

# Concurrent Data Structures

Thesis

Submitted in partial fulfilment of the requirements of  
BITS C421T/422T Thesis

By

Shreya Inamdar

ID.No. 2010A7TS144P

Under the supervision of

Prof. Madhavan Mukund

Professor and Dean of studies, Chennai Mathematical Institute

and

Prof. S.P. Suresh

Associate Professor, Chennai Mathematical Institute



BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI (RAJASTHAN)

Date: 01/05/2014

## **Acknowledgements**

I would like to extend sincere gratitude to my supervisors Prof. Madhavan Mukund and Prof. S. P. Suresh for giving me an opportunity to work with them, their excellent advice, optimism and endless patience to guide me through the thesis. I would also like to thank Prof. Murali P, for his continual guidance and support that was instrumental for this opportunity to take effect.

## List of symbols and abbreviations

Through-out the text, unless explicitly mentioned all the notation refers to the ones used in [1].

CAS	compareAndSet
Node	Node in the log for universal wait-free construction
TNode	A node in the binary search tree
STM	Software Transactional Memory

## Abstract

Concurrent objects are shared memory data structures that are concurrently read and written by multiple threads asynchronously. These data structures have different design principles and correctness conditions as compared to their sequential counterparts which help programmers to reason about their behaviour using techniques from sequential domain.

This thesis aims to get a practical understanding of principles behind the design and implementation of concurrent data structures while using linearizability as a correctness condition. Highlights of basic concurrent data structures like heaps, binary search trees and skip lists is presented at different levels of concurrency, followed by a study of a well-known type of binary search tree in recent literature. Consensus objects and their application in making any sequential data structure into lock-free or wait-free one, and the efficiency of such generalized techniques is discussed.

Based on the analysis of the well studied data structures and universal constructions, a new practical wait-free algorithm for binary search trees is proposed, which is shown to be more efficient than current practical implementations of non-blocking counterparts, in both space and run-time, at the cost of handling concurrency using many consensus objects.

**Keywords:** Concurrency, linearizability, progress, linked list, heap, skip-list, binary search tree, practical wait-free construction

## Table of Contents

1. Introduction
2. Properties of concurrent data structures
  - 2.1 Correctness
  - 2.2 Progress
3. Types of synchronization
  - 3.1 Coarsegrained
  - 3.2 Finegrained
  - 3.3 Optimistic
  - 3.4 LazyList
  - 3.5 Non-blocking
4. Implementations of some concurrent data structures
  - 4.1 Concurrent Heap
  - 4.2 Concurrent Skip List
  - 4.3 Non-blocking Binary Search Tree
5. Universal Constructions
  - 5.1 Lock-free Universal Construction
  - 5.2 Wait-free Universal Construction
6. Wait-free binary search tree
  - 6.1 Optimizations in Wait-free Universal Constuction
  - 6.2 Classes and Hierarchy
  - 6.3 Implementation
  - 6.4 Correctness
  - 6.5 Performance
  - 6.6 Future Scope
7. Conclusion
8. Bibliography
9. Appendix
  - 9.0 Classes used in Wait-free Binary Search Tree
  - 9.1 Implementation of Concurrent heap
  - 9.2 Implementation of Concurrent SkipList
  - 9.3 Implementation of Non-blocking Binary Search Tree
  - 9.4 Implementation of Wait-free Binary Search Tree

Note: Appendices 9.1 – 9.4 have been appended separately to the report to maintain the legibility of the lines of code.

# 1. Introduction

With increased applications of distributed computing due to the advent of multi-core processors. Being able to design, reason and implement concurrent data structures has become an important skill to harness the huge advantage of scalability. Designing correct and practical concurrent data structures requires an understanding and experience with the basic design principles and a few standard techniques, as well as reasoning to formally prove correctness. Given the complexity of the state space of these programs, due to multiple possible interleavings among concurrently executing threads, this becomes a crucial skill. This thesis is an attempt to familiarize with the reasoning and testing of concurrent data structures and getting a hands-on experience by applying them to the ones that exist in literature. We then take-up the problem of designing a wait-free binary search tree data structure based on the universal construction and engineering the optimizations to make the solution practical.

The text consists of five major sections. First deals with basic properties of shared objects – correctness and progress. The second section quickly summarizes the different levels of synchronization currently in literature and how they influence the degree of concurrency. The third section gives an overview of some non-trivial yet simple concurrent data structures, with their complete implementations in the appendix. Section four summarizes universal constructions that apply for all sequential objects. Finally section five elaborates the design and implementation of a wait-free binary search tree, which is a novel contribution of this thesis.

