

Quadboost: A Scalable Concurrent Quadtree

Keren Zhou, Guangming Tan, Wei Zhou

Abstract—Building concurrent spatial trees is more complicated than binary search trees since a space hierarchy should be preserved during modifications. We present a non-blocking quadtree—*quadboost*—that supports concurrent insert, remove, move, and contain operations. To increase its concurrency, we propose a decoupling approach that separates physical adjustment from logical removal within the remove operation. In addition, we design a continuous find mechanism to reduce its search cost. The move operation combines the searches for different keys together and modifies different positions with atomicity. The experimental results show that *quadboost* scales well on a multi-core system with 32 hardware threads. More than that, it outperforms existing concurrent trees in retrieving two-dimensional keys with up to 109% improvement when the number of threads is large. The move operation proved to perform better than the best-known algorithm, with up to 47%.

Index Terms—Concurrent Data Structures, Quadtree, Continuous Find, Decoupling, LCA



1 INTRODUCTION

MULTI-CORE processors are currently the universal computing engine in computer systems. Therefore, it is urgent to develop data structures that provide an efficient and scalable multi-thread execution. At present, concurrent data structures [1], such as stacks, linked-lists, and queues have been extensively investigated. As a fundamental building block of many parallel programs, these concurrent data structures provide significant performance benefits [2], [3].

Recently, research on concurrent trees is focused on binary search trees (BSTs) [4], [5], [6], [7], [8], [9], [10], which are at the heart of many tree-based algorithms. The concurrent paradigms of BSTs were extended to design concurrent spatial trees like R-Tree [11], [12]. However, there remains another unaddressed spatial tree—quadtree, which is widely used in applications for multi-dimensional data. For instance, spatial databases, like PostGIS [13], adopt octree, a three-dimensional variant of quadtree, to build spatial indexes. Video games apply quadtree to handle collision detection [14]. In image processing [15], quadtree is used to decompose pictures into separate regions.

There are different categories of quadtree according to the type of data a node represents, where two major types are region quadtree and point quadtree [16]. Point quadtree stores actual keys in each node, yet it is hard to design concurrent algorithms for point quadtree since an insert operation might involve re-balance issues, and a remove operation needs to re-insert the whole subtree under a removed node. The region quadtree divides a given region into several sub-regions, where internal nodes represent regions and leaf nodes store actual keys. Our work focuses on region quadtree for two reasons: (1) The shape of region quadtree is independent of insert/remove operations' order, which allow us to avoid complex re-balance rules and devise specific concurrent techniques for it. (2) Further, region quadtree could be regarded as a typical external tree that

we can adjust existing concurrent techniques from BSTs. Therefore, we will refer to region quadtree as *quadtree* in the following context.

In this paper, we design a non-blocking quadtree, referred to as *quadboost*, that supports concurrent insert, remove, contain, and move operations. Our key contributions are as follows:

- We propose the first non-blocking quadtree. It records traversal paths, compresses *Empty* nodes if necessary, adopts a decoupling technique to increase the concurrency, and devises a continuous find mechanism to reduce the cost of retries induced by CAS failures.
- We design a lowest common ancestor (LCA) based move operation, which traverses a common path for two different keys and modifies two distinct nodes with atomicity.
- We prove the correctness of *quadboost* algorithms and evaluate them on a multi-core system. The experiments demonstrate that *quadboost* is highly efficient for concurrent updating at different contention levels.

The rest of this paper is organized as follows. In Section 2, we overview some basic operations. Section 3 describes a simple CAS quadtree to motivate this work. Section 4 provides detailed algorithms for *quadboost*. We provide a sketch of correctness proof in Section 5. Experimental results are discussed in Section 6. Section 7 summarizes related works. Section 8 concludes the paper.

2 PRELIMINARY

Quadtree can be considered as a dictionary for retrieving two-dimensional keys, where $\langle keyX, keyY \rangle$ is never duplicated. Figure 1 illustrates a sample quadtree and its corresponding region, where we use numbers to indicate keys. Labels on edges are routing directions—Southwest (sw), Northwest (nw), Southeast (se), and Northeast (ne).

- K.Zhou and G.Tan are with Institute of Computing Technology, Chinese Academy of Sciences.
- W.Zhou is with School of Software, Yunnan University.

The right picture is a mapping of the quadtree on a two-dimensional region, where keys are located according to their coordinates, and regions are divided by their corresponding width (w) and height (h). There are three types of nodes in the quadtree, which represent different regions in the right figure. *Internal* nodes are circles on the left figure, and each of them has four children which indicate four equal sub-regions on different directions. The root node is an *Internal* node, and it is the largest region. *Leaf* nodes and *Empty* nodes are located at the terminal of the quadtree; they indicate the smallest regions on the right figure. *Leaf* nodes are solid rectangles that store keys; they represent regions with the same numbers on the right figure. *Empty* nodes are dashed rectangles without any key.

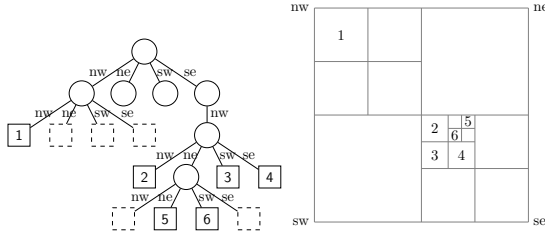


Fig. 1: A sample quadtree and its corresponding region

We describe the detailed structures of a quadtree in Figure 2. An *Internal* node maintains its four children and a two-dimensional routing structure $\langle x, y, w, h \rangle$, where $\langle x, y \rangle$ stands for the upper left coordinate and $\langle w, h \rangle$ are the width and height of the region. A *Leaf* node contains a key $\langle keyX, keyY \rangle$ and its corresponding *value*. An *Empty* node does not have any field. To avoid some corner cases, we initially split the root node and its children to form two layers of dummy *Internal* nodes with a layer of *Empty* nodes at the terminal. Also, we present routing functions *find* and *getQuadrant* in the figure. For instance, if we have to locate key 1 in Figure 1, we start with the root node and compare key 1 with its routing structures by *getQuadrant*. Then we reach its *nw* and perform a comparison again. In the end, we find the terminal node that contains key 1.

```

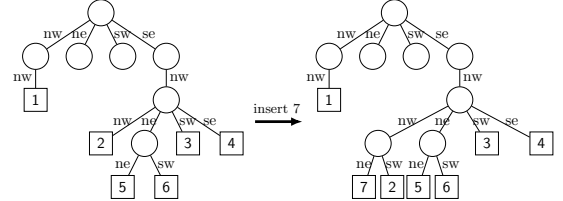
1  class Node<V> {}
2  class Internal<V> extends Node<V> {
3      final double x, y, w, h;
4      Node nw, ne, sw, se;
5  }
6  class Leaf<V> extends Node<V> {
7      final double keyX, keyY;
8      final V value;
9  }
10 class Empty<V> extends Node<V> {}
11 void find(Node& l, double keyX, double keyY) {
12     while (l.class() == Internal) {
13         getQuadrant(l, keyX, keyY);
14     }
15 }
16 void getQuadrant(Node& l, double keyX, double keyY) {
17     if (keyX < l.x + l.w / 2) {
18         if (keyY < l.y + l.h / 2) l = l.nw;
19         else l = l.sw;
20     } else {
21         if (keyY < l.y + l.h / 2) l = l.ne;
22         else l = l.se;
23     }
24 }

