1 Introduction

The dictionary is one of the most often-used abstract data types (ADT) in computer science. Much work has been done to implement this ADT in the concurrent setting. Most of this work uses locks (see, e.g., [2], [8]). Although locks are simple to use, this approach has a major disadvantage: it is hard to design scalable locking strategies due to problems such as deadlock, priority inversion, and convoying [6]. For this reason, it is desirable to build a non-blocking implementation. Furthermore, the non-blocking property ensures that, while a single operation may be delayed, the system as a whole will always make progress.

Some other dictionary implementations use operations that are not commonly supported by multi-core machines, such as load-link/store-conditional [1] and multi-word compare-and-swap (CAS) [7]. The dictionary has also been implemented by means of software transactional memory (STM) (see, e.g., [9]). However, such an implementation is currently not efficient [2].

Most multi-core machines support single-word CAS operations. Non-blocking dictionary implementations based on linked lists and skip lists have been implemented by Sundell and Tsigas [10], Fomitchev and Ruppert [5], Fraser [7], and Valois [11]. Valois also presented a sketch of a non-blocking binary search tree (BST) [11]. The first complete non-blocking BST algorithm was presented by Ellen et al. [4]. This was also the first practical non-blocking tree data structure.

In [3], we have generalized their BST to a $k$-ary search tree ($k$-ST). Each internal node contains $k - 1$ keys and has $k$ children. Larger $k$ values decrease the average depth of nodes, allowing faster searches. However, this also increases the local work done at each internal node for routing searches and performing updates to the tree. We have implemented both the BST of Ellen et al. and our $k$-ST in Java. We conducted an experiment to compare both implementations against the concurrent Skip List (SL) from the Java class library and the lock-based AVL tree of Bronson et al [2]. The AVL tree is the current leading concurrent search tree. In [2], Bronson et al. presented experimental results comparing their tree with SL, a lock-based red-black tree, and a red-black tree implemented using STM. Since SL and AVL drastically outperform the other two implementations, we did not include the others in our comparison. In our experimental results, BST and 4-ST ($k$-ST with $k = 4$) are the top performers in both high and low contention. When the tree is small and the contention is high, the simplicity of BST gains advantages. On the other hand, when the tree is large (low contention), the shallower tree depth of 4-ST makes it faster than other algorithms. In our experimental setup, we did not observe any significant benefit of using values of $k$ greater than four.

Our previous work focused on giving an empirical analysis of BST’s performance. In this paper, we present a modification to the non-blocking BST implementation of Ellen et al. and an analysis of the modified implementation’s amortized time complexity. Given a finite concurrent execution of a number of data structure operations, the amortized cost is defined as the total amount of work done by the system divided by the number of operations invoked. Amortized analysis ensures that, even in the worst-case execution,
the average cost of an operation is small, even though some operations might cost much more than others. Our concurrent BST data structure is non-blocking, which means that even though there might be some operations that do not make any progress, the system as a whole always makes progress. For this reason, an amortized analysis is best suited for our algorithm.

In the next section, we start by giving a brief overview of the original BST implementation of Ellen et al. Then we explain the motivation behind each modification to the original algorithms, followed by complete pseudocode for the new algorithms. We explain our amortized analysis in Section 4. We bound the total steps in an execution as follows. We assign each step to some operation (not necessarily the operation that performed the step). We call this our blaming scheme. Then we bound the number of steps blamed on each operation as function of \( c \) (the maximum point contention during the operation) and \( h \) (the height of the tree at the beginning of the operation). The amortized cost of each operation is calculated as the total number of steps assigned to that operation. Our blaming scheme proves that the amortized cost is \( O(h) \) per Find operation, and \( O(h + c^2) \) per update operation. However, we could not find any concrete example that shows this bound is tight. In fact, we believe that this bound can be further improved to \( O(h + c) \). Section 5 describes an additional lemma that would be sufficient to prove a bound of \( O(h + c) \) on the amortized cost of update operations.

2 Overview of the Original BST Algorithm

In [4], Ellen et al. presented a non-blocking BST. Their BST is a leaf-oriented tree. The keys of the dictionary are stored in the leaves, while the internal nodes serve the purpose of directing searches. To avoid handling special cases when the tree has less than three nodes, they added two special keys to the initial tree, namely \( \infty_1 \) and \( \infty_2 \). These keys are larger than any other keys, and \( \infty_1 < \infty_2 \) (see Figure 1). All the real keys that are inserted into the tree will always be inserted as descendants of \( \infty_1 \).

Each leaf node only stores a key. Each internal node stores a key (for directing searches) and an update field. This update field contains the state of the node and a pointer to an Info object. The state field captures the current condition of a node. If a node has a clean state, it means there is no operation going on at that location. If a node has a flag state, it means some operation is trying to change one of that node’s child pointers. If a node has a mark state, it means the node is going to be removed from the tree, or has already been removed. The Info object stores information about the operation that is performing the change at the node (if the node’s state is not clean). Initially, a node’s state is clean and its info is null (⊥). If an operation is delayed because another operation is updating a node it needs to change, it can help finish that other operation by using the information that is stored in the Info object.

Find works as in a sequential BST. It traverses down the tree until it finds a leaf. If the key stored in that leaf matches the one it is looking for, then it returns \( \text{true} \), and otherwise it returns \( \text{false} \). Since Find never needs to modify any node, it never needs to help finish other operations.

Insert and Delete are the operations that modify the tree. Since the tree is leaf-oriented, updates always occur at a leaf of the tree. When inserting a new key, Insert starts by searching for that key. If the key is already in the tree, the operation returns \( \text{false} \). If the key is not present in the tree, then it will proceed by trying to replace a leaf (at the correct position for inserting the new key) with a small subtree containing one internal node, a leaf with the new key, and a leaf with the same key as the leaf that got replaced. Before replacing the leaf, Insert flags the parent of that leaf. If the flag is successful, then it replaces the leaf, and the operation finishes. However, if the flag is unsuccessful, it will help the operation...
currently operating on the parent node. After helping, INSERT then retries its own operation from the beginning.

When deleting a key, DELETE starts by searching for the key that it wants to remove. If the key is not present in the tree, the operation returns false. If the key is found, then it proceeds to delete the key by removing the leaf that contains that key and the leaf’s parent, leaving the sibling of that leaf in the tree. To delete a leaf, DELETE first flags the grandparent node of the leaf, marks the parent, and then changes the child pointer of the grandparent to point to the sibling of that leaf. If flagging of the grandparent (dflag CAS) is unsuccessful, DELETE would help the other operation that is operating on the grandparent node, and then restart its own operation from the beginning. If the dflag CAS is successful, but marking the parent (mark CAS) is unsuccessful, then DELETE helps the operation that is currently operating on the parent node, removes its own flag (backtrack CAS), and then retries its own operation from the beginning.

When an operation \( x \) helps another operation \( y \), it is possible that \( y \) also needs to help another operation \( z \). In this situation, \( x \) will recursively help \( z \) (and any operation that \( z \) needs to help), and so on.

## 3 Modification to the Original Algorithm

In this section, we present our modifications to the non-blocking BST algorithm of Ellen et al. We shall prove in the next sections that this modified algorithm has \( O(h + c^2) \) amortized cost per operation, while the original algorithm has \( \Omega(c \cdot h) \) amortized cost per operation.

First, we give an example of a run of the original algorithm that leads to \( \Omega(c \cdot h) \) amortized cost per operation. Consider a system with \( c \) processes concurrently running and a BST of height \( h \). Suppose one process \( p \) performs a deletion of a leaf of depth \( h \), and all other operations want to do insertions in that same location. Suppose \( p \) succeeds in its deletion, so all other operations fail their attempt to flag and they have to retry. (When one operation makes another operation fail and retry, we call it thwarting.) Before the other operations start their next attempt, \( p \) quickly re-inserts the key it just deleted. Then during the next attempt of all the insertions, \( p \) deletes the node again (and thwarts all other operations). This scenario can be repeated over and over again. Process \( p \) always succeeds in performing its operation, but the other \( c − 1 \) processes never succeed. In this situation, all \( c \) processes traverse down the tree of height \( h \), but only two operations finish during each iteration. Thus the amortized cost per operation is \( \Omega(c \cdot h) \).

Our modifications consist of three main ideas. The first one is that we want to avoid having an update operation repeat its SEARCH after it fails an attempt. Rather than restarting the SEARCH from the root, an operation should be able to recover from its latest position and continue its SEARCH from there. To do this, we let an operation remember its search path. An operation can recover from its failed attempt by using its knowledge of this path.

Suppose an update operation tries to search for key \( k \), and its first attempt’s SEARCH returns some leaf \( l \). If this operation makes another attempt, it will search for the same key \( k \). Let \( x \) be some ancestor of \( l \). If the first SEARCH traverses \( x \), and \( x \) is still in the tree during the second SEARCH, then \( x \) is still on the search path for \( k \). So instead of starting from the root, the second search can start from the lowest unmarked node that was traversed during the first search. In this situation, the second attempt behaves just as if the operation was delayed, and never performed the traversals that it undoes. We shall see in Section 4.2.1 (Lemma 4.6), if a node \( v \) is pushed onto the stack, then there exists some tree configuration during the operation such that \( v \) is on the search path for \( k \). This lemma is analogous to Lemma 20 from [4], which is an important lemma for defining the linearization points of operations.

In our implementation, we use two stacks to store the path that an update operation traverses. The first one stores pointers to Node objects (nodeStack), and the other one stores corresponding pointers to Update objects (updateStack). Storing Update object pointers on a stack is important to maintain the order of reading pointers in the original algorithm (i.e., grandparent’s update pointer is read before parent node pointer). This property ensures the correctness of the algorithm. Whenever an operation enters a node, it pushes the node onto the stack (line 120 and 121). It pops a node when undoing the traversal to that node (line 108 and 109). The stack used is local to each operation. Empty stacks are created in the beginning of each FIND (line 27 and 28), INSERT (line 34 and 35), or DELETE (line 66 and 67) operation, and the same
stacks are used throughout their attempts. (In fact, when implementing the data structure in program code, these two stacks could be created local to each process, and used for every operation of that process. A process would just need to empty the stacks at the beginning of each operation.) The Find operation never has to retry, so it does not need to store the information about its search path, so the stacks can be omitted in the real implementation of Find. In this paper, we use the same routine for traversal in all operations for the sake of simplicity in both the pseudocode and the time complexity analysis.

There are two cases where an operation \( op \) needs to pop a node \( v \) off its stack. The first case is if \( v \) is already marked when \( op \) retries its attempt (line 108). The second case (line 98) is when a Delete operation fails its attempt. If the attempt fails to flag the grandparent node, then it would help the operation that is currently operating on that node. On the next attempt, the grandparent node might already be deleted, or have a new value on its update field. If the deletion attempt failed to mark its parent node, then its flag would be backtracked before it retries another attempt. So the update field of the grandparent node will have a new value. In order to pop the grandparent or read the new value of its update field, one must pop the parent node first, even though the parent node is still in the tree (and thus still on the search path).