## 2. Properties of Concurrent data structures

Correctness and progress are two independent properties that are used to describe the behavior of concurrent objects. Both of them give programmers a way to reason about concurrent data structures in relation to the concepts from sequential domain. Formally, correctness is known as *safety* and progress is known as *liveness* property.

### 2.1 Correctness

In sequential systems, where an object goes through different states sequentially, correctness can be explained through preconditions and postconditions. In a concurrent system, operations can be invoked by concurrent processes and it becomes necessary to give meaning to different possible interleavings of invocations. There are different notions of correctness for concurrent objects based on notion of equivalence with sequential systems, as given by Maurice and Herlihy in [1]. These are based on the following principles:

1. Method calls should appear to happen in one at a time sequential order
2. Method calls should appear to take effect in program order
3. Each method call should appear to take effect instantaneously at some moment between its invocation and response

The strongest of all these principles, termed *linearizability*, is 3. It implies both 1 and 2.

The basic idea behind linearizability is that - every concurrent history is equivalent to some sequential history following the condition that - method calls must take effect in their precedence order and overlapping method calls can be ordered in any convenient way[3].

Properties of Linearizability:

1. Compositional property - A system is linearizable if each and every object is linearizable. This is also known as local property and allows modular design, construction and verification of correctness of complex systems by reasoning about individual objects and methods independently.
2. Non-blocking property - The pending invocation of a total operation is never required to wait for another pending invocation to complete.  
An operation is *total* if it is defined for every object value. Otherwise it is *partial*.  
For instance, enqueue is total as it is defined for every value of an unbounded queue, but dequeue is partial as it is not defined for an empty queue.  
Thus linearizability is an appropriate correctness condition for systems where concurrency and real-time response are important.

## 2.2 Progress

There are different levels of progress conditions in multiprocessor programming. They dictate how the behaviour of one thread might affect the behaviour of other threads.

1. A method is *blocking* if an unexpected delay by one thread can prevent other threads from progressing.
2. A method is said to be *lock-free* if at least one thread, and thus a system as a whole, is guaranteed to progress at all times.
3. A method is said to be *wait-free* if every thread completes its operation in a finite number of steps.

## 3. Types of synchronization

There are different levels of synchronization which dictate varying degrees of concurrency.

1. Coarsegrained synchronization
2. Finegrained synchronization
3. Optimistic synchronization
4. LazyList synchronization
5. Non-blocking synchronization

### 3.1 Coarsegrained synchronization

Synchronization is done at the level of method calls. This precludes concurrency to a large extent, as calling a synchronized method attains an intrinsic lock on the object on which this method is called[4]. Thus preventing other methods from proceeding till this method returns or catches an exception.

### 3.2 Finegrained synchronization

Generally shared objects (tree, list etc) have multiple components linked by references. Improving over coarse-grained synchronization, we try to lock individual components that the method requires, instead of locking the object as a whole. Thus two method calls interfere only when they want to read or modify components that cannot be altered independently.



### 3.3 Optimistic synchronization

When a method wants to modify a component in the object, it traverses without acquiring any locks, and on finding the component of interest, acquires locks, verifies to check if the components have been altered in the meantime and subsequently proceeds to completion. This takes concurrency to another level, as methods don't acquire locks on the components they do not intend to modify. The trade-off in this approach is that – in case the object was modified in the meantime, the process has to re-start. This might also lead to starvation of a few threads unless special care is taken by the programmer.

### 3.4 LazyList Synchronization

This type of synchronization splits deletion of data components in two phases - logical removal and physical removal. Without acquiring locks, a node can be logically removed, generally done by storing an additional atomic *mark* variable in each component. The responsibility to physically delete that component then is handled by a specialized method or other methods.

### 3.5 Non-blocking synchronization

By using atomic compareAndSet functions, locks can be eliminated altogether. Thus progress of at least one thread is guaranteed. Starvation, however, might still be possible.

## 4 Implementations of Concurrent Data Structures

To get a practical understanding of concurrent data structures, some standard and well studied data structures were implemented as given in [1,5,6,7]. A bird's-eye view of the design is presented while highlighting the key points and complete implementation given in the appendix.

### 4.1 Concurrent Heap

The FineGrainedHeap class extends naturally from sequential heap implementation.

The method add() creates a new leaf node, and pushes it up the tree until the heap property is restored. To permit multiple threads to handle the same heap, nodes are percolated up the tree in discrete steps while setting required flags. The method removeMin() stores the root node for returning, swaps a leaf node with the root, and pushes that node down the tree until the heap property is restored. The locks are acquired only when the node is percolating down the tree. While the nodes that are being pushed-up use flags to declare their status.

Status: AVAILABLE / EMPTY / BUSY	
Priority	Owner thread ID
item	lock

Fig 4.1 A Heap Node

**Status:** A node is busy when it is being percolated up in the heap and is not yet at the right place. The root is empty when removeMin() call swaps the leaf node. Otherwise the node is available.