```

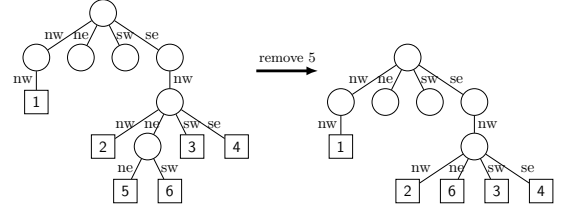
Fig. 2: Quadtree nodes and routing functions

There are four basic operations that rely on *find*:

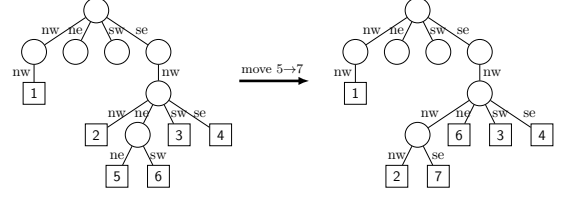
- 1) *insert(key, value)* adds a node that contains *key* and *value* into a quadtree.
- 2) *remove(key)* deletes an existing node with *key* from a quadtree.
- 3) *contain(key)* checks whether *key* is in a quadtree.
- 4) *move(oldKey, newKey)* replaces an existing node with *oldKey* and *value* by a node with *newKey* and *value* that is not in a quadtree.



(a) Insert node 7 into the sample quadtree



(b) Remove node 5 from the sample quadtree



(c) Move the value of node 5 to node 7 from the sample quadtree

Fig. 3: Sample quadtree operations¹

To insert a node, we first locate its position by calling *find*. If we find an *Empty* node at the terminal, we directly replace it by the node. Otherwise, before replacing the candidate node, we recursively divide a *Leaf* node into four corresponding sub-regions until the candidate region is *Empty*. Figure 3a illustrates a scenario of inserting node 7 (*insert(7)*) as a neighborhood of node 2. The parent node is split, and node 7 is added on the *ne* direction. Likewise, to remove a node, we also begin by locating it and then erase it from a quadtree. Next, we check whether its parent contains a single *Leaf* node. If so, we record the node and traverse up until reaching a node that contains at least a *Leaf* node or the child of the root node. Finally, we use that node as a new parent and re-insert the recorded node as its child. Take Figure 3b as an example, if we remove node 5 (*remove(5)*) from a quadtree, node 6 will be linked to the upper level. The move operation is a combination of the insert operation and the remove operation. It first removes the node with *oldKey* and then adds a new node with *newKey* into a quadtree. Consider the scenario in Figure 3c, after removing node 5 and inserting node 7 with the same value (*move(1, 7)*), the new tree appears on the right part. Because *contain(key)* just checks whether the node returned by *find* has the same *key*, it does not need an extra explanation.

1. *Empty* nodes are not drawn

3 CAS QUADTREE

```

25 bool contain(double keyX, double keyY) {
26     Node p, l = root;
27     // l: terminal node for retrieving <keyX, keyY>
28     // p: parent of l
29     find(p, l, keyX, keyY);
30     if (inTree(l, keyX, keyY)) return true;
31     return false;
32 }

33 bool insert(double keyX, double keyY, V value) {
34     while (true) {
35         Node p, l = root;
36         find(p, l, keyX, keyY);
37         if (inTree(l, keyX, keyY)) return false;
38         Node newNode = createNode(l, p, keyX, keyY, value); // creates a
// sub-tree by split l until the candidate region of <keyX, keyY> is
// not a Leaf node and returns the sub-tree's root node.
39         if (helpReplace(p, l, newNode)) return true;
40     }
41 }

42 bool remove(double keyX, double keyY, V value) {
43     Node newNode = new Empty();
44     while (true) {
45         Node p, l = root;
46         find(p, l, keyX, keyY);
47         if (!inTree(l, keyX, keyY)) return false;
48         if (helpReplace(p, l, newNode)) return true;
49     }
50 }

51 void find(Node& p, Node& l, double keyX, double keyY) {
52     while (l.class() == Internal) {
53         p = l; // record the parent node
54         getQuadrant(l, keyX, keyY);
55     }
56 }

57 bool helpReplace(Internal p, Node oldChild, Node newChild) {
58     if (p.nw == oldChild) return CAS(p.nw, oldChild, newChild);
59     else if (p.ne == oldChild) return CAS(p.ne, oldChild, newChild);
60     else if (p.sw == oldChild) return CAS(p.sw, oldChild, newChild);
61     else if (p.se == oldChild) return CAS(p.se, oldChild, newChild);
62     return false;
63 }

64 bool inTree(Node node, double keyX, double keyY) {
65     return node.class == Leaf && node.keyX == keyX && node.keyY == keyY;
66 }

```

Fig. 4: CAS quadtree algorithm

There are a plenty of concurrent tree algorithms, yet a formal concurrent quadtree has not been studied. Intuitively, we can devise a concurrent quadtree by modifying the sequential algorithm described in Section 2 and adopting CAS instruction, which we name CAS quadtree. Figure 4 depicts CAS quadtree’s algorithms. Both the insert operation and the remove operation follow such a paradigm: it starts by locating a terminal node; then, it checks whether the node satisfies some conditions. If conditions are not satisfied, it returns false. Otherwise, it tries to apply a single CAS to replace the terminal node by a new node. After a successful CAS, it returns true. Or else, it restarts locating a terminal node from the root node. For an insert operation, if $\langle \text{keyX}, \text{keyY} \rangle$ is not in the tree, it creates a corresponding sub-tree that contains $\langle \text{keyX}, \text{keyY} \rangle$ and *value* (line 38). Then it uses the root of the sub-tree to replace the terminal node by a single CAS (line 39). To remove a node, it creates an *Empty* node at the beginning (line 43). If $\langle \text{keyX}, \text{keyY} \rangle$ is in the tree, it uses the *Empty* node to replace the terminal node (line 48). Different from the sequential algorithm, we do not adjust the structure as it involves several steps that cannot be implemented with atomicity. The algorithm is non-blocking. In other words, the algorithm provides a guaranteed whole progress, even if some threads starve.

However, CAS quadtree has several limitations. First, consider if there are a considerable proportion of remove operations. By applying the simple mechanism, we still have a large number of nodes in the quadtree because we substitute existing nodes with *Empty* nodes without structural

T₁: remove 3

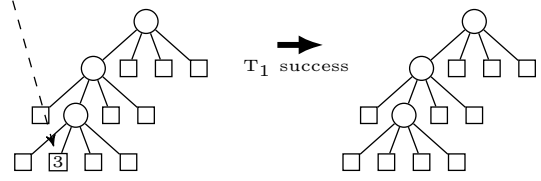


Fig. 5: Thread T₁ intends to remove node 3 from quadtree. After its removal, there remains a chain of *Empty* nodes

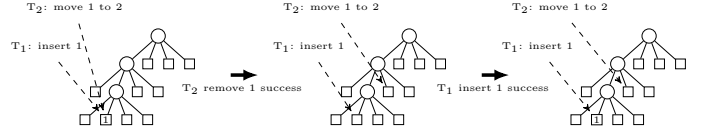


Fig. 6: An example of an incorrect move operation. Thread T₁ intends to insert node 1 into a quadtree, and thread T₂ plans to move the value from node 1 to node 2. T₂ first successfully removes node 1 and then attempts to insert node 2 into the quadtree. In the interval node 1 is added back by T₁, but T₂ is not aware of the action and reports success.

adjustment. Figure 5 illustrates a detailed example showing that there remains a chain of *Empty* nodes. Hence, not only we have to traverse a long path to locate the terminal node for basic operations, but also a plenty of nodes are maintained in the memory.

