Properties of Contention Adapting Search Trees

Kjell Winblad

Department of Information Technology, Uppsala University, Sweden

Abstract. Contention adapting search trees (CA trees) is a family of concurrent data structures that can be used to represent ordered sets and maps. This technical report contains pseudocode for the CA tree operations as well as detailed proof sketches for the properties that they provide, namely deadlock freedom, livelock freedom and linearizability.

1 Introduction

The purpose of this technical report is to complement the previous CA tree works ([2] and [3]) with more detailed proofs for the correctness properties deadlock freedom, livelock freedom and linearizability that the CA tree operations maintain. Readers interested in a detailed description of CA trees and the CA tree algorithm for operations that operate on a single element or key such as insert, remove, lookup etc are referred to a previously published work [2]. A detailed description of operations that atomically operate on multiple elements such as bulk insert/delete and range operations can be found in a technical report available online [3]. The pseudocode that is available in these previous works is also available in this technical report for convenience.

2 Properties

The CA tree operations maintain the correctness properties deadlock freedom, livelock freedom and linearizability for which we will present proofs for here.

Definition 1. We make the following definitions:

- We say that a CA tree routing node or base node is inside a CA tree if it is reachable from the root of the CA tree.
- We say that a thread T has validated a base node or routing node N if T has read the valid flag in N and the flag’s value was true.
- A key is said to be inside a base node B if it is inside the sequential data structure rooted in B.
- We define the quiescent state version of a CA tree C at time t as the CA tree created by blocking all threads that are not holding any base node locks of base nodes inside C at time t and continuing executing all threads that are holding base node locks until no base node lock is held by any thread.
- The abstract set represented by a CA tree C at a time point t is defined as the set of all keys that are inside base nodes that are inside the quiescent state version of C at time t.
```c
void statLock(StatLock slock) {
    if (statTryLock(slock)) {
        slock.statistics -= SUCC_CONTRIB;
        return;
    }
    lock(slock.lock);
    slock.statistics += FAIL_CONTRIB;
}
```

Fig. 1: Pseudocode for statistics collecting lock.

- The linearization point for a single-key operation, i.e. insert, remove, get etc (Fig. 2), is anytime while the operation execute lines 15–10.
- The linearization point of a bulk operation, i.e. bulk insert, remove etc, (Fig. 7) is anytime while the operation is holding all base node locks of the base nodes that the operation operate on.
- The linearization point of a range operation (Fig. 8), is anytime while the operation is holding all base node locks of the base nodes that the operation operate on.
- A CA tree satisfies the binary search tree property if all keys in base nodes reachable from the left branch of a routing node \( R \) are smaller then the key in the routing node \( R \) and all keys in base nodes reachable from the right branch of a routing node are greater than or equal to the key in the routing node \( R \).

**Lemma 1.** The following are invariants for the CA tree and its operations:

1. An operation never changes the sequential data structure in a base node or a base node’s valid flag without holding the lock of the base node.
2. When one of the operations have locked a base node \( B \) and just validated \( B \), then \( B \) is inside the tree and \( B \)’s parent is the same as it has been since \( B \) was inserted into the tree.
3. When one of the operations have locked a routing node \( R \) and just validated \( R \), then this routing node is inside the tree.
4. The root pointer of a CA tree, the left and right pointers of a routing node are never changed without first locking the lock in the node that the pointer points to.
5. The execution of the functions for high-contention split (Fig. 3) and low-contention join (Fig. 4) do not change the abstract set represented by the CA tree.
6. If a thread \( T \) has searched for a key \( k \) in a CA tree \( C \) using the binary search tree property and ended up in a base node \( B \) that \( T \) has locked and just validated at time point \( t \), then a search for \( k \) using the binary search tree property in a quiescent state version of \( C \) at time \( t \) would have ended up in \( B \) as well.
7. The quiescent state version of a CA tree at anytime \( t \) provide the binary search tree property. Note that this together with statement 1, 2 and 5 implies that if a search using the binary search tree property for a key \( k \) in the CA tree \( C \) ended up in a valid base node \( B \) which is validated after it has been locked then:
   (a) \( k \) is in the abstract set represented by \( C \) if and only if \( k \) is inside the sequential data structure \( S \) rooted at \( B \) and,
   (b) if the minimum key in \( S \) is \( k_1 \) and the maximum key in \( S \) is \( k_2 \) then the keys in \( [k_1, k_2] \) is in the abstract set represented by \( C \) if and only if they are in \( S \).
Object doOperation(CATree tree, Op operation, Key key) {
  RoutingNode prevNode = null;
  Object currNode = tree.root;
  while (currNode isInstanceOf RoutingNode) {
    prevNode = currNode;
    if (key < currNode.key) currNode = currNode.left;
    else currNode = currNode.right;
  }
  BaseNode base = currNode asInstanceOf BaseNode;
  statLock(base.lock);
  if (base.valid == false) {
    statUnlock(base.lock);
    return doOperation(tree, operation, key); // retry
  } else {
    Object result = operation.execute(base.root, key);
    if (base.lock.statistics > MAX_CONTENTION) {
      if (size(base.root) < 2) base.lock.statistics = 0;
      else highContentionSplit(tree, base, prevNode);
    } else if (base.lock.statistics < MIN_CONTENTION) {
      if (prevNode == null) base.lock.statistics = 0;
      else lowContentionJoin(tree, base, prevNode);
    }
    statUnlock(base.lock);
    return result;
  }
}