**Priority:** The priority of nodes in the max heap

**Item:** The item to be stored in the node

**Lock:** The lock used by removeMin() while pushing the nodes down the tree

The complete implementation is given in the appendix 9.1

## 4.2 Concurrent Skip List

Lock-based concurrent skip list is an example of LazyList type of concurrency.

Each node has

- a lock
- marked field indicating whether it is in the abstract set, or has been logically removed.
- FullyLinked flag is set to true once it has been linked in all its levels

The following invariant is always maintained: Higher-level lists are always contained in lower-level lists.

An add() call begins with find(), which returns the node's predecessors and successors at all levels. Validation that the predecessors still refer to the successors is done by locking the predecessors at every level. Once added at all levels, fullyLinked flag is set. Until a node is fully linked, all access to that node is put on a hold till fullyLinked flag is set.

A `remove()` call also begins with `find()` to check if the victim node is already the list and is fully linked and unmarked (ready to be deleted). If these conditions hold, then the mark bit is set, indicating the logical removal of the node. Physical removal is done after validation and from top to bottom maintaining the skiplist property.

A `find()` call traverses the skiplist and checks the `unmarked` and `fullyLinked` flags to decide whether the node is in the list. In case any of the locks in `add()` or `remove()` fail, `find()` is called again to get a fresh list of predecessors and successors. The `find()` method ignores all locks and thus is wait-free.

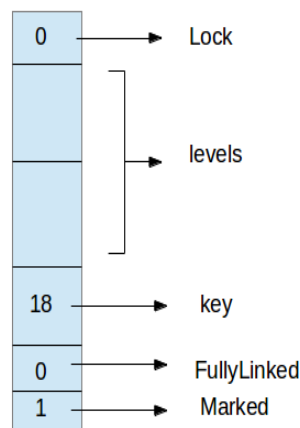


Fig 4.2 A SkipList Node

Complete implementation is in appendix 9.2.

### 4.3 Non-blocking Binary Search Tree

This data structure is introduced in the widely cited paper on Non-blocking Binary search Trees by Ellen Et. al.[2] To bypass the need of acquiring locks on the nodes of a binary search tree, techniques have been suggested on a method for converting lock-based operations into non-blocking ones. The main idea is to replace locks owned by processes with the locks owned by operations. To allow for more concurrency, routing nodes are introduced.

Here is a bird's-eye view of the algorithm:

A search operation inputs a key to be searched and returns a tuple with the parent, grand-parent, matching node and, the state of the parent and grandparent node.

An insert() calls search to check if the node with the key is already in the tree. The details of the returned values are used in insert to check if the parent is itself busy in some operation. If that is the case, the thread calling this insert, helps the parent node. This happens recursively.

Similarly a delete() also calls a search and checks the grandparent's status along with parent's status, as it atomically alters the child pointer of grandparent to bypass the node being deleted. If grand-parent or parent are themselves busy, then the thread helps these operations before running its own operation.

Insert and delete use several flags to notify the thread of their status. A complete implementation is given in the appendix 9.3.

## 5. Universal Constructions

Any sequential object can be converted into a linearizable wait-free concurrent object[1]. This generalized conversion may or may not be have practical complexity, but it is possible[8].

A class C is universal if one can construct a wait-free implementation of any object from some number of objects of C and some number of read–write registers. We use CAS Class to convert the sequential binary search tree into its wait-free linearizable version. Consensus number of CAS class is infinite, as it can handle consensus among any number of threads.

### 5.1 Lock-free Universal Construction

A deterministic sequential object is modelled as a combination of an initial state and a log(linked list) of state transitions ( method calls). The sequential object is required to be deterministic, as there should be only one possible final state given an initial state and an operation applied on the initial state.

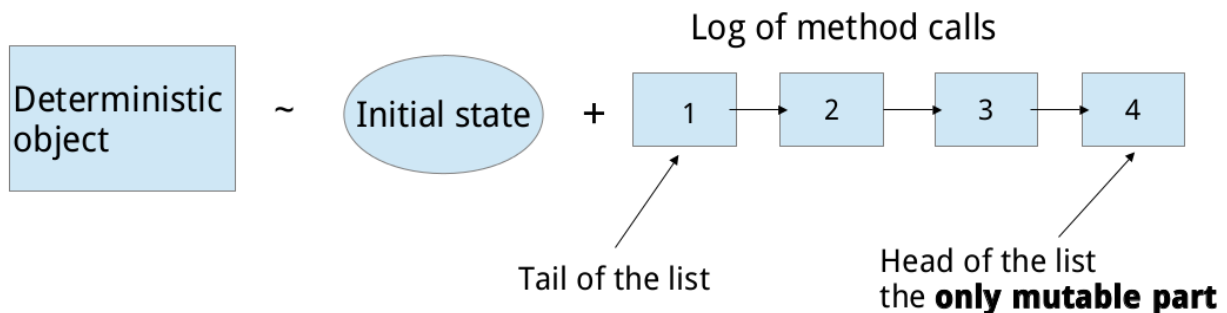


Fig 5.1.1 Modelling a sequential deterministic object

A lock free universal construction converts a sequential deterministic object into a lock-free linearizable concurrent object.