Second, we cannot implement the move operation in the simple algorithm. There might be two different nodes in a quadtree that were under modifying. If we directly erase the node with *oldKey* and add the node with *newKey*, the move operation cannot be correctly linearized. Figure 6 shows such an incorrect move operation by combining an insert operation and a remove operation together.

Therefore, these drawbacks motivate us to develop a new concurrent algorithm to make the move operation correct and employ an efficient mechanism to compress nodes.

4 QUADBOOST

4.1 Rationale

In this section, we describe how to design *quadboost* algorithms to solve the two problems addressed in Section 3.

To make the move operation correct, we should ensure that other threads know whether a terminal node is under moving. Hence, we attach an internal node with a separate object-*Operation* (*op*) to represent a node’s state and record sufficient information to complete the operation. We instantiate the attachment behavior as a CAS and call it a *flag* operation. We design different *ops* for the insert operation, the remove operation and the move operation. The detailed description of structures and a state transition mechanism is presented in Section 4.2.

To erase *Empty* nodes from a quadtree, we can apply a similar paradigm in the concurrent BST’s removal [4] as shown in Figure 7a, which flags both the parent and the

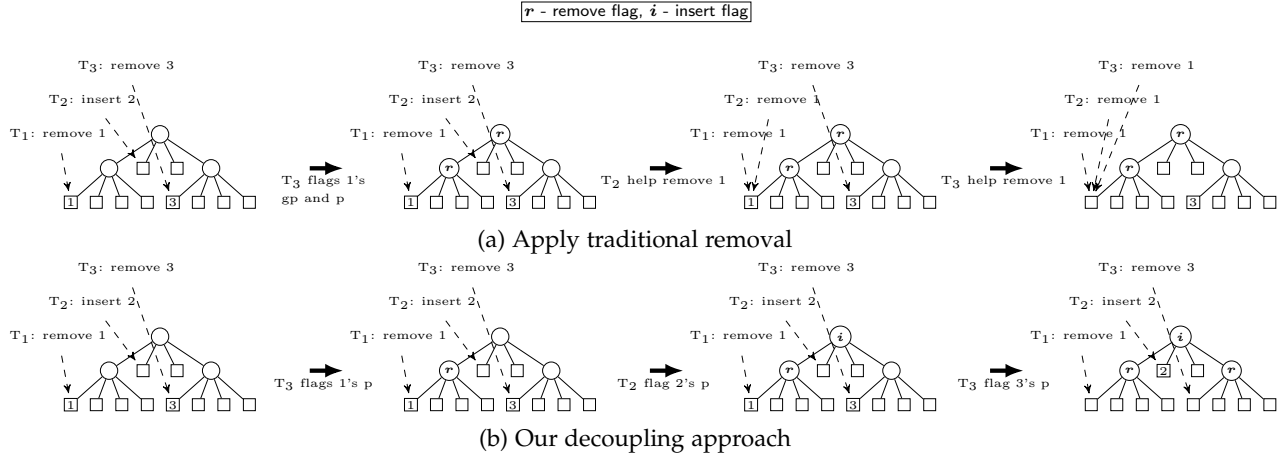


Fig. 7: At the beginning, three threads are performing different operations concurrently: (a) T_1 removes key 1 in the lower level. (b) T_2 inserts key 2 in the upper level. (c) T_3 removes key 3 in the lower level. Consider the scenario that T_1 precedes others threads, and then both 1's parent and grandparent will be flagged. Hence, T_2 and T_3 will help T_1 remove key 1 before restarting their operations. By applying our decoupling method, only the parent node should be flagged. Hence three threads could run without intervention.

grandparent of a terminal node. This mechanism mixes logical removal and physical removal together. Different from the BST's removal in which every time the parent node has to be adjusted, we only have to compress the parent when there is only a single *Leaf* child. Hence, we could separate the removal of a node and the adjustment of the structure into two phases. Meanwhile, we design an *op-Compress* to indicate a node is underlying the structural adjustment. Figure 7b illustrates how this method increases concurrency: three threads that handle different *ops* could run in parallel. Further, for simplicity, we relax the adjustment condition to where the parent could be compressed if all children are *Empty*.

However, there is still a problem left after applying the above two methods. Recall the example in Figure 5. We flag the bottom *Internal* node to indicate that it should be compressed, but after replacing the bottom *Internal* node with an *Empty* node, it results in four *Empty* nodes in the last level. How do we remove a series of nodes from a quadtree in a bottom-up way? We record the entire traversal path from the root to a terminal node in a stack. Since the traversal path will be altered when a node is compressed, we only have to restart locating the terminal node from its parent if any flag operation other than the flag of *Compress* fails. This is called the continuous find mechanism.

4.2 Structures and State Transitions

As mentioned before, we add an *Operation* object to handle concurrency issues. Figure 8 shows the data structures of *quadboost*. Four sub-classes of *Operation*, including *Substitute*, *Compress*, *Move*, and *Clean*, describe all states in our algorithm. *Substitute* provides information on the insert operation and the remove operation that are designed to replace an existing node by a new node. Hence, we shall let other threads be aware of its parent, child, and a new node for substituting. *Compress* provides information for physical adjustment. We erase the parent node, previously connected to the grandparent, by swinging the link to an *Empty* node.

```

67 class Node<V> {}
68 class Internal<V> extends Node<V> {
69     final double x, y, w, h;
70     volatile Node nw, ne, sw, se;
71     volatile Operation op = new Clean();
72 }
73 class Leaf<V> extends Node<V> {
74     final double keyX, keyY;
75     final V value;
76     volatile Move op;
77 }
78 class Empty<V> extends Node<V> {}
79 class Operation {}
80 class Substitute extends Operation {
81     Internal parent;
82     Node oldChild, newNode;
83 }
84 class Compress extends Operation {
85     Internal grandparent, parent;
86 }
87 class Move extends Operation {
88     Internal iParent, rParent;
89     Node oldChild, oldRChild, newChild;
90     Operation oldOp, oldROp;
91     volatile bool allFlag = false, iFirst = false;
92 }
93 class Clean extends Operation {}

```

Fig. 8: quadboost structures

Move stores both *oldKey*'s and *newKey*'s terminal nodes, their parents, their parents' prior *ops*, and a new node. Moreover, we use a bool variable *allFlag* to indicate whether two parents have been attached to a *Move op* or not. Another bool variable *iFirst* is used to indicate the attaching order. For instance, if *iFirst* is true, *iParent* will be attached with a *Move op* before *rParent*. *Clean* means that there is no thread modifying the node. In contrast to Figure 4, the *Internal* class adds an *op* field to hold its state and related information. Note that a flag operation (CAS) is only applied on *Internal* nodes. We atomically set *Move ops* in *Leaf* nodes to linearize the move operation correctly.