Line 112 and 113 of Recover-and-Traverse read the most recent update field of the topmost node inside nodeStack after the first loop (which pops out marked nodes). This ensures that for each attempt, the value of parent node’s update field is read during the attempt itself. We shall see later in Lemma 4.12 that popping one node after a failed deletion attempt and re-reading the top node’s update field on line 112 and 113 ensures that two failed attempts of the same operation are not thwarted by the same flag.

Our second modification is to remove the recursive helping mechanism. In the original algorithm, an operation can recursively help many other operations with which it does not directly conflict. For example, consider the tree in Figure 2. Since process \( Q \) has flagged node \( B \), process \( R \) will fail its mark CAS, so we say \( Q \) thwarts \( R \). But process \( Q \) also failed its mark CAS because node \( A \) has been flagged by process \( P \). In this situation, \( P \) indirectly thwarts \( R \). In the original algorithm, process \( R \) would help \( Q \), and then recursively help \( P \). This chain of indirect thwarting can be of unbounded length (i.e., \( R \)’s flag on node \( C \) could cause another attempt’s mark CAS to fail, and so on).

The following example shows that only applying the first modification to the original algorithm would lead us to \( \Omega(h + c^2) \) amortized cost. (Our goal was to achieve \( O(h + c) \) amortized time. We conjecture that the modified algorithm achieves this even though we were only able to prove that it achieves \( O(h + c^2) \) amortized time so far.) Consider the tree in Figure 3, and \( c \) processes running on the system. Let \( p_1, p_2, \ldots, p_c \) be \( c \) DELETE operations that are deleting keys 1, 2, ..., \( c \), respectively. Suppose all operations traverse the tree at the same time, and they succeed in flagging their grandparent nodes. Process \( p_1 \) would succeed in marking its parent node (internal node 1), but all other operations would be thwarted (i.e., \( p_i \) would be thwarted by \( p_{i-1} \)). Then, all the operations which failed to mark would help all operations occurring lower in the tree: \( p_2 \) would help \( p_1 \), \( p_3 \) would help \( p_2 \) and then recursively help \( p_1 \), and so on. After \( p_1 \) finishes,

![Figure 2: Example of situation that causes recursive helping.](image-url)
and before all other operations retry, the process $p_1$ quickly re-inserts key 1, and then it wants to delete 1 again. We get the exact same setting as when we started, except that the $c - 1$ operations that are retrying do not need to do their traversals again. If we repeat this scenario, in each iteration we delete key 1 and re-insert it, while all other operations keep failing. So in each iteration the system does $\Omega(h + c^2)$ steps but only two operations finish. So, the amortized cost for each operation is $\Omega(h + c^2)$.

The purpose of the helping mechanism is to ensure that, when an operation makes another attempt, it will not be thwarted again by the same cause. But operations that happen in non-overlapping locations cannot directly thwart one another, so it is not necessary to help those operations. In the example of Figure 2 above, it is enough for $R$ just to help backtrack $Q$'s failed attempt, because $R$ would not need to flag or mark the same location that $P$ has written to.

The third modification is to HELP as we go back up the tree on line 107 (when undoing traversal steps). An operation $op$ will undo the traversal to a node $v$ if $v$ is already marked when $p$ recovers from its failed attempt. However, $v$ might be still in the tree (i.e., if the $dchild$ CAS of the operation that marked it has not yet been performed), so on $p$'s next attempt, $v$ might be pushed back onto the stack. We modify the algorithm so that $p$ helps $v$’s operation as it pops $v$ to ensure that every node is only pushed once onto the stack during any operation. As we have removed the recursive helping mechanism, the helping procedure takes a constant number of steps, so adding this helping when popping will only change the constant factor of popping steps. However, we shall see later that this modification makes the time complexity analysis easier.

Most of our modifications do not significantly affect the proofs in [4], except for some exceptions. Firstly, the modification to SEARCH so that a new attempt does not restart its traversal from the top of the tree, affects the correctness proof. This modification significantly affects Lemma 20 of [4], which is the most important lemma for the correctness proof. Lemma 4.6 is analogous to Lemma 20 of [4], but is modified to reflect the changes we have made to the algorithm. Secondly, removing recursive helping, while it does not affect the correctness proof, it does affect the progress proof. However, in this paper we give the time complexity of the modified algorithm. The bound implies that the modified algorithm maintains its non-blocking property.
type Update { // stored in one CAS word
    {clean, dflag, iflag, mark} state
    Info *info
}

type Node {
    Key ∪ {∞₁, ∞₂} key
}

type Internal { // subtype of Node
    Update update
    Node *left, *right
}

type Leaf { // subtype of Node
}

type Info {
    Internal *p
    Leaf *l
}

type IInfo { // subtype of Info
    Internal *newInternal
}

type DInfo { // subtype of Info
    Internal *gp
    Update pupdate
}

// Initialization:
shared Internal *Root = pointer to new Internal node
    with key field ∞₂, update field {clean, ⊥}, and pointers to new Leaf nodes
    with keys ∞₁ and ∞₂, respectively, as left and right fields.

FIND(Key k):boolean
Stack nodeStack = empty stack
Stack updateStack = empty stack
Node *l

PUSH(nodeStack, Root)
PUSH(updateStack, Root, update)
l = RECOVER-AND-TRaverse(nodeStack, updateStack, k)
return l.key == k
**Insert**(*Key k*):boolean

34 Stack `nodeStack` = empty stack
35 Stack `updateStack` = empty stack
36 Internal *newInternal
37 Leaf *newSibling
38 Leaf *new = pointer to new Leaf node whose key field is `k`
39 Update `pupdate`, `result`
40 IInfo *op
41 Node *l, *removed

42 **PUSH**(nodeStack, Root)
43 **PUSH**(updateStack, Root. update)

44 **while** TRUE
45 
46   l = **RECOVER-AND-TRAVERSE**(nodeStack, updateStack, k)
47   if l.key == k  // key is already present in the tree
48     return FALSE
49   end if
50   `pupdate` = top element of updateStack
51   p = top element of nodeStack
52   if `pupdate` ≠ CLEAN
53     HELP(`pupdate`)
54   else
55     newSibling = pointer to a new Leaf whose key is l.key
56     newInternal = pointer to a new Internal node with key max(`k`, l.key) ,
57       update field ⟨CLEAN, ⊥⟩, and with two child fields equal to new and newSibling
58       (the one with smaller key is left child)
59     op = pointer to a new IInfo record containing ⟨p, l, newInternal⟩
60     `result` = **CAS**(p.update, `pupdate`, ⟨IFlag, op⟩)
61     if `result` == `pupdate`
62       HELPINSERT(op)
63     return TRUE
64   else
65     HELP(`result`)
66   end if
67 end while
DELETE(Key k):boolean

66 Stack nodeStack = empty stack
67 Stack updateStack = empty stack
68 Update pupdate, gpupdate, result
69 DInfo *op
70 Node *l, *p, *gp

71 PUSH(nodeStack, Root)
72 PUSH(updateStack, Root. update)

73 while true
74     l = RECOVER-AND-TRAVERSE(nodeStack, updateStack, k)
75     if l.key ≠ k  // key does not exist
76         return FALSE
77     end if
78     gpupdate = second from top element of updateStack
79     pupdate = top element of updateStack
80     gp = second from top element of nodeStack
81     p = top element of nodeStack
82     if gpupdate ≠ CLEAN
83         HELP(gpupdate)
84     else
85         if pupdate ≠ CLEAN
86             HELP(pupdate)
87             else op = pointer to a new DInfo record containing ⟨gp, p, l, pupdate⟩
88                 result = CAS(gp.update, gpupdate, ⟨Dflag, op⟩)
89                 if result == gpupdate
90                     if HELPDELETE(op)
91                         return TRUE
92                     end if
93                 else HELP(result)
94             end if
95         end if
96     end if

97     // pop p if the deletion attempt failed
98     POP(nodeStack)
99     POP(updateStack)
100 end while
RECOVER-AND-TRaverse(Stack *nodeStack, Stack *updateStack, Key k):Node

**Precondition:** (1) nodeStack and updateStack are not empty, they at least contains the root of the tree and the root’s Update record
(2) Let \( x_1, \ldots, x_n \) and \( y_1, \ldots, y_n \) be elements inside nodeStack and updateStack respectively (in order they are pushed), then these statements hold:
(2a) \( x_i \) points to an internal node
(2b) at some time, \( x_i \) was child of \( x_{i-1} \) and \( x_i \) is on the search path for \( k \)
(2c) at some earlier time, \( x_i \)’s update field equals \( y_i \)

**Postcondition:** (1) \( l \) points to a leaf
(2) Let \( x_1, \ldots, x_n \) and \( y_1, \ldots, y_n \) be elements inside nodeStack and updateStack respectively (in order they are pushed), then these statements hold:
(2a) \( x_i \) points to an internal node
(2b) at some time, \( x_i \) was child of \( x_{i-1} \) and \( x_i \) is on the search path for \( k \)
(2c) at some earlier time, \( x_i \)’s update field equals \( y_i \)

Node *removed

// recover from last attempt
removed = NIL

while nodeStack.top.state == Mark // remove marked node from the stack
    HELPMarked(nodeStack.top.update)
    POP(nodeStack)
    POP(updateStack)
end while

p = nodeStack.top
POP(updateStack)
PUSH(updateStack, p.update)
if \( k < p.key \)
    \( l = p.left \)
else \( l = p.right \)
end if

// traverse down
while \( l \) points to an internal node
    PUSH(nodeStack, l)
    PUSH(updateStack, l.update)
    if \( k < l.key \)
        \( l = l.left \)
    else \( l = l.right \)
    end if
end while
return \( l \)

CAS-Child(Internal parent Node *old, Node *new)
if \( new.key < parent.key \)
    CAS(parent.left, old, new)
else CAS(parent.right, old, new)
end if

HELPInsert(IInfo op)
CAS-Child(op.p, op.l, op.newInternal) // ichild CAS
CAS(op.update, ⟨IFlag, op⟩, ⟨Clean, op⟩) // iunflag CAS
HELPDELETE(Dinfo *op)

Update result, result2

result = CAS(op. p. update, op. pupdate, ⟨MARK, op⟩)
if result == op. pupdate OR result = ⟨MARK, op⟩
    HELPMARKED(op)
    return TRUE
    // helps only the direct thwarting attempt
else if result. state == IFLAG
    HELPINSERT(result. info)
else if result. state == MARK
    HELPMARKED(result. info)
else if result. state == DFlag
    result2 = CAS(result. info. p. update, op2. pupdate, ⟨MARK, result. info⟩)
    if result2 == result. info. pupdate OR result == ⟨MARK, result. info⟩
        HELPMARKED(result. info)
        else CAS(result. info. gp. update, ⟨DFlag, result. info⟩, ⟨Clean, result. info⟩)
        end if
    end if

// backtracks op because the mark attempt was failed
CAS(op. gp. update, ⟨DFlag, op⟩, ⟨Clean, op⟩)
return FALSE
end if

HELPMARKED(DInfo *op)

Node *other

if op. p. right == op. l
    // set other to point to the sibling of op.l
    other = op. p. left
else other = op. p. right
end if
CAS-CCHILD(op. gp, op. p, other)
CAS(op. gp. update, ⟨DFlag, op⟩, ⟨Clean, op⟩)

HELP(Update u)
if u. state == IFLAG
    HELPINSERT(u. info)
else if u. state == DFlag
    HELPDELETE(u. info)
else if u. state == MARK
    HELPMARKED(u. info)
end if

4 Analysis of Amortized Time Complexity

4.1 The cost of Find, Insert, and Delete are proportional to the number of traversals and attempts

In this subsection, we show that the cost of Find, Insert, and Delete are proportional to the number of pushes to nodeStack done in that method and the number of executions of line 44 (for Insert) or 73 (for
DELETE). This will be useful to simplify the proof of the amortized time complexity bound, since we can focus only on these three steps. We call a single push onto nodeStack a traversal step. Note that counting line 44 and 73 is basically counting the number of iterations of INSERT and DELETE’s main loop, which is the number of attempts made in that operation. Let TRAVERSAL$_m$ and ATTEMPT$_m$ be the number of traversals and attempts by an operation $m$, respectively (including the traversals that are done in any RECOVER-AND-TRAVVERSE invoked by $m$).