A sequential thread of this class, when invoked with the `apply(invoc)` method call, would

1. Create its node with the invocation it was called with
2. Append the node to the head of the log
3. Creates its own copy of the object in the initial state
4. Applies all the operations starting from the tail to the head, returning the result of its invocation

In a concurrent version, there are more than one threads waiting to append their own node to the head of the log. To handle this contention, the next field of the current head in the linked list is made a consensus object. By using the CAS consensus object, contention among any number of threads can be handled. The first one to call the consensus object wins. No subsequent writes happen to the next field. In the figure we have a CASconsensus object that is implemented using an atomic register.

```
class CASConsensus extends ConsensusProtocol {
    private final int FIRST = -1;
    private AtomicInteger r = new AtomicInteger(FIRST);
    public Object decide(Object value) {
        propose(value);
        int i = ThreadID.get();
        if (r.compareAndSet(FIRST, i)) // I won
            return proposed[i];
        else // I lost
            return proposed[r.get()];
    }
}
```

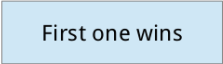


Image source: The Art of Multiprocessor Programming, Maurice and Herlihy.

Fig 5.1.2 Implementation of CAS Consensus class

The winner of the contention then makes a private copy of the initial state of the object and applies all operations in the log till it calls apply() on its own invocation and returns the response. The other threads update their head to the winner node and try to append their node to the updated head.

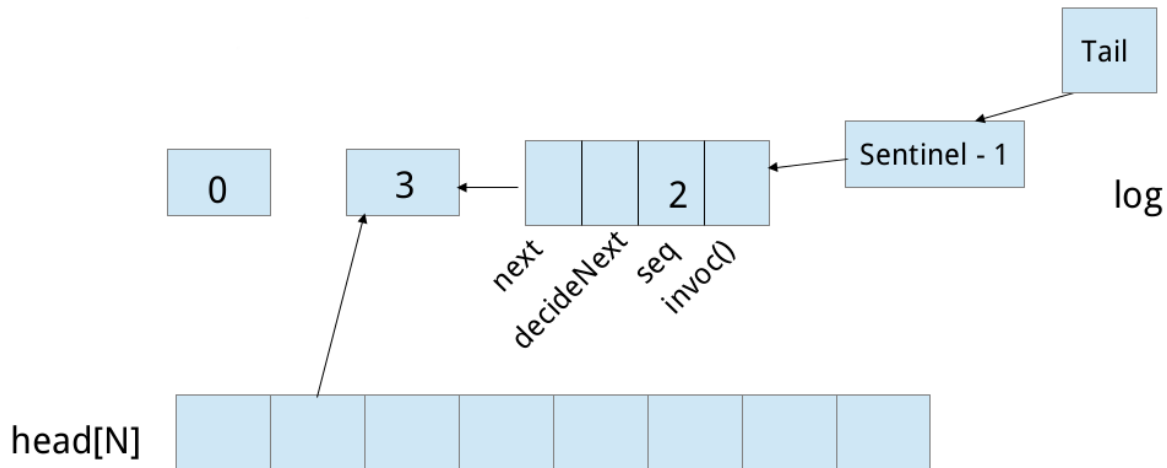


Fig 5.1.3 Snapshot of a lock-free universal construction

The above figure shows a snapshot of lock-free construction object. The log contains nodes. Each node has four fields.

1. **Invoc** – the invocation which is assigned to the thread appending this node
2. **seq** – the sequence number of the node in the log. This is a strictly increasing sequence. Tail node points to a sentinel whose sequence number is 1. Seq of a node is zero when initialized and becomes a non-zero value on getting appended to the log.
3. **Head[]** - head[i] points to the node with maximum sequence number as seen by thread i. Node with maximum value of seq is the head to which a thread tries to append its node.
4. **next** – since multiple threads traverse the list many times, winner of consensus is stored in next field for use by subsequent threads

The following figure gives pseudo code for apply() operation of a lock-free construction.

apply(invoc)

1. make a new node with current invocation
2. while the new node is not appended to the log
3.     before  $\leftarrow$  node with maximum sequence number
4.     after  $\leftarrow$  winner of cas consensus on before.decideNext
5.     update sequence number of after
6.     head[i]  $\leftarrow$  after
7.     from tail to head
8.     apply invocations to thread's private copy
9. return response of invocation assigned to this thread

Fig 5.1.4 Pseudo code of a lock-free universal construction class

To prove that this algorithm is lock-free we need to prove that system progress is guaranteed. The only reason that one thread may not progress is because it constantly keeps losing consensus. But this in-turn implies that other threads are constantly winning the consensus. And the system as a whole is progressing.