Each basic operation, except for the contain operation, starts by changing an *Internal* node's *op* from *Clean* to other states. The three basic operations, therefore, generate a cor-

responding state transition diagram which provides a high-level description of our algorithm. After locating a terminal node, insert, remove, compress, and move transitions are executed as shown in Figure 9. In the figure, we specify the flag operation that restores an *op* to *Clean* as *unflag*, the flag operation that changes an *op* from *Clean* to *Substitute* in *insert* as *iflag*, the flag operation that changes an *op* from *Clean* to *Substitute* in *remove* as *rflag*, and the flag operation that changes an *op* from *Clean* to *Compress* as *cflag*. We describe how these transitions execute when a thread detects a state as follows:

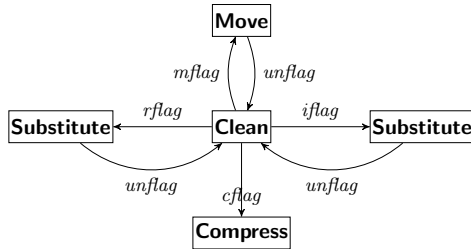


Fig. 9: State transition diagram

- **Clean.** For the insert transition, a thread constructs a new node and changes the parent's *op* to *Substitute* by *iflag*. For the remove transition, the thread creates an *Empty* node and changes the parent's *op* from *Clean* to *Substitute* by *rflag*. For the move transition, the thread flags both *newKey*'s and *oldKey*'s parents by *mflag*. For the compress transition, the thread uses a *cflag* operation to flag the parent if necessary.
- **Substitute.** The thread uses a CAS to change the existing node by a given node stored in the *op*. It then restores the parent's state to *Clean* by *unflag*.
- **Move.** The thread first determines the flag order by comparing *oldKey*'s parent with *newKey*'s parent. Suppose it flags *newKey*'s parent first, it will flag *oldKey*'s parent later. Then, the thread replaces *oldKey*'s terminal with an *Empty* node and replaces *newKey*'s terminal with a new node. Finally, it *unflags* their parents' *op* to *Clean* in the reverse order.
- **Compress.** The thread erases the node from the tree so that it cannot be detected. Different than other states, the node with a *Compress op* cannot be set to *Clean* by *unflag*.

4.3 Concurrent Algorithms

Figure 10 reflects quadboost's insert and contain operations. Both of them start by a find process that locates a terminal node. The find operation (line 98) pushes *Internal* nodes into a stack (line 95) and keeps recording the parent node's *op* (line 96).

The contain operation executes in a similar way as CAS quadtree. It calls the find function to locate a terminal node (line 98). The *op* at line 96 and the stack at line 95 are created for a modular presentation. We can omit them in a real implementation.

In the insert operation, we create a stack at the beginning to record the traversal path (line 98) and a *pOp* to record the parent node's *op* (line 96). After locating a terminal

```

94 bool contain(double keyX, double keyY) {
95     Stack<Node> path;
96     Operation pOp;
97     Node l = root;
98     find(l, pOp, path, keyX, keyY);
99     if (inTree(l, keyX, keyY) && !moved(l)) return true;
100    return false;
101 }
102
103 bool insert(double keyX, double keyY, V value) {
104     Stack<Node> path;
105     Operation pOp;
106     Node l = root;
107     find(l, pOp, path, keyX, keyY);
108     while (true) {
109         if (inTree(l, keyX, keyY) && !moved(l)) return false;
110         p = path.pop();
111         if (pOp.class == Clean) {
112             Node newNode = createNode(l, p, keyX, keyY, value);
113             Operation op = new Substitute(p, l, newNode);
114             if (helpFlag(p, pOp, op)) {
115                 helpSubstitute(op);
116                 return true;
117             } else pOp = p.op;
118         }
119         help(pOp); // help complete the operation of pOp
120         continueFind(pOp, path, l, p); // find a new terminal in a bottom up
121         way
122     }
123 }
124
125 void continueFind(Operation& pOp, Stack& path, Node& l, Internal p) {
126     if (pOp.class != Compress) l = p; // start from the parent node
127     else {
128         while (!path.isEmpty()) { // find a node in the path that is not in
129             the Compress state
130             l = path.pop();
131             pOp = l.op;
132             if (pOp.class == Compress) helpCompress(pOp);
133             else break; // find the node
134         }
135     }
136     find(l, pOp, path, keyX, keyY);
137 }
138
139 void find(Node& l, Operation& pOp, Stack& path, double keyX, double keyY)
140 while (l.class() == Internal) {
141     path.push(l);
142     pOp = l.op;
143     getQuadrant(l, keyX, keyY);
144 }
145
146 bool helpFlag(Internal node, Operation oldOp, Operation newOp) {
147     return CAS(node.op, oldOp, newOp);
148 }
149
150 void helpSubstitute(Substitute op) {
151     helpReplace(op.parent, op.oldChild, op.newNode);
152     helpFlag(op.parent, op, new Clean()); // unflag the parent node to Clean
153 }
154
155 void help(Operation op) {
156     if (op.class == Compress) helpCompress(op);
157     else if (op.class == Substitute) helpSubstitute(op);
158     else if (op.class == Move) helpMove(op);
159 }
160
161 bool moved(Node l) {
162     return l.class == Leaf && l.op != null && !hasChild(l.op.iParent,
163         l.op.oldChild);
164 }
165
166 bool hasChild(Internal parent, Node oldChild) {
167     return parent.nw == oldChild || parent.ne == oldChild || parent.sw ==
168         oldChild || parent.se == oldChild;
169 }

```

Fig. 10: quadboost *insert* and *contain*

node, we check whether the key of the node is in the tree and whether the node is moved at line 108. We will show the reason why we have to check whether a node is moved in Section 4.4. Then, we flag the parent of the terminal node before replacing it. In the next step, we call the *helpSubstitute* function at line 114, which first invokes the *helpReplace* function at line 145 to replace the terminal node and then *unflag* the parent node at line 146. If the flag operation fails, we update *pOp* at line 116 and help it finish at line 118. More than that, we have to execute the *continueFind* function to restart from the nearest parent. The *continueFind* function (line 122) pops nodes from the path until reaching a node whose *op* is not *Compress*. If the node's *op* is *Compress*, it helps the *op* finish. Or else, it breaks the loop and performs the find operation from the last node

popped at line 132.

The remove operation has a similar paradigm as the insert operation. It first locates a terminal node and checks whether the node is in the tree or moved. After that, it flags the parent node and replaces the terminal node with an *Empty* node. An extra step in the remove operation is the compress function at line 171. In the algorithm, we perform the compress function before the remove operation returns true. In this way, the linearization point of the remove operation belongs to the execution of itself, and extra adjustment induced by the compress function do affect the effectiveness. The compress function should examine three conditions before compressing the parent node. First, because the remove operation must return true, we do not compress nor help if the state of the parent is not *Clean* (line 182). Second, we check if the grandparent node (*gp*) is the root (line 184) as we maintain two layers of dummy *Internal* nodes. At last, we check whether four children of a parent node are all *Empty* (line 186).

```

159 bool remove(double keyX, double keyY, V value) {
160     Stack<Node> path;
161     Node l = root, newNode = new Empty();
162     Operation pOp;
163     find(l, keyX, keyY, pOp, path);
164     while (true) {
165         if (!inTree(l, keyX, keyY) || moved(l)) return false;
166         p = path.pop();
167         if (pOp.class == Clean) {
168             Operation op = new Substitute(p, l, newNode);
169             if (helpFlag(p, pOp, op) {
170                 helpSubstitute(op);
171                 compress(path, p); // compress the path if necessary
172                 return true;
173             } else pOp = p.op;
174         }
175         help(pOp);
176         continueFind(pOp, path, l, p);
177     }
178 }

179 void compress(Stack& path, Internal p) {
180     while (true) {
181         Operation pOp = p.op;
182         if (pOp.class == Clean) {
183             gp = path.pop();
184             if (gp == root) return; // two layers of Internal nodes are
                                   // maintained
185             Operation op = new Compress(gp, p);
186             if (!check(p) || !helpFlag(p, pOp, op)) return;
187             else helpCompress(op);
188             p = gp;
189         } else return;
190     }
191 }

192 bool helpCompress(Compress op) {
193     return helpReplace(op.grandparent, op.parent, new Empty<V>());
194 }

195 bool check(Internal node) {
196     return node.nw.class == Empty && node.ne.class == Empty && node.sw.class ==
           Empty && node.se.class == Empty;
197 }

```

Fig. 11: quadboost *remove*

4.4 LCA-based Move Operation

Tree-based structures share a property that two nodes in the tree have a common path starting from the root, and the lowest node in the path is called the lowest common ancestor (LCA). Figure 12 demonstrates the LCA node of a quadtree by a concrete example. Based on the observation, our LCA-based move operation is defined to find two different terminal nodes sharing a common path, remove a node with *oldKey*, and insert a node with *newKey*.

