CONCURRENT BINARY SEARCH TREES: DESIGN AND OPTIMIZATIONS

by

Arunmoezhi Ramachandran

APPROVED BY SUPERVISORY COMMITTEE:

Neeraj Mittal, Chair

Balaji Raghavachari

Venkatesan Subbarayan

Rob F Van Der Wijngaart

To my parents and sister

CONCURRENT BINARY SEARCH TREES: DESIGN AND OPTIMIZATIONS

by

ARUNMOEZHI RAMACHANDRAN, BE, MS

DISSERTATION

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY IN

COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT DALLAS

May 2016

ACKNOWLEDGMENTS

Ack page

December 2015

PREFACE

This dissertation was produced in accordance with guidelines which permit the inclusion as part of the dissertation the text of an original paper or papers submitted for publication. The dissertation must still conform to all other requirements explained in the "Guide for the Preparation of Master's Theses and Doctoral Dissertations at The University of Texas at Dallas." It must include a comprehensive abstract, a full introduction and literature review, and a final overall conclusion. Additional material (procedural and design data as well as descriptions of equipment) must be provided in sufficient detail to allow a clear and precise judgment to be made of the importance and originality of the research reported.

It is acceptable for this dissertation to include as chapters authentic copies of papers already published, provided these meet type size, margin, and legibility requirements. In such cases, connecting texts which provide logical bridges between different manuscripts are mandatory. Where the student is not the sole author of a manuscript, the student is required to make an explicit statement in the introductory material to that manuscript describing the student's contribution to the work and acknowledging the contribution of the other author(s). The signatures of the Supervising Committee which precede all other material in the dissertation attest to the accuracy of this statement.

CONCURRENT BINARY SEARCH TREES: DESIGN AND OPTIMIZATIONS

Publication No. ____________________

Arunmoezhi Ramachandran, PhD
The University of Texas at Dallas, 2016

Supervising Professor: Neeraj Mittal

abstract goes here

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

LOCK BASED CONCURRENT BINARY SEARCH TREES

1.1 The Lock-Based Algorithm

We first provide an overview of our algorithm. We then describe the algorithm in more detail and also give its pseudo-code. For ease of exposition, we describe our algorithm assuming no memory reclamation, which can be performed using the well-known technique of hazard pointers (**?**).

1.1.1 Overview of the Algorithm

Every operation in our algorithm uses *seek* function as a subroutine. The seek function traverses the tree from the root node until it either finds the target key or reaches a non-binary node whose next edge to be followed points to a null node. We refer to the path traversed by the operation during the seek as the *access-path*, and the last node in the access-path as the *terminal node*. The operation then compares the target key with the stored key (the key present in the terminal node). Depending on the result of the comparison and the type of the operation, the operation either terminates or moves to the execution phase. In certain cases in which a key may have moved upward along the access-path, the seek function may have to restart and traverse the tree again; details about restarting are provided later. We now describe the next steps for each of the type of operation one-by-one.

Search: A search operation starts by invoking seek operation. It returns `true` if the stored key matches the target key and `false` otherwise.

Insert: An insert operation starts by invoking seek operation. It returns false if the target key matches the stored key; otherwise, it moves to the execution phase. In the execution phase, it attempts to insert the key into the tree as a child node of the last node in the access-path using a CAS instruction. If the instruction succeeds, then the operation returns true; otherwise, it restarts by invoking the seek function again.

Delete: A delete operation starts by invoking seek function. It returns false if the stored key does not match the target key; otherwise, it moves to the execution phase. In the execution phase, it attempts to remove the key stored in the terminal node of the access-path. There are two cases depending on whether the terminal node is a binary node (has two children) or not (has at most one child). In the first case, the operation is referred to as *complex delete operation*. In the second case, it is referred to as *simple delete operation*. In the case of simple delete, the terminal node is removed by changing the pointer at the parent node of the terminal node. In the case of complex delete, the key to be deleted is replaced with the *next largest* key in the tree, which will be stored in the *leftmost node* of the *right subtree* of the terminal node.

1.1.2 Details of the Algorithm

As in most algorithms, to make it easier to handle special cases, we use sentinel keys and sentinel nodes. The structure of an empty tree with only sentinel keys (denoted by ∞_1 and ∞_2 with $\infty_1 < \infty_2$) and sentinel nodes (denoted by $\mathbb{R}$ and $\mathbb{S}$) is shown in Figure 1.1.

Our algorithm, like the one in (Natarajan and Mittal, 2014), operates at edge level. A delete operation obtains ownership of the edges it needs to work on by locking them. To enable locking of an edge, we steal a bit from the child addresses of a node referred to as *lock-flag*. We also steal another bit from the child addresses of a node to indicate that the node is undergoing deletion and will be removed from the tree. We denote this bit by

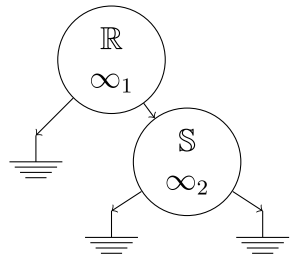

Figure 1.1. Sentinel keys and nodes ($\infty_1 < \infty_2$)

mark-flag. Finally, to avoid the ABA problem, as in Howley and Jones (Howley and Jones, 2012), we use *unique* null pointers. To that end, we steal yet another bit from the child address, referred to as *null-flag*, and use it to indicate whether the address points to a null or a non-null address. So, when an address changes from a non-null value to a null value, we only set the null-flag and the contents of the address are not otherwise modified. This ensures that all null pointers are unique.

We next describe the details of the seek operation, which is executed by all operations (search as well as modify) after which we describe the details of the execution phase of insert and delete operations.

The Seek Phase

A seek function keeps track of the node in the access-path at which it took the last “right turn” (*i.e.*, it last followed a right edge). Let this “right turn” node be referred to as *anchor node* when the traversal reaches the terminal node. Note that the terminal node is the node whose key matched the target key or whose next child edge is set to a null address. For an illustration, please see Figure 1.2. In the latter case (stored key does not match the target key), it is possible that the key may have moved up in the tree. To ascertain that the seek function did not miss the key because it may have moved up during the traversal, we use the following set of conditions that are *sufficient* (but not necessary) to guarantee that the seek function did not miss the key. First, the anchor node is still part of the tree. (For an illustration, see Figure 1.3) Second, the key stored in the anchor node has not changed since it first encountered the anchor node during the (current) traversal. To check for the above two conditions, we determine whether the anchor node is undergoing deletion by examining it right child edge. We discuss the two cases one-by-one.

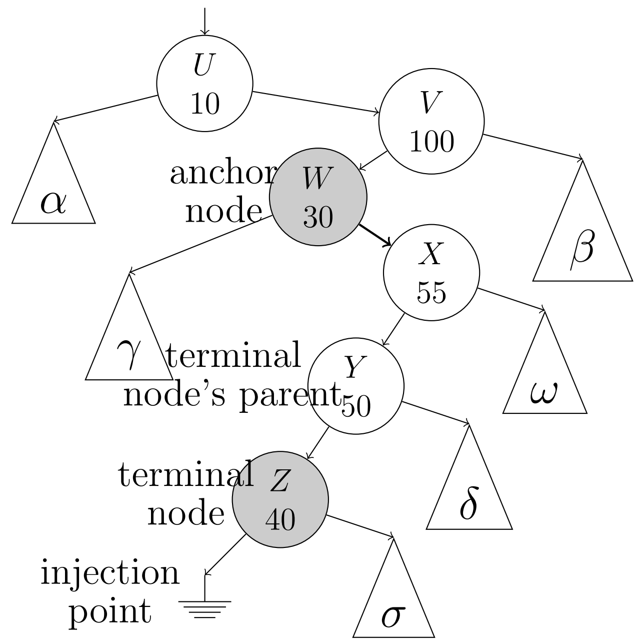

Figure 1.2. Nodes in the access path of seek

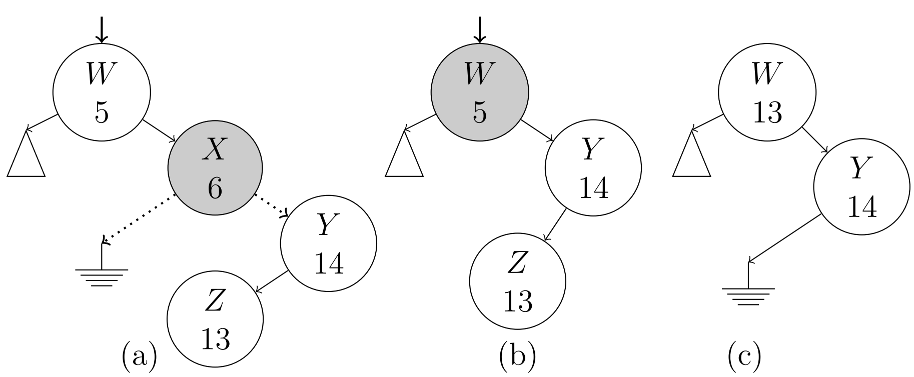

Figure 1.3. A scenario in which the last right turn node is no longer part of the tree

(a) *Right child edge not marked:* In this case, the anchor node is still part of the tree. We next check whether the key stored in the anchor node has changed. If the key has not changed, then the seek function returns the results of the (current) traversal, which consists of three addresses: (i) the address of the terminal node, (ii) the address of its parent, and (iii) the null address stored in the child field of the terminal node that caused the traversal to terminate. The last address is required to ensure that an insert operation works correctly (specifically to ascertain that the child field of the terminal node has not undergone any change since the completion of the traversal). We refer to it as the *injection point* of the insert operation. On the other hand, if the key has changed, then the seek function restarts from the root of the tree. A possible optimization is that the seek function restarts only if the target key is now less than the anchor node's key.

