited in this thesis project. Follow-up studies about algorithms of wait-free data structures is possible.

3. CONCURRENT QUEUES

3. CONCURRENT QUEUES

In this chapter, Michael and Scott's [10] two kinds of concurrent queue is presented. One is the lock-free concurrent queue and the other is the two-lock concurrent queue. The lock-free concurrent queue uses CAS to update its pointers. In Java, which is the programing language used for implementation in this thesis, the `AtomicReference<V>` class can be used to implement the CAS function. Furthermore, by using the helping technique [14], the queue structure achieves lock-freedom.

3.1 ABA PROBLEM

A problem called the ABA problem may occur with the CAS primitive [11]. This problem cannot be detected by CAS and leads a value to be incorrectly written into a location. The problem is that a value *A* at the expected location is changed by another thread to *B* and then changed back to *A*. When the CAS primitive examines this address it will consider it as equal to the expected value, which is *A*, but be changed twice, and do the update that should not be done. Since Java language has a garbage collection mechanism, the ABA problem is automatically fixed. The codes in this thesis do not specifically deal with ABA problem.

3.2 ALGORITHM OF THE LOCK-FREE CONCURRENT QUEUE

The lock-free concurrent queue is basically implemented by singly-linked list. Two pointers are introduced as "head" and "tail". The enqueue operation is done from the tail of the list and the dequeue operation is done from the head of the list. In order to keep the nodes in the list always being linked when moving the pointers and safely freeing the removed nodes, this algorithm uses a dummy node to keep the head pointer always pointing to it, which is the last dequeued node. The dummy node guarantees either of the head and tail pointers does not point to `NULL`. Figure 5 gives the pseudo code of such operations of the lock-free concurrent queue.

```
1       class LockFreeQueue<E>
2           Node<E> sentinel; //dummy node
3           head = new AtomicReference<Node<E>>(sentinel);
4           tail = new AtomicReference<Node<E>>(sentinel);
5
6           void Enqueue(E elem)
7               newNode = new Node<E>(elem);
8               loop
9                   curTail = tail.elem;
10                  tailNext = curTail.next;
11                  if(curTail == tail.elem)
12                      if(tailNext == NULL)
13                          if(curTail.next.CAS(tailNext, newNode))
14                              tail.CAS(curTail, newNode);
15                              return;
16                      else
17                          tail.CAS(curTail, tailNext);
18
19          E Dequeue()
20              loop
21                  dummy = head.elem;
22                  realHead = dummy.next;
23                  curTail = tail.elem;
24                  if(dummy == head.elem)
25                      if(dummy == curTail)
26                          if(realHead == NULL)
27                              return NULL;
28                          tail.CAS(curTail, realHead);
29                      else
30                          E nodeElem = realHead.elem
31                          if(head.CAS(dummy, realHead))
32                              return nodeElem;
```

Figure 5. Enqueue and dequeue of the lock-free concurrent queue

When enqueue a node into the lock-free concurrent queue, two pointers need to be updated. One is the tail pointer and the other is the next pointer which is included in the current tail node and pointing to `NULL` (line 9 - line 10). So two CAS operations are needed. With this implementation, one thread can know the status of the queue, and also can help another thread to finish updating this queue. The current thread firstly checks the consistence of the tail pointer (line 11). If the tail pointer is not moved, the next pointer of the tail node may not be

`NULL` because the other thread has accessed the queue and linked a new node to the tail of this queue, but has not updated tail pointer yet. That indicates that thread is running between the line 13 and line 14. In this case, the current thread directly jumps to the line 17 in order to help the other thread to finish the pointer-moving step, and then starts over the loop. Otherwise, it enqueues a new node and updates the pointers with CAS (line 13 - line 14).

When dequeue happens, the current thread first checks the consistence of the `dummy`, `realHead` and `curTail` (line 24). The checking of line 25 divides into two situations. One is that the queue is empty, which is verified by line 26. If line 26 returns false, that means the queue is not empty and another thread has already done a successful insertion but the tail pointer has not updated yet. In that case, the current thread helps to move the tail pointer to the right location (line 28). In other situations, the current thread uses CAS to update the pointers and dequeues the appropriate node (line 30 - line 32).

3.3 ALGORITHM OF THE TWO-LOCK CONCURRENT QUEUE

The two-lock concurrent queue also uses a dummy node to protect the head and tail pointers of the queue. Instead of using CAS primitive to atomically update the pointers, the queue gives two locks on the critical sections, where some operations execute on the head and tail pointers. After one thread has acquired the access permission to a critical section, other threads are blocked until the accessed thread finishes its execution and the lock is released.

```
33     class TwoLockQueue<E>
34         Node<E> sentinel; //dummy node
35         head = new Node<E>(sentinel);
36         tail = new Node<E>(sentinel);
37
38         void Enqueue(E elem)
39             newNode = new Node<E>(elem);
40             newNode.next = NULL;
41             lock
42                 tail.next = newNode;
43                 tail = newNode;
44                 unlock
45
46         E Dequeue()
47             lock
48                 curHead = head;
```

```
49                newHead = curHead.next;
50                if(newHead = NULL)
51                    return NULL;
52                item = newHead.elem;
53                head = newHead;
54                unlock
55            return item;
```

Figure 6. Enqueue and dequeue of the concurrent two-lock queue

Figure 6 shows the pseudo code of the concurrent two-lock queue. Same as the lock-free concurrent queue, the two-lock concurrent queue set a dummy node at the head of the queue to where the head pointer always points. The dummy node avoids the situation that the head and tail pointer point to `NULL` after executions of enqueue and dequeue. The two locks are set at line 41 and line 47 respectively. The codes lead the updating of the head and tail pointers are considered as the critical sections, which need the protection by locks.

4. CONCURRENT SEARCH TREES

4. CONCURRENT SEARCH TREES

In this chapter, concurrent AVL tree [1] is presented first. It introduces version numbers and a hand-over-hand optimistic validation to achieve mutual exclusion when different threads intend to rebalance the tree after insertions or deletions. Then CBTree [2] is introduced, which is implemented by also using the same mechanism as in [1] to achieve concurrency, and also implemented with a different approach of restructuring the tree.

4.1 ALGORITHMS OF CONCURRENT AVL TREE

An AVL tree (Figure 7) is a balanced binary search tree, which minimizes its height by rotations when the heights of a node's left and right subtree differ by more than one. Such a property of an AVL tree is called "self-balancing". The concurrent version of it aims to make multiple threads be able to concurrently access a particular subtree of the tree in order to do some changes.

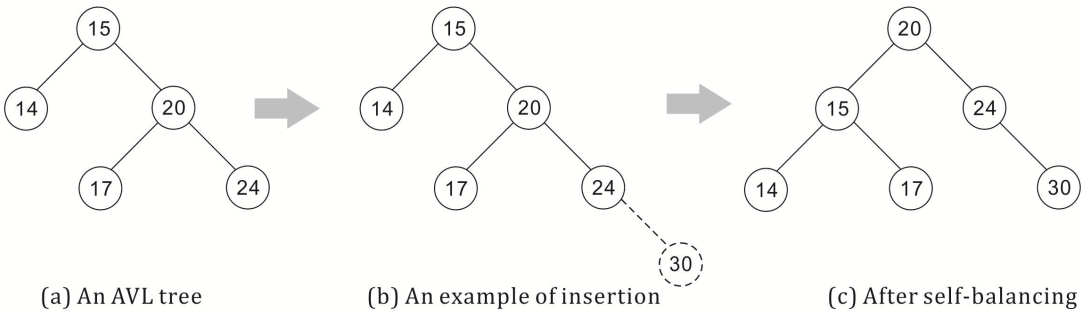


Figure 7. An example of AVL tree

The node structure

A concurrent AVL tree stores contents in nodes, which is same as a normal AVL tree and an additional version number. The structure of the node of the tree is presented in Figure 8. It shows that each node contains the contents as a generic type “key” and “value” variations. Since this algorithm uses map object to build the node structure, “key” and “value” have been combined as an association that a specific content of “value” can be identified with the “key". Moreover, this structure includes the height of the node itself, and the links of its parent, left and right subtree.

```
56    Node<K,V> {
57        int height;
58        long ver;
59        K key;
60        V value;
61        Node<K,V> par;
62        Node<K,V> leftC;
63        Node<K,V> rightC;
64        . . .
65    }
```