Lock-free constructions have a drawback of starvation. There is no upper limit on how many times a thread will have to re-try for getting its node appended to the log. Once there is an upper bound on the number of steps it takes for a thread to complete its apply() call, we have a wait-free construction.

## 5.2 Universal Wait-free Construction

To ensure that every thread completes its `apply()` method call in a finite number of steps, threads should help the starving threads. Helping a thread requires appending the helped thread to append its node to the log before the helper. For this every thread must announce the node it

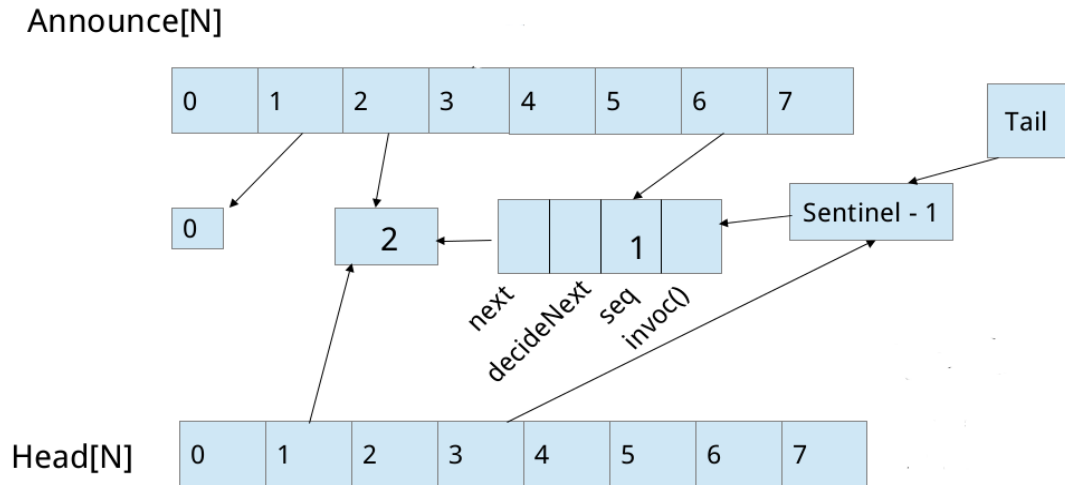


Fig 5.2.1 Snapshot of a Wait-free universal construction object

wants to append to the log. For this purpose announce array is used. `Announce[i]` stores the node which thread `i` wants to append to the log. Following figure shows a snapshot for universal wait-free construction which is same as the lock-free construction, with an addition of announce array.

Once `apply()` call of a thread is invoked, it initializes its node and announces it first. Then tries to append it to the log similar to the lock-free construction with a small change – it tries to help the node ahead of it in the announce array that needs help. After appending it in the log, the thread tries to append its own node.



Following figure is the pseudo-code for wait-free construction's apply() call that describes the small modifications as compared to lock-free construction.

```
apply(Invoc invoc)
1.  announce its new node
2.  read head[] to get node with max sequence number
3.  while (announce[i] is not yet appended to list)
4.    before ← head[i]
5.    if a node ahead of it in announce [] needs help
6.      prefer ← announce[(before.seq+1)%n]
7.    else
8.      prefer ← announce[i]
9.    after ← winner of consensus on before.decideNext
10.   update after.seq
11.   update head[i]
12.   <proceed like lock-free construction>
```

Fig 5.2.2 Pseudo-code for apply(invoc) of Wait-free implementation class

In the above figure, notice that every thread tries to help the node at  $(\text{before.seq} + 1) \% n$  th position in the announce array. This index is so crafted that any node that is announced is appended to the log after no more than  $n$  appends to the head as seen at the time of its announcement.

The following condition always holds.

*If more than  $n$  nodes have been appended to the log at the head as seen by a thread A at the itime of its announcement, then the node announced by thread A is already in the log.*

This puts an upper limit on the number of steps it takes for a thread to append its node to the log. Thus the lock-free algorithm is made wait-free.

## 6. Report on wait-free binary search trees

Starting from the wait-free universal construction, we have a brute-force solution in which the `apply()` method call does the job of calling the intended method - insert / find / delete.