(b) *Right child edge marked:* In this case, we compare the information gathered in the current traversal about the anchor node with that in the previous traversal, if one exists. Specifically, if the anchor node of the previous traversal is same as that of the current traversal and the keys found in the anchor node in the two traversals also match, then the seek function terminates, but returns the results of the previous traversal (instead of that of the current traversal). This is because the anchor node was definitely part of the tree during the previous traversal since it was reachable from the root of the tree at the beginning of the current traversal. Otherwise, the seek function restarts from the root of the tree.

For insert and delete operations, we refer to the terminal node as the *target node*.

The Execution Phase of an Insert Operation

In the execution phase, an insert operation creates a new node containing the target key. It then adds the new node to the tree at the injection point using a CAS instruction. For an illustration, see Figure 2.2. If the CAS instruction succeeds, then (the new node becomes a part of the tree and) the operation terminates; otherwise, the operation restarts from the seek phase. Note that the insert operations are lock-free.

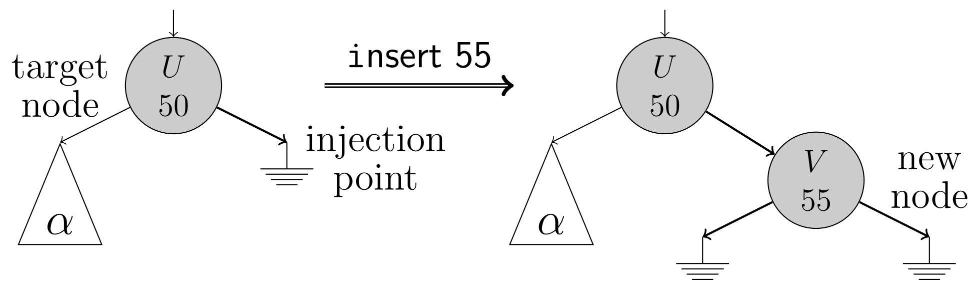

Figure 1.4. An illustration of an insert operation

The Execution Phase of a Delete Operation

The execution of a delete operation starts by checking if the target node is a binary node or not. If it is a binary node, then the delete operation is classified as complex; otherwise it is classified as simple.

For a tree node X, let $X.parent$ denote its parent node, and $X.left$ and $X.right$ denote its left and right child node, respectively. Also, hereafter in this section, let T denote the target node of the delete operation under consideration.

(a) *Simple Delete:* In this case, either $T.left$ or $T.right$ is pointing to a null node. Note that both $T.left$ and $T.right$ may be pointing to null nodes in which case T will be a leaf node. Without loss of generality, assume that $T.right$ is a null node. The removal of T involves locking the following three edges: $\langle T.parent, T\rangle$, $\langle T, T.left\rangle$ and $\langle T, T.right\rangle$. For an illustration, see Figure 2.3.

A lock on an edge is obtained by setting the lock-flag in the appropriate child field of the parent node using a CAS instruction. For example, to lock the edge $\langle X, Y\rangle$, where Y is the left child of X, the lock-flag in the left child of X is set to one. If all the edges are locked successfully, then the operation validates that the key stored in the

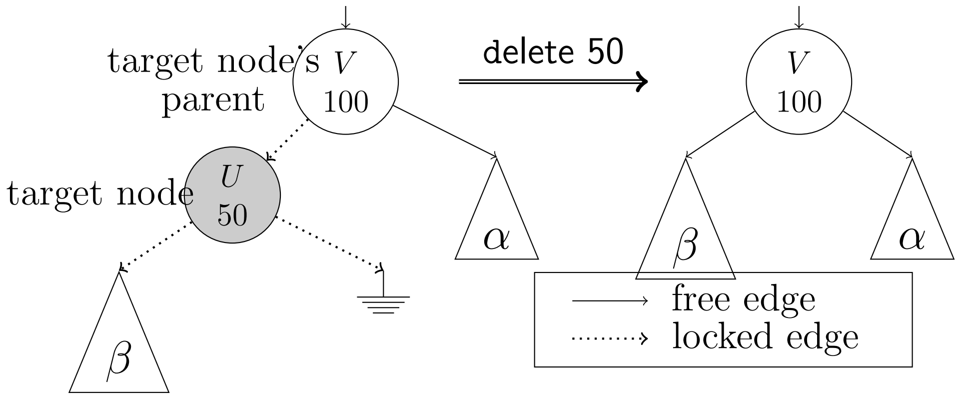

Figure 1.5. An illustration of a simple delete operation

target node still matches the target key. If the validation succeeds, then the operation marks both the children edges of T to indicate that T is going to be removed from the tree. Next, it changes the child pointer at $T.parent$ that is pointing to T to point to $T.left$ using a simple write instruction. Finally, the operation releases all the locks and returns true.

(b) *Complex Delete:* In this case, both $T.left$ and $T.right$ are pointing to non-null nodes. The operation locates the next largest key in the tree, which is the smallest key in the subtree rooted at the right child of T. We refer to this key as the *successor key* and the node storing this key as the *successor node*. Hereafter in this section, let S denote the successor node. Deletion of the key stored in T involves copying the key stored in S to T and then removing S from the tree. To that end, the following edges are locked by setting the lock-flag on the edge using a CAS instruction: $\langle T, T.right\rangle$, $\langle S.parent, S\rangle$, $\langle S, S.left\rangle$ and $\langle S, S.right\rangle$. For an illustration, see Figure 2.4. Note that the first two edges may be same which happens if the successor node is the right child of the target node. Also, since we do not lock the left edge of the target node, the left edge may change and may possibly start pointing to a null address. But, that does not impact the correctness of the complex delete operation.

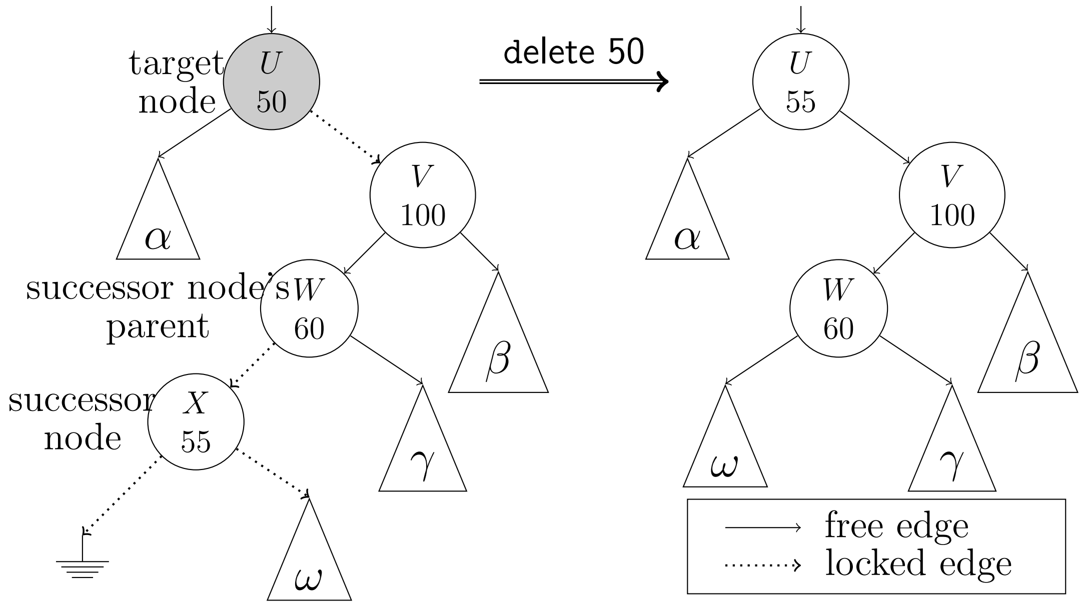

Figure 1.6. An illustration of a complex delete operation

If all the edges are locked successfully, then the operation validates that the key stored in the target node still matches the target key. If the validation succeeds, then the operation copies the key stored in S to T, and marks both the children edges of S to indicate that S is going to be removed from the tree. Next, it changes the child pointer at $S.parent$ that is pointing to S to point to $S.right$ using a simple write instruction. Finally, the operation releases all the locks and returns true.