Figure 8. The structure of class Node [1]

Version numbers

The mechanism of version numbers (Figure 9) is the key to achieve mutual exclusion in the algorithm. Each of these version numbers represents a status of the node that is inquired for an access. When changes like insertions or deletions need to be done in the tree, the algorithm checks the version numbers stored in the current node, to see if it has been changed. If the version numbers are not the same before and after the change, the changes are considered to be invalid, then the inquiring of the access must start over; otherwise the trying of access is permitted and changes will be validated. Such mechanism only allows one thread to do a validated changing at a time and other threads are blocked until the last operation is validated.

```
66      Unlinked = 1L;
67      Growing = 2L;
68      GrowShift = 3;
69      GrowMask = 0xffL << 3;
70      Shrinking = 4L;
71      ShrinkCountShift = 1L << 11;
72      Ignores = ~(Growing | GrowMask);
```

Figure 9. Version numbers [1]

Searching

Searching is a basic operation of a tree data structure. In order to achieve concurrency, during a search operation the concurrent AVL tree uses hand-over-hand locking to lock the critical sections of the structure.

Hand-over-hand locking aims to decrease the duration of waiting nodes to release locks. Nodes on the searching path release locks as soon as the correctness of the search is not affected by rotations. In this thesis, version numbers are used to achieve this locking mechanism.

A searching in a concurrent AVL tree is almost the same as it is in a normal binary search tree, except that the contention of multiple threads must be additionally considered. When a searching starts in a concurrent AVL tree, the difference is merely that it begins at a special node called root holder. The root holder is a node that links the actual root of the tree as the right child of itself, and without the part of key or value. The version number of the root holder is always zero. The root holder is not affected by any changes in the tree. Consequently, it significantly simplifies the implementation of the concurrent AVL tree.

Moreover, in a concurrent AVL tree, before a searching choose the left or right subtree to go, it must check the nodes' status. The node that has already been accessed need to be checked first, and then follows the next node along the searching path. If anyone of these checks breaks the synchronization, the searching retries from the node that has been accessed until the restructuring of the tree is finished. This scheme asks the searching in the tree to validate its condition before every access of the next node to avoid contentions. The pseudo code of the searching implementation is showed in Figure 10.

```
73      //search in concurrent AVL tree
74
75      search(K key) {
76          return trySearch(key, rtHolder, 1, 0);
77      }
78
79      trySearch(
80          K key, Node n, int path, long curV) {
81          while (true) {
82              ch = n.child(path);
83              if (((n.ver^curV) & Ignores) != 0)
84                  return RETRY;
85              if (ch == NULL)
86                  return NULL;
87              nextPath = key.compareTo(ch.key);
88              if (nextPath == 0)
89                  return ch.value;
90              chVer = ch.ver;
91              if ((chVer & Shrinking) != 0) {
92                  waitUntilNotChanging(ch);
93              } else if (chVer != Unlinked &&
94                          ch == n.child(path)) {
95                  if (((n.ver^curV) & Ignores) != 0)
96                      return RETRY;
97                  q = trySearch(key, ch, nextPath, chVer);
98                  if (q != RETRY)
99                      return q;
100     }   }   }
```

Figure 10. The searching method of concurrent AVL tree [1]

At the line 75, the searching begins with a given key. Line 76 shows the searching is implemented by the method `trySearch` and starts at the root holder. In line 83, `Ignores` indicates that the changes that grow the node and its subtrees should be ignored. In that case, with a rotation to balance the subtree of a certain node, the subtree is grown by means of the height values of all the nodes in the subtree added by one. Meanwhile, a shrink happens on the grown subtree's parent and its children by the height values of them are all decreased. Such situation points out that the searching would be still valid with a grown node, and invalid with a shrunk node as well as their subtrees. Because the path that leads to an expected node is not misdirected from a grown node, but leads to an unsuccessful searching from a shrunk node (Figure 11). As a result, if there is a shrink of node detected in line 83, the searching retries. As well, from line 90 to line 96, if the status of the next node on the path detected as shrinking, the thread waits until the changing is finished. Or if the node is temporarily unlinked, which is discussed later, the searching also retries from the beginning of the loop.

After all of these validations, the searching can access to the next node in order to recursively invoke the `trySearch` method to return the value of the target node (line 89), or return `NULL` in the case of no such node with the given key in the tree (line 86).

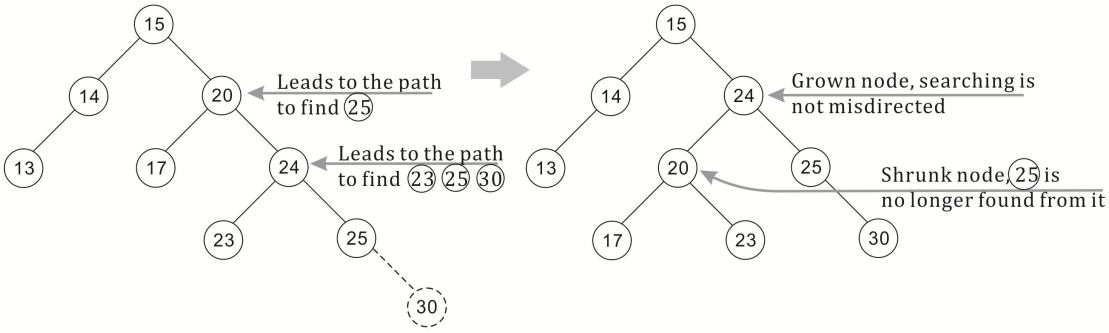


Figure 11. An example of the grown node and the shrunk node

Insertion

An AVL tree inserts a new node by comparing the value of the key stored in the node with other nodes to locate the new node. According to the comparison, the insertion algorithm decides the path to go down along the tree structure to continue the insertion until the new node is linked as a leaf or an update of value occurred in a node. Deriving from this algorithm, an insertion in a concurrent AVL tree follows the rules of insertion in basic binary search trees but uses hand-over-hand optimistic validation and locks to guarantee concurrently operations.

```
101     insert(K key, V value) {
102         return tryInsert(key, value, rtHolder, 1, 0);
103         }
104
105     tryInsert(
106             K key, V value, Node n, int path, long curV) {
107         q = RETRY;
108             do {
109                 ch = n.child(path)
110                 if (((n.ver^curV) & Ignores) != 0)
111                     return RETRY;
112                 if (ch == NULL) {
113                     q = doInsert(key, value, n, path, curV);
114                 } else {
115                     nextPath = key.compareTo(ch.key);
```

```
116                 if (nextPath == 0) {
117                     q = doUpdate(ch, value);
118                 } else {
119                     long chVer = ch.ver;
120                     if ((chVer & Shrinking) != 0) {
121                         waitUntilNotChanging(ch);
122                     } else if (chVer != Unlinked &&
123                         ch == n.child(path)) {
124                         if (((n.ver^curVer) & Ignores)!=0)
125                             return RETRY;
126                         q = tryInsert(key, value, ch, nextPath, chVer);
127             }   }   }
128         } while (q == RETRY);
129             return q;
130     }
131     doInsert(
132         K key, V value, Node n, int path, long curV) {
133         synchronized (n) {
134             if (((n.ver^curV) & Ignores) != 0 ||
135                     n.child(path) != NULL)
136                 return RETRY;
137             n.setNewCh(path, new Node(
138                 1, 0, key, value, n, NULL, NULL));
139         }
140         fixHeightAndRebalance(n);
141             return NULL;
142     }
143     doUpdate(Node n, V value) {
144         synchronized (n) {
145             if (n.ver == Unlinked) return RETRY;
146             preValue = n.value;
147             n.value = value;
148             return preValue;
149        }
150     }
```

Figure 12. The insertion method of concurrent AVL tree [1]

Figure 12 shows the implementation of insertion. As the procedure of search, it begins at the root holder (line 102) in order to keep an unaffected parent of the root in the tree. Before accessing in every node, it checks the status of each node. If the node is shrunk or in the process of restructuring, the current insertion process retries (line 105 - line 125) until it arrives at the exact position where the insertion should be done (line 113). It is either a node that has already been in the tree and only needed to update its value stored in itself, or a leaf of the tree that must be the inserted node's parent.