Figures 13 and 14 present the algorithm of the move operation. In contrast with the insert operation and the remove operation, the move operation begins by calling

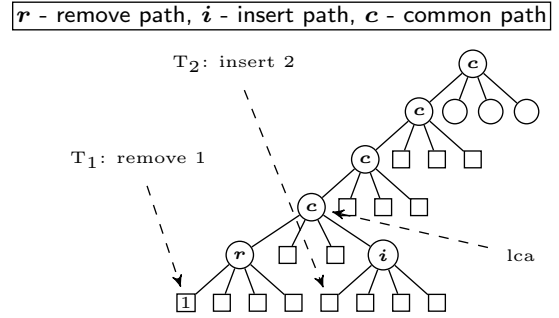


Fig. 12: Paths of two operations—insert node 2 and remove node 1 that share the LCA node

the findCommon function (line 212) that combines searches for two nodes together. The two searches share a common path that allows us to record them only once. We use two stacks to record nodes at line 199. The shared path and the remove path are pushed into *rPath*; and the insert path is pushed into *iPath*. By doing this, we could avoid considering complicated corner cases in terms of node compressions in *rPath*. The findCommon function begins by reading a child node from the parent for *oldKey* and checks whether *newKey* is in the same direction (line 271). If not, it terminates the traversal, and searches for individual keys separately (line 282 and line 279). If *oldKey* is not in the tree or is moved, or if *newKey* is in the tree and not moved, it returns false.

The move operation then checks two parents' *op* (*iOp* and *rOp*) before flag operations. If neither of them is *Clean* (line 216-217), or *rp* and *ip* are the same but their *ops* are different (line 218), it starts the continueFindCommon function at line 243 which we will discuss later in the section. Otherwise, it creates a new node for inserting and an *op* to hold essential information for a CAS at line 220-224. There are two specific cases. If two terminal nodes share a common parent, we directly call the helpMove function at line 228. Or else, to avoid live locks, we flag two nodes in a specific order. In our algorithm, we use the getSpatialOrder function at line 223 to compare *ip* and *rp* in the following order: $x \rightarrow y \rightarrow w$, where x , y , and w are the fields of an *Internal* node. We prove that this method produces a unique order among all *Internal* nodes in a quadtree in appendix A.

To start, the helpMove function flags a parent and checks whether both parents are successfully flagged at line 248-250. If both flag operations succeed, it sets *allFlag* to true at line 252. Note that we assign the *op* to *oldRChild* at line 253, letting other threads know whether the CAS on *iParent* has finished, which is the linearization point of the move operation. Because simultaneously *oldRChild* is aware that it is removed, it also explains why we should examine whether a node is moved. Then, we check whether two terminal nodes are the same. If so, we could combine two CASs into a single one. If not, we should remove *oldKey*'s terminal and replace *newKey*'s terminal. In the end, we reset the parents' *op* to *Clean* in the reverse order (line 261-267).

The continueFindCommon function is also invoked when one of the flag operations fails. If *rp* cannot be flagged, it pops nodes from *rPath* until it reaches the LCA node (line 291) or a node's *op* is *Compress* (line 297). It then starts from the last popped node to search for a new terminal of

```

198 bool move(double oldKeyX, double oldKeyY, double newKeyX, double newKeyY) {
199     Stack rPath, iPath;
200     // rl: terminal node of the remove path
201     // il: terminal node of the insert path
202     // rp: parent node of the remove path
203     // ip: parent node of the insert path
204     Node rl = root, il, rp, ip;
205     // rOp: op object of rp
206     // iOp: op object of ip
207     Operation rOp, iOp;
208     // rFail: whether the remove path fail
209     // iFail: whether the insert path fail
210     // cFail: whether the common path fail
211     bool rFail = false, iFail = false, cFail = false;
212     if (!findCommon(il, rl, lca, rOp, iOp, iPath, rPath, oldKeyX, oldKeyY,
213         newKeyX, newKeyY)) return false;
214     ip = iPath.pop();
215     rp = rPath.pop();
216     while (true) {
217         if (rOp.class != Clean) rFail = true;
218         if (iOp.class != Clean) iFail = true;
219         if (iOp != rOp && ip == rp) cFail = true; // the same parent node has
220             two different states
221         if (!cFail && !iFail && !rFail) {
222             if (il.class == Empty || il == rl) newNode = new Leaf(newKeyX,
223                 newKeyY, rl.value);
224             else newNode = createNode(il, ip, newKeyX, newKeyY, rl.value);
225             //iFirst: flag which node first
226             bool iFirst = getSpatialOrder(ip, rp);
227             Operation op = new Move(ip, rp, il, rl, newNode, iOldOp, rOldOp,
228                 iFirst);
229             if (rp != ip) { // two parents are different
230                 bool hasFlag = iFirst ? helpFlag(ip, iOp, op) : helpFlag(rp,
231                     rOp, op);
232                 if (hasFlag) {
233                     if (helpMove(op)) {
234                         compress(rp, rPath);
235                         return true;
236                     } else { // all flag operation fail
237                         rFail = iFail = true;
238                     }
239                 } else { // one of two paths fail
240                     if (iFirst) {iOp = ip.op; iFail = true;}
241                     else {rOp = rp.op; rFail = true;}
242                 }
243             } else { // two parents are the same
244                 if (helpMove(op)) return true;
245                 else {rOp = iOp = rp.op; cFail = true;}
246             }
247         } if (!continueFindCommon(iFail, rFail, cFail, il, rl, ip, rp, lca, rOp,
248             iOp, oldKeyX, oldKeyY, newKeyX, newKeyY, iPath, rPath)) return
249             false; // continuous find insert and remove paths
250         cFail = iFail = rFail = false;
251     }
252 }
253
254 bool helpMove(Move op) {
255     if (op.iFirst) helpFlag(op.rParent, op.oldROp, op); // flag the parent of
256         the oldKey's terminal first
257     else helpFlag(op.iParent, op.oldIOp, op); // flag the parent of the
258         newKey's terminal first
259     bool doCAS = op.iFirst ? op.rParent.op == op : op.iParent.op == op; //
260         whether the flag operation succeed
261     if (doCAS) { // all flags have been done
262         op.allFlag = true;
263         op.oldRChild.op = op;
264         if (op.oldRChild == op.oldRChild) { // combine two CASes in two one
265             helpReplace(op.rParent, op.oldRChild, op.newChild);
266         } else {
267             helpReplace(op.iParent, op.oldIChild, op.newChild);
268             helpReplace(op.rParent, op.oldRChild, new Empty());
269         }
270     }
271     if (op.iFirst) { // unflag in a reverse order
272         if (op.allFlag) helpFlag(op.rParent, op, new Clean());
273         if (op.rParent != op.rParent) helpFlag(op.iParent, op, new Clean());
274     } else {
275         if (op.allFlag) helpFlag(op.iParent, op, new Clean());
276         if (op.rParent != op.rParent) helpFlag(op.rParent, op, new Clean());
277     }
278     return op.allFlag;
279 }