In both cases (simple as well as complex delete), if the operation fails to obtain any of the locks, then it releases all the locks it was able to acquire up to that point, and restarts from the seek phase. Also, after obtaining all the locks, if the key validation fails, then it implies that some other delete operation has removed the key from the tree while the current execution phase was in progress. In that case, the given delete operation releases all the locks, and simply returns false. Note that using a CAS instruction for setting the lock-flag also enables us to *validate that the child pointer has not changed* since it was last observed in a single step.

Algorithm 1: Data Structures Used

```
// a tree node
struct Node {
    Key key;
    { boolean, boolean, boolean, NodePtr } child[2];
    // each child field contains four subfields: lFlag, mFlag, nFlag and address
};
// used to store the results of a tree traversal
struct SeekRecord {
    NodePtr node;
    NodePtr parent;
    NodePtr nullAddress;
};
// used to store information about an anchor node
struct AnchorRecord {
    NodePtr node;
    Key key;
};
// used to store information about an edge to lock
struct LockRecord {
    NodePtr node;
    enum { LEFT, RIGHT } which;
    { boolean, NodePtr } addressSeen;
    // addressSeen contains two subfields: nFlag and address
};
// local seek record used when looking for a node
SeekRecordPtr seekTargetKey, seekSuccessorKey;
// local array used to store the set of edges to lock
LockRecord lockArray[4];
```

1.1.3 Formal Description

We refer to our algorithm as CASTLE (Concurrent Algorithm for Binary Search Tree by Locking Edges).

A pseudo-code of our algorithm is given in Algorithms 8-7. Different data structures used in our algorithm are shown in Algorithm 8. Besides tree node, we use three additional records: (a) *seek record:* to store the outcome of a tree traversal both when looking for the target key and the successor key, (b) *anchor record:* to store information about the anchor node during the seek phase, and (c) *lock record:* to store information about a tree edge that needs to be locked.

Algorithm 2: Seek Function

```
Seek( key, seekRecord )
begin
    while true do
        // initialize the variables used in the traversal
        pNode := ℝ;        cNode := 𝕊;
        address := 𝕊→child[RIGHT].address;
        anchorRecord := {ℝ, ∞₁};
        while true do
            // reached terminal node; read the key stored in the current node
            cKey := cNode→key;
            if key = cKey then
                seekRecord := {cNode, pNode, address};
                return;
            which := key < cKey ? LEFT : RIGHT;
            // read the next address to dereference along with mark and null flags
            ⟨*, *, nFlag, address⟩ := cNode→child[which];
            if nFlag then                    // the null flag is set; reached terminal node
                aNode := anchorRecord→node;
                if aNode→child[RIGHT].mFlag then
                    // the anchor node is marked; it may no longer be part of the tree
                    if anchorRecord = pAnchorRecord then
                        // the anchor record of the current traversal matches that of
                           the previous traversal
                        seekRecord := pSeekRecord;
                        return;
                    else break;
                else                         // the anchor node is definitely part of the tree
                    if aNode→key < key then                    // seek can terminate now
                        seekRecord := {cNode, pNode, address};
                        return;
                    else break;

            // update the anchor record if needed
            if which = RIGHT then
                // the next edge to be traversed is a right edge; keep track of
                   current node and its key
                anchorRecord := {cNode, cKey};

            // traverse the next edge
            pNode := cNode;        cNode : =address;
```

Algorithm 3: Search Operation

```
boolean Search( key )
begin
    Seek( key, seekTargetKey );
    node := seekTargetKey→node;
    if node→key = key then
        return true;                                        // key found
    else
        return false;                                       // key not found
```

Algorithm 4: Insert Operation

```
boolean Insert( key )
begin
    while true do
        Seek( key, seekTargetKey );
        node := seekTargetKey→node;
        if node→key = key then
            return false;                                   // key found
        else
            // key not found; add the key to the tree
            newNode := create a new node;
            // initialize its fields
            newNode→key := key;
            newNode→child[LEFT] := ⟨0_l, 0_m, 1_n, null⟩;
            newNode→child[RIGHT] := ⟨0_l, 0_m, 1_n, null⟩;
            // determine which child field (left or right) needs to be modified
            which := key < node→key ? LEFT : RIGHT;
            // fetch the address observed by the seek function in that field
            address := seekTargetKey→nullAddress;
            result := CAS( node→child[which],
                           ⟨0_l, 0_m, 1_n, address⟩,
                           ⟨0_l, 0_m, 0_n, newNode⟩ );
            if result then
                // new key successfully added to the tree
                return true;
```

Algorithm 5: Delete Operation

```
boolean DELETE( key )
begin
    while true do
        SEEK( key, seekTargetKey );
        node := seekTargetKey→node;
        if node→key ≠ key then                                    // key not found
            return false;
        else                  // key found; read contents of target node's children fields
            lField := CLEARFLAGS( node→child[LEFT] );
            rField := CLEARFLAGS( node→child[RIGHT] );
            if lField.nFlag  or  rField.nFlag then                // simple delete operation
                parent := seekTargetKey→parent;
                if key < parent→key then  which := LEFT;
                else  which := RIGHT;
                lockArray[0] := {parent, which, ⟨0, node⟩};
                lockArray[1] := {node, LEFT, lField};
                lockArray[2] := {node, RIGHT, rField};
                result := LOCKALL( lockArray, 3 );
                if result then                 // all locks acquired; perform the operation
                    if node→key = key then         // key still matches; remove the node
                        REMOVECHILD( parent, which ); match := true;
                    else  match := false;
                    UNLOCKALL( lockArray, 3 );
                    return match;
            else                  // complex delete operation; locate the successor node
                FINDSMALLEST(node, rField.address, seekSuccessorKey);
                sNode := seekSuccessorKey→node; sParent := seekSuccessorKey→parent;
                // determine the edges to be locked
                lockArray[0] := {node, RIGHT, rField};
                if node ≠ sParent then
                    // successor node is not the right child of target node
                    lockArray[1] := {sParent, LEFT, ⟨0, sNode⟩}; size := 4;
                else  size := 3 ;
                lField := CLEARFLAGS( sNode→child[LEFT] );
                rField := CLEARFLAGS( sNode→child[RIGHT] );
                lockArray[size − 2] := {sNode, LEFT, lField};
                lockArray[size − 1] := {sNode, RIGHT, rField};
                result := LOCKALL( lockArray, size );
                if result then                 // all locks acquired; perform the operation
                    if node→key = key then
                        // key still matches; copy key in successor node to target node
                        node→key := sNode→key;
                        REMOVECHILD( sParent, LEFT ); match := true;
                    else  match := false;
                    UNLOCKALL( lockArray, size );
                    return match;
```

Algorithm 6: Lock and Unlock Functions

```
boolean LockAll( lockArray, size )
begin
    for i ← 0 to size − 1 do
        // acquire lock for the i-th entry
        node := lockArray[i].node;
        which := lockArray[i].which;
        lockedAddress := lockArray[i].addressSeen;
        lockedAddress.lFlag := true;
        // set the lock flag in the child edge
        result := CAS(node→child[which], lockArray[i].addressSeen, lockedAddress);
        if not (result) then
            // release all the locks acquired so far
            UnlockAll( lockArray, i − 1 );
            return false;
    return true;

UnlockAll( lockArray, size )
begin
    for i ← size − 1 to 0 do
        node := lockArray[i].node;
        which := lockArray[i].which;
        // clear the lock flag in the child edge
        node→child[which].lFlag := false;
```

The pseudo-code for the seek function is shown in Algorithm 9. The pseudo-codes for search, insert and delete operations are shown in Algorithm 10, Algorithm 11 and Algorithm 12, respectively. Algorithm 6 contains the pseudo-code for locking and unlocking a set of tree edges, as specified in an array. Finally, Algorithm 7 contains the pseudo-codes for three helper functions used by a delete operation, namely: (a) ClearFlags: to clear lock and mark flags from a child field, (b) FindSmallest: to locate the smallest key in a subtree, and (c) RemoveChild: to remove a given child of a node.