We first bound the time required by subroutines used by the main operations. The following lemma shows that all helping subroutines take constant time.

**Lemma 4.1** The cost of CAS-CHILD, HelpInsert, HelpMarked, HelpDelete, and Help are $O(1)$

**Proof** CAS-CHILD does not call any other method, and a constant number of steps are performed in this method. So the cost of CAS-CHILD is $O(1)$.

HelpInsert consists of a call to CAS-CHILD, whose cost is $O(1)$, and a CAS on line 133. So the cost of HelpInsert is $O(1)$.

HelpMarked consists of a call to CAS-CHILD, whose cost is $O(1)$. All other steps run in constant time, so the cost of HelpMarked is $O(1)$.

Depending on the if-then-else conditions, HelpDelete can make one call either to HelpMarked or HelpInsert. We have showed that these methods take constant time. All other steps in HelpDelete also take constant time, so the cost of HelpDelete is $O(1)$.

Depending on the state of $u$, Help calls one of HelpInsert, HelpDelete, HelpMarked, or no other routine. We have showed that all of these methods take $O(1)$ steps. Other than those methods, there is one if-then-else, which also takes a constant number of steps, so the cost of Help is $O(1)$. 

Next, we consider the RECOVER-AND-TRAVVERSE method. This is the main method for traversing the tree, as well as undoing the traversals of marked nodes when an attempt fails. The idea is to assign the undoing step to the matching traversal step. Let RTcall$_m$ be the number of calls to RECOVER-AND-TRAVVERSE by operation $m$. First we bound the total cost of all calls to RECOVER-AND-TRAVVERSE during operation $m$.

**Lemma 4.2** The total cost of all calls to RECOVER-AND-TRAVVERSE by an operation $m$ is $O(TRAVERSAL_m + RTcall_m)$.

**Proof** There are two main loops in RECOVER-AND-TRAVVERSE. The first while-loop (line 106-110) pops one element out of nodeStack in each iteration. Let $x$ be the node that is popped during an iteration of the first while-loop. We have showed that HelpMarked costs $O(1)$. We assign the checking of the while-loop condition, the call to HelpMarked, and the two Pops on line 108 and 109 to the Push step of $m$ that previously pushed $x$ onto nodeStack. (This push exists since the stack is empty at the beginning of $m$.) Now, we are left with the last check for the while-loop condition, which causes the loop to finish. Since the loop finishes, there are no more iterations to be done, and there is no Push to assign this step to. We assign this last step to the call to RECOVER-AND-TRAVVERSE.

The second while-loop (line 119-126) is the main routine for traversing down the tree. Each iteration pushes one node to nodeStack, as well as pushing that node’s update record to updateStack. Let $x$ be the node that is pushed onto nodeStack during an iteration of the second while-loop. The if-then-else on line 122 to 125 takes constant time. We assign the work of this if-then-else, pushing $x$’s update record to updateStack, and checking the while-loop condition to the Push step that pushes $x$ onto nodeStack (the traversal step for node $x$). As we did for the first loop, we assign the last check of the while-loop condition to the call to RECOVER-AND-TRAVVERSE.

All steps other than the two main loops take constant time. We also assign these steps to the call to RECOVER-AND-TRAVVERSE. So each call to RECOVER-AND-TRAVVERSE in $m$ is assigned $O(1)$ steps, giving $O(TRAVERSAL_m)$ total work for all calls. So, the total cost for all calls to RECOVER-AND-TRAVVERSE in $m$ is $O(TRAVERSAL_m + RTcall_m)$. 

**Lemma 4.3** The cost of a Find operation $m$ is $O(TRAVERSAL_m)$. 

11
Proof Find consists of one call to Recover-and-Traverse on line 32. By Lemma 4.2, the total cost for Recover-and-Traverse of operation $m$ is $O(\text{TRAVERSAL}_m + 1)$. Recall that the tree always has at least one special internal node at the top of the tree (with an infinite key), so Recover-and-Traverse must traverse at least once. So, the cost of the call to Recover-and-Traverse on line 32 is $O(\text{TRAVERSAL}_m)$.

Other than the call to Recover-and-Traverse, there is one traversal step on line 30 and several constant-time steps which can be assigned to this traversal. So the total cost of a Find operation $m$ is $O(\text{TRAVERSAL}_m)$.

Lemma 4.4 The cost of an Insert operation $m$ is $O(\text{TRAVERSAL}_m + \text{ATTEMPT}_m)$.

Proof Insert consists of one main loop and several constant-time steps outside the loop. The step on line 42 is a traversal step. We assign the Push on line 43 to this traversal step.

Now observe the main loop of Insert. Each iteration is an attempt of the Insert operation, which consists of a call to Recover-and-Traverse, followed by several constant-time steps. By Lemma 4.2, the cost of all calls to Recover-and-Traverse during operation $m$ is $O(\text{TRAVERSAL}_m + RTcall_m)$. We assign all the steps inside the loop block to the step on line 44 (which is an attempt step). Since we call Recover-and-Traverse once every attempt, $RTcall_m$ is equal to $\text{ATTEMPT}_m$ (or $\text{ATTEMPT}_m - 1$ if it exits the loop or dies before executing line 45). Thus, the total time for an Insert operation $m$ is $O(\text{TRAVERSAL}_m + \text{ATTEMPT}_m)$.

Lemma 4.5 The cost of a Delete operation $m$ is $O(\text{TRAVERSAL}_m + \text{ATTEMPT}_m)$.

Proof The structure of Delete is similar to Insert: it consists of one main loop and several constant-time steps outside the loop. The step on line 71 is a traversal step. We assign the Push step on line 72 to this traversal step.

The main loop of Delete is also similar to Insert. Each iteration is an attempt to delete key $k$ from the tree. By Lemma 4.2, the cost of all calls to Recover-and-Traverse is $O(\text{TRAVERSAL}_m + RTcall_m)$. We assign all steps inside the loop to the execution of line 73 (which is an attempt step). We call Recover-and-Traverse once every iteration, so $RTcall_m$ is equal to $\text{ATTEMPT}_m$ (or $\text{ATTEMPT}_m - 1$ if it exits the loop or dies before executing line 74). So the total time for a Delete operation $m$ is $O(\text{TRAVERSAL}_m + \text{ATTEMPT}_m)$.

4.2 Blaming Scheme

We have shown in the previous section that the cost of each of the main methods is proportional to the number of attempts and traversals. In this subsection, we describe to which operation we assign each attempt or traversal step (we call this our blaming scheme). Then, we give a bound on the number of steps that are assigned to each operation, thus proving the bound on amortized time complexity.

There are two main ideas for our blaming scheme. The first one is for blaming traversals. The second is for blaming attempts. Section 4.2.1 and 4.2.2 describe these two parts.

4.2.1 Blaming Scheme for Traversals

Consider operations that run without any concurrency. Focusing only on the number of attempts and traversals done in each operation, we can see the three main operations as follows.

- Find consists of traversals.
- Insert consists of traversals and one insertion attempt.
- Delete consists of traversals and one deletion attempt.
This is because, without concurrency all CASes would succeed and no operation would need to retry its attempt. However, when concurrent operations can occur, some operation might find that the location it needs to flag or mark is not clean and thus cannot continue its work. Even when the node is clean, the CAS might fail to write a value because there is some other operation that successfully does a CAS to the same location. In the case where an operation performs more than one attempt, an attempt that is not the last attempt of that operation is called a failed attempt. Thus, each attempt of an operation is either a failed attempt or the last attempt of that operation.

Since we have concurrency, it is possible that the tree is constantly changing during an operation. Thus, the number of traversals done by an operation can be unbounded. But we know that, unless there exists another operation modifying the tree, the number of traversals must be bounded by the height of the tree. We blame a traversal of a node that is newly inserted on the operation that inserted it. Here, “newly inserted” means that the node was not present in the tree at the time the operation that traversed it started.

In the sequential setting, the number of traversals is bounded by the height of the tree since an operation always traverses down the tree. However, in our concurrent algorithm, an operation can undo its traversal, and thus go back up the tree. Consider the tree in Figure 4. Suppose an operation op wishes to insert key 11. On the first attempt it will traverse down the tree and reach the leaf with key 12. But suppose there is another operation that concurrently deletes 12, so op’s attempt fails. Before op begins its next attempt, key 13 also gets deleted, so op’s next attempt will undo the traversal to 13 and 14, and continue the search until it reaches 15. If this scenario is repeated in the next attempt (15 and 16 get deleted), then op’s next attempt will traverse down to 18. So in our setting, it is possible that an operation traverses all the internal nodes in the tree. But again, we know that an operation undoes its traversal if and only if that node was present in the tree during one of its attempts, and was deleted before its next attempt. So, we blame the traversal that got undone to the operation that deletes the node. Later, we shall prove that each node is only traversed once by each operation, so the number of traversals that are blamed on the operation itself is bounded by the height of the tree when the operation started. Also, since we only blame a traversal step on a concurrent update, the number of traversals blamed on each update operation is bounded by the contention when the update’s child CAS occurred.

We now give the blaming scheme more precisely. A traversal done by operation op to enter node v is blamed as follows.

1. If v is eventually popped off the stack on line 98, then the traversal is blamed on the attempt that popped it (there is at most one of this kind of traversal per deletion attempt, so this assignment does not affect the constant cost of deletion attempts).

2. Otherwise, if v was not present in the tree when op started, then the traversal is blamed on the operation that inserted v.

3. Otherwise, if v was in the tree when op started, and v is still inside nodeStack of op during op’s last attempt, then the traversal is blamed on op itself.
4. Otherwise, $v$ was in the tree when $op$ started, but $v$ is no longer inside nodeStack of $op$ during $op$’s last attempt. Then, the traversal is blamed on the operation that deleted $v$.

The first rule of our blaming scheme handles the situation when a node is popped on line 98. We blame the work to the deletion attempt (not the operation) that pops the parent node, adding a constant number of steps to the cost of that attempt.