The `doInsert` method from line 131 to line 142 presents the process to insert a new node into the tree. Because the algorithm involves concurrent operations of multiple threads, it is highly possible that there are other threads attempting to insert a new node with the same key at the time as the current thread. To prevent such conflicts, an insertion acquires a lock on the future parent of the new leaf (line 133). Meanwhile, to guarantee that no harmful structural changes occur when an insertion is about to be done, which can lead the new node to be at a wrong position, a validation as in search method is used (line 134 - line 136). In contrast, if an insertion happens when the node with the given key has already showed in the tree, the only thing to do by the insertion is to update the value of the node with the given key. Since it does not bring a structural change, the algorithm only acquires a lock to prevent a simultaneous update by other threads at this node, and does not perform any validations (line 143 - line 148).

AVL tree is a self-balancing tree that is able to adjust the height in order to rebalance itself. It usually happens after a successful insertion or deletion of node in the tree. The concurrent AVL tree calls a `fixHeightAndRebalance` method (line 140) to recompute the height of each node and to decide how to rotate. This method is discussed latter.

Deletion

Deletion is algorithmically more complex than other operations in a binary search tree. If a node, which is intended to be deleted, is a leaf or only has one child, it is directly unlinked from the tree and then its child is linked to its parent if any. If a node has two children, it is called an internal node. Firstly, a node with the smallest key in the right subtree of the deleted node should be found as an alternative node. Then the key-value association stored in the alternative node are copied and moved into the deleted node in order to replace the original key-value association stored in this node. Finally, the alternative node must be unlinked from the original position and its children should be linked to its parent. Since the deletion algorithm of a binary search tree changes the positions of nodes along the path that is possibly followed by a search executed by another thread, a concurrent binary search tree must obtain the lock mechanism to guarantee atomic synchronizations. However, such excessive locking impacts the performance and scalability [1]. Bronson et al. presented a partially external trees based on previous research [17] to compensate for the drawback of the locking scheme.

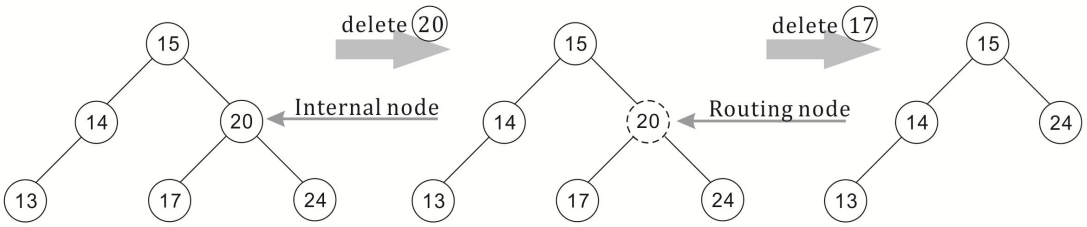


Figure 13. An example of a routing node in a partially external tree

Figure 13 gives the perspective of partially external trees. When an internal node is needed to be deleted, it only deletes the "value" field of the key-value association by setting it to `NULL`, and keeps the "key" field. If a subsequent searching heads to this deleted node, the searching can be routed by the comparison between the reserved key and the target's key. In this scenario, the deleted node is called a routing node as it can route a searching in the path. Moreover, if a subsequent operation makes the routing node an uninternal node, the routing node needs to be unlinked directly by the hand-over-hand optimistic validation.

A node with one child or no children is deleted directly, where "delete" means unlinking the node from the tree and linking its child (if there is one) to its parent. It is same with how the deletion is done in a normal binary search tree. This process is showed in Figure 14. Likewise, in order to achieve concurrency, the operation of deletion follows the same pattern as the insertion. What makes the differences is that in line 113 of Figure 12 where `tryDelete` returns `NULL`, and in line 117, the `tryDelete` calls the method `deleteNode`.

```
151    delete(K key) {
152        return tryDelete(key, rtHolder, 1, 0);
153    }
154    ... // tryDelete here is similar to tryInsert in Figure 12
155    boolean unlinkOrNot(Node node) {
156        return node.leftC == NULL || node.rightC == NULL;
157    }
158    deleteNode(Node p, Node node) {
159        if (node.value == NULL) return NULL;
160
161        if (!unlinkOrNot(node)) {
162            synchronized (node) {
163                if (node.ver == Unlinked || unnlinkOrNot(node))
164                    return RETRY;
165                preValue = node.value;
166                node.value = NULL;
167            }
168        } else {
```

```
169             synchronized (p) {
170                 if (p.ver == Unlinked || node.parent != p
171                         || node.ver == Unlinked)
172                     return RETRY;
173                 synchronized (node) {
174                     preValue = node.value;
175                     node.value = NULL;
176                     if (unlinkOrNot(node)) {
177                         t = node.leftC == NULL ? node.rightC : node.leftC;
178                         if (p.leftC == node)
179                             p.leftC = t;
180                         else
181                             p.rightC = t;
182                         if (t != NULL) t.par = p;
183                         node.ver = Unlinked;
184             }   }   }
185             fixHeightAndRebalance(p);
186         }
187         return preValue;
188     }
```

Figure 14. The deletion method of Bronson et al.s tree [1]

The `deleteNode` method performs in order to create partially external tree and directly unlink non-internal nodes. It acquires a lock after it checks the node whether it is an internal node or not. By implementing OCC, the version number of the target node is checked before a further operation, either converting it into a routing node or unlinking it. If the unlinking is not possible, the removal retries. Meanwhile, it also checks if the node is still internal or not after the lock has been held in case of other threads restructuring the tree at the same time. At last, `fixHeightAndRebalance` method is called to rebalance the tree after an unlinking, which alters the height of the subtree.

fixHeightAndRebalance

After the insertions and deletions, the `fixHeightAndRebalance` method is called for recomputing the height value of each node and adjusting the structure of the concurrent binary search tree by rotations. Meanwhile, this method also unlinks routing nodes that have less than two children. By checking the condition of the accessed node, this method updates the height field in each related node and decides which kind of rotation will be done by verifying that if the balance factor is smaller than -1 or bigger than 1.

```
189    fixHeightAndRebalance(Node<K,V> n){
190        while(n != NULL && n.par != NULL){
191            status = nStatus(n);
192            if(status == NothingRequired || isUnlinked(n)){
193                return;
194            }
195            if(status != UnlinkRequired &&
196                    status != RebalanceRequired){
197                synchronized(n){
198                    n = fixHeight(n);
199                }
200            } else {
201                Node<K,V> pa = n.parent;
202                synchronized(pa){
203                    if(!isUnlinked(pa) &&
204                            n.par == pa){
205                        synchronized(n){
206                            n = rebalance(pa,n);
207    }   }   }   }   }   }
```

Figure 15. The method of `fixHeightAndRebalance` [1]

Figure 15 shows the code of this method. It indicates that before it proceeds to recompute or adjust the height, it locks the relevant nodes. In order to prevent the damage of the links between each node, locks are required both at the current node and at its parent when there is either a single rotation or double rotations. The `fixHeight` method aims to update the height value of each node after operations. The `rebalance` is used to decide which kind of rotations should be done here. Similarly, before implementing both of these methods, the status of the current node is checked by inspecting its version numbers (line 192, line 195 - 196).