In the pseudo-code, to improve clarity, we sometimes use subscripts l, m and n to denote lock, mark and null flags, respectively.

Algorithm 7: Helper Functions used by Delete Operation

```
word ClearFlags( word field )
begin
    newField := field with lock and mark flags cleared;
    return newField;

FindSmallest( parent, node, seekRecord )
begin
    // initialize the variables used in the traversal
    pNode := parent;        cNode := node;

    while true do
        ⟨*, *, nFlag, address⟩ := cNode→child[LEFT];
        if not (nFlag) then
            // visit the next node
            pNode := cNode;        cNode := address;
        else
            // reached the successor node
            seekRecord := {cNode, pNode, address};
            break;

RemoveChild( parent, which )
hu begin
    // determine the address of the child to be removed
    node := parent→child[which];

    // mark both the children edges of the node to be removed
    node→child[LEFT].mFlag := true;
    node→child[RIGHT].mFlag := true;

    // determine whether both the child pointers of the node to be removed are null
    if node→child[LEFT].nFlag   and   node→child[RIGHT].nFlag then
        // set the null flag only
        parent→child[which].nFlag := true;
    else
        // switch the pointer at the parent to point to its appropriate grandchild
        if node→child[LEFT].nFlag then
            address := node→child[RIGHT].address;
        else  address := node→child[LEFT].address ;
        parent→child[which].address := address;
```

1.1.4 Correctness Proof

It is convenient to treat insert and delete operations that do not change the tree as search operations. We call a tree node *active* if it is reachable from the root of the tree. We call a tree node *passive* if it was active earlier but is not active any more. Note that, before an active node is made passive by a delete operation, both its children edges are *marked*. Also, a CAS instruction performed on an edge (by either an insert operation or a delete operation as part of locking) is successful only if the edge is unmarked. As a result, clearly, if an insert operation completes successfully, then its target node was active when its edge was modified to make the new node (containing the target key) a part of the tree. Likewise, if a delete operation completes successfully, then all the nodes involved in the operation (up to three nodes) were active when their edges were locked.

All Executions are Linearizable

We show that an arbitrary execution of our algorithm is linearizable by specifying the *linearization point* of each operation. Note that the linearization point of an operation is the point during its execution at which the operation appeared to have taken effect. Our algorithm supports three types of operations: search, insert and delete. We now specify the linearization point of each operation.

1. *Insert operation:* The operation is linearized at the point at which it performed the successful CAS instruction that resulted in its target key becoming part of the tree.

2. *Delete operation:* There are two cases depending on whether the delete operation is simple or complex. If the operation is simple delete, then the operation is linearized at the point at which a successful write step was performed at the parent of the target node that resulted in the target node becoming passive. Otherwise, it is linearized at the point at which the original key of the target node was replaced with its successor key.

3. *Search operation:* There are two cases depending on whether the target node was active when the operation read the key stored in the node. If the target node was not active, then the operation is linearized at the point at which the target node became passive. Otherwise, it is linearized at the point at which the read step was performed.

It can be easily verified that, for any execution of the algorithm, the sequence of operations obtained by ordering operations based on their linearization points is legal, *i.e.*, all operations in the sequence satisfy their specification.

Thus we have:

Theorem 1. *Every execution of our algorithm is linearizable.*

All Executions are Deadlock-Free

We say that the system is in a *quiescent state* if no modify operation completes hereafter. We say that the system is in a *potent state* if it has one or more pending modify operations. Note that quiescence is a *stable property*; once the system is in a quiescent state, it stays in a quiescent state. We show that our algorithm is deadlock-free by proving that a potent state is necessarily non-quiescent.

Note that, in a quiescent state, no edges in the tree can be marked. This is because a delete operation marks edges only after it has successfully obtained all the locks, after which it is guaranteed to complete. This also implies that the tree cannot undergo any changes now because that would imply eventual completion of a modify operation. Thus, once a system has reached a quiescent state, all modify operation currently pending repeatedly alternate between seek and execution phases. We say that the system is in a *strongly-quiescent state* if all pending modify operations started their most recent seek phase *after* the system became quiescent. Note that, like quiescence, strong quiescence is also a stable property. Now, once the system has reached a strongly quiescent state, the following can be easily verified. First, for a given modify operation, every traversal of the tree in the seek phase returns the same target node. Second, for a given delete operation, the set of edges it needs to lock remains the same.

Now, assume that the system eventually reaches a state that is both potent and quiescent. Clearly, from this state, the system will eventually reach a state that is potent and strongly-quiescent. Note that a delete operation in our algorithm locks edges in a *top-down*, *left-right* manner. As a result, there cannot be a "cycle" involving delete operations. If a delete operation continues to fail in the execution phase, then it is necessarily because it tried to acquire lock on an already locked edge. (Recall that the set of edges does not change any more and there are no marked edges in the tree.) We can construct a chain of operations such that each operation in the chain tried to lock an edge already locked by the next operation in the chain. Clearly, the length of the chain is bounded. This implies that the last operation in the chain is guaranteed to obtain all the locks and will eventually complete. This contradicts the fact that the system is in a quiescent state.

Thus, we have:

Theorem 2. *Every execution of our algorithm is deadlock-free.*

CHAPTER 2

LOCK FREE CONCURRENT BINARY SEARCH TREE

2.1 The Lock-Free Algorithm

For ease of exposition, we describe our algorithm assuming no memory reclamation.

2.1.1 Overview of the Algorithm

Every operation in our algorithm uses *seek* function as a subroutine. The seek function traverses the tree from the root node until it either finds the target key or reaches a non-binary node whose next edge to be followed points to a null node. We refer to the path traversed by the operation during the seek as the *access-path*, and the last node in the access-path as the *terminal node*. The operation then compares the target key with the stored key (the key present in the terminal node). Depending on the result of the comparison and the type of the operation, the operation either terminates or moves to the execution phase. In certain cases in which a key may have moved upward along the access-path, the seek function may have to restart and traverse the tree again; details about restarting are provided later. We now describe the next steps for each of the operations one-by-one.

Search A search operation starts by invoking seek operation. It returns `true` if the stored key matches the target key and `false` otherwise.

Insert An insert operation ((shown in Figure 2.2)) starts by invoking seek operation. It returns `false` if the target key matches the stored key; otherwise, it moves to the execution phase. In the execution phase, it attempts to insert the key into the tree as a child node of the last node in the access-path using a CAS instruction. If the instruction succeeds, then the operation returns true; otherwise, it restarts by invoking the seek function again.

Delete A delete operation starts by invoking seek function. It returns false if the stored key does not match the target key; otherwise, it moves to the execution phase. In the execution phase, it attempts to remove the key stored in the terminal node of the access-path. There are two cases depending on whether the terminal node is a binary node (has two children) or not (has at most one child). In the first case, the operation is referred to as *complex delete operation*. In the second case, it is referred to as *simple delete operation*. In the case of simple delete (shown in Figure 2.3), the terminal node is removed by changing the pointer at the parent node of the terminal node. In the case of complex delete (shown in Figure 2.4), the key to be deleted is replaced with the *next largest* key in the tree, which will be stored in the *leftmost node* of the *right subtree* of the terminal node.

2.1.2 Details of the Algorithm

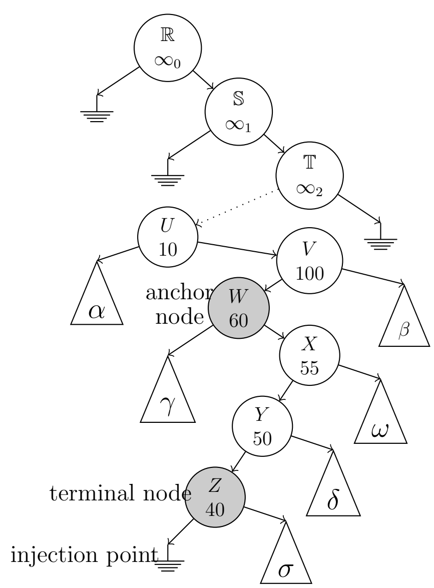

Figure 2.1. Nodes in the access path of seek along with sentinel keys and nodes ($\infty_0 < \infty_1 < \infty_2$)

As in most algorithms, we use sentinel keys and three sentinel nodes to handle the boundary cases easily. The structure of an empty tree with only sentinel keys (denoted by ∞_0, ∞_1 and ∞_2 with $\infty_0 < \infty_1 < \infty_2$) and sentinel nodes (denoted by $\mathbb{R}$, $\mathbb{S}$ and $\mathbb{T}$) is shown in Figure 2.1.