Excluding the previous case, a node is only popped from the stack if it is marked (on line 108). Before a node is popped, the operation helps it on line 107, ensuring it is physically removed from the tree. (Lemma 11 from [4] shows that if a node is successfully marked, then no backtrack CAS belongs to the operation that marks the node, and the first $d$ child that belongs to that operation succeeds.) So, in the next traversal attempt, this node is no longer reachable, and thus never pushed back onto the stack. So, except for the special case (which is handled by the first rule), an operation can only push each node once onto its nodeStack.

The following lemmas show that for the second and fourth cases above, the operation that is blamed exists and is concurrent with $op$. Thus, we can bound the number of traversals that are blamed on an INSERT and DELETE operation by the contention at the time its child CAS occurred. First, we prove a lemma that guarantees correctness of searches. The lemma is analogous to Lemma 20 of [4], but is modified to reflect the changes we have made to the algorithm.

**Lemma 4.6** Let $v$ be a node that is inside nodeStack at some time during an execution of operation $op$ whose key is $k$. Let $RaT$ be the RECOVER-AND-TRAVERSAL that pushes $v$ onto the stack if $v$ is not the root node. Otherwise, $RaT$ is the FIND, INSERT, or DELETE method that pushed the root node onto nodeStack. There is a configuration $C$ such that

1. $v$ is on the search path for $k$ in configuration $C$,
2. $C$ is before $v$ is pushed onto the stack, and
3. $C$ is after $RaT$ is invoked.

**Proof** Let $v_1, ..., v_j = v$ be the nodes on the stack in the order they are pushed. Let $RaT_j$ be the RECOVER-AND-TRAVERSAL that pushed $v_j$. We prove the claim by induction on $j$.

**Base Case** ($j=1$): Since Root never changes, $v_1$ is always the root node. Once the root node is pushed on line 30, 42, or 71, it is never popped off of the stack, because the root node is never marked. The root is never popped on line 98 either, because the root never has a parent, so if a DELETE operation found the root to be its $p$ node, then $gp$ is null (the dictionary is empty) and the operation would return false. Let $C_1$ be the configuration immediately before $v_1$ is pushed onto nodeStack. Claim 1 is true because the root node is always on the search path for any key. Claim 2 and 3 follow from the definition of $C_1$.

**Induction Step:** Let $j > 1$. Assume the lemma holds for $v_{j-1}$ and that $op$ pushed $v_j$ onto the stack. We prove that the lemma holds for $v_j$. Let $enter_j$ be the step when $RaT_j$ reads the pointer to $v_j$ in a child field of $v_{j-1}$ (line 115, 116, 123, or 124).

There is some child CAS $ccas$ that writes a pointer to $v_j$ in a child field of $v_{j-1}$ before $enter_j$. (In fact, there is exactly one such child CAS, by Lemma 14(7) of [4].) Let $C_j$ be the configuration just after $ccas$.

First we define $C_j$ based on two cases.

**Case 1:** $RaT_{j-1} = RaT_j$. In this case, we define $C_j$ to be either $C_{j-1}$ or $C$, whichever is later.

**Case 2:** $RaT_{j-1} \neq RaT_j$. In this case, $RaT_j$ must have seen that $v_{j-1}$ is unmarked on line 106. Let $C_{unmarked}$ be the configuration at the time when $RaT_j$ saw that $v_{j-1}$ is unmarked. We define $C_j$ to be either $C_{unmarked}$ or $C$, whichever is later.

Now we prove the three claims in turn. For the first case, the proof is very similar to the proof of Lemma 20 of [4].

By the induction hypothesis, $C_{j-1}$ is after $RaT_{j-1}$ is invoked. Thus, for the first case, $C_j$ must also be after $RaT_j$ is invoked. For the second case, $C_{unmarked}$ is after $RaT_j$ is invoked (by definition), so $C_j$ must also be after $RaT_j$ is invoked.

By the induction hypothesis, $C_{j-1}$ precedes $enter_{j-1}$, which precedes $enter_j$. Configuration $C_j$ and $C_{unmarked}$ precede $enter_j$ (by definition). So $C_j$ precedes $enter_j$, which precedes the push step of $v_j$.  

14
It remains to prove that \( v_j \) is on the search path for \( k \) in \( C_j \). We consider three cases. The second and third cases are identical to the proof of Lemma 20 of [4].

**Case A:** \( C_j = C_{\text{unmarked}} \). In this case \( c\text{cas} \) wrote a pointer to \( v_j \) in the child field of \( v_{j-1} \) before \( C_{\text{unmarked}} \), and the pointer was still there when \( \text{enter}_j \) read that field after \( C_{\text{unmarked}} \). By the induction hypothesis, \( v_{j-1} \) is on the search path for \( k \) in \( C_{j-1} \). Since \( v_{j-1} \) is unmarked in \( C_{\text{unmarked}} \), by Lemma 19 of [4], it is still on the search path for \( k \) in \( C_{\text{unmarked}} \). Step \( \text{enter}_j \) reads the appropriate child of \( v_{j-1} \) (i.e., the left child if \( k < v_{j-1}.\text{key} \) and the right child otherwise), so \( v_j \) is the appropriate child of \( v_{j-1} \) in configuration \( C_{\text{unmarked}} \). Thus, \( v_j \) is on the search path for \( k \) in \( C_{\text{unmarked}} = C_j \).

**Case B:** \( C_j = C_{j-1} \). This means that \( c\text{cas} \) wrote a pointer to \( v_j \) in the child field of \( v_{j-1} \) before \( C_{j-1} \), and the pointer was still there when \( \text{enter}_j \) read that field after \( C_{j-1} \). By Lemma 14(7) of [4], the child pointer must have contained the pointer to \( v_j \) at \( C_{j-1} \). Thus, at \( C_{j-1}, v_{j-1} \) was on the search path for \( k \) (according to the induction hypothesis) and the child pointer of \( v_{j-1} \) that would be read by a search for \( k \) contained a pointer to \( v_j \). Thus, \( v_j \) is on the search path for \( k \) at \( C_{j-1} = C_j \).

**Case C:** \( C_j = C \). The successful child CAS, \( c\text{cas} \) changes a child pointer of \( v_{j-1} \) immediately before \( C \). By Lemma 12 and Lemma 14(2) of [4], \( v_{j-1} \) is flagged (and hence not marked) when \( c\text{cas} \) occurs. By Lemma 17 of [4], \( v_{j-1} \) is in the tree at configuration \( C \). In some configuration \( C_{j-1} \) before \( C \), \( v_{j-1} \) was on the search path for \( k \), by the induction hypothesis. By Lemma 19 of [4], \( v_{j-1} \) is still on the search path for \( k \) in configuration \( C \). Step \( \text{enter}_j \) reads the appropriate child of \( v_{j-1} \) (i.e., the left child if \( k < v_{j-1}.\text{key} \) and the right child otherwise), so \( v_j \) is the appropriate child of \( v_{j-1} \) in configuration \( C \). Thus, \( v_j \) is on the search path for \( k \) in \( C = C_j \).

**Lemma 4.7** *If an operation \( op \) traverses to node \( v \), but \( v \) was not in the tree when \( op \) started, then the successful \( \text{ichild} \) CAS of the operation that inserts \( v \) is concurrent with \( op \).*

**Proof** By Lemma 4.6, there exists a configuration \( C \) during the call to \( \text{RECOVER-AND-TRaverse} \) that traverses to node \( v \) (which is during the execution of \( op \)), and \( v \) is on \( op \)'s search path in configuration \( C \). But \( v \) was not in the tree when \( op \) started, so it must be inserted between the time \( op \) started and the time of configuration \( C \). So \( op \) is concurrent with the \( \text{ichild} \) CAS that inserts \( v \).

**Lemma 4.8** *If an operation \( op \) traverses to node \( v \), and that traversal was later popped by line 108, then there exists a \( \text{DELETE} \) operation whose \( \text{dchild} \) CAS physically removes \( v \) from the tree, and that \( \text{dchild} \) CAS is concurrent with \( op \).*

**Proof** Since \( op \) traverses to \( v \), by Lemma 4.6, there is some configuration \( C \) during \( op \) in which \( v \) is reachable. An operation would only pop \( v \) on line 108 if that node is marked, and it always helps the marked node on line 107 before popping it. So \( v \)'s update field is marked at some time during \( op \)'s execution, and either \( v \) is already physically removed from the tree when \( op \) popped it, or \( op \) helps to remove \( v \). (By Lemma 23 of [4], the first \( \text{dchild} \) that belongs to the Info object in \( v \) succeeds.) So, the \( \text{dchild} \) that physically removes \( v \) from the tree must be concurrent with \( op \).

The following lemma shows that the number of traversals that are blamed on the operation that performed them is bounded by the height of the tree when the operation started.

**Lemma 4.9** *Let \( h_{\text{start}} \) be the height of the tree when an operation \( op \) started. The number of traversals that are blamed on \( op \) does not exceed \( h_{\text{start}} \).*

**Proof** Let \( t_{\text{start}} \) be the time when \( op \) started, and \( Tree_{\text{start}} \) be the tree at time \( t_{\text{start}} \). The cost of traversals to nodes that are not in \( Tree_{\text{start}} \) are not assigned to \( op \). The only traversal steps that are blamed on \( op \) are those which stay inside \( nodeStack \) during \( op \)'s last attempt. At the beginning of a \( \text{FIND} \), \( \text{INSERT} \), or \( \text{DELETE} \) operation, the root of the tree is pushed onto \( nodeStack \). Other than the root, a node is pushed to \( nodeStack \) only by line 120. If a node is pushed onto \( nodeStack \), it must be the child of the top element of \( nodeStack \) at the preceding execution of line 115, 116, 123, or 124. Let \( v_1, v_2, \ldots, v_n \) be the nodes that stay in \( nodeStack \) during \( op \)'s last attempt and are members of \( Tree_{\text{start}} \). Since a node never gets a new ancestor
(see Lemma 18 from [4]) and \( v_{i+1} \) is a child of \( v_i \) when \( op \) traverses to \( v_{i+1} \), then \( v_i \) is an ancestor of \( v_{i+1} \) at \( Tree_{start} \). So for all \( 1 \leq i < n \), \( v_i \) is an ancestor of \( v_{i+1} \) in \( Tree_{start} \). So \( v_1, v_2, ..., v_n \) all belong to the path from the root to \( v_n \) in \( T_{start} \). Thus, \( n \leq h_{start} \). □

4.2.2 Blaming Scheme for Attempts

The second main part of our blaming scheme is for blaming attempts. An operation only retries its attempt if the previous attempt was thwarted by an attempt of some other operation. We have seen, in a brief example in Section 3, that an attempt can thwart another attempt directly or indirectly. It is possible that the indirect thwarting attempt is not concurrent with the thwarted attempt. For example, suppose an operation \( op_2 \) wants to flag a node, and it was not clean because another operation \( op_2 \) has performed a dflag CAS on it. But \( op_2 \)'s mark CAS failed because another operation \( op_3 \) flagged the same node and finishes its operation. In this situation, \( op_1 \) is indirectly thwarted by \( op_3 \)'s attempt. While \( op_1 \)'s attempt must be concurrent with \( op_2 \), it is not necessarily concurrent with \( op_3 \). In this situation, the operation that owns \( op_1 \) might not be concurrent with the operation that owns \( op_3 \). We do not want to assign work to a non-concurrent operation, because we want to bound the amount of work assigned to each operation based on the contention at some point of its execution time.