The codes about rotations are omitted since the algorithms are similar to the

rotations of an AVL tree. Except that they have to lock the nodes that might be rotated and update the version numbers of the growing and shrinking nodes. The algorithms of rotations also involve the OCC to retry in a recursive loop if there are any concurrent updates happen simultaneously.

4.2 ALGORITHMS OF THE SEQUENTIAL CBTREE

CBTree is another self-adjusting binary search tree, which is derived from the splay tree. This section describes details of the algorithm of the sequential CBTree and its concurrent implementation.

Splay tree

The splay tree [3], which is developed by Sleator and Tarjan, is also a self-adjusting binary search tree. It moves the accessed node towards the root by rotations, which is called splaying. The way of rotations is based on the positions of the involved nodes: the currently accessed node, its parent and its grandparent. The three types of rotations are presented in Figure 16.

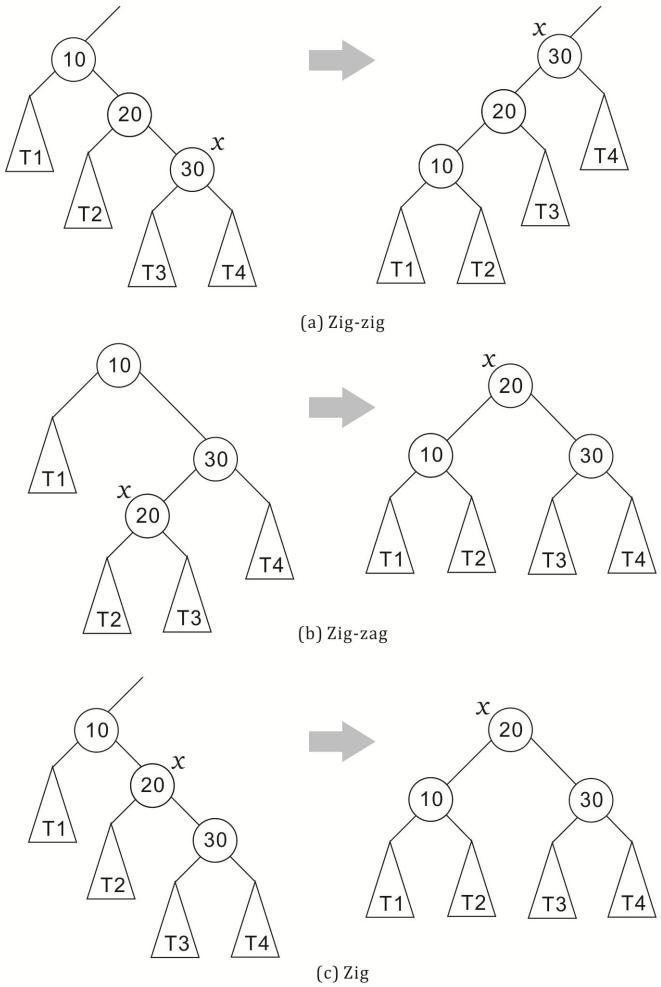


Figure 16. Three types of splaying. *x* is the accessed node.

If a node is accessed by operations like insertion, deletion or searching, the structure of the splay tree is transformed. Unlike an AVL tree, a splay tree does not balance itself by comparing the height of each subtree of a node in order to keep the overall running time $O(log_2 N)$. In the worst case of a splay tree, the nodes are only accessed linearly. That leads to the overall running time of this tree $O(N)$. However, to a binary search tree, the running time of $O(N)$ is not very bad only if such an operation happens quite infrequently [18]. Considering the situation that if there are 100 operations running in a certain data structure, only 5 operations are with the running time of $O(N)$ and the running time of other 95 operations is $O(log_2 N)$, the overall performance of the structure is considered relatively satisfactory. In such cases, an analysis of the amortized running time of a data structure or an algorithm is performed [19]. It aims to consider the whole sequence of operations and establishes average performance of the entire algorithm.

With amortized analysis, a splay tree has an amortized run time of $O(log_2 N)$ [3]. The particular property of the splay tree is that it moves the recently accessed node towards the root, which means if a node is accessed very frequently, its position in the tree is near the root. Such situation is very common in practical applications. It is highly possible that an item is searched and used again and again in a period of time. According to this special property, splay trees provide a better performance in such kind of applications.

Sequential CBTree

The CBTree is a kind of splay tree. The term CBTree is the abbreviation of counting-based tree. It is invented to achieve a good scalability in concurrency. According to that, the ways of rotations in the CBTree are altered. The CBTree only does semi-splaying, which is only parts of the algorithm of rotations in a splay tree. Figure 17 shows the details of the semi-splaying. The CBTree examines the current node's relative positions and the position of its child and grandchild to decide which type of rotations should be performed. It is similar to the AVL tree. Unlikely, the CBTree maintains a weight field in each node to present the number of accesses in its subtrees. This field is divided into three parts: *leftWeight* and *rightWeight* that are used to store the number of accesses in either subtree respectively, and *selfWeight* is the current node's number of accesses. The CBTree does a semi splaying by computing the weight values of the current node and the grandchild along the path to the destination.

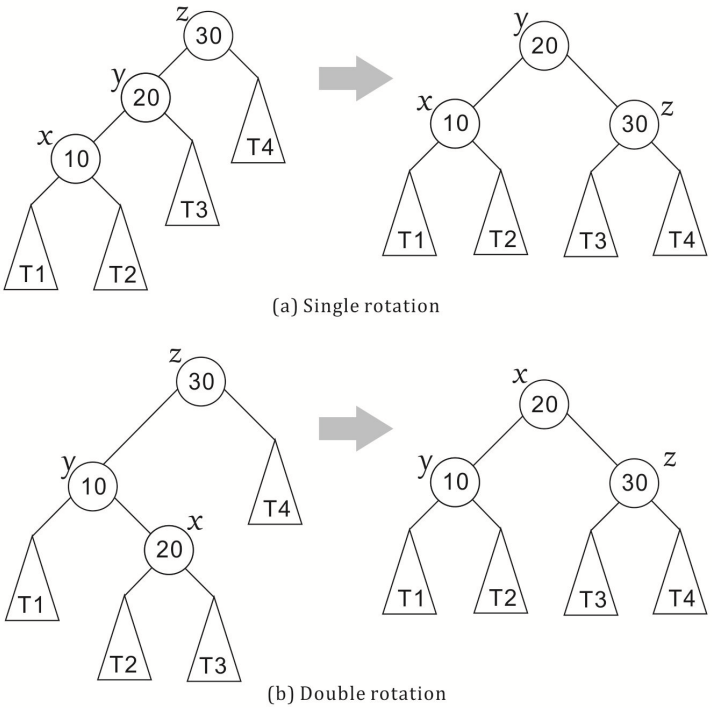


Figure 17. Semi-splaying. *x* is the current node. The case when *x*'s parent is the right child is symmetric.