Our algorithm, like the one in (Natarajan and Mittal, 2014), operates at edge level. A delete operation obtains ownership of the edges it needs to work on by marking them. To enable marking, we steal bits from the child addresses of a node. Specifically, we steal *three* bits from each child address to distinguish between three types of marking: (i) marking for *intent*, (ii) marking for *deletion* and (iii) marking for *promotion*. The three bits are referred to as *intent-flag*, *delete-flag* and *promote-flag*. To avoid the ABA problem, as in Howley and Jones (Howley and Jones, 2012), we use *unique* null pointers. To that end, we steal another bit from the child address, referred to as *null-flag*, and use it to indicate whether the address field contains a null or a non-null value. So, when an address changes from a non-null value to a null value, we only set the null-flag and the contents of the address field are not otherwise modified. This ensures that all null pointers are unique.

Finally, we also steal a bit from the key field to indicate whether the key stored in a node is the original key or the replacement key. This information is used in a complex delete operation to coordinate helping among processes.

We next describe the details of the seek function, which is used by all operations (search as well as modify) to traverse the tree after which we describe the details of the execution phase of insert and delete operations.

The Seek Phase

A seek function keeps track of the node in the access-path at which it took the last "right turn" (*i.e.*, it last followed a right edge). Let this "right turn" node be referred to as *anchor node* when the traversal reaches the terminal node. Note that the terminal node is the node whose key matched the target key or whose next child edge is set to a null address. For an illustration, please see Figure 2.1. In the latter case (stored key does not match the target key), it is possible that the key may have moved up in the tree. To ascertain that the seek function did not miss the key because it may have moved up during the traversal, we use the following set of conditions that are *sufficient* (but not necessary) to guarantee that the seek function did not miss the key. First, the anchor node is still part of the tree. Second, the key stored in the anchor node has not changed since it first encountered the anchor node during the (current) traversal. To check for the above two conditions, we determine whether the anchor node is undergoing removal (either delete or promote flag set) by examining its right child edge. We discuss the two cases one-by-one.

(a) *Right child edge not marked:* In this case, the anchor node is still part of the tree. We next check whether the key stored in the anchor node has changed. If the key has not changed, then the seek function returns the results of the (current) traversal, which consists of three addresses: (i) the address of the terminal node, (ii) the address of its parent, and (iii) the null address stored in the child field of the terminal node that caused the traversal to terminate. The last address is required to ensure that an insert operation works correctly (specifically to ascertain that the child field of the terminal node has not undergone any change since the completion of the traversal). We refer to it as the *injection point* of the insert operation. On the other hand, if the key has changed, then the seek function restarts from the root of the tree.

(b) *Right child edge marked:* In this case, we compare the information gathered in the current traversal about the anchor node with that in the previous traversal, if one exists. Specifically, if the anchor node of the previous traversal is same as that of the current traversal and the keys found in the anchor node in the two traversals also match, then the seek function terminates, but returns the results of the previous traversal (instead of that of the current traversal). This is because the anchor node was definitely part of the tree during the previous traversal since it was reachable from the root of the tree at the beginning of the current traversal. Otherwise, the seek function restarts from the root of the tree.

The seek function also keeps track of the *second-to-last* edge in the access-path (whose endpoints are the parent and grandparent nodes of the terminal node), which is used for helping, if there is a conflict. For insert and delete operations, we refer to the terminal node as the *target node*.

The Execution Phase of an Insert Operation

In the execution phase, an insert operation creates a new node containing the target key. It then adds the new node to the tree at the injection point using a CAS instruction. If the CAS instruction succeeds, then (the new node becomes a part of the tree and) the operation terminates; otherwise, the operation determines if it failed because of a *conflicting* delete operation in progress. If there is no conflicting delete operation in progress, then the operation restarts from the seek phase; otherwise it performs helping and then restarts from the seek phase.

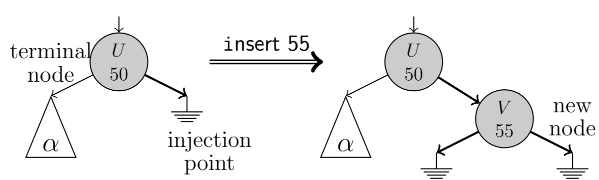

Figure 2.2. An illustration of an insert operation.

The Execution Phase of a Delete Operation

The execution of a delete operation starts in *injection mode*. Once the operation has been injected into the tree, it advances to either *discovery mode* or *cleanup mode* depending on the type of the delete operation.

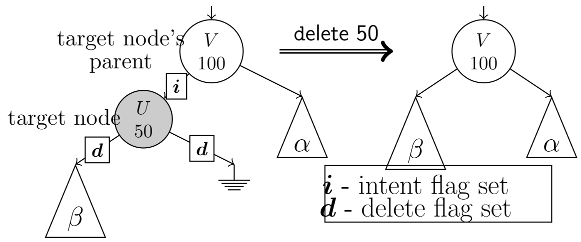

Figure 2.3. An illustration of a simple delete operation.

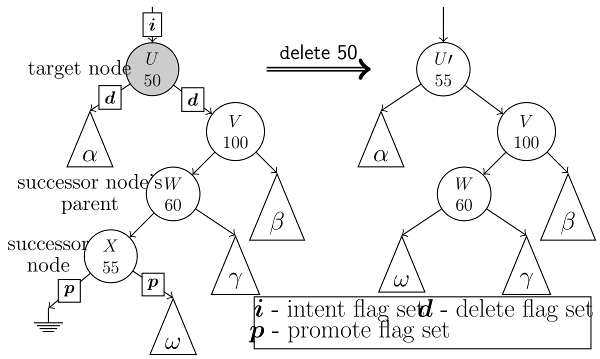

Figure 2.4. An illustration of a complex delete operation.

Injection Mode In the injection mode, the delete operation marks the three edges involving the target node as follows: (i) It first sets the intent-flag on the edge from the parent of the target node to the target node using a CAS instruction. (ii) It then sets the delete-flag on the left edge of the target node using a CAS instruction. (iii) Finally, it sets the delete-flag on the right edge of the target node using a CAS instruction. If the CAS instruction fails at any step, the delete operation performs helping, and either repeats the same step or restarts from the seek phase. Specifically, the delete operation repeats the same step when setting the delete-flag as long as the target node has not been claimed as the successor node by another delete operation. In all other cases, it restarts from the seek phase.

We maintain the invariant that an edge, once marked, cannot be unmarked. After marking both the edges of the target node, the operation checks whether the target node is a binary node or not. If it is a binary node, then the delete operation is classified as complex; otherwise it is classified as simple. Note that the type of the delete operation cannot change once all the three edges have been marked as described above. If the delete operation is complex, then it advances to the discovery mode after which it will advance to the cleanup mode. On the other hand, if it is simple, then it directly advances to the cleanup mode (and skips the discovery mode). Eventually, the target node is either removed from the tree (if simple delete) or replaced with a "new" node containing the next largest key (if complex delete).

For a tree node X, let $X.parent$ denote its parent node, and $X.left$ and $X.right$ denote its left and right child node, respectively. Also, hereafter in this section, let T denote the target node of the delete operation under consideration.

Discovery Mode In the discovery mode, a complex delete operation performs the following steps:

1. **Find Successor Key:** The operation locates the next largest key in the tree, which is the smallest key in the subtree rooted at the right child of T. We refer to this key as the *successor key* and the node storing this key as the *successor node*. Hereafter in this section, let S denote the successor node.
2. **Mark Child Edges of Successor Node:** The operation sets the promote-flag on both the child edges of S using a CAS instruction. Note that the left child edge of S will be null. As part of marking the left child edge, we also store the address of T (the target node) in the edge. This is done to enable helping in case the successor node is obstructing the progress of another operation. In case the CAS instruction fails while marking the left child edge, the operation repeats from step 1 after performing helping if needed. On the other hand, if the CAS instruction fails while marking the right child edge, then the marking step is repeated after performing helping if needed.
3. **Promote Successor Key:** The operation replaces the target node's original key with the successor key. At the same time, it also sets the mark bit in the key to indicate that the current key stored in the target node is the replacement key and not the original key.