Fig. 2: Generic pseudocode for single-element operations like insert, remove lookup etc.

Proof: Statement 1 can easily be verified from inspecting the pseudocode. Statements 2 to 5 can be proven to be invariant by induction. Initially, they trivially hold since a CA tree initially only consists of one base node pointed to by the root pointer. The only statements in the pseudocode that might make statements 2 to 5 false are changes that change pointers in the routing layer or the valid flags of base nodes or routing nodes. All these changes are located in the functions highContentionSplit (Fig. 3) and lowContentionJoin (Fig. 4). If a thread execute one of these functions alone without the possibility of any other thread interfering and assuming that they are true when the function start to execute, then it is easy to verify that all changes in the functions preserve the statements by studying the pseudocode. Furthermore, by assuming that the statements hold before and after lock acquisitions it is also easy to verify that all reads and writes that are critical for the preservation of the truth of the statements are protected by locks. Other threads can thus not interfere with the changes in a way that make them not preserve the truth of the statements, so by induction the statements 2 to 5 are invariants.

The only thing that might affect the truth of statement 6 is that concurrently with the operation whose search that ended up in the validated base node $B$, another operation that holds a lock of at least one base node (which cannot be $B$) is about to change a pointer on the search path to $B$. This is because the routing nodes traversed in the search
void highContentionSplit(CATree tree, BaseNode base, RoutingNode parent) {
    Key splitKey = pickSplitKey(base.root);
    Tuple<Tree> split = splitTree(splitKey, base.root);
    RoutingNode newRoute =
        RouteNode(BaseNode(split.elem1), splitKey, BaseNode(split.elem2));
    if (parent == null) tree.root = newRoute;
    else if (parent.left == base) parent.left = newRoute;
    else parent.right = newRoute;
    base.valid = false;
}

Fig. 3: High-contention adaptation.

would otherwise be identical in both the actual CA tree and its quiescent state version. The only change of this kind is in lowContentionJoin (lines 21–27) which splices out exactly one base node and its parent routing node. It is easy to see that such concurrent changes can not affect the truth of statement 6 since the change only splices out exactly one routing node from the search path to B.

Finally, statement 7 can be proven by induction. Initially, statement 7 trivially holds since a CA tree initially only consists of one base node pointed to by the root pointer. It is easy to see that the only changes that can affect the truth of statement 7 are changes to the sequential data structures of base nodes, the root pointer, the left and right pointers of routing nodes. Assuming that statement 7 holds before such changes are done in highContentionSplit (Fig. 3) and lowContentionJoin (Fig. 4) it is easy to see from the pseudocode and by relying on statement 1 to 5 (which we have already shown are invariants) that the execution of these functions preserves the truth of statement 7.