Before we describe our blaming scheme, first we give a formal description of a direct thwarting attempt and an indirect thwarting attempt.

**Definition** An attempt \( a \) of an operation \( op \) is a failed attempt if \( a \) is not the last attempt of \( op \).

**Definition** An attempt fails its flag or mark CAS if it or its helper perform the CAS but none of them successfully write the attempt’s Info object to the desired update field.

**Definition** If \( a \) is a failed attempt, and it does not perform its flag CAS because it sees an un-clean value, then the direct thwarting attempt is the attempt that created the Info object \( pupdate \) on line 52, \( gpupdate \) on line 83, or \( pupdate \) on line 86.

If \( a \) is a failed attempt, and it fails its flag CAS, then the direct thwarting attempt is the attempt that created the Info object \( result \) on line 62 or \( result \) on line 93.

If \( a \) is a failed attempt, and it fails its mark CAS, then the direct thwarting attempt is the attempt that created the Info object \( result \) that is read by the first process that performs line 135 among all executions of HELPDELETE\((op)\), where \( op \) is \( a \)'s Info object.

**Definition** If a failed attempt \( a \) fails its flag or mark CAS on an update field \( u \), and its direct thwarting attempt \( b \) wrote an iflag, dflag, or mark CAS on \( u \), then we say that \( a \) is thwarted by \( b \)'s iflag, dflag, or mark CAS respectively.

Let \( u \) be the update field of a node \( x \), then we say that \( a \) failed at node \( x \).

**Definition** An attempt \( a \) indirectly thwarts another attempt \( a' \) if there exists a sequence of \( n \geq 1 \) attempts \( b_1, b_2, ..., b_n \) such that for all \( 1 \leq i < n \), \( b_i \) directly thwarts \( b_{i+1} \), \( a \) directly thwarts \( b_1 \), and \( b_n \) directly thwarts \( a' \).

**Definition** Let \( a \) be a failed attempt. Let \( b_1, b_2, ..., b_n \) be the sequence of attempts such that each attempt is directly thwarted by the next in the sequence. Then, \( a \)'s ultimate thwarting attempt is \( b_n \). (Since the ultimate thwarting attempt is not thwarted by any other attempt, it must be the last attempt of its operation.)

**Lemma 4.10** For each failed attempt \( a \), there exists an ultimate thwarting attempt.

**Proof** Since a failed attempt is not the last attempt of its operation, it must finish (otherwise the next attempt could not start). So \( a \) must have seen an un-clean value or executed line 62, line 93, or line 135. Thus, there exists another attempt that directly thwarts \( a \).
Let \( a, b_1, b_2, \ldots \) be the sequence of attempts such that each attempt is directly thwarted by the next in the sequence. Assume this sequence is infinite to derive a contradiction. Observe that attempts \( b_1, b_2, \ldots \) must be deletion attempts, because each has a successful \texttt{dflag} CAS (so it can thwart the previous attempt) and a failed \texttt{mark} CAS (so it can be thwarted by the next attempt). Let \( gp_1, gp_2, \ldots \) be the nodes that are flagged by \( b_1, b_2, \ldots \), respectively. So for all \( i \), \( b_i \) flags node \( gp_i \), and fails to mark node \( gp_{i+1} \). This implies that at the time \( b_i \) traverses to \( gp_{i+1} \), \( gp_i \) was the parent of \( gp_{i+1} \). Let \( G_\alpha \) be the graph whose vertices are all internal nodes created during execution \( \alpha \) and there is an edge from node \( x \) to node \( y \) if \( y \) is a child of \( x \) at some time during \( \alpha \). By Lemma 36 from [4], graph \( G_\alpha \) contains no cycle. So \( gp_1, gp_2, \ldots \) are all different nodes. Hence, \( b_1, b_1, \ldots \) are all different attempts. This contradicts the fact that the execution is finite. So, there exists an attempt \( b_n \) which is the last attempt in the sequence and it is the ultimate thwarting attempt of \( a \).

Now we give the general idea of our scheme for blaming attempts. The goal is to design the blaming scheme such that for any two concurrent operations \( x \) and \( y \), only a constant number of attempts of \( x \) are blamed on \( y \). Additionally, we define a constant number of specific points during \( y \)'s execution, and all operations containing the attempts blamed on \( y \) must be concurrent with at least one of these points of time. So the number of attempts that are blamed on \( y \) is bounded by the contention at these points of time.

Initially we tried blaming the last attempt of an operation \( op \) on the operation itself, and each failed attempt \( a \) on the operation that owns its ultimate thwarting attempt \( b \). However, \( b \) might not run concurrently with \( op \), so we do not want to blame the work of attempt \( a \) to \( b \)'s operation. So, we modified our scheme as follows. If \( b \) is not concurrent with \( op \), we blame \( a \) on \( op \) itself. More specifically, we blame the work of \( a \) to \( b \)'s operation if \( op \) is running at the time attempt \( b \) started.

Since there is only one last attempt of each operation, there is only one such attempt blamed on the operation itself. We shall prove later that the number of \( op \)'s failed attempts that are blamed on \( op \) itself is bounded by the number of concurrent operations when \( op \) started and ended. The blaming scheme ensures that only operations that are concurrent with \( op \) at the time \( op \)'s last attempt started can blame their failed attempt on \( op \). If each operation can only blame a constant number of attempts on \( op \) then the total number of attempts blamed on \( op \) can be bounded by the point contention at the time \( op \)'s last attempt started. However, the following example shows that it is possible that other operations can have more than one failed attempts that are ultimately thwarted by \( op \)'s last attempt if certain changes in the tree structure occur.

Consider the tree in Figure 5. Suppose \( op \) wants to \texttt{INSERT(1)} and there are operations \( A, B, \) and \( C \) that already flagged 3, 5, and 7 respectively. \( A \) wants to \texttt{DELETE(4)}, \( B \) wants to \texttt{DELETE(6)}, and \( C \) wants to \texttt{INSERT(9)}. First \( op \) traverses down the tree and arrives at leaf 2. But it sees a \texttt{dflag} of \( A \), so it tries to help \( A \). Since \( B \) has already flagged node 5, \( op \) fails to finish \( A \)'s operation, and thus it backtracks \( A \)'s \texttt{dflag} and retries its own attempt. Before \( op \) makes its second attempt, some other operation deletes 2 (thus removing leaf 2 and internal node 3 from the tree). So, \( op \)'s second attempt will arrive at leaf 4. But if sees \( B \)'s \texttt{dflag} on node 5, it tries to help \( B \). But again, \( C \) already flagged node 7, so \( op \) cannot finish \( B \)'s operation. In this case, both the first and second attempt of \( op \) are ultimately thwarted by \( C \). This example can be generalized.

Figure 5: Example where more than one failed attempt that belong to the same operation can be ultimately thwarted by the same attempt.
so that many attempts of an operation have the same ultimate thwarting attempt.

This leads to a problem, because our goal is to keep the number of attempts that an operation can blame on each other operation constant. In the previous example, there is a change to the tree in between \( op \)'s first and second attempt, which leads \( op \) to a different leaf. Thus, instead of blaming both of \( op \)'s attempts on the ultimate thwarting attempt \( C \), we can blame one on the update operation that changed the tree. Based on this observation, now we describe the complete blaming scheme for blaming attempts. An attempt \( a \) of operation \( op \) is blamed as follows.

1. If \( a \) is the last attempt of \( op \), then \( a \) is blamed on \( op \) itself.
2. If \( a \) is a failed attempt, let \( b \) be the ultimate thwarting attempt of \( a \).
   (a) If \( op \) has an earlier failed attempt \( a' \) whose ultimate thwarting attempt is also \( b \), and there exists an update operation whose child CAS is after attempt \( a' \) started and before attempt \( a \) finishes, then \( a \) is blamed on the operation whose child CAS was the last to modify the tree during that time.
   (b) Otherwise, if \( op \) is running at the time attempt \( b \) started, then \( a \) is blamed on the operation that performs \( b \).
   (c) Otherwise, \( a \) is blamed on \( op \) itself.

The following lemmas show that for each ultimate thwarting attempt \( b \) of \( op \)'s failed attempt, only a constant number of \( op \)'s attempts are blamed on the operation that performed \( b \) by rule 2(b).

**Lemma 4.11** If \( a \) is a deletion attempt that has a successful dflag CAS, and \( gp_a \) and \( p_a \) are the two nodes that \( a \) stores in its Info object as \( gp \) and \( p \) respectively, then \( p_a \) is the child of \( gp_a \) from the time \( a \)'s operation reads \( p_a \)'s update field during its traversal, until the first dchild that uses \( a \)'s Info object occurs or \( a \)'s flag on \( gp_a \) is backtracked.

**Proof** Since \( a \) performs its dflag CAS, \( a \) reads a clean value in both \( gp_a \) and \( p_a \)'s update field during its traversal. Since \( a \)'s dflag CAS was successful, \( gp_a \)'s clean value remains the same from when \( a \) read it until the time \( a \) flagged it, so there was no other operation that changes \( gp_a \)'s child pointers during this period of time (see Lemma 12 from [4]). After \( a \) flagged \( gp_a \) and before this flag is removed, there is no other operation that can change the child field of \( gp_a \) (see Lemma 10 from [4]). So \( a \)'s (or \( a \)'s helper's) dchild is the only CAS that can change \( gp_a \)'s child field. If \( a \) is a failed attempt, then \( p_a \) remains a child of \( gp_a \) until \( a \)'s flag is backtracked.

**Lemma 4.12** Two distinct failed attempts that belong to the same operation cannot have the same direct thwarting attempt.

**Proof** Let \( a_1 \) and \( a_2 \) be two failed attempts that belong to the same operation and assume \( a_1 \) is performed earlier than \( a_2 \). Let \( b \) be the attempt that directly throttles \( a_1 \). Since an operation cannot retry another attempt before finishing the previous attempt, \( a_1 \) must execute line 52, 62, 83, 86, 93, or 135-150 before starting \( a_2 \). The following arguments show that \( b \)'s flag is removed before \( a_1 \) finishes.

**Case 1**: \( a_1 \) reads an un-clean value or failed its flag CAS. In this case, \( b \) either finishes its operation, or is backtracked (by itself or by the help of \( a_1 \) on line 52, 62, 83, 86, or 93) before \( a_1 \) finishes.

**Case 2**: \( a_1 \) failed its mark CAS. If \( a_1 \) executes line 135 earlier than any of its helpers, or \( a_1 \)'s helper reads the same Info object as \( a_1 \) on line 135, then the same argument as in the first case applies. Otherwise, some helper of \( a_1 \) executes line 135 earlier than \( a_1 \), and it reads a different Info object than the one \( a_1 \) reads on that line later on. Then, \( b \)'s flag is removed before \( a_1 \) executes line 135. Thus, \( b \)'s flag is removed before \( a_1 \) finished.