4. **Remove Successor Node:** The operation removes S (the successor node) by changing the child pointer at$S.parent$ that is pointing to S to point to the right child of S using a CAS instruction. If the CAS instruction succeeds, then the operation advances to the cleanup mode. Otherwise, it performs helping if needed. It then finds S again by performing another traversal of the tree starting from the right child of T. If the traversal fails to find S (recall that the left edge of S is marked for promotion and contains the address of T), then S has already been removed from the tree by another operation as part of helping, and the delete operation advances to the cleanup mode. On advancing to the cleanup mode, the operation sets a flag in T indicating that S has been removed from the tree (and T can now be replaced with a new node) so that other operations trying to help it know not to look for S.

Cleanup Mode There are two cases depending on whether the delete operation is simple or complex.

(a) **Simple Delete:** In this case, either $T.left$ or $T.right$ is pointing to a null node. Note that both $T.left$ and $T.right$ may be pointing to null nodes (which in turn will imply that T is a leaf node). Without loss of generality, assume that $T.right$ is a null node. The removal of T involves changing the child pointer at $T.parent$ that is pointing to T to point to $T.left$ using a CAS instruction. If the CAS instruction succeeds, then the delete operation terminates; otherwise, it performs another seek on the tree. If the seek function either fails to find the target key or returns a terminal node different from T, then T has been already removed from the tree (by another operation as part of helping) and the delete operation terminates; otherwise, it attempts to remove T from the tree again using possibly the new parent information returned by seek. This process may be repeated multiple times.

Algorithm 8: Data Structures Used

```
struct Node {
    {Boolean, Key} mKey;
    {Boolean, Boolean, Boolean, Boolean, NodePtr} child[2];
    Boolean readyToReplace;
};
struct Edge {
    NodePtr parent, child;
    enum which { LEFT, RIGHT };
};
struct SeekRecord {
    Edge lastEdge, pLastEdge, injectionEdge;
};
struct AnchorRecord {
    NodePtr node;
    Key key;
};
struct StateRecord {
    Edge targetEdge, pTargetEdge;
    Key targetKey, currentKey;
    enum mode { INJECTION, DISCOVERY, CLEANUP };
    enum type { SIMPLE, COMPLEX } ;
    // the next field stores pointer to a seek record; it is used for finding the
       successor if the delete operation is complex
    SeekRecordPtr successorRecord;
};
// object to store information about the tree traversal when looking for a given key
   (used by the seek function)
SeekRecordPtr targetRecord := new seek record;
// object to store information about process' own delete operation
StateRecordPtr myState := new state;
```

(b) **Complex Delete:** Note that, at this point, the key stored in the target node is the replacement key (the successor key of the target key). Further, the key as well as both the child edges of the target node are marked. The delete operation attempts to replace target node with a *new* node, which is basically a copy of target node except that all its fields are unmarked. This replacement of T involves changing the child pointer at $T.parent$ that is pointing to T to point to the new node. If the CAS instruction succeeds, then the delete operation terminates; otherwise, as in the case of simple delete, it performs another seek on the tree, this time looking for the successor key. If the seek function either fails to find the successor key or returns a terminal node different from T, then T has been already replaced (by another operation as part of helping) and the delete operation terminates. Otherwise, it attempts to replace T again using possibly the new parent information returned by seek. This process may be repeated multiple times.

Discussion It can be verified that, in the absence of conflict, a delete operation performs three atomic instructions in the injection mode, three in the discovery mode (if delete is complex), and one in the cleanup mode.

Helping

To enable helping, as mentioned earlier, whenever traversing the tree to locate either a target key or a successor key, we keep track of the *last two* edges encountered in the traversal. When a CAS instruction fails, depending on the reason for failure, helping is either performed along the last edge or the second-to-last edge.

2.1.3 Formal Description

A pseudo-code of our algorithm is given in Algorithms 8-19.

Algorithm 8 describes the data structures used in our algorithm. Besides Node, three important data types in our algorithm are: Edge, SeekRecord and StateRecord. The data type Edge is a structure consisting of three fields: the two endpoints and the direction (left or right). The data type SeekRecord is a structure used to store the results of a tree traversal. The data type StateRecord is a structure used to store information about a delete operation (*e.g.*, target edge, type, current mode, etc.). Note that only objects of type Node are shared between processes; objects of all other types (*e.g.*, SeekRecord, StateRecord) are *local* to a process and not shared with other processes.

The pseudo-code of the seek function is described in Algorithm 9, which is used by all the operations. The pseudo-codes of the search, insert and delete operations are given in Algorithm 10, Algorithm 11 and Algorithm 12, respectively. A delete operation executes function INJECT in injection mode, functions FINDANDMARKSUCCESSOR and REMOVESUCCESSOR in discovery mode and function CLEANUP in cleanup mode. Their pseudo-codes are given in Algorithm 13, Algorithm 14, Algorithm 15 and Algorithm 16, respectively. The pseudo-codes for helper routines (used by multiple functions) are given in Algorithm 17 and Algorithm 18. Finally, the pseudo-codes of functions used to help other (conflicting) delete operations are given in Algorithm 19.

2.1.4 Correctness Proof

It can be shown that our algorithm satisfies linearizability and lock-freedom properties (Herlihy and Shavit, 2012). Broadly speaking, linearizability requires that an operation should appear to take effect instantaneously at some point during its execution. Lock-freedom requires that some process should be able to complete its operation in a finite number of its own steps. It is convenient to treat insert and delete operations that do not change the tree as search operations. We call a tree node *active* if it is reachable from the root of the tree. We call a tree node *passive* if it was active earlier but is not active any more. It can be verified that, if an insert operation completes successfully, then its target node was active when it performed the successful CAS instruction on the node's child edge. Likewise, if a delete operation completes successfully, then its target node was active when it marked the node's left edge for deletion. Further, for a complex delete, the successor node was active when it marked the node's left edge for promotion.

All Executions are Linearizable

We show that an arbitrary execution of our algorithm is linearizable by specifying the *linearization point* of each operation. Note that the linearization point of an operation is the point during its execution at which the operation appeared to have taken effect. Our algorithm supports three types of operations: search, insert and delete. We now specify the linearization point of each operation.

1. *Insert operation:* The operation is linearized at the point at which it performed the successful CAS instruction that resulted in its target key becoming part of the tree.
2. *Delete operation:* There are two cases depending on whether the delete operation is simple or complex. If the operation is simple delete, then the operation is linearized at the point at which a successful CAS instruction was performed at the parent of the target node that resulted in the target node becoming passive. Otherwise, it is linearized at the point at which the original key of the target node was replaced with its successor key.
3. *Search operation:* There are two cases depending on whether the target node was active when the operation read the key stored in the node. If the target node was not active, then the operation is linearized at the point at which the target node became passive. Otherwise, it is linearized at the point at which the read action was performed.

It can be easily verified that, for any execution of the algorithm, the sequence of operations obtained by ordering operations based on their linearization points is legal, *i.e.*, all operations in the sequence satisfy their specification. This establishes that *our algorithm generates only linearizable executions.*

All Executions are Lock-Free

We say that the system is in a *quiescent state* if no modify operation completes hereafter. We say that the system is in a *potent state* if it has one or more pending modify operations. Note that a quiescence is a *stable* property; once the system is in a quiescent state, it stays in a quiescent state. We show that our algorithm is lock-free by proving that a potent state is necessarily non-quiescent provided assuming that some process with a pending modify operation continues to take steps.

Assume, by the way of contradiction, that there is an execution of the system in which the system eventually reaches a state that is potent as well as quiescent. Note that, once the system has reached a quiescent state, it will eventually reach a state after which the tree will not undergo any structural changes. This is because a modify operation makes at most two structural changes to the tree. So, if the tree is undergoing continuous structural changes, then it clearly implies that modify operations are continuously completing their responses, which contradicts the assumption that the system is in a quiescent state. Further, on reaching such a state, the system will reach a state after which no new edges in the tree are marked. Again, this is because a modify operation marks at most four edges and the set of edges in the tree does not change any more. We call such a system state after which neither the set of edges nor the set of *marked* edges in the tree change any more as a *strongly quiescent state*. Note that, like quiescence, strong quiescence is also a stable property.

From the above discussion, it follows that the system in a quiescent state will eventually reach a state that is strongly quiescent. Consider the search tree in such a strongly quiescent state. It can be verified that no more modify operations can now be injected into the tree, and, moreover, all modify operations already injected into the tree are delete operations currently "stuck" in either discovery or cleanup mode. Now, consider a process, say p, that continues to take steps to execute either its own operation or another operation blocking its progress (directly or indirectly) as part of helping. Consider the recursive chain of the *helpee* operations that p proceeds to help in order to complete its own operation. Let α_i denote the i^{th} helpee operation in the chain. It can be shown that:

Lemma 1. *Let $\mathcal{C}_D$ denote the set of all complex delete operations already injected into the tree that are "stuck" in the discovery mode. Then,*

1. $\alpha_1 \in \mathcal{C}_D$, and

2. Suppose p is currently helping α_i for some $i \geq 1$ and assume that $\alpha_i \in \mathcal{C}_D$. Let α_{i+1} denote the next operation that p selects to help. Then, (a) α_{i+1} exists, (b) $\alpha_{i+1} \in \mathcal{C}_D$, and (c) the target node of α_{i+1} is at strictly larger depth than the target node of α_i.