Let $W(v)$ be the total weight of the current nodes in the subtree rooted at node *v*. Then $W(v) = v.selfWeight + v.leftWeight + v.rightWeight$. Sleator and Tarjan proposed a potential function [3] to analyze the amortized complexity of splaying. It is also used for analyzing the CBTree. Let $r(v) = \log_2 W(v)$ be the rank of *v*. The potential of a CBTree is $\Phi = \Sigma\, r(v)$ over all nodes *v* in the tree. Afek et al. [2] gave a bound for the potential change caused by a rotation, which is represented by an inequality: $2 + \Delta\Phi \leqq 2(r(z) - r(x))$. In this inequality, $\Delta\Phi$ indicates the difference of the potential of the CBTree before and after a semi-splay. Node *z* is the current node and node *x* is the grandchild of *z* on the path.

Sleator and Tarjan defined the amortized time *a* of an operation by $a = t + \Delta\Phi$, where *t* is the actual time of the operation [3]. To make the CBTree efficient, the amortized time should be decreased after a rotation. Derived from the inequality, we can get:

$$
\begin{aligned}
& 2 + \Delta\Phi \leqq 2(r(z) - r(x)) \\
\rightarrow \quad & \Delta\Phi \leqq 2(r(z) - r(x) - 1) \\
\rightarrow \quad & r(z) - r(x) \leqq 1 \\
\rightarrow \quad & \log_2(W(z)/W(x)) \leqq 1 \\
\rightarrow \quad & W(z)/W(x) \leqq 2
\end{aligned}
$$

As the Figure 17 presents, a rotation occurs when $W(z)/W(x) < \alpha$, while α is a constant less than 2.

The operations of searching, insertion and deletion in a sequential CBTree are almost the same as they are in an AVL tree. The difference is when and how to do the rotations. Assume the current accessed node is z with its child y and grandchild x as Figure 16 shows, when a searching runs, it follows the path from the root to the desired item. The searching maintains z and checks the value of $W(z)/W(x)$ in order to decide whether to perform a rotation. The position of x decides whether a single rotation or a double rotation should be done. If the value of $W(z)/W(x)$ is less than α, a rotation occurs, and the current node will be replaced by y or x, which depends on what kind of rotation it does. Otherwise, if the value of $W(z)/W(x)$ is equal to or bigger than 2, which is out of α's range, no rotations would happen and the current node z skips to x. That means the searching does not check the rule of rotations and hence does not perform any restructuring at y. After the searching ends on the desired item v, the algorithm increases $W(v)$ and the weights of all the nodes along the path from the v to the root by 1.

Before an insertion and deletion in a sequential CBTree, the algorithm firstly searches for the objective item by doing rotations in the tree. After the searching finishes, the weight of each node along the searching path is increased, and then the desired insertion or deletion is performed. An insertion inserts a new item into the CBTree if the searching returns `NULL`, or updates the value of a certain item if the item is successfully found by the searching operation. A deletion happens if the searching finds the item that is about to be removed. Like a deletion happens in an optimistic concurrent binary search tree, which has been mentioned before, if the item has two children, which means it is an internal node, it is marked as a routing node and only logically deleted. The value field of this node is set as `NULL`. When any restructuring leads to the routing node only has a single child or no children, this node is unlinked by the hand-over-hand optimistic validation scheme.

4.3 ALGORITHMS OF THE CONCURRENT CBTREE

The concurrent CBTree is a concurrent implementation of the sequential CBTree using Bronson et al.'s hand-over-hand optimistic validation mechanism. Version numbers are used to handle certain operations in critical sections. Differently, the implementation must decide wether to skip a node when there is no rotation. The most importantly, the weight value of each node must be updated after a successful operation or restructuring.

Searching

The searching in a concurrent CBTree is implemented almost the same as the searching in concurrent AVL tree. It recursively looks up the desired node with the hand-over-hand optimistic validation. Moreover, the `tryGet` method adds a skip parameter that is initially set to be `TRUE` to indicate that if no rotations needed then skip the child of the current node along the searching path (Figure 18). In this method, the current accessed node is *z*, and the `trySemiSpl` (Figure 19) method in line 221 checks wether the rule of rotations is satisfied. If there is a rotation, the structure of the three involved nodes is changed and the algorithm returns `RETRY` in order to replace *z* by the current node after the restructure. Otherwise, if no rotation happens, the recursive method in line 231 reverses the skip parameter. According to that, in the next recursive method's process, the `trySemiSpl` method will not be implemented and the algorithm will directly jump to another recursive method. Line 217 - 218 shows that if y, which is the child of current node, is the found node, its weight is updated. On the other hand, the codes of line 232 - 239 describe how to increase the weight after the recursive method has returned. Figure 19 gives the implementation of the method `trySemiSpl`. It runs to check the rule of rotations in the concurrent CBTree and decides which kind of rotations to do. Obviously, it also adjusts the weight value of each node after a restructuring.

```
208    tryGet(k, z, path, curV , SKIP) {
209        while (true) {
210            y = z.child(path);
211            if (y == NULL) {
212                if (node.ver != curV)
213                    return RETRY;
214                return NULL; // no such node
215            }
216            nextPath = compare(k, y.key)
217            if (nextPath == E) {
218                y.selfNum++;
219                return y.value;
220            }
221            if (!SKIP and trySemiSpl(z, y, nextpath))
222            // validations
223                return RETRY;
224            yVer = y.ver;
225            if (yVer is changing)
226                waitUntilNotChanging(y);
227            else if (y == z.child(path)
228                    and y is not unlinked) {
229                if (z.ver != curV)
230                    return RETRY;
231                q = tryGet(k,y,nextPath,yVer,!SKIP);
232                if (q != RETRY) {
233                    if (q != NULL) {
234                        if (nextPath == L)
235                            z.leftNum++;
236                        else
237                            z.rightNum++;
238                    }
239                    return q;
240                }
241    }   }   }
```

Figure 18. The `tryGet` method of CBTree [2]

```
242    trySemiSpl(z, y, nextPath) {
243        rotate = NONE;
244        if (z.leftC == y) {
245            if (nextPath == L) {
246                x = y.leftC;
247                if (x != NULL && ΔΦ(z,x) < α)
248                    rotate = SINGLE;
249            }
250            else if (nextPath == R) {
251                x = y.rightC;
252                if (x != NULL && ΔΦ(z,x) < α)
253                    rotate = DOUBLE;
254            }
255            if (rotate == NONE)
256                return false;
257            gr = z.par;
258            synchronized (gr) {
259                if (gr.leftC == z || gr.rightC == z) {
260                    synchronized (z) {
261                        if (z.leftC == y) {
262                            synchronized (y) {
263                                if (rotate == SINGLE) {
264                                    singleRight(gr, z, y, y.rightC);
265                                    z.leftNum = y.rightNum;
266                                    y.rightNum += z.selfNum + z.rightNum;
267                                } else {
268                                    Node x = y.rightC;
269                                    if (x != NULL) {
270                                        synchronized (x) {
271                                            doubleRightThenLeft(grand,
272                                                                z,y,x);
273                                            z.leftNum = x.rightNum;
274                                            y.rightNum = x.leftNum;
275                                            x.rightNum += z.selfNum +
276                                                          z.rightNum;
277                                            x.leftNum += y.selfNum +
278                                                         y.leftNum;
279                                }   }   }
280                                return true;
281            }   }   }   }   }
282        } else {
283            ...
284        }
285        return false;
286    }
```

Figure 19. The `trySemiSpl` method of the CBTree. It shows when y is a left child; the right child case is symmetric. It gives the process of the rotation in this case, which is the same with the process of concurrent AVL tree's implementation [2].

Insertion and deletion

The implementation of the concurrent CBTree's insertion and deletion follows the algorithm of the concurrent binary search tree. They both do a searching in the tree first. If there is no such a node in the tree, it inserts a new node into the tree; otherwise it updates the value of this node. The deletion procedure transforms the found node into a routing node in order to build a partially external tree. After the hand-over-hand optimistic validation in a recursive method has made the node a single-child node or a leaf, this node is directly unlinked. In the concurrent CBTree, the algorithm of insertion and deletion increases the weight value of each node after every successful operation. The alternative rule of rotation and the skip mechanism have been showed earlier and used in the insertion and deletion's navigation part.