### 6.1 Optimizing on universal wait-free construction

The reasons why this is not a practical method to implement concurrent wait-free algorithms for sequential data structures are the redundant application of operations to a new thread-local copy of the object. This is also not scalable because with the number of threads, required storage also increases. We first try to first optimize for time.

First point to note is that one thread can continue the operations from where it left off at the last function call, instead of starting all over again. This gives an improvement in time, but not in space, as previously also, there were as many copies of the object as there were threads. To save extra space, we try to maintain a global copy of the object instead of thread-local copies. This comes with a challenge to handle the concurrency and maintain a consistent global object for all the threads to access/modify.

Assume a global internal binary search tree, which performs operations as suggested by the wait-free construction. Consider the following contentions:

Concurrent inserts - Inserting a node in a binary search tree always happens at the leaf. Concurrent inserts can be handled by using `compareAndSet()` operations. In all the existing models, if we have two concurrent inserts contesting for the same parent, if one succeeds, the other has to re-try from the start.

Concurrent insert and delete - For instance, consider two concurrent operations - one inserts a new node, and another deletes the node which is set to be a parent to the newly inserted node. If the `delete()` on the node that is being inserted with a child gets linearized before `insert()`, then the insert doesn't take effect. To avoid draining time in these re-tries and invalid insertions, we have come-up with a solution in which every insert and delete calls `search()` only once. There are no re-tries.

The crux of the idea is to use consensus objects as reference to left and right children. This idea has been borrowed from the universal construction's next pointer, where concurrence is handled by calling `CASConsensus()` on the next field.

When using consensus objects for deciding left and right children, all operations get simplified.

Every call to Insert () performs a search on the tree for the location of insertion and if there doesn't already exist a node with same key, returns the parent node. Then, a new node with the required key is created and decideLeft or decideRight objects invoke their decideNext() method. If in the mean time of return of search operation, some nodes get added, the decideNext() call fails. But, since we are already on the right path, we continue traversing the nodes that have been inserted in the mean-time and try to call decideNext() on the appropriate left or right reference. If a node with same key is found, then function returns false. If successful, the new node gets inserted in the right place.

A call to delete() calls search() and simply sets an atomic freeze flag to true. Note that there is no physical deletion in our binary search tree. But since delete calls are significantly less in practice, we can afford retaining these nodes. We still do better in space than the other non-blocking implementations, as we have a completely internal tree with no extra internal nodes.

Find() just returns the results from search() after checking the freeze flag.

## 6.2 Classes and hierarchy

The following is a complete description of all the classes used in the implementation -

Class TNode:

Tree object has nodes of this class. Each node has the following new fields.

1. level - the level at which the node is inserted. This plays the role of seq in universal wait-free construction. Each node is initialized with its level as 0. Setting of level to a non-zero value indicates that the node has won consensus and has been successfully inserted.
2. freeze - the atomic boolean flag which indicates if the node has been logically removed from the tree or still exists as an element in the total order that is represented by the tree.
3. decideLeft & decideRight - these are CASConsensus objects that can be written only once.
4. left & right - they are the left and right pointers that store the winning node of the consensus

Class Node:

Node class has its objects in the list of invocations in the universal construction. In addition to those in the universal construction, we have a status flag that is used to record if that invocation has returned. Checking this flag saves us the time in helping some operations that have already taken effect on the global tree and precede in the program order with respect to this invocation.

Consensus classes:

We start with a consensus interface, with two decide() operators, one for Node class objects and other for TNode class objects. ConsensusProtocol is an abstract class that implements this interface. It defines propose() function for both the consensus. Finally CASConsensus is the concrete class that implements the decide() functions.

Model Overview

Internal Binary Search Tree is implemented and is made wait-free using the universal construction.

## 6.3 Implementation

Insert :

Like a sequential insert function, we first search for a suitable parent as the first step of insert. In concurrent case, however, there might be insertions between the time interval where search linearizes and insert linearizes. To account for these inserts, we keep calling CAS consensus objects for every left or right child of the parent returned by search till the new node which is to be inserted wins the consensus. In case another node with the same key gets inserted in the tree in the meanwhile, we return false.

There might also be cases where there is a contention between threads to insert their node at same position. All such threads perform a CAS consensus. The winning thread inserts its node and the other thread continue on their path downwards with the newly inserted node as the parent.

Delete:

Like a sequential deletion, the node to be deleted is first searched in the tree. In case this search linearizes before an insert of the same key, we return false. Otherwise, we atomically try to freeze the node if freeze is not already set. Unlike a sequential delete, there is no physical deletion of nodes involved. Only logical deletion is allowed. This is a side-effect of using CAS Consensus as they come with a one-time write restriction.

Find: find() calls search and checks for the freeze flag.