Again, assuming that statement 7 holds before single-key operations (Fig. 2) or bulk operations (Fig. 7) that remove or insert keys to the sequential data structures of base nodes, it follows naturally by using statement 6 and the definitions of the linearization points that these changes also preserve the truth of statement 7. To see that range operations (Fig. 8) also preserve the truth of statement 7 assuming it is true before the range operations insert or removes any keys to base nodes, note that: (i) By the binary search tree property, the base node B_2 that can contain the smallest key that is greater than the greatest key that a locked base node B_1 can contain is located in the leftmost base node in the first routing node on the reverse path to B_1 from the root that has a right pointer that does not lead to B_1. (ii) Also by the binary search tree property, if the parent routing node R of the base node B has key k and another routing node then R on the path to B has a key greater than k, then R must have a right pointer from which B is not reachable. Using the knowledge stated above it is easy to verify that the pseudocode for range operations (Fig. 8) only perform an operation for a specific key in a sequential data structure of a base node if this does not violate statement 7 assuming that statement 7 holds just before the modification is performed.
Theorem 1. All operations appear to happen instantly to the abstract set represented by a CA tree at the time of their linearization points (See Definition 1) and are thus linearizable.

Proof: By Lemma 1 and the structure of the code it follows that when an operation adds, removes or lookups a key in a sequential data structure $S$ of one of the base nodes, then this key can not be in the sequential data structure of another base node and the key is inside $S$ if and only if it is in the abstract set represented by the CA tree at the time of the linearization point. Furthermore, an operation is always holding the locks of all base nodes which sequential data structures it is using to perform the operation, thus preventing any other operations to notice any intermediate steps of the operation. We can therefore conclude that all CA tree operations are linearizable.

Theorem 2. The CA tree operations are deadlock free.

We will show that the CA tree operations are deadlock free by showing that all operations either obtain locks in a specific order so a deadlock cannot occur, or prevent a deadlock situation by using tryLock which, if unsuccessful, is followed by the release of the currently held locks.

We will first prove that a call to the function $\text{lowContentionJoin}$ (Fig. 4) cannot cause a deadlock. We first note that everywhere $\text{lowContentionJoin}$ is called the lock of the base node given as parameter is held, the caller holds no other CA tree locks and the base node lock that is held is released after the call to $\text{lowContentionJoin}$ has returned. Operations that call $\text{lowContentionJoin}$ can use $\text{tryLock}$ (Fig. 4, line 5) to lock another base node. If the $\text{tryLock}$ is unsuccessful, $\text{lowContentionJoin}$ will return and the currently held lock will be released (e.g. Fig. 2, line 23). Also, $\text{lowContentionJoin}$ is the only function that acquires locks in the routing nodes. Routing nodes are always locked after the base node locks. $\text{lowContentionJoin}$ always acquires the parent routing node’s lock before the grandparent routing node’s lock, so locking of routing node locks are ordered by the distance to the root of the tree. (Since no operation ever holds two routing node locks that are at the same level, it is not a problem that there is no order between routing nodes at the same level.)

Except $\text{lowContentionJoin}$ which we have already proven cannot cause a deadlock, the only functions that can hold more than one base node locks are $\text{doBulkOp}$ (Fig. 7) and $\text{rangeOp}$ (Fig. 8). We will now prove that these functions always lock base nodes that are inside the tree in the left to right order (when depicted as in Fig. 9) and can thus not cause a deadlock situation. They both only hold the lock of at most one invalid base node since they immediately unlock a base node that is invalid after it has been locked, so we only need to consider base nodes that are inside the CA tree.

The function $\text{doBulkOp}$ sorts the keys (Fig. 7, line 7) so smaller keys are before larger keys. Therefore, by statement 6 and 7 of Lemma 1 it is easy to see that a valid base node that is locked on line 15 is ordered after all base nodes that the function already holds the lock for.

The function $\text{rangeOp}$ (Fig. 8) finds the next base node to lock in the subtree rooted at the right branch of the first routing node on the reverse path to the previously locked base node that does not contain the previously locked base node (cf. the proof of Theorem 1). Therefore, by statement 6 and 7 of Lemma 1 and the fact that no routing node can be
added to the search path from the root to a locked and valid base node, it is easy to see that base nodes that are locked by rangeOp are ordered after all base nodes that the operation has previously locked.

It can therefore be concluded that the CA tree operations are deadlock free since all locks are acquired in a specific order or otherwise the lock is acquired by a tryLock call and all currently held locks are released if the tryLock fails.