```

Fig. 13: quadboost *move*

oldKey at line 302. If *ip* cannot be flagged, it pops nodes from *iPath* until it is empty (line 315) or a node's *op* is not *Compress* (line 317). It again continues to search for *newKey* at line 323. If either *rPath* has popped the LCA node or *iPath* is empty, it clears *iPath* at line 335 and pops all nodes above the LCA node from *rPath* at line 334. In the end, it calls the *findCommon* function to locate terminal nodes again at line 342. We prove that if *oldKey*'s terminal and *newKey*'s terminal share an LCA node, the common path will never be changed unless the LCA is altered.

```

270 bool findCommon(Node& il, Node& rl, Internal& lca, Operation& rOp, Operation&
271     iOp, Stack& iPath, Stack& rPath, double oldKeyX, double oldKeyY, double
272     newKeyX, double newKeyY) {
273     while (rl.class == Internal) {
274         rPath.push(rl);
275         rOp = rl.op;
276         getQuadrant(rl, oldKeyX, oldKeyY);
277         if (!sameDirection(oldKeyX, oldKeyY, newKeyX, newKeyY, il, rl))
278             break // check whether two nodes are in the same direction
279     }
280     lca = rPath.top();
281     find(rl, oldKeyX, oldKeyY, rOp, rPath); // find oldKey's terminal
282     if (!inTree(rl, oldKeyX, oldKeyY) || moved(rl)) return false;
283     il = lca;
284     find(il, newKeyX, newKeyY, iOp, iPath); // find newKey's terminal
285     if (!inTree(il, newKeyX, newKeyY) && !moved(il)) return false;
286 }
287
288 bool continueFindCommon(Node& il, Node& rl, Internal& lca, Operation& rOp,
289     Operation& iOp, Stack& iPath, Stack& rPath, Internal ip, Internal rp,
290     double oldKeyX, double oldKeyY, double newKeyX, double newKeyY, bool
291     iFail, bool rFail, bool cFail) {
292     if (rFail && !cFail) { // restart to find oldKey's terminal in cast that
293         the lca's op remains the same
294         help(rOp);
295         if (rOp.class != Compress) rl = rp;
296     } else {
297         while (!rPath.empty()) {
298             if (rPath.size() <= indexof(lca)) { // reach the lca node,
299                 cFail = true; // break the loop the find the common path
300                 again
301                 break;
302             }
303             rl = rPath.pop();
304             rOp = rl.op;
305             if (rOp.class == Compress) helpCompress(rOp);
306             else break;
307         }
308     } if (!cFail) { // the lca node has not been changed
309         find(rl, oldKeyX, oldKeyY, rOp, rPath);
310         if (!inTree(rl, oldKeyX, oldKeyY) || moved(rl)) return false;
311     } else {
312         rp = rPath.pop();
313         return true;
314     }
315 } if (iFail && !cFail) {
316     help(iOp);
317     if (iOp.class != Compress) il = ip;
318     else {
319         while (!iPath.empty()) {
320             il = iPath.pop();
321             iOp = il.op;
322             if (iOp.class == Compress) helpCompress(iOp);
323             else break;
324         }
325     } if (iOp.class == Compress) cFail = true; // check the last op which
326         must be the op of the lca node
327     if (!cFail) {
328         find(il, newKeyX, newKeyY, iOp, iPath);
329         if (!inTree(il, newKeyX, newKeyY) && !moved(il)) return false;
330     } else {
331         ip = iPath.pop();
332         return true;
333     }
334 } if (cFail) {
335     help(iOp);
336     help(rOp);
337     rPath.setIndex(indexof(lca)); // pop out all nodes above the lca node
338     iPath.clear(); // clean the input path
339     while (!rPath.empty()) { // first time must be not empty
340         rl = rPath.pop();
341         rOp = rl.op;
342         if (rOp.getClass() == Compress) helpCompress(rOp);
343         else break;
344     } // pop up nodes in the common path
345     if (!findCommon(il, rl, lca, rOp, iOp, iPath, rPath, oldKeyX, oldKeyY,
346         newKeyX, newKeyY) return false;
347     else { // find two nodes
348         rp = rPath.pop();
349         ip = iPath.pop();
350     }
351 }

```

Fig. 14: quadboost *findCommon* and *continueFindCommon*

4.5 One Parent Optimization

In practice, we notice that pushing a whole stack during a traversal is highly expensive. When detecting a failed flag operation, we only have to restart from the parent of a terminal node because we do not change *Internal* nodes unless the compress function erases them from a quadtree. Thus, to reduce the pushing cost, we could only record the parent of the terminal node during a traversal.

Meanwhile, we have to change the continuous find

mechanism. For the insert operation and the remove operation, encountering a *Compress op*, we straightforwardly restart from the root. For the move operation, if either *oldKey*'s or *newKey*'s parent is under compression, we restart from the LCA node. If the LCA node also has a *Compress op*, we restart from the root.

5 PROOF SKETCH

In this section, we prove that quadboost is both linearizable and non-blocking, and we propose a lengthy proof in appendix A.

There are four kinds of *basic operations* in quadboost: the insert operation, the remove operation, the move operation, and the contain operation. Other functions are called *sub-routines*, which are invoked by basic functions. The insert operation and the remove operation only modify one terminal node, whereas the move operation might operate two different terminals—one for inserting a node with *newKey*, the other for removing a node with *oldKey*. We call them *newKey*'s terminal and *oldKey*'s terminal, and we call their parents *newKey*'s parent and *oldKey*'s parent, respectively. We define $snapshot_{T_i}$ as the state of our quadtree at some time, T_i .

In our proofs, a CAS that changes a node's *op* is a *flag* operation and a CAS that changes a node's child is a *replace* operation. Specifically, we use *iflag*, *rflag*, *mflag*, and *cflag* to denote flag operations for the insert operation, the remove operation, the move operation, and the compress operation separately. Likewise, we use *ireplace*, *rreplace*, *mreplace*, and *creplace* for replace operations. Moreover, we specify a flag operation that attaches a *Clean op* on a node as an unflag operation.

First, we present some observations from quadboost. Then, we propose lemmas to show that our subroutines satisfy their pre-conditions and post-conditions because proofs on basic operations depend on these conditions. Next, we demonstrate that there are three categories of successful CAS transitions according to Figure 9. We also derive some invariants of these transitions. Using above post-conditions of subroutines and invariants, we could demonstrate that a quadtree's structure is maintained during concurrent modifications. In following proofs, we show that quadboost is linearizable because it can be ordered equivalently as a sequential one by its linearization points. In the last part, we prove the non-blocking progress condition of quadboost.