Since \( b \)'s flag is removed before \( a_1 \) finished, \( b \) cannot write another flag or mark on any active node (see Lemma 9 from [4]). If \( b \) marked a node \( x \), then \( x \) has been removed from the tree (see Lemma 11 and 14(2) from [4]) by the end of \( a_1 \). If \( x \) is not in nodeStack when \( a_2 \) started, by Lemma 4.6, \( a_2 \) will not traverse to \( x \),
because it is no longer reachable. Now we show that if \( x \) is in nodeStack after \( a_1 \) finishes its Recover-and-Traversal, then \( a_1 \)'s next attempt will pop \( x \). We consider two cases: If \( a_1 \) is an insertion attempt, then \( x \) is \( a_1 \)'s parent node. The first loop of Recover-and-Traversal in \( a_1 \)'s next attempt will pop \( x \) because it has been marked. If \( a_1 \) is a deletion attempt, then \( x \) can be \( a_1 \)'s parent or grandparent node. Line 98 pops the parent node of a failed deletion attempt, and the first loop iteration of the Recover-and-Traversal of the attempt after \( a_1 \) will pop \( a_1 \)'s grandparent node if it has been marked. So \( x \) is not in nodeStack after the attempt after \( a_1 \) finishes its Recover-and-Traversal (which is an earlier attempt or the same attempt as \( a_2 \)).

It remains to show that \( a_2 \) is not thwarted by \( b \)'s flag. This can only happen is \( b \)'s flag is read and stored in updateStack by an attempt before \( a_2 \). Line 112-113 ensures that \( a_2 \)'s pupdate is read during \( a_2 \)'s execution, so if \( b \)'s flag was located on \( a_1 \)'s parent node, \( a_2 \) would not read it as its pupdate value. Thus, if \( a_2 \) is an insertion attempt, it cannot be thwarted by \( b \)'s flag. If \( a_2 \) is a deletion attempt, and \( b \)'s flag was located on \( a_1 \)'s grandparent node, \( a_2 \) will pop its parent node on line 98 and \( a_2 \) will re-read gpupdate on line 112-113. So in this case, \( a_2 \) also cannot be thwarted by \( b \)'s flag.

We are left with the case where \( b \)'s flag was located on some node \( v \), which was a parent of \( a_1 \)'s grandparent node (i.e., \( a_1 \) is thwarted by \( b \)'s mark on its grandparent node). If \( v \) is not inside nodeStack after \( a_1 \) finishes its Recover-and-Traversal, \( a_2 \) cannot be thwarted by \( b \)'s flag. Consider the case where \( v \) is inside nodeStack after \( a_1 \) finishes its Recover-and-Traversal. Let \( gp \) and \( p \) be the grandparent and parent node found by \( a_1 \). Since \( gp \) was a child of \( v \) when \( b \)'s operation traverses to \( gp \), \( v \) must be pushed onto nodeStack before \( gp \). Let \( v, u_1, ..., u_n, gp, p \) be the top nodes inside nodeStack after \( a_1 \) finishes its Recover-and-Traversal in the order they are pushed. Since \( b \)'s operation sees \( gp \) as a child of \( v \), \( u_1, ..., u_n \) must be already removed from the tree by the time \( b \)'s operation traverses to \( gp \). So they must be already marked when \( b \)'s operation traverses to \( gp \), which is earlier than \( b \)'s successful mark CAS that thwarts \( a_1 \). Attempt \( a_2 \) pops \( p \) on line 98, and the next Recover-and-Traversal will pop \( gp \) because it has been marked by \( b \). It will also pop \( u_1, ..., u_n \) because they have been marked. If \( v \) is not marked (thus not popped), Recover-and-Traversal will re-read its update field on line 112-113. So \( a_2 \) also cannot be thwarted by \( b \)'s flag.

**Lemma 4.13** An attempt that fails its mark CAS cannot be directly thwarted by another mark CAS.

**Proof** Let \( a \) be a failed attempt that failed its mark CAS. Let \( b \) be the attempt that directly thwarts \( a \). To derive a contradiction, assume \( b \) has a successful mark CAS on the same node as \( a \)'s parent node. Let \( p_a \) and \( gp_a \) be the two nodes that \( a \) wrote in its Info object as \( p \) and \( gp \) respectively. Let \( p_b \) and \( gp_b \) be the two nodes that \( b \) wrote in its Info object as \( p \) and \( gp \) respectively.

Since \( a \) has a successful dflag, \( a \)'s operation must have seen a clean value on \( gp_a \) and \( p_a \)'s update field. Let \( t_1 \) and \( t_2 \) be the time when \( a \)'s operation reads clean values in \( gp_a \) and \( p_a \)'s state, respectively. Let \( t_3 \) be the time when a successfully performed its dflag CAS, and \( t_4 \) be the time when the first mark CAS that uses \( a \)'s Info object is performed (it fails because \( b \) has marked the same location). Let \( t_b \) be the time when \( b \) performed its mark CAS on \( p_b \) (which equals \( p_a \)), it must be between \( t_2 \) and \( t_4 \).

By Lemma 4.11, \( p_a \) is a child of \( gp_a \) from \( t_1 \) until \( t_4 \). So at time \( t_b \), \( gp_b \) equals \( gp_a \). By Lemma 9 of [4], \( b \)'s mark CAS must be preceded by a successful dflag CAS on \( gp_b \). Since \( gp_b \) is clean from \( t_1 \) until \( t_3 \), \( t_b \) cannot be within this duration of time. But \( t_b \) cannot be between \( t_3 \) and \( t_4 \) either, because during this period of time, \( gp_a \) is flagged by \( a \), thus cannot be flagged by \( b \) simultaneously. This contradicts the fact that \( t_b \) is between \( t_2 \) and \( t_3 \).

**Lemma 4.14** If each of two deletion attempts (not necessarily from same operation) fails its mark CAS, then the two attempts cannot have the same direct thwarting attempt.

**Proof** Let \( a \) and \( b \) be two deletion attempts that each have a failed mark CAS. To derive a contradiction, assume some attempt \( x \) directly thwarts both of them. By Lemma 4.13, \( a \) and \( b \) cannot be thwarted by \( x \)'s mark CAS, so \( a \) and \( b \) are thwarted by \( x \)'s flag CAS. Let \( p_a \) and \( gp_a \) be the two nodes that \( a \) wrote in its Info object as \( p \) and \( gp \) respectively. Let \( p_b \) and \( gp_b \) be the two nodes that \( b \) wrote in its Info object as \( p \) and \( gp \) respectively. So \( p_a \) is a child of \( gp_a \) when \( a \)'s operation traversed \( p_a \) and \( p_b \) is a child of \( gp_b \) when \( b \)'s operation traversed \( p_b \). Let \( t_a \) be the time when \( a \)'s operation reads \( gp_a \) is clean, or when \( a \) started,
whichever is later. Let \( t_b \) be the time when \( b \)'s operation reads \( gp_b \) is clean, or when \( b \) started, whichever is later. Since \( a \) and \( b \) have a successful dirflag, \( gp_a \) and \( gp_b \) are still clean at time \( t_a \) and \( t_b \). Without loss of generality, assume \( t_a \leq t_b \).

Since \( a \) and \( b \) are both directly thwarted by \( x \)'s flag, \( p_a \) equals \( p_b \). By the definition of direct thwarting, the first call to HELPDELETE that uses \( a \)'s Info object that executes line 135 sees \( x \)'s Info object in the update field of \( p_a \). The same argument applies to the first HELPDELETE that uses \( b \)'s Info object that executes line 135. So \( x \)'s flag CAS happens after both \( t_a \) and \( t_b \), but before the first mark CAS that uses \( a \)'s Info object and before the first mark CAS that uses \( b \)'s Info object. By Lemma 4.11, \( p_a \) is a child of \( gp_a \) from time \( t_a \) until the first mark CAS that belongs to \( a \) fails, and \( p_b \) is a child of \( gp_b \) from time \( t_b \) until the first mark CAS that belongs to \( b \) fails. So at the time \( x \) performed its flag CAS, \( gp_a \) equals \( gp_b \). Both \( a \) and \( b \) flagged this node (otherwise they will not try to perform their mark CAS). Thus, \( t_b \) must be after \( a \)'s flag is removed since \( b \)'s flag succeeds. Thus, \( t_b \) is after the first mark CAS that belongs to \( b \) fails. This contradicts the fact that \( x \)'s flag CAS occurs after \( t_b \) but before \( a \)'s failed mark CAS. 

The following lemmas show that when an operation has many failed attempts with the same ultimate thwarting attempt, only a constant number of them can be blamed on the ultimate thwarting attempt by rule 2(b) (since the rule 2(a) will blame most of them to concurrent successful update operations instead).

**Lemma 4.15** Let \( p \) be the parent node for some attempt \( a \) that searches for key \( k \). There is a configuration \( C \) during \( a \)'s execution, such that \( p \) is reachable and is on the search path for \( k \).

**Proof** If \( p \) is pushed to nodeStack by \( a \)'s RECOVER-AND-TRAVERSAL, then by Lemma 4.6, \( C \) exists. If \( p \) is pushed by some earlier attempt, it means \( a \) sees \( p \) is not marked on line 106. So \( C \) is the configuration when \( a \) executes this line. By Lemma 19 of [4], since \( p \) is unmarked and was previously on the search path for \( k \), it is still on the search path for \( k \) in \( C \).

**Lemma 4.16** Let \( a_1, a_2, ..., a_n \) be a sequence of failed attempts of an operation \( op \) (in order they occur) such that there is no successful child CAS between the start of \( a_1 \) and the end of \( a_n \). Let \( b_m \) be the last attempt of some operation (not necessarily the same as \( op \)). There are at most three attempts of \( a_1, a_2, ..., a_n \) that are ultimately thwarted by \( b_m \).

**Proof** Let \( T \) be the tree when \( a_1 \) started. Since no successful child CAS occurs until \( a_n \) ends, the tree is \( T \) throughout this period of time. Let \( l_i \) be the leaf node found by the RECOVER-AND-TRAVERSAL in \( a_i \). Let \( gp_i \) and \( p_i \) be the top two nodes on nodeStack after \( a_i \)'s RECOVER-AND-TRAVERSAL finishes. Let \( b_1, ..., b_m \) be the maximal sequence of deletion attempts such that each attempt is directly thwarted on its mark CAS by the next in sequence. By Lemma 4.14, if \( b_i \) is directly thwarts \( b_{i-1} \) on its mark CAS, it cannot directly thwart any other deletion attempts on its mark CAS, so this sequence is unique. So the only attempts that can be indirectly thwarted by \( b_m \) are those which are directly thwarted by \( b_1, ..., b_m-1 \). We have seen in the proof of Lemma 4.10 that each of \( b_1, ..., b_m \) flagged a different node. The following argument shows that a total of at most three attempts of \( a_1, a_2, ..., a_n \) are thwarted by a flag or mark that belongs to any of the attempts \( b_1, ..., b_m \).