**Theorem 3.** The CA tree operations are livelock free.

A livelock occurs when threads perform some actions that interfere with each other so that none of them makes any actual progress. There are only two situations when CA tree operations need to redo some steps because of interference from other threads:

- A thread needs to retry an operation or part of an operation if an invalid base node is seen. The interfering thread must have completed an operation in this case. Otherwise no split or join could have happened. Furthermore, since a base node or routing node is made invalid and linked out from the tree while it is locked, a search will never end up in the same invalid base node when it is retried. For example, consider the case when the search for the next base node in rangeOp (Fig. 8) ends up in an invalid base node $B$ because $B$ and its parent $R$ has been spliced out from the tree and $R$ was previously on the path to the previously locked base node. Then when the search for the next base node to lock is retried it will not end up in $B$ again because of the validity check on line 17 (Fig. 6).
- Similarly, if the code in Fig. 4 (lines 15–20) needs to be retried to find the grandparent of a base node, another thread must have spliced out a routing node and has thus made progress.

The CA tree operations are therefore livelock free.

**References**

1. CA Trees. [http://www.it.uu.se/research/group/languages/software/ca_tree](http://www.it.uu.se/research/group/languages/software/ca_tree).
void lowContentionJoin(CATree tree, BaseNode base, RoutingNode parent) {
    if (parent.left == base) {
        BaseNode neighborBase = leftmostBaseNode(parent.right);
        if (!statTryLock(neighborBase.lock)) {
            base.lock.statistics = 0;
            } else if (!neighborBase.valid) {
                statUnlock(neighborBase.lock);
                base.lock.statistics = 0;
            } else {
                lock(parent.lock);
                parent.valid = false;
                neighborBase.valid = false;
                base.valid = false;
                RoutingNode gparent = null; // gparent = grandparent
                do {
                    if (gparent != null) unlock(gparent.lock);
                    gparent = parentOf(parent, tree);
                    if (gparent != null) lock(gparent.lock);
                } while (gparent != null && !gparent.valid);
                if (gparent == null) {
                    tree.root = parent.right;
                } else if (gparent.left == parent) {
                    gparent.left = parent.right;
                } else {
                    gparent.right = parent.right;
                }
            unlock(parent.lock);
            if (gparent != null) unlock(gparent.lock);
            BaseNode newNeighborBase =
                BaseNode(joinTrees(base.root, neighborBase.root));
            RoutingNode neighborBaseParent = null;
            if (parent.right == neighborBase) neighborBaseParent = gparent;
            else neighborBaseParent = leftmostRouteNode(parent.right);
            if (neighborBaseParent == null) {
                tree.root = newNeighborBase;
            } else if (neighborBaseParent.left == neighborBase) {
                neighborBaseParent.left = newNeighborBase;
            } else {
                neighborBaseParent.right = newNeighborBase;
            }
            statUnlock(neighborBase.lock);
        }
    } else if (statUnlock(neighborBase.lock)) {
        base.lock.statistics = 0;
    } else {
        /* This case is symmetric to the previous one */
    }

Fig. 4: Low-contention adaptation.
```java
void manageCont(BaseNode base, boolean contended) {
    if (contended) base.lock.statistics += FAIL_CONTRIB;
    else base.lock.statistics -= SUCC_CONTRIB;
    if (base.lock.statistics > MAX_CONTENTION) {
        if (size(base.root) < 2) base.lock.statistics = 0;
        else highContentionSplit(tree, base, base.parent);
    } else if (base.lock.statistics < MIN_CONTENTION) {
        if (base.parent == null) base.lock.statistics = 0;
        else lowContentionJoin(tree, base, base.parent);
    }
}
```

Fig. 5: Manage contention.

```java
BaseNode, List<RouteNode>
getNextBaseNodeAndPath(BaseNode b, List<RouteNode> p) {
    List<RouteNode> newPathPart;
    BaseNode bRet;
    if (p.isEmpty()) {
        // The parent of b is the root
        return null, null;
    } else {
        List<RouteNode> rp = p.reverse();
        if (rp.head().left == b) {
            bRet, newPathPart =
            leftmostBaseNodeAndPath(rp.head().right);
            return bRet, p.append(newPathPart);
        } else {
            K pKey = rp.head().key; // pKey = key of parent
            rp.removeFirst();
            while (rp notEmpty()) {
                if (rp.head().valid && pKey <= rp.head().key){
                    bRet, newPathPart =
                    leftmostBaseNodeAndPath(rp.head().right);
                    return bRet, rp.reverse().append(newPathPart);
                } else {
                    rp.removeFirst();
                }
            }
        }
        return null, null;
    }
}
```