Observation 1. *The key field of a Leaf node is never changed. The op field of a Leaf node is initially null. The space information of an Internal node is never changed.*

Observation 2. *The root node is never changed.*

Observation 3. *A flag operation attaches an op on a Internal node.*

Observation 4. *The allFlag and iFirst field in Move are initially false, and they will never be set back after assigning to true.*

Observation 5. *If a Leaf node is moved, its op is set before the replace operation on op.iParent, which is before the replace operation on op.rParent.*

Observation 6. *The help function, the helpCompress function, the helpMove function, and the helpSubstitute function are not called in a mutual way. (If method A calls method B, and method B also calls method A, we say A and B are called in a mutual way.)*

5.1 Basic Invariants

We use $find(keys)$ to denote a set of find operations: *find*, *continueFind*, *findCommon*, and *continueFindCommon*. The find function and the continueFind function return a tuple $\langle l, pOp, path \rangle$. The findCommon function and the continueFindCommon function return two such tuples. We specify functions outside the *while* loop in the insert operation, the remove operation, and the move operation are at $iteration_0$, and functions inside the *while* loop are at $iteration_i, 0 < i$ ordered by their invocation sequence.

We suppose that $find(keys)$ executes from a valid $snapshot_{T_i}$ to derive the following conditions. Proofs for conditions of other subroutines are included in appendix A.

Lemma 1. *The post-conditions of $find(keys)$ returned at T_i , with tuples $\langle l^k, pOp^k, path^k \rangle, 0 \leq k < |keys|$.*

- 1) l^k is a Leaf node or an Empty node.
- 2) At some $T_{i1} < T_i$, the top node in $path^k$ has contained pOp^k .
- 3) At some $T_{i2} < T_i$, the top node in $path^k$ has contained l^k .
- 4) If pOp^k is read at T_{i1} , and l^k is read at T_{i2} , then $T_{i1} < T_{i2} < T_i$.
- 5) For each node n in the $path^k$, $size(path^k) \geq 2$, n_t is on the top of n_{t-1} , and n_t is on the direction $d \in \{nw, ne, sw, se\}$ of n_{t-1} at $T_{i1} < T_i$.

Based on these post-conditions, we show that each *op* created at T_i store their corresponding information.

Lemma 2. *For op created at T_i :*

- 1) If *op* is *Substitute*, *op.parent* has contained *op.oldChild* that is a Leaf node at $T_{i1} < T_i$ from the results of $find(keys)$ at the prior iteration.
- 2) If *op* is *Compress*, *op.grandparent* has contained *op.parent* that is an Internal node at $T_{i1} < T_i$ from the results of $find(keys)$ at the prior iteration.
- 3) If *op* is *Move*, *op.iParent* has contained *op.oldICChild* that is a Leaf node and *op.oldIOp* before T_i , and *op.rParent* has contained *op.oldRCChild* that is a Leaf and *op.oldROp* before T_i from the results of $find(keys)$ at the prior iteration.

Though we have not presented details of the createNode function, our implementation could guarantee that it has following conditions.

Lemma 3. *For createNode($l, p, newKeyX, newKeyY, value$) that returns a newNode invoked at T_i , it has the post-condition:*

- 1) The newNode returned is either a Leaf node with newKey and value, or a sub-tree that contains both *l.key* node and newKey node with the same parent.

Using prior conditions, we derive some invariants during concurrent executions and prove that there are three

kinds of successful CAS transitions. We put successful flag operations that attach ops on nodes at the beginning of each CAS transition. We say every successive replace operations that read op belongs to it and follows the flag operations.

Let $flag_0, flag_1, \dots, flag_n$ be a sequence of successful flag operations. $flag_i$ reads pOp_i and attaches op_i . $replace_i$ and $unflag_i$ read op_i and follows it. Therefore, we say $flag_i$, $replace_i$, and $unflag_i$ belong to the same op . In addition, if there are more than one replace operation belongs to the same op_i , we denote them as $replace_i^0, replace_i^1, \dots, replace_i^n$ ordered by their successful sequence. Similarly, if there is more than one flag operation that belonging to op_i on different nodes, we denote them as $flag_i^0, flag_i^1, \dots, flag_i^n$. A similar notation is used for $unflag_i$.

The following lemmas prove the correct ordering of three different transitions.

Lemma 4. For a new node n :

- 1) It is created with a Clean op .
- 2) $rflag$, $iflag$, $mflag$ or $cflag$ succeeds only if n 's op is Clean.
- 3) $unflag$ succeeds only if n 's op is Substitute or Move.
- 4) Once n 's op is Compress, its op will never be changed.

Lemma 5. For an Internal node n , it never reuses an op that has been set previously.

Lemma 6. $replace_i^k$ will not occur before $flag_i^k$ that belongs to the same op that has been done.

Lemma 7. The $flag \rightarrow replace \rightarrow unflag$ transition occurs when $rflag_i$, $iflag_i$ or $mflag_i$ succeeds, and it has following properties:

- 1) $replace_i$ never occurs before $flag_i$.
- 2) $flag_i^k, 0 \leq k < |flag_i|$ is the first successful flag operation on $op_i.parent^k$ after T_{i1} when pOp_i^k is read.
- 3) $replace_i^k, 0 \leq k < |replace_i|$ is the first successful replace operation on $op_i.parent^k$ after T_{i2} when $op_i.oldChild^k$ is read.
- 4) $replace_i^k, 0 \leq k < |replace_i|$ is the first successful replace operation on $op_i.parent^k$ that belongs to op_i .
- 5) $unflag_i^k, 0 \leq k < |unflag_i|$ is the first successful unflag operation on $op_i.parent^k$ after $flag_i^k$.
- 6) There is no successful unflag operation that occurs before $replace_i$.
- 7) The first replace operation on $op_i.parent^k$ that belongs to op_i must succeed.

Lemma 8. The $flag \rightarrow replace$ transition occurs only when $cflag_i$ succeeds, and it has following properties:

- 1) $creplace_i$ never occurs before $cflag_i$.
- 2) $cflag_i$ is the first successful flag operation on $op_i.parent$ after T_{i1} when pOp_i is read.
- 3) $creplace_i$ is the first successful replace operation on $op_i.grandparent$ after T_{i2} when $op_i.parent$ is read.
- 4) $creplace_i$ is the first successful replace operation on $op_i.grandparent$ that belongs to op_i .
- 5) There is no unflag operation after $creplace_i$.
- 6) The first replace operation on $op_i.grandparent$ that belongs to op_i must succeed.

Lemma 9. For the $flag \rightarrow unflag$ transition, it only results from $mflag$ (Suppose $iFirst$ is false) when:

- 1) $unflag_i$ is the first successful unflag operation on $op_i.rParent$.
- 2) The first flag operation on $op_i.iParent$ must fail, and no later flag operation succeeds.
- 3) $op_i.iParent$ and $op_i.rParent$ are different.

Claim 1. There are three kinds of successful transitions belong to an op : (1) $flag \rightarrow replace \rightarrow unflag$, (2) $flag \rightarrow unflag$, (3) $flag \rightarrow replace$.

Then, we prove that a quadtree maintains its properties during concurrent modifications.

Definition 1. Our quadtree has the following properties:

- 1) Two layers of dummy Internal nodes are never changed.
- 2) An Internal node n has four children located in the direction $d \in \{nw, ne, sw, se\}$ according to their $\langle x, y, w, h \rangle$, or $\langle keyX, keyY \rangle$.

For Internal nodes residing on four directions:

- $n.nw.x = n.x, n.nw.y = n.y;$
- $n.ne.x = n.x + w/2, n.ne.y = n.y;$
- $n.sw.x = n.x, n.sw.y = n.y + n.h/2;$
- $n.se.x = n.x + w/2, n.se.y = n.y + h/2,$

All children have their $w' = n.w/2, h' = n.h/2$.