4.4 SINGLE ADJUSTER

The CBTree maintains a weight counter in each node. In the process of traversals or after a successful operation in the tree, the counters have to be updated and synchronized. Such updates occur in the private cache of some multicore processors of Intel architectures, which means the core must acquire exclusive ownership of the cache line. Therefore, when all of the cores of the processor update the same counters of some nodes, each core has to wait for other cores in order to acquire the cache line. It degrades the scalability and loses the performance of the concurrent CBTree.

In order to bypass this limitation, an optimization mechanism, which is called a single adjuster, is implemented. When multiple threads are running, one of the threads is set as a dedicated thread. This special thread periodically alters between doing operations with restructuring the tree and without restructuring. All other threads only do the read-only operations, which means an item is searched, inserted or deleted without rotations and updates of the weight counters. Since rotations in a concurrent CBTree is rare in this mechanism [2], all other threads' operations benefit from the single adjuster's restructuring. In the chapter 5, the advantages of using such mechanism is showed. Under certain configurations, the searching path length of the concurrent CBTree is shortened with the scalability of the tree and so it is optimized as well.

5. EXPERIMENTAL EVALUATION

5. EXPERIMENTAL EVALUATION

The performance of the concurrent AVL tree and the concurrent CBTree is compared and analyzed in this chapter. We greatly acknowledge the use of the source code available at [23] as a part of our own implementation. Moreover, the performance of the concurrent lock-free queue and the two-lock queue is presented.

5.1 EXPERIMENTAL MACHINE

All of the experiments are run on a work station with two Intel Xeon E5645 (Nehalem) processors. Each processor has six cores running at 2.4GHz. Each core runs two SMT threads. This makes 24 hardware threads available on the machine. The machine has 24 GB main memory. It runs on Ubuntu Linux 12.04.

The Intel Nehalem architecture maintains a three-level cache. Each core of the processor has a two-level private write-back cache and the last level shared cache. As it has been mentioned in section 4.4, the single adjuster mechanism is used for implementing the concurrent CBTree to reduce the impact of updating counters on such architecture. The performance of the CBTree that uses the single adjuster scheme is compared with the CBTree without the single adjuster in this chapter, which shows a significant difference between these two implementations.

5.2 IMPLEMENTATIONS OF THE CONCURRENT BINARY SEARCH TREES

Three algorithms of tree structures are evaluated in this section. One is the concurrent CBTree without the single adjuster mechanism, which is referred to as **Cbt** in the following sections. By comparing the performance of it with a concurrent CBTree that uses the single adjuster, the impact of implementing concurrent CBTree with weight counter on the Intel Xeon architecture is analyzed. The other algorithm is the concurrent CBTree with single adjuster

mechanism, it is referred to as **CbtOneAdjuster**. The third algorithm for evaluation is Bronson et al.'s concurrent AVL tree, which is referred to as **Opt** in the following sections.

In order to reduce the impact of other operations, the experiments only run "searching" operation in these tree structures. It makes few restructuring and clearly shows the superiorities of using concurrent CBTree in certain circumstances. According to Afek et al.'s paper of the CBTree [2], the single adjuster mechanism is implemented as: the dedicated thread running this mechanism does the searching operations with restructuring for 1 millisecond, and then alters to do searching operations without restructuring for 20 milliseconds. This thread keeps altering its implementation between such two patterns until it finishes its running. All other threads only do read-only searching operations. Every thread repeatedly searches a set of items for 1000 times. Moreover, for initialization, each implementation begins with a maximum balanced complete binary search tree. To achieve that, we set the median item as the root of the tree, and the first quartile as the root's left child and the third quartile as the root's right child and so on [2].

In the experiments, 1 million sequential numbers are initially inserted in these tree structures as the node's keys for searching. Searching operations are run in different sections of height and are also run with different range of random numbers. So through the results of such experiments, performances of different structures can be evaluated to describe the advantages or drawbacks of each algorithm, as a direction to improve them in the future.

Different sections of height

Since the CBTree is based on the principle of splay tree, it aims to move frequently accessed nodes towards the root. The aim is to make the searching faster because the path length of the frequently searched node is decreased by splaying. This experiment searches nodes in three different sections of height in each tree structure. They are in the domain of the first, second and third trisection of the height of the trees, which indicates that the testing nodes distribute in the deepest part, the middle part and the highest part of each tree.

Since the number of nodes in each initialized tree is 1000000, the height of each tree structure is $log_2(1000000) + 1 \approx 19$. The searching operations are implemented in the section of height 0 to 6, 7 to 13 and 14 to 19 respectively. Figure 20 describes the performance (operations per milliseconds) of each tree in each section of height.