Fig. 6: Find next base node.
K[] doBulkOp(CATree tree, Op op, K[] keys) {
    keys = keys.clone();
    values = values.clone();
    K[] returnArray = new K[keys.size];
    boolean first = true;
    boolean firstContended = true;
    sort(keys);
    Stack<BaseNode> lockedBaseNodes = new Stack<>
    int i = 0; while( i < keys.size ){
        find_base_node_for_key:
        BaseNode baseNode = getBaseNode(tree, keys[i]);
        if(baseNode != lockedBaseNodes.top){
            if(first) {
                firstContended = baseNode.lockIsContended();
            } else baseNode.lockNoStats();
            if (!baseNode.valid) {
                baseNode.unlock();
                goto find_base_node_for_key; // retry
            } else {
                first = false;
            }
            lockedBaseNodes.push(baseNode);
        }
        returnArray[i] = op.execute(baseNode.root, keys[i]);
        i++; 
        K baseNodeMaxKey = baseNode.maxKey();
        while(keys[i] <= baseNodeMaxKey){
            returnArray[i] = op.execute(baseNode.root, keys[i]);
            i++; 
        }
    }
    BaseNode[] lockedBaseNodesArray = lockedBaseNodes.toArray();
    if( lockedBaseNodes.size() == 1 ) {
        manageCont(lockedBaseNodesArray[0], firstContended);
        lockedBaseNodesArray[0].unlock();
    } else {
        for(int i = 0; i < lockedBaseNodes.size(); i++){
            if( i == (lockedBaseNodes.size() -1 ) ) {
                manageCont(lockedBaseNodesArray[0], false);
            } else baseNode.lock.statistics -= SUCC_CONTRIB;
            baseNode.unlock();
        }
    }
    return returnArray;
}

Fig. 7: Bulk operations.
```java
K[] rangeOp(CATree tree, Op op, K lo, K hi)
{
    List<RouteNode> path;
    Stack<BaseNode> lockedBaseNodes = new Stack<>();
    fetch_first_node: baseNode, path = getBaseNodeAndPath(lo);
    boolean firstContended = baseNode.lockIsContended();
    if (!baseNode.valid) {
        baseNode.unlock();
        goto fetch_first_node;
    }
    while (true) {
        lockedBaseNodes.push(baseNode);
        K baseNodeMaxKey = baseNode.maxKey();
        if (baseNodeMaxKey != null && hi < baseNodeMaxKey) {
            break; // All needed base nodes are locked
        }
        BaseNode lastLockedBaseNode = baseNode;
        List<RouteNode> pathBackup = path.clone();
        search_next_base_node:
        baseNode, path = getNextBaseNodeAndPath(lastLockedBaseNode, path);
        if (baseNode == null) {
            break; // All needed base nodes are locked
        }
        baseNode.lockNoStats();
        tryAgain = !baseNode.valid;
        if (tryAgain) {
            baseNode.unlock();
            path = pathBackup;
            goto search_next_base_node;
        }
    }
    Buffer<K> retBuff = Buffer<K>;
    BaseNode[] lockedBaseNodeArray = lockedBaseNodes.toArray();
    if (lockedBaseNodeArray.size() == 1) {
        retBuff.add(performOpToKeysInRange(baseNode, lo, hi, op));
        manageCont(lockedBaseNodeArray[0], firstContended);
        lockedBaseNodeArray[0].unlock();
    } else {
        for (int i = 0; i < lockedBaseNodeArray.size(); i++) {
            baseNode = lockedBaseNodeArray[i];
            retBuff.add(performOpToKeysInRange(baseNode, lo, hi, op));
            if (i == (lockedBaseNodeArray.size() - 1)) {
                manageCont(lockedBaseNodeArray[0], false);
            } else baseNode.lock.statistics -= SUCC_CONTRIB;
            baseNode.unlock();
        }
    }
    return retBuff.toArray();
}
```

Fig. 8: Range operations.
Fig. 9: The structure of a CA tree. Numbers denote keys, a node whose flag is valid is marked with a green hook; an invalid one with a red cross.