Using the above lemma, we can easily construct a chain of distinct helpee operations whose length exceeds the number of processes—a contradiction. This establishes that *our algorithm only generates lock-free executions.*

Algorithm 9: Seek Function

```
Seek( key, seekRecord )
begin
    pAnchorRecord := {𝕊, ∞₁};
    while true do
        // initialize all variables used in traversal
        pLastEdge := {ℝ, 𝕊, RIGHT};    lastEdge := {𝕊, 𝕋, RIGHT};
        curr := 𝕋;    anchorRecord := {𝕊, ∞₁};

        while true do
            // read the key stored in the current node
            ⟨∗, cKey⟩ := curr→mKey;
            // find the next edge to follow
            which := key < cKey ? LEFT: RIGHT;
            ⟨n, ∗, d, p, next⟩ := curr→child[which];
            // check for the completion of the traversal
            if key = cKey  or n then
                // either key found or no next edge to follow; stop the traversal
                seekRecord→pLastEdge := pLastEdge;
                seekRecord→lastEdge := lastEdge;
                seekRecord→injectionEdge := {curr, next, which};

                if key = cKey then // keys match
                    return;
                else  break;

            if which = RIGHT then
                // next edge to be traversed is a right edge; keep track of the
                   current node and its key
                anchorRecord := ⟨curr, cKey⟩;

            // traverse the next edge
            pLastEdge := lastEdge;    lastEdge := {curr, next, which};    curr := next;

        // key was not found; check if can stop
        ⟨∗, ∗, d, p, ∗⟩ := anchorRecord.node→child[RIGHT];
        if not (d)  and not (p) then
            // anchor node is still part of the tree; check if anchor node's key has
               changed
            ⟨∗, aKey⟩ := anchorRecord.node→mKey;
            if anchorRecord.key = aKey then  return;
        else
            // check if the anchor record (the node and its key) matches that of the
               previous traversal
            if pAnchorRecord = anchorRecord then
                // return the results of the previous traversal
                seekRecord := pSeekRecord;
                return;

        // store the results of the traversal and restart
        pSeekRecord := seekRecord;    pAnchorRecord := anchorRecord;
```

Algorithm 10: Search Operation

```
Boolean Search( key )
begin
    Seek( key, mySeekRecord );

    node := mySeekRecord→lastEdge.child;
    ⟨∗, nKey⟩ := node→mKey;

    if nKey = key then return true;
    else return false;
```

Algorithm 11: Insert Operation

```
Boolean Insert( key )
begin
    while true do
        Seek( key, targetRecord );

        targetEdge := targetRecord→lastEdge;
        node := targetEdge.child;
        ⟨∗, nKey⟩ := node→mKey;
        if key = nKey then return false;

        // create a new node and initialize its fields
        newNode := create a new node;
        newNode→mKey := ⟨0_m, key⟩;
        newNode→child[LEFT] := ⟨1_n, 0_i, 0_d, 0_p, null⟩;
        newNode→child[RIGHT] := ⟨1_n, 0_i, 0_d, 0_p, null⟩;
        newNode→readyToReplace := false;

        which := targetRecord→injectionEdge.which;
        address := targetRecord→injectionEdge.child;
        result := CAS(node→child[which], ⟨1_n, 0_i, 0_d, 0_p, address⟩, ⟨0_n, 0_i, 0_d, 0_p, newNode⟩);
        if result then return true;

        // help if needed
        ⟨∗, ∗, d, p, ∗⟩ := node→child[which];
        if d then  HelpTargetNode( targetEdge ) ;
        else if p then  HelpSuccessorNode( targetEdge ) ;
```

Algorithm 12: Delete Operation

```
Boolean Delete( key )
begin
    // initialize the state record
    myState→targetKey := key;
    myState→currentKey := key;
    myState→mode := INJECTION;
    while true do
        Seek( myState→currentKey, targetRecord );
        targetEdge := targetRecord→lastEdge;
        pTargetEdge := targetRecord→pLastEdge;
        ⟨∗, nKey⟩ := targetEdge.child→mKey;
        if myState→currentKey ≠ nKey then
            // the key does not exist in the tree
            if myState→mode = INJECTION then
                return false;
            else return true;
        // perform appropriate action depending on the mode
        if myState→mode = INJECTION then
            // store a reference to the target edge
            myState→targetEdge := targetEdge;
            myState→pTargetEdge := pTargetEdge;
            // attempt to inject the operation at the node
            Inject( myState );
        // mode would have changed if injection was successful
        if myState→mode ≠ INJECTION then
            // check if the target node found by the seek function matches the one
               stored in the state record
            if ( myState → targetEdge.child
                 ≠
                                targetEdge.child ) then
                return true;
            // update the target edge information using the most recent seek
            myState→targetEdge := targetEdge;
        if myState→mode = DISCOVERY then
            // complex delete operation; locate the successor node and mark its child
               edges with promote flag
            FindAndMarkSuccessor( myState );
        if myState→mode = DISCOVERY then
            // complex delete operation; promote the successor node's key and remove
               the successor node
            RemoveSuccessor( myState );
        if myState→mode = CLEANUP then
            // either remove the target node (simple delete) or replace it with a new
               node with all fields unmarked (complex delete)
            result := Cleanup( myState );
            if result then return true;
            else
                ⟨∗, nKey⟩ := targetEdge.child→mKey;
                myState→currentKey := nKey;
```

Algorithm 13: Injecting a Deletion Operation

```
Inject( state )
begin
    targetEdge := state→targetEdge;
    // try to set the intent flag on the target edge
    // retrieve attributes of the target edge
    parent := targetEdge.parent;
    node := targetEdge.child;
    which := targetEdge.which;
    result := CAS( parent→child[which],
                   ⟨0_n, 0_i, 0_d, 0_p, node⟩,
                   ⟨0_n, 1_i, 0_d, 0_p, node⟩ );
    if not (result) then
        // unable to set the intent flag; help if needed
        ⟨*, i, d, p, address⟩ := parent→child[which];
        if i then HelpTargetNode( targetEdge ) ;
        else if d then
            HelpTargetNode( state→pTargetEdge );
        else if p then
            HelpSuccessorNode( state→pTargetEdge );
        return;
    // mark the left edge for deletion
    result := MarkChildEdge( state, LEFT );
    if not (result) then return;
    // mark the right edge for deletion; cannot fail
    MarkChildEdge( state, RIGHT );
    // initialize the type and mode of the operation
    InitializeTypeAndUpdateMode( state );
```

Algorithm 14: Locating the Successor Node

```
FindAndMarkSuccessor( state )
begin
    // retrieve the addresses from the state record
    node := state→targetEdge.child;
    seekRecord := state→successorRecord;
    while true do
        // read the mark flag of the key in the target node
        ⟨m, ∗⟩ := node→mKey;
        // find the node with the smallest key in the right subtree
        result := FindSmallest( state );
        if m  or not (result) then
            // successor node had already been selected before the traversal or the
               right subtree is empty
            break;
        // retrieve the information from the seek record
        successorEdge := seekRecord→lastEdge;
        left := seekRecord→injectionEdge.child;
        // read the mark flag of the key under deletion
        ⟨m, ∗⟩ := node→mKey;
        if m then // successor node has already been selected
            continue;
        // try to set the promote flag on the left edge
        result := CAS( successorEdge.child →
                       child[LEFT],
                       ⟨1_n, 0_i, 0_d, 0_p, left⟩,
                       ⟨1_n, 0_i, 0_d, 1_p, node⟩ );
        if result then break;
        // attempt to mark the edge failed; recover from the failure and retry if
           needed
        ⟨n, ∗, d, ∗, ∗⟩ := successorEdge.child→child[LEFT];
        if n  and d then
            // the node found is undergoing deletion; need to help
            HelpTargetNode( successorEdge );
    // update the operation mode
    UpdateMode( state );
```