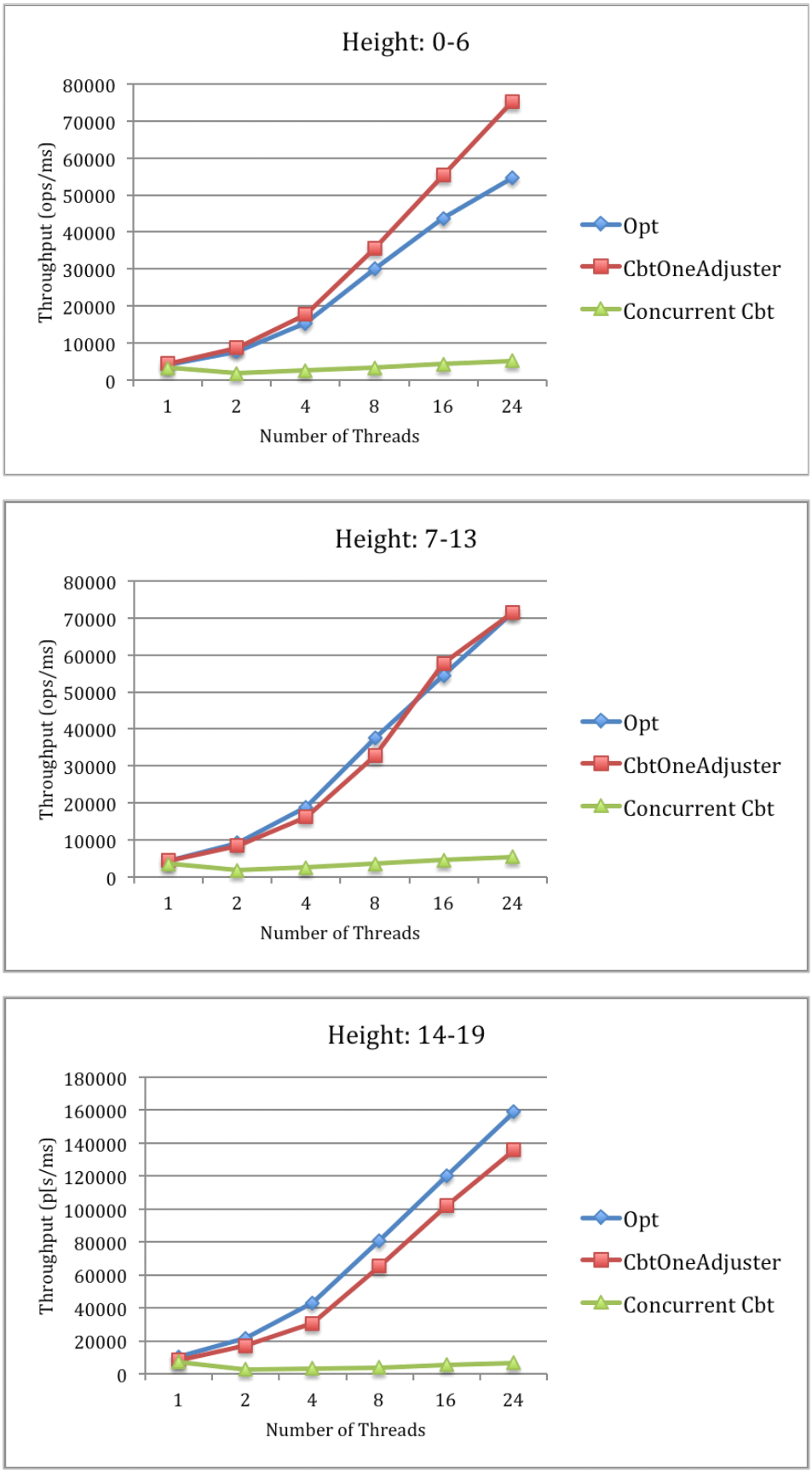


Figure 20. Results of experimenting with searching in different heights

In each section of height, **Cbt** gives similar performance. When multiple threads search items in **Cbt** at the same time, the tree is restructuring frequently. The performance is much reduced by the restructuring of the tree. Moreover, the main drawback of this implementation is the limitation of the processor's architecture. As it is mentioned in section 4.4 and 5.1, the Intel Xeon E5645 processor updates the counter serially and that leads to the poor scalability of the concurrent CBTree. Although **Cbt** has a better searching path length than **Opt**, **Opt** performs much better than **Cbt** since it does the read-only searching without any restructuring and its implementation is not impacted by the processor's architecture.

With the increase in the height, **Opt**'s performance has been improving. Because the path length is shorter when the search node is higher. However, comparing with splay trees, **Opt** has poor scalability of structure when it is doing read-only searching. Without any restructuring, when the search nodes are in the very deep locations, **Opt** takes a long path to get them, even if these nodes are searched frequently. According to that, from the Figure 20, in the section of height 0 to 6, **Opt** performs worse than **CbtOneAdjuster**. In the section of height 7 to 13, Opt performs very close to **CbtOneAdjuster**, and in the section of height 14 to 19, **Opt**'s performance overpasses **CbtOneAdjuster**. When the search nodes locate in a high position, the reduced path length by implementing **CbtOneAdjuster** is small. It makes the improvement in scalability of **CbtOneAdjuster** not so obvious. On the other hand, **CbtOneAdjuster** also takes time for restructuring. So the read-only searching implemented by **Opt** in high positions of the tree gives a better performance.

By using the single adjuster mechanism in CBTree, **CbtOneAdjuster** gives the best performance when the search nodes are in the very deep positions. It moves the frequently searched nodes to the root in order to reduce the path length, which could be very long according to how deep the searched nodes locate. Consequently, the searching time is shorter after the nodes have been searched several times. Even in the middle trisection of the tree, **CbtOneAdjuster** also performs very similar to **Opt**. Furthermore, comparing to **Cbt**, the advantage of implementing the single adjuster mechanism is distinct, since the mechanism decreases the impact of restructuring as far as possible.

Different range of searching

According to the algorithm of the CBTree, if a node is frequently searched, it should be moved towards the root. In order to complete this process, the tree is restructured by several rotations. In the implementation of concurrent CBTree,

when multiple searching operations are running in the CBTree, it possibly happens that the height of one searching node *A* is increased, and the height of the other searching node *B*, which is *A*'s sibling, is decreased at the same time. It drops the advantages brought by the splaying of the CBTree, since the path length of the searching nodes may not be shortened a lot. Figure 21 shows an example of such situation.

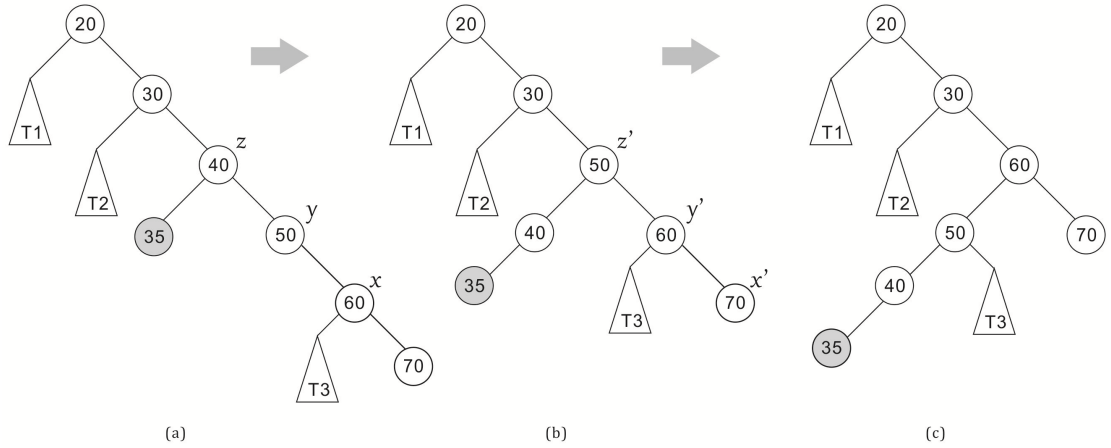


Figure 21. (a) The current node is *x*. (b) The current node is *x'*. (c) The height of node 35 is lower than before and its searching path is longer.

When the search nodes are distributed in a large range, such affections by restructuring are aggravated. The path length of a search node may be increased several times by the splaying implementation of another subtree. Even if its own splaying shortens its path to the root, the average performance will still be degraded with such affection brought by concurrently splaying.