**Case 1:** \( a_1, a_2, ..., a_n \) belong to an insertion attempt. By Lemma 4.15, there exists a configuration during \( a_1 \) such that \( p_1 \) is reachable. So \( p_1 \) is reachable in tree \( T \). If \( p_1 \) is marked at some time after \( a_1 \) started and before \( a_n \) executes line 52 or 62, then \( p_1 \) will be removed from the tree by the first \( a_i \) that sees that mark when helping or when popping \( p_1 \) out of nodeStack. Since no successful child CAS occurs, \( p_1 \) must be unmarked during this time. So \( p_1 \) is the parent node for all \( a_i \). Since \( b_1, ..., b_m \) flagged different nodes, at most one of them flags \( p_1 \). Let \( x \) be the \( b_i \) that flags \( p_1 \). By Lemma 4.12, each of \( a_1, a_2, ..., a_n \) has a different direct thwarting attempt, so only one of them is directly thwarted by \( x \).

**Case 2:** \( a_1, a_2, ..., a_n \) belongs to a deletion operation. By Lemma 4.15, there exists a configuration during \( a_1 \) such that \( p_1 \) is reachable. So, \( p_1 \) is reachable in tree \( T \). A failed deletion attempt will pop its parent node on line 98, but since \( p_1 \) is reachable in \( T \), the next attempt will re-push \( p_1 \) to the stack.

**Case 2(a):** If \( gp_1 \) is not in \( T \), it must be traversed by an earlier attempt, and already marked before \( a_1 \) started. In this case, \( a_1 \) would be thwarted by the attempt that marks \( gp_1 \), and \( a_2 \) would pop \( gp_1 \) out of
Let $v$ be the top node of nodeStack when $a_2$ exited the first loop of Recover-and-Traversal (line 106-110). Node $v$ is unmarked when $a_2$ executes line 106, so it is reachable in $T$. If some later $a_i$ sees $v$ is marked on line 106, it will help remove $v$ from the tree and pop it out of nodeStack. But there is no child CAS that occurs between the time $a_1$ starts and the time $a_i$ finishes, so all $a_i$ must have seen $v$ unmarked. We have argued that $a_2$ will re-push $p_1$ onto nodeStack. Node $v$ was pushed onto nodeStack earlier than $p_1$, so it is an ancestor of $p_1$. Since a node cannot have a new ancestor (see Lemma 18 of [4]), there is no new node on the path from $v$ to $p_1$. Since no successful child CAS occurs after $a_1$ starts until $a_n$ finishes, there are no new nodes inserted as a descendant of $p_1$. So $gp_2$ equals $v$, and $p_2$ equals $p_1$. Furthermore, $v$ stays on nodeStack until $a_n$ finishes its Recover-and-Traversal, and $p_1$ is always popped by $a_i$ and re-pushed by $a_{i+1}$, so for all $i > 1$, $gp_i$ equals $v$ and $pi$ equals $p_1$. We know that $a_1$ is directly thwarted by the attempt that marks $gp_1$, so the only attempt among $b_1, \ldots, b_m$ that can directly thwart $a_1$ is $b_m$, because each of $b_1, \ldots, b_m-1$ failed its mark CAS. By Lemma 4.12, each of $a_1, \ldots, a_n$ has a different direct thwarting attempt, so at most two of them are directly thwarted by the attempts $b_i$ and $b_j$ that flag $v$ and $p_1$. So at most three attempts of $a_1, a_2, \ldots, a_n$ are ultimately thwarted by $b_m$.

**Case 2(b):** If $gp_1$ is in $T$, then with the same argument as for $a_2, \ldots, a_n$ in Case 2(a), $gp_1$ and $p_1$ will be the grandparent and parent node for all $a_i$. So at most two attempts of $a_1, a_2, \ldots, a_n$ are directly thwarted by the attempts $b_i$ and $b_j$ that flag $gp_1$ and $p_1$. 

By rule 2(a), if two failed attempts of an operation $op_1$ are ultimately thwarted by the same attempt $b$, and a successful child CAS ccas occurs between the two failed attempts, we blame the later attempt to the operation that owns ccas. So we are left with attempts that are ultimately thwarted by $b$ such that no successful child CAS occurred between them (by rule 2(b)). Lemma 4.16 shows that $op_1$ cannot blame more than three failed attempts on the operation that owns $b$ using rule 2(b). Furthermore, if this happens, $op$ must be concurrent with the beginning of $b$. Thus, the total number of steps blamed on each operation by rule 2(b) is $O(c)$. Lastly, we shall bound the number of failed attempts that are blamed on an operation that successfully modified the tree (by rule 2(a)) and to the operation that owns the failed attempt itself (by rule 2(c)).

**Lemma 4.17** For any failed attempt $a$ of operation $op$, there is a point in time after $op$ started and before $a$ finishes, such that $op$ runs concurrently with $a$’s direct thwarting attempt $b$.

**Proof** If $a$ does not perform its CAS because it sees an un-clean value on a node $x$ (which can be $a$’s parent or grandparent), then by the time $op$ traverses to $x$, $b$ has already written its Info object in $x$, but has not yet performed an unflag or backtrack CAS on it. So $op$ and $b$ are concurrently running at the time $op$ traverses to $x$.

If $a$ performed its CAS on some node $x$ (which can be $a$’s parent or grandparent), but failed to write its Info object because $b$ has written to it, then $b$’s CAS must happen after $op$ traverses to $x$ and before $a$’s failed CAS. So at the time when $b$ successfully wrote its Info object on $x$, $op$ is concurrently running with $b$.

**Lemma 4.18** For any failed deletion attempt $a$ that fails its mark CAS, there is a point in time where a runs concurrently with its direct thwarting attempt $b$.

**Proof** Since $a$ performed its mark CAS, its pupdate and gpupdate are clean. Attempt $a$ re-reads the top node’s update field on line 112-113, so pupdate is either read by these line, or pushed during the second loop of Recover-and-Traversal. So pupdate was clean at some time during $a$’s execution, but it is un-clean when the first mark that uses $a$’s Info object occurs (line 135). So, when the direct thwarting attempt $b$ writes its Info object on $a$’s parent node, $a$ is running.

**Lemma 4.19** Let $b_0, \ldots, b_n$ be a sequence of deletion attempts, such that each attempt has a successful dfalg and is directly thwarted by the next ($b_0$ is not necessarily the ultimate thwarting attempt). Let $t_{start}$ be the earliest of the starting times of $b_0$ and $b_n$. Let $t_{end}$ be the latest of the finishing times of $b_0$ and $b_n$. At any time between $t_{start}$ and $t_{end}$, at least one attempt of $b_0, \ldots, b_n$ is running.
Proof Let \( \text{start}(x) \) be the starting time of attempt \( x \), and \( \text{end}(x) \) be the finishing time of attempt \( x \). Let \( s_i \) be the earliest of \( \text{start}(b_0), \ldots, \text{start}(b_i) \), and let \( e_i \) be the latest of \( \text{end}(b_0), \ldots, \text{end}(b_i) \). We prove by induction on \( i \) that at any time between \( s_i \) and \( e_i \), at least one of \( b_0, \ldots, b_i \) is active. The lemma then follows from the facts that \( s_n \leq \text{start} \) and \( e_n \geq \text{end} \).

**Base Case:** \( i = 0 \). In this case, \( s_0 = \text{start}(b_0) \), and \( e_0 = \text{end}(b_0) \). \( b_0 \) is running throughout this duration of time, so the claim holds for the base case.

**Induction Step:** Assume that the claim holds for \( i \), we prove that the claim also holds for \( i + 1 \). If \( \text{start}(b_{i+1}) \) is earlier than \( s_i \), then \( s_{i+1} = \text{start}(b_{i+1}) \). Otherwise, \( s_{i+1} = s_i \). If \( \text{end}(b_{i+1}) \) is later than \( e_i \), then \( e_{i+1} = \text{end}(b_{i+1}) \). Otherwise, \( e_{i+1} = e_i \).

By the induction hypothesis, at least one of \( b_0, \ldots, b_i \) is active between \( s_i \) and \( e_i \), so at least one of \( b_0, \ldots, b_{i+1} \) is running between \( s_i \) and \( e_i \). If \( s_{i+1} \) is earlier than \( s_i \), then \( b_{i+1} \) must be running from time \( s_{i+1} \) until \( s_i \) (by Lemma 4.18, \( b_i \)'s execution time overlaps with \( b_{i+1} \)'s). Similarly, if \( e_{i+1} \) is later than \( e_i \), then \( b_{i+1} \) must be running from time \( e_i \) until \( e_{i+1} \) (by Lemma 4.18). So at least one of \( b_0, \ldots, b_{i+1} \) is running from time \( s_{i+1} \) until \( e_{i+1} \).

**Lemma 4.20** Let \( c_{\text{start}} \) and \( c_{\text{end}} \) be the number of concurrent operations at the time an operation \( op \) started and ended respectively. The number of \( op \)'s failed attempts that are blamed on \( op \) itself by rule 2(c) is \( O(c_{\text{start}} + c_{\text{end}}) \).

**Proof** Lemma 4.16 shows that each ultimate thwarting attempt can be blamed for at most three failed attempts of each other operation. If a failed attempt \( a \), which has ultimate thwarting attempt \( b_n \), is blamed on \( op \) itself, it means that \( op \) is not active at the starting time of \( b_n \). Let \( a, b_1, \ldots, b_n \) be the sequence of attempts such that each attempt is directly thwarted by the next one. There are two possible cases: the first one is if the time \( b_n \) started is earlier than the time \( op \) started. By Lemma 4.17, \( b_i \) is concurrent with \( op \). So the time \( op \) starts is between the time \( b_i \) starts and the time \( b_i \) finishes. By Lemma 4.19, at any time between the starting time of \( b_i \) and the ending time of \( b_i \), there exists an attempt of \( b_1, \ldots, b_i \) that is active at the time \( op \) started. Let \( x \) be such an attempt. By Lemma 4.14, since \( x \) directly thwarts the preceding attempt in the sequence, it cannot directly thwart any other deletion attempts on its mark CAS, so \( x \) is unique. Thus, for each attempt \( x \) running at the time \( op \) started, at most three attempts of \( op \) are blamed on \( op \) itself. So, the number of failed attempts of \( op \) that are blamed on \( op \) itself is bounded by \( c_{\text{start}} \). With a similar argument, if \( b_n \)'s starting point is later than \( op \)'s finishing time, then there exists a unique deletion attempt that is active at the time \( op \) finishes, so the number of \( op \)'s failed attempts that are blamed on \( op \) itself is bounded by \( c_{\text{end}} \). So the total number of \( op \)'s failed attempts that are blamed on \( op \) itself is \( O(c_{\text{start}} + c_{\text{end}}) \).

**Lemma 4.21** For each successful child CAS \( ccas \), let \( c \) be the contention at the time \( ccas \) occurs. The total number of failed attempts that are blamed on the operation that owns \( ccas \) by rule 2(a) is bounded by \( O(c^2) \).