Following figures given show the implementation of insert, search and delete

### Insert

```
boolean insert(int key){
    TNode [] ret = ROOT.search(key);
    if(ret[0] != null){
        return ret[0].freeze.compareAndSet(true, false);
    }
    //if node is not found in the tree already
    TNode parent = ret[1];
    TNode toInsert = new TNode(key);
    TNode q;
    while(toInsert.level == 0){
        if(key > parent.key){
            q = parent.decideRight.decide(toInsert);
            parent.right = q;
            q.level = parent.level + 1;
            parent = q;
        }
        else if (key < parent.key){
            q = parent.decideLeft.decide(toInsert);
            parent.left = q;
            q.level = parent.level + 1;
            parent = q;
        }
        else{
            return false;
        }
    }
    return true;
}
```

Fig 6.3.1 Implementation of insert(key)

Insert calls search with the key to be inserted. If a match is found, and if the freeze flag is already true i.e. the element was logically not present in the tree, then it is atomically set to false. In case the freeze flag was set to false i.e. The key was present in the tree, then insert returns false. This is taken care of by compareAndSet(false, true) function on the freeze flag.

If search doesn't return a match, then there is a chance that the parent node already has inserted some child nodes. Assuming this to happen, we continue the search in the same path. At every node on the way down, the thread calls consensus object of one of left or right child. On winning the consensus, insert returns true. In case the consensus is lost, we continue with the winning node as the parent and retry inserting the key to the new parent.

Consider the case with two threads A and B. Both want to insert the same key. If insert of A linearizes after search of B, but before insert of B, we encounter a node with the same key as the one that is to be inserted. In such a case we correctly return false.

## Search

```
public TNode [] search (int key){
    TNode curr = this;
    TNode parent=curr;
    TNode [] ret = new TNode[2];
    while(curr != null){
        if (key == curr.key){
            ret[0] = curr;
            ret[1] = parent;
            return ret;
        }
        else if(key < curr.key){
            parent = curr;
            curr = curr.left;
        }
        else{
            parent = curr;
            curr = curr.right;
        }
    }
    ret[0] = null;
    ret[1] = parent;
    return ret;
}
```

Fig 6.3.2 Implementation of search(key)

Search function takes the key to be searched as input and returns an array of two TNodes in ret[]. ret[0] has the node that matched the search or null if there was no match. ret[1] has the parent at which the match was found. Since we have a root that has the maximum key possible, parent is never returned as null. Search for a matching key continues till a null node is encountered on the path from root. Encountering a null is a linearization point for an unsuccessful search, while encountering a matching key is the linearization point for a successful search.

## Delete

```
public boolean delete(int key){
    TNode [] ret = ROOT.search(key);
    if(ret[0] == null)
        return false;
    else{
        return ret[0].freeze.compareAndSet(false, true);
    }
}
```

Fig 6.3.3 Implementation of delete(key)

Delete function is only logical because of restriction on CAS object that it can be written only once. Delete calls search for the key to be deleted and on receiving a null, they return false. If the node is present, and freeze is not set, then freeze is atomically set to true. If freeze was already set, delete returns false. These operations are done atomically using compareAndSet on freeze flag which is declared as an AtomicBoolean.

## Find

```
public boolean find(int key){
    TNode [] ret = ROOT.search(key);
    if(ret[0] == null)
        return false;

    else{
        return !ret[0].freeze.get();
    }
}
```

fig 6.3.4 Implementation of find(key)

find() calls search() with the same key and atomically checks whether the key is logically present in the tree. In case a matching key was found and freeze was not set, it returns true. In all other cases, it returns false.

## 6.4 Correctness

To prove correctness, we need to prove wait-free nature and linearizability. These two are independent conditions. Wait-free nature follows from the universal construction. To prove linearizability, it is sufficient to prove that all methods are individually linearizable.

Linearization Points:

Successful search linearizes on hitting a node with matching key. Unsuccessful search linearizes when the search hits a null node. Linearization points of find are same as that of search.

Successful insert has two linearization points. One is when the key is already present and the freeze flag is to be unset. The other is when the node with key to be inserted wins a consensus of decideLeft or decideRight. As after this any search call sees the newly inserted node as a child and no insert can win consensus at that position again. Unsuccessful insert also has two linearization points. One when a node with matching key was found but compareandset fails as the freeze flag is already set. Next is when a node with the same key as toInsert is found. This happens when the search of one thread linearizes before insert of other.

Successful delete has the atomic set of freeze flag to true as linearization point. Unsuccessful delete linearizes at unsuccessful search and an unsuccessful compareAndSet() call to freeze flag node with matching key.