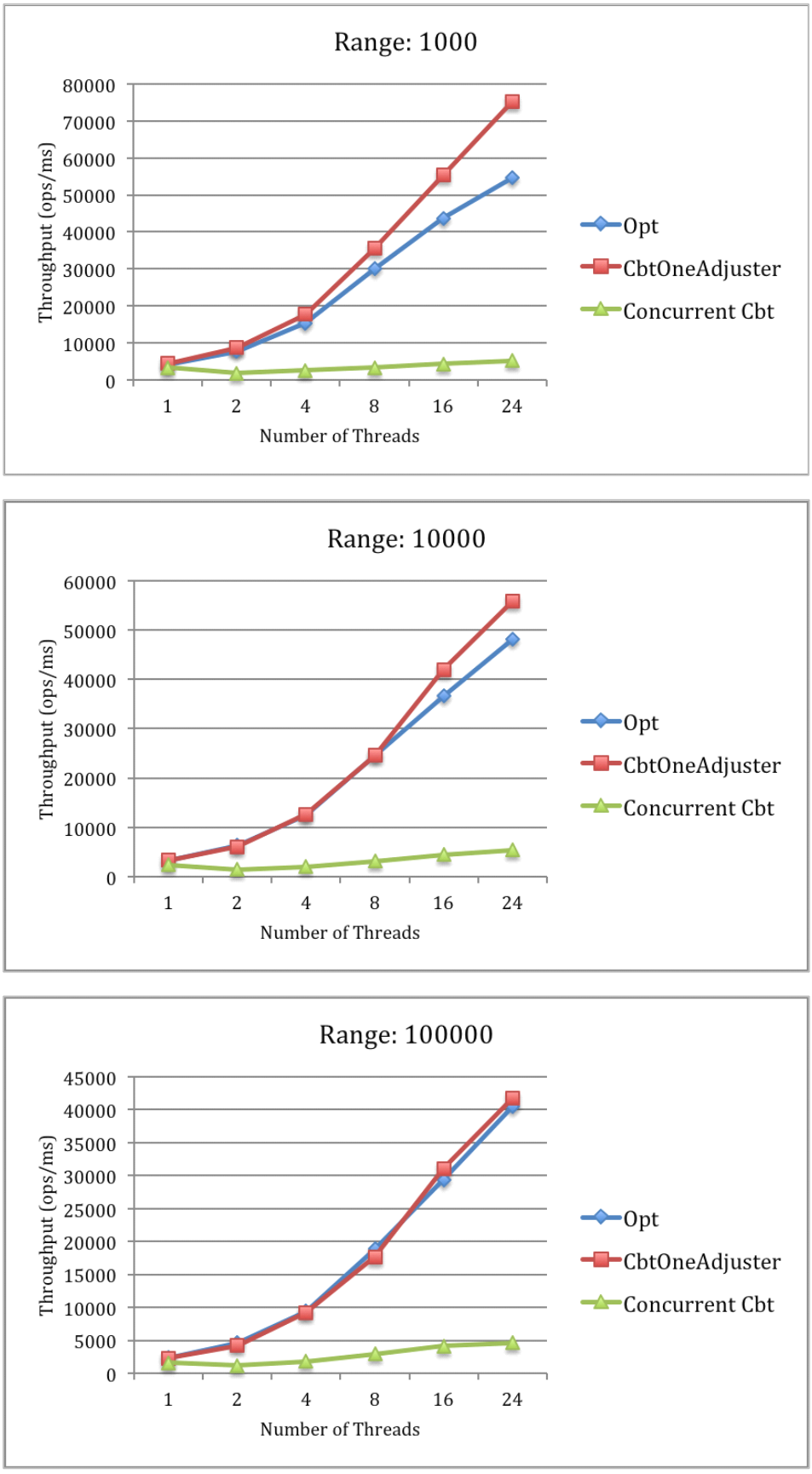


Figure 22. Results of experimenting with searching in different ranges

Experiments of searching in different range of nodes are implemented in this section. Figure 22 presents the distinction of the implementation of each tree structure with different searching range. The implementation is done on the 0 - 6 section of height in each tree, which is with the longest path of searching. When the searching range is 1000, **CbtOneAdjuster** outperforms the others. Even if no restructuring, **Opt** also has a worse performance because its average path length is longer than **CbtOneAdjuster**. Despite of **Cbt** also having a shorter path length since the algorithm of restructuring is same with **CbtOneAdjuster**, **Cbt** is limited with the feature of the architecture used in the experiments. Moreover, from Figure 22, when the searching range increases to 10000 and 100000, the performance of **CbtOneAdjuster** is close to **Opt**. Even if the path length is shortened, the advantages of splaying gradually can not cover the loss of performance brought by the restructuring. As a result, when the searching range is 100000, **CbtOneAdjuster**'s performance is almost the same as **Opt**.

5.3 IMPLEMENTATIONS OF THE CONCURRENT QUEUES

The concurrent lock-free queue and the concurrent two-lock queue are evaluated in this section. Each thread does 1 million random operations of enqueue and dequeue. Figure 23 shows the time consumption of each concurrent queue when multiple threads concurrently run in the implementation.

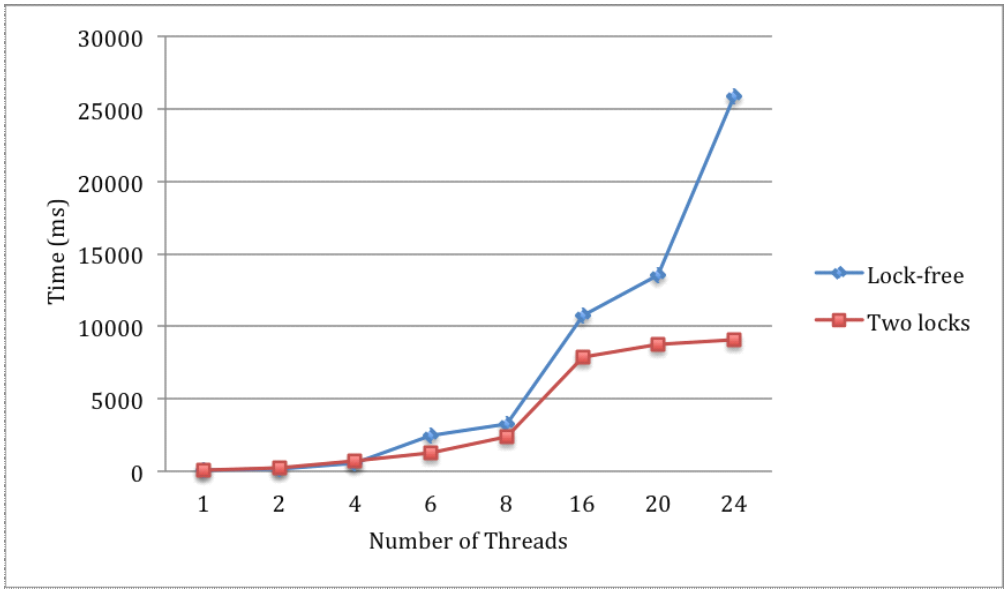


Figure 23. Performance of the two concurrent queues

Under low contention of threads, the lock-free concurrent queue outperforms the two-lock concurrent queue. CAS makes the current thread to help another thread that has accessed the shared resource finish its operation. At this contention level, CAS usually returns true and new value is written into the shared resource at the first trying. However, the lock-free concurrent queue has a low performance under a high contention of threads. Threads executing on the two-lock concurrent queue do not compete for accessing the critical sections. They just wait the accessed thread to release the lock. Contrarily, threads may execute CAS several times in order to update the pointers. The level of competition between threads are very hight in this case. That makes the throughput of the two-lock concurrent queue is better than the lock-free concurrent queue under a high contention level.

6. CONCLUSION AND FUTURE WORK

6. CONCLUSION AND FUTURE WORK

This Master thesis mainly discusses two concurrent tree data structure algorithms: concurrent AVL tree and concurrent CBTree. Bronson et al. developed the hand-over-hand optimistic validation mechanism to concurrently implement AVL tree. It blocks the bad contentions in the tree by checking the version numbers involved in each node. CBTree aims to concurrently implement the splay tree by counting the number of accesses to each node. It also uses Bronson et al.'s hand-over-hand optimistic validation to achieve concurrency. By experimenting the performance of each algorithm based on searching in each tree, even CBTree has a short path length, its scalability is also affected by the processor's architecture. Afek et al. developed a single adjuster mechanism to bypass this limitation. In addition, a big searching range could make the searching path be shortened not very prominently, which can not cover the loss of performance from the restructuring.

Michael and Scott's two concurrent queue data structure algorithms: lock-free concurrent queue and two-lock concurrent queue, are briefly discussed in this thesis as well. By introducing a dummy node, which locates in front of the queue and the head pointer always points to, the head and tail pointers of both the two concurrent queue data structures are protected from pointing to `NULL`. The lock-free concurrent queue uses CAS operation to atomically update the shared resource. The two-lock concurrent queue only locks the critical section to achieve mutual exclusion. Experimental implementation shows that under a low contention of threads, the lock-free concurrent queue outperforms the two-lock concurrent queue. But when the contention level is high, the two-lock concurrent queue has a better performance.

The two tree data structure algorithms respectively provide a method to concurrently implement the AVL tree and the splay tree. Bronson et al.'s hand-over-hand optimistic validation gives a mechanism that would be useful for implementing other kinds of concurrent trees. It is also based on the locking mechanism, which causes the threads that can not access the current nodes to wait. However, the optimization in implementation of CBTree is stonewalled by the cache level's arrangement of many popular and widely available processors. In addition, the lock-free concurrent queue loses efficiency under a high contention of threads. In future, how to develop a lock-free concurrent mechanism to implement tree data structure and how to bypass the limitation brought out by the architecture of a certain processor are the possible directions of researches, which could improve the performance of concurrent tree

algorithms. Moreover, how to improve the performance of the lock-free concurrent queue could also be a possible direction of following study.