**Proof** If an operation \( op \) blames a failed attempt on \( ccas \), by definition, \( ccas \) must happen during \( op \)'s attempts, so \( op \) is running at the time \( ccas \) occurs. Thus, the number of operations that can blame attempts on the operation that owns \( ccas \) using rule 2(a) is bounded by \( c \). Now we shall bound the number of attempts that each concurrent operation can blame on the operation that owns \( ccas \).

Let \( a_1, \ldots, a_n \) be \( op \)'s attempts that are blamed by rule 2(a) on the operation that owns \( ccas \) (in order they occurred). By the definition of rule 2(a), for each \( a_i \), there exists an earlier attempt \( a'_i \) of \( op \) that has the same ultimate thwarting attempt as \( a_i \), and \( ccas \) occurs after \( a'_i \) starts and before \( a_i \) finishes. Since \( ccas \) occurs before \( a_1 \) finishes, it must be before any of \( a_2, \ldots, a_n \) start. Since rule 2(a) blames \( a_1, \ldots, a_n \) on the operation that owns \( ccas \), by the definition of rule 2(a), \( ccas \) is the last successful child CAS that occurs before each of \( a_1, \ldots, a_n \) finishes. Thus, there is no other successful child CAS after \( a_2 \) starts and before \( a_n \) finishes (\( ccas \) can occur during \( a_1 \) or before \( a_1 \)). Attempt \( a_1 \) is a special attempt, there is one such attempt (i.e., the earliest attempt of \( op \) that is blamed on the operation that owns \( ccas \)). Now we bound the number of attempts \( a_2, \ldots, a_n \).

Since no successful child CAS occurs after \( a_2 \) starts and before \( a_n \) finishes, at most three attempts of \( a_2, \ldots, a_n \) have the same ultimate thwarting attempt, by Lemma 4.16. We use a similar argument as
for Lemma 4.20 to show that for each ultimate thwarting attempt of an attempt $a_i$, there exists a unique attempt that runs at the time $ccas$ occurs. Since at most three attempts can share the same ultimate thwarting attempt, $n$ is at most $3c$.

Let $x_i$ be the ultimate thwarting attempt of $a_i$ and $a'_i$. Let $b_i$ and $b'_i$ be the direct thwarting attempts of $a_i$ and $a'_i$, respectively. By Lemma 4.12, $a'_i$ and $a_i$ cannot be directly thwarted by the same attempt, so $b_i \neq b'_i$. Let $c_1, \ldots, c_m$ be the maximal sequence of deletion attempts such that each attempt has a successful $dflag$ and is thwarted on its mark CAS by the next attempt in the sequence, and attempt $c_m$ is directly thwarted by $x_i$. By Lemma 4.14, $c_{i-1}$ is the only deletion attempt that fails its mark CAS and is thwarted by $c_i$. Attempts $b'_i$ and $b_i$ are attempts that thwart $a'_i$ and $a_i$, so they must belong to the sequence $c_1, \ldots, c_m, x_i$, because $a_i$ and $a'_i$ are ultimately thwarted by $x_i$. So either $b'_i$ is thwarted by $b_i$ (possibly indirectly), or vice versa. We consider the case where $b'_i$ is thwarted by $b_i$, and a symmetric argument applies for when $b_i$ is thwarted by $b'_i$.

By Lemma 4.17, there exists a point in time before $a'_i$ finishes such that $b'_i$ is running, and there exists a point in time before $a_i$ finishes such that $b_i$ is running. Let $t_{ccas}$ be the time when $ccas$ occurs. We consider three cases based on the time $b'_i$ and $b_i$ occur relative to $t_{ccas}$.

**Case 1:** both $b'_i$ and $b_i$ finish before $t_{ccas}$. For $i \geq 2$, $a_i$ starts later than $t_{ccas}$. Since $b_i$ is finished before $t_{ccas}$ (which is before $a_i$ started), $b_i$'s $flag$ has been removed when $a_i$ started, and if $b_i$ has marked a node, it is no longer reachable. By Lemma 4.15, the parent node of $a_i$ is reachable, and RECOVER-AND-TRAVERE re-reads the top node's update field on line 112-113, so $a_i$ cannot be thwarted by $b_i$ on its parent node. So $a_i$ can only be thwarted by $b'_i$ if $a_i$ is a deletion attempt whose $gpupdate$ is read by an earlier attempt.

Let $p$ be the parent node for $a_i$. When $a_i$ fails, it pops its parent node on line 98, so the next attempt can pop the grandparent node (if it is already marked) or re-read its update field. Since no child CAS occurs between the start of $a_2$ and the end of $a_n$, $p$ must be still reachable, and is re-pushed by the next attempt after $a_i$. So $a_i$ is the only attempt that does not read its $gpupdate$ during its attempt itself. (Each subsequent attempt will reach the same grandparent and parent node, and will read the update field of the grandparent node during the call to RECOVER-AND-TRAVERSE.) Thus, at most one attempt falls into case 1.

**Case 2:** both $b'_i$ and $b_i$ start after $t_{ccas}$. By Lemma 4.17, there exists a point in time before $a'_i$ finishes such that $b'_i$ is running, so the only case where both $b'_i$ and $b_i$ start after $t_{ccas}$ is when $a'_i$ running at time $t_{ccas}$. There is only one attempt of $op$ that runs at $t_{ccas}$, so at most one attempt $a'_i$ of $op$ that falls into this category. As argued above, there are at most three attempts $a_j$ with the same ultimate thwarting attempt as this $a'_i$.

**Case 3:** otherwise. Then, either (1) one of $b'_i$ and $b_i$ is running at $t_{ccas}$ or (2) one of $b'_i$ and $b_i$ ends before $t_{ccas}$, and the other begins after $t_{ccas}$. Either way, one of $b'_i$ and $b_i$ starts before $t_{ccas}$ and the other ends after $t_{ccas}$.

We have showed that $b'_i$ and $b_i$ belong to the sequence $c_1, \ldots, c_m, x_i$. By Lemma 4.19, at least one of $b'_i, \ldots, b_i$ is active at time $t_{ccas}$. By Lemma 4.16, at most three attempts of $a_2, \ldots, a_n$ are ultimately thwarted by $x_i$. Thus, the number of failed attempts that $op$ blames on the operation that owns $ccas$ that falls into this category is bounded by $3c$.

We have argued that $op$ has one special attempt ($a_1$), at most one attempt of the first case, at most one attempt of the second case, and at most $3c$ attempts of the third case. So $n$ is bounded by $3 + 3c$. There are at most $c$ operations running at time $t_{ccas}$, so the total number of attempts blamed on the operation that owns $ccas$ by rule 2(a) is $O(c^2)$. We have argued that $op$ has one special attempt ($a_1$), at most one attempt of the first case, at most one attempt of the second case, and at most $3c$ attempts of the third case. So $n$ is bounded by $O(4 + 3c)$. There are at most $c$ operations running at time $t_{ccas}$, so the total number of attempts blamed on the operation that owns $ccas$ by rule 2(a) is $O(c^2)$.  

**4.3 Time Complexity**

**Lemma 4.22** Let $h_{start}$ be the height of the tree when a FIND operation $op$ started. The amortized cost of $op$ is $O(h_{start})$. 

23
The last attempt of \( op \) is blamed on \( op \) itself. By Lemma 4.21, the number of failed attempts that are blamed on \( op \) by rule 2(a) is bounded by \( O(c_{\text{child}}^2) \). By Lemma 4.16, each operation that runs at the time \( op \)'s last attempt started can blame at most three attempts on \( op \), so the total number of attempts blamed on \( op \) by rule 2(b) is bounded by \( O(c_{\text{lastattempt}}) \). By Lemma 4.20, the number of failed attempts that are assigned to the operation itself by rule 2(c) is bounded by \( O(c_{\text{start}} + c_{\text{end}}) \). So in total, the amortized cost of \( op \) is \( O(h_{\text{start}} + c^2) \).

**Lemma 4.24** Let \( c_{\text{start}}, c_{\text{end}}, c_{\text{lastattempt}}, \) and \( c_{\text{dchild}} \) be the contention at the time a DELETE operation \( op \) starts, ends, begins its last attempt, and when \( op \)'s successful \( \text{dchild} \) CAS (if any) occurs, respectively. Let \( c \) be \( \max(c_{\text{start}}, c_{\text{end}}, c_{\text{lastattempt}}, c_{\text{dchild}}) \). Let \( h_{\text{start}} \) be the height of the tree when \( op \) started. The amortized cost of \( op \) is \( O(h_{\text{start}} + c) \).}

**Proof** By Lemma 4.9, the number of traversals that are blamed on the operation itself is \( O(h_{\text{start}}) \). By Lemma 4.7, the number of traversals that are blamed on \( op \) by the second rule of the traversal blaming scheme is bounded by \( c_{\text{child}} \).

The last attempt of \( op \) is blamed on \( op \) itself. By Lemma 4.21, the number of failed attempts that are blamed on \( op \) by rule 2(a) is bounded by \( O(c_{\text{child}}^2) \). By Lemma 4.16, each operation that runs at the time \( op \)'s last attempt started can blame at most three attempts on \( op \), so the total number of attempts blamed on \( op \) by rule 2(b) is bounded by \( O(c_{\text{lastattempt}}) \). By Lemma 4.20, the number of failed attempts that are assigned to the operation itself by rule 2(c) is bounded by \( O(c_{\text{start}} + c_{\text{end}}) \). So in total, the amortized cost of \( op \) is \( O(h_{\text{start}} + c^2) \).

**5 Obstacle to obtaining an \( O(h + c) \) bound**

We have showed that the amortized cost of the FIND operation is \( O(h_{\text{start}}) \), and the amortized cost of INSERT and DELETE operations are \( O(h_{\text{start}} + c^2) \). Our original goal was to prove that the amortized cost of the modified algorithm is \( O(h + c) \). However the tightest bound we got so far for the number of attempts blamed on an update by the second rule (i.e., Lemma 4.21) is \( O(c^2) \). Although the bound we get is \( O(c^2) \), we could not find any concrete example of an execution that would show this amortized cost is tight. We believe that for each successful \( \text{child} \) CAS \( c_{\text{cas}} \), each concurrent operation with \( c_{\text{cas}} \) can only blames \( O(1) \) failed attempts to the operation that owns \( c_{\text{cas}} \). So the total number of failed attempts blamed on an update operation by the second rule is \( O(c) \).

**Conjecture** Let \( a_1, \ldots, a_n \) be a sequence of failed attempts that belong to the same operation \( op \) (in order), such that there is no successful \( \text{child} \) CAS that occurs after \( a_1 \) starts and before \( a_{n-1} \) finishes, and there are \( O(1) \) successful \( \text{child} \) CAS that occurs after \( a_{n-1} \) starts and before \( a_n \) finishes. Let \( a'_1, \ldots, a'_n \) be the ultimate thwarting attempts of \( a_1, \ldots, a_n \) respectively. At most \( O(1) \) attempts of \( op \) after \( a_n \) are ultimately thwarted by any of \( a'_1, \ldots, a'_n \).
References