## 6.5 Performance

In general, the performance of wait-free implementations is considered to be less efficient than the non-blocking ones[8]. Performance of this algorithm, which has been measured against a non-blocking binary search tree implementation by Ellen Et. Al [2] shows otherwise. Both these implementations are run on Ubuntu 12.04, kernel 3.14.0-rc3+ system running on quad-core Intel(R) Core(TM) i3 CPU with 3.8 GB RAM at 2.53GHz and having 3072 KB of L2 cache.

A series of executions were made and the third of a consecutive set of consistent readings is noted in the tables that follow. 'A' stands for our implementation and 'B' stands for the non-blocking BST by faith Ellen.



N = 20						Average per thread time		Max time per thread	
Total time						A	B	A	B
S.No	I	D	S	A	B	A	B	A	B
1	33	33	34	133	150	6	8	96	60
2	18	2	80	140	223	7	11.2	100	110
3	8	2	90	144	120	7.2	6	105	50
4	80	10	10	138	200	6.5	10	68	50

N = 60						Average per thread time		Max time per thread	
Total time						A	B	A	B
S.No	I	D	S	A	B	A	B	A	B
1	33	33	34	170	450	2.8	9.0	133	150
2	18	2	80	180	590	3.0	10	110	98
3	8	2	90	165	460	2.9	7.5	105	40
4	80	10	10	170	600	10	2.9	50	68

N = 100						Average per thread time		Max time per thread	
Total time						A	B	A	B
S.No	I	D	S	A	B	A	B	A	B
1	33	33	34	170	1000	1.7	10	107	80
2	18	2	80	174	930	1.74	9.3	105	6350
3	8	2	90	170	824	1.7	8.24	105	53
4	80	10	10	164	648	1.64	6.48	67	56

Fig 6.4.1 Performance comparison with values of  
N = 20, 60 and 100

The following measurements have been made – program execution time, average per-thread execution time and maximum thread execution time. These wall times are measured using Java's in-built functions. I,D,S are the percentage of insertions, deletions and searches. Rows 2 and 3 are the best indications of performance in a practical scenario.

Trends and explanations:

1. The total execution time of A is almost always less than B and Average per-thread execution time of A is less than B. As the number of threads increases, the difference in execution times becomes more significant. This difference can be accounted because of the fact that insertions and deletions do not do as many flag checks as they do in B and other implementations derived from B. Also there is no retrying in A.
2. The maximum run-time for any thread in A is almost always greater than B. This is because if a thread keeps losing consensus at the left or right child, there is no helper thread in this case. Statistically it is possible that there is at least one thread which faces this condition.

## 6.6 Future Scope

One important goal is to extend this implementation to provide a guarantee of logarithmic height. In sequential implementations, lock-based versions, many techniques have emerged, but working with the one-time write restriction will prove to be more challenging. Theoretical analysis is needed to provide bounds on the number of times a thread might fail consensus while trying to insert at a left or right child. Also, more experimental work is required to benchmark this implementation against other practical implementations available for a dictionary abstract data type.

## 7. Conclusion

Starting with the basic properties of shared objects like liveness and safety, we moved towards the design of concurrent data structures like heap, binary search tree and skip list. Then we presented the universal constructions to convert a sequential object to a lock-free and wait-free linearizable concurrent object. Using these basic constructions, we optimize based on the specific needs of binary search trees and emerged with a novel wait-free binary search tree algorithm and its implementation that performs better than non-blocking counterparts both space and time complexity-wise.

## 8. Bibliography

[1] M. Herlihy, N. Shavit. The Art of Multiprocessor Programming

- [2] F. Ellen, P. Fatourou, E. Ruppert, F. V. Breugel, Non-blocking Binary Search Trees. *In Proc. 29th ACM Symp. Principles of Distributed Computing 2010*
- [3] M. Herlihy, J. Wing. Linearization: A Correctness Condition for Concurrent Object. *In Proc. ACM transactions on programming Languages and Systems, Vol. 2, No. 3, July 1990*
- [4] V. K. Garg. Concurrent and Distributed Computing
- [5] M. Moir, N. Shavit. Concurrent Data Structures
- [6] N. G. Bronson , J. Casper , H. Chafi , K. Olukotun . A Practical Concurrent Binary Search Tree. *In Proc. of the 15th ACM SIGPLAN Symposium on Principals and Practice of Parallel Programming, January, 2010*
- [7] M. Herlihy , Y. Lev, V. Luchangco, N. Shavit. A Provably Correct Scalable Concurrent Skip List
- [8] H. Attiya, N. Lynch, N. Shavit. Are Wait Free Algorithms Fast ?. *In Proc. of the 31st Annual Symposim on Foundations of Computer Science (St. Louis, Mo., Oct.). IEEE, New York, 1990, pp. 55-64.*

## 9 . Appendix

### 9.0 Classes used in Wait-free Binary Search Tree