Algorithm 15: Removing the Successor Node

```
RemoveSuccessor( state )
begin
    // retrieve addresses from the state record
    node := state→targetEdge.child;
    seekRecord := state→successorRecord;
    // extract information about the successor node
    successorEdge := seekRecord→lastEdge;

    // ascertain that the seek record for the successor node contains valid
       information
    ⟨∗, ∗, ∗, p, address⟩ := successorEdge.child→child[LEFT];
    if not (p) or (address ≠ node) then
        node→readyToReplace := true;
        UpdateMode( state );
        return;

    // mark the right edge for promotion if unmarked
    MarkChildEdge( state, RIGHT );

    // promote the key
    node→mKey := ⟨1_m, successorEdge.child→mKey⟩;
    while true do
        // check if the successor is the right child of the target node itself
        if successorEdge.parent = node then
            // need to modify the right edge of the target node whose delete flag is
               set
            dFlag := 1;      which := RIGHT;
        else
            dFlag := 0;      which := LEFT;

        ⟨∗, i, ∗, ∗, ∗⟩ := successorEdge.parent→child[which];

        ⟨n, ∗, ∗, ∗, right⟩ := successorEdge.child→child[RIGHT];
        oldValue := ⟨0_n, i, dFlag, 0_p, successorEdge.child⟩;
        if n then
            // only set the null flag; do not change the address
            newValue :=

            ⟨1_n, 0_i, dFlag, 0_p, successorEdge.child⟩;
        else
            // switch the pointer to point to the grand child
            newValue := ⟨0_n, 0_i, dFlag, 0_p, right⟩ ;

        result := CAS( successorEdge.parent      →
                       child[which],
                       oldValue, newValue );
        if result or dFlag then break;

        ⟨∗, ∗, d, ∗, ∗⟩ := successorEdge.parent→child[which]; pLastEdge :=
        seekRecord→pLastEdge;
        if d and (pLastEdge.parent ≠ null) then
            HelpTargetNode( pLastEdge );

        result := FindSmallest( state );
        lastEdge := seekRecord→lastEdge;
        if ( not (result) or
             lastEdge.child                ≠
             successorEdge.child ) then
            // the successor node has already been removed
            break;

        else successorEdge := seekRecord→lastEdge ;
```

Algorithm 16: Cleaning Up the Tree

```
Boolean CLEANUP( state )
begin
    ⟨parent, node, pWhich⟩ := state→targetEdge;
    if state→type = COMPLEX then
        // replace the node with a new copy in which all fields are unmarked
        ⟨∗, nKey⟩ := node→mKey;
        newNode→mKey := ⟨0_m, nKey⟩;
        // initialize left and right child pointers
        ⟨∗, ∗, ∗, ∗, left⟩ := node→child[LEFT];
        newNode→child[LEFT] := ⟨0_n, 0_i, 0_d, 0_p, left⟩;
        ⟨n, ∗, ∗, ∗, right⟩ := node→child[RIGHT];
        if n then
            newNode→child[RIGHT] := ⟨1_n, 0_i, 0_d, 0_p, null⟩;
        else newNode→child[RIGHT] := ⟨0_n, 0_i, 0_d, 0_p, right⟩ ;
        // initialize the arguments of CAS instruction
        oldValue := ⟨0_n, 1_i, 0_d, 0_p, node⟩;
        newValue := ⟨0_n, 0_i, 0_d, 0_p, newNode⟩;
    else // remove the node
        // determine to which grand child will the edge at the parent be switched
        if node→child[LEFT] = ⟨1_n, ∗, ∗, ∗, ∗⟩ then
            nWhich := RIGHT;
        else nWhich := LEFT;
        // initialize the arguments of the CAS instruction
        oldValue := ⟨0_n, 1_i, 0_d, 0_p, node⟩;
        ⟨n, ∗, ∗, ∗, address⟩ := node→child[nWhich];
        if n then // set the null flag only
            newValue := ⟨1_n, 0_i, 0_d, 0_p, node⟩;
        else // change the pointer to the grand child
            newValue := ⟨0_n, 0_i, 0_d, 0_p, address⟩ ;

    result := CAS( parent→child[pWhich],
                   oldValue, newValue );
    return result;
```

Algorithm 17: Helper Routines

```
Boolean MarkChildEdge( state, which )
begin
    if state→mode = INJECTION then
        edge := state→targetEdge;
        flag := DELETE_FLAG;
    else
        edge := (state→successorRecord)→lastEdge;
        flag := PROMOTE_FLAG;
    node := edge.child;
    while true do
        ⟨n, i, d, p, address⟩ := node→child[which];
        if i then
            helpeeEdge := {node, address, which};
            HelpTargetNode( helpeeEdge );
            continue;
        else if d then
            if flag = PROMOTE_FLAG then
                HelpTargetNode( edge );
                return false;
            else return true;
        else if p then
            if flag = DELETE_FLAG then
                HelpSuccessorNode( edge );
                return false;
            else return true;
        oldValue := ⟨n, 0_i, 0_d, 0_p, address⟩;
        newValue := oldValue | flag;
        result := CAS( node → child[which],
                       oldValue,
                       newValue );
        if result then break;
    return true;

Boolean FindSmallest( state )
begin
    // find the node with the smallest key in the subtree rooted at the right child
       of the target node
    node := state→targetEdge.child;
    seekRecord := state→seekRecord;
    ⟨n, ∗, ∗, ∗, right⟩ := node→child[RIGHT];
    if n then // the right subtree is empty
        return false;
    // initialize the variables used in the traversal
    lastEdge := ⟨node, right, RIGHT⟩;
    pLastEdge := ⟨node, right, RIGHT⟩;
    while true do
        curr := lastEdge.child;
        ⟨n, ∗, ∗, ∗, left⟩ := curr→child[LEFT];
        if n then // reached the node with the smallest key
            injectionEdge := ⟨curr, left, LEFT⟩;
            break;
        // traverse the next edge
        pLastEdge := lastEdge;
        lastEdge := ⟨curr, left, LEFT⟩;
```

Algorithm 18: Helper Routines

```
InitializeTypeAndUpdateMode( state )
begin
    // retrieve the target node's address from the state record
    node := state→targetEdge.child;
    ⟨lN, ∗, ∗, ∗, ∗⟩ := node→child[LEFT];
    ⟨rN, ∗, ∗, ∗, ∗⟩ := node→child[RIGHT];
    if lN  or rN then
        // one of the child pointers is null
        ⟨m, ∗⟩ := node→mKey;
        if m then  state→type := COMPLEX;
        else  state→type := SIMPLE;
    else // both child pointers are non-null
        state→type := COMPLEX;
    UpdateMode( state );

UpdateMode( state )
begin
    // update the operation mode
    if state→type = SIMPLE then // simple delete
        state→mode := CLEANUP;
    else // complex delete
        node := state→targetEdge.child;
        if node→readyToReplace then
            state→mode := CLEANUP;
        else state→mode := DISCOVERY;
```

Algorithm 19: Helping Conflicting Delete Operations

```
HelpTargetNode( helpeeEdge )
begin
    // intent flag must be set on the edge
    // obtain new state record and initialize it
    state→targetEdge := helpeeEdge;
    state→mode := INJECTION;
    // mark the left and right edges if unmarked
    result := MarkChildEdge( state, LEFT );
    if not (result) then return;
    MarkChildEdge( state, RIGHT );
    InitializeTypeAndUpdateMode( state );
    // perform the remaining steps of a delete operation
    if state→mode = DISCOVERY then
        FindAndMarkSuccessor( state );
    if state→mode = DISCOVERY then
        RemoveSuccessor( state );
    if state→mode = CLEANUP then Cleanup( state ) ;

HelpSuccessorNode( helpeeEdge )
begin
    // retrieve the address of the successor node
    parent := helpeeEdge.parent;
    node := helpeeEdge.child;
    // promote flat must be set on the successor node's left edge
    // retrieve the address of the target node
    ⟨*, *, *, *, left⟩ := node→child[LEFT];
    // obtain new state record and initialize it
    state→targetEdge := {null, left, _};
    state→mode := DISCOVERY;
    seekRecord := state→successorRecord;
    // initialize the seek record in the state record
    seekRecord→lastEdge := helpeeEdge;
    seekRecord→pLastEdge := {null, parent, _};
    // promote the successor node's key and remove the successor node
    RemoveSuccessor( state );
    // no need to perform the cleanup
```

REFERENCES

Herlihy, M. and N. Shavit (2012). *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann.

Howley, S. V. and J. Jones (2012, June). A Non-Blocking Internal Binary Search Tree. In *Proceedings of the 24th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pp. 161–171.

Natarajan, A. and N. Mittal (2014, February). Fast Concurrent Lock-Free Binary Search Trees. In *Proceedings of the 19th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pp. 317–328.