For Leaf nodes residing on four directions:

- $n.x \leq n.nw.keyX < n.x + n.w/2,$
 $n.y \leq n.nw.keyY < n.y + n.h/2;$
- $n.x + n.w/2 \leq n.ne.keyX < n.x + n.w,$
 $n.y \leq n.ne.keyY < n.y + n.h/2;$
- $n.x \leq n.sw.keyX < n.x + n.w/2,$
 $n.y + n.h/2 \leq n.sw.keyY < n.y + n.h;$
- $n.x + n.w/2 \leq n.se.keyX < n.x + n.w,$
 $n.y + n.h/2 \leq n.se.keyY < n.y + n.h.$

To help clarify a quadtree's properties during concurrent executions, we define *active* set and *inactive* set for different kinds of nodes. For an Internal node or an Empty node, if it is reachable from the root in $snapshot_{T_i}$, it is active; otherwise, it is inactive. For a Leaf node, if it is reachable from the root in $snapshot_{T_i}$ and not moved, it is active; otherwise, it is inactive. We say a node n is moved in $snapshot_{T_i}$ if the function $moved(n)$ returns true at T_i . We denote $path(keys^k), 0 \leq k < |replace_i|$ as a stack of nodes pushed by $find(keys)$ in a snapshot. We define $physical_path(keys^k)$ to be the path for $keys^k$ in $snapshot_{T_i}$, consisting of a sequence of Internal nodes with a Leaf node or an Empty node at the end. We say a subpath of $path(keys^k)$ is an *active_path* if all nodes from the root to node $n \in path(keys^k)$ are active. Hence a *physical_path(keys^k)* is active only if the end node is not moved.

Lemma 10. Two layers of dummy nodes are never changed.

Lemma 11. Children of a node with a Compress op will not be changed.

Lemma 12. Only an Internal node with all children Empty could be attached with a Compress op .

Lemma 13. An Internal node whose op is not Compress is active.

Lemma 14. After the invocation of $find(keys)$ which reads l^k , there is a snapshot in which the path from the root to l^k is $physical_path(keys^k)$.

Lemma 15. After $ireplace$, $rreplace$, $mreplace$, and $creplace$, a quadtree's properties remain.

Proof. We shall prove that in any $snapshot_{T_i}$, a quadtree's properties remain.

First, Lemma 10 shows that two layers of dummy nodes remain in the tree. We have to consider other layers of nodes that are changed by replace operations.

Consider $creplace$ that replaces an *Internal* node by an *Empty* node. Because the *Internal* node has been flagged on a *Compress op* before $creplace$ (Lemma 8), all of its children are *Empty* and not changed (Lemma 11 and Lemma 12). Thus, $creplace$ does not affect the second claim of Definition 1.

Consider $ireplace$, $rreplace$, or $mreplace$ that replaces a terminal node by an *Empty* node, a *Leaf* node, or a sub-tree. By Lemma 7, before $replace^k$, $op.parent^k$ is flagged with op such that no successful replace operation could happen on $op.parent^k$. Therefore, if the new node is *Empty*, it does not affect the tree property. If the new node is a *Leaf* node or a sub-tree, based on the post-conditions of the $createNode$ function (Lemma 3), after replace operations the second claim of Definition 1 still holds. \square

Claim 2. A quadtree maintains its properties in every snapshot.

5.2 Linearizability

In this section, we define linearization points for basic operations. As the compress function is included in the move operation and the remove operation that return true, it does not affect the linearization points of these operations. If an algorithm is linearizable, its result be ordered equivalently as a sequential history by the linearization points. Since all modifications depend on $find(keys)$, we first point out its linearization point. For $find(keys)$, we define its linearization point at T_i such that l^k returned is on the $physical_path(keys^k)$ in $snapshot_{T_i}$.

For the contain operation that returns true, we show that there is a corresponding snapshot in which l^k in $physical_path(keys^k)$ is active. For the contain operation, the insert operation, the remove operation, or the move operation that returns false, we show there is a corresponding snapshot in which l^k in $physical_path(keys^k)$ is inactive. For the insert, remove, and move operations that returns true, we define the linearization point to be its first successful replace operation— $replace_i^0$. To make a reasonable demonstration, we first show that $replace_i^k$, $0 \leq k < |replace_i|$ belongs to each operation that creates op_i and illustrate that op is unique for each operation.

Lemma 16. For $find(keys)$ that returns tuples $\langle l^k, pOp^k, path^k \rangle$, there is a $snapshot_{T_i}$ such that $path^k$ returned with l^k at the end is $physical_path(key^k)$ in $snapshot_{T_i}$.

Lemma 17. If the insert operation, the remove operation, or the move operation returns true, the first successful replace operation occurs before returning, and it belongs to the op created by the operation itself at the last iteration in the while loop.

Lemma 18. If the insert operation, the remove operation, or the move operation returns false, there is no successful replace happens during the execution.

The next lemma points out the **linearization points** of the contain operation.

Lemma 19. For the contain operation that returns true, there is a corresponding snapshot that l^k in $physical_path(keys^k)$ is active in $snapshot_{T_i}$. For the contain operation that returns false, there is a corresponding snapshot that l^k in $physical_path(keys^k)$ is inactive in $snapshot_{T_i}$.

We list out the **linearization points** of other operations as follows:

- $insert(key)$. The linearization point of the insert operation that returns false is T_i after calling $find(key)$ at which l at the end of $physical_path(key)$ contains the key and is not moved. For the insert operation that returns true, the linearization point is at the first successful replace operation (line 145).
- $remove(key)$. The linearization point of the remove operation that returns false is T_i after calling $find(key)$ at which l at the end of $physical_path(key)$ does not contain the key or is moved. For the successful remove operation, we define the linearization point where the node with key is replaced by an *Empty* node (line 145).
- $move(oldKey, newKey)$. For the unsuccessful move operation, the linearization point depends on both $newKey$ and $oldKey$. If rl at the end of $physical_path(oldKey)$ does not contain $oldKey$ or is moved, the linearization point is T_{i1} after calling $find(keys)$. Or else, if il at the end of $physical_path(newKey)$ contains $newKey$ and is not moved, the linearization point is at T_{i2} after calling $find(keys)$. For the successful move operation, the linearization point is the first successful replace operation. (line 255 or line 257)

Claim 3. $quadboost$ is linearizable.

5.3 Non-blocking

Finally, we prove that $quadboost$ is non-blocking, which means that the system as a whole is making progress even if some threads are starving.

Lemma 20. A node with a *Compress op* will not be pushed into path more than once.

Lemma 21. For $path(keys^k)$, if n_t is active in $snapshot_{T_i}$, then n_0, \dots, n_{t-1} pushed before n_t are active.

Lemma 22. If n is the LCA node in $snapshot_{T_i}$ on $physical_path$ for $oldKey$ and $newKey$. Then at $T_{i1}, T_{i1} > T_i$, n is still the LCA on $active_path$ for $oldKey$ and $newKey$ if it is active.

Lemma 23. $path^k$ returned by $find(keys)$ consists of finite number of keys.

Lemma 24. There is a unique spatial order among nodes in a quadtree in every snapshot.

Lemma 25. *There are a finite number of successful $flag \rightarrow replace \rightarrow$, $flag \rightarrow replace$, $flag \rightarrow unflag$ transitions.*

Lemma 26. *If the help function returned at T_i , and $find(keys)$ at the prior iteration reads $p^k.op$ at $T_{i1} < T_i$, keys in $snapshot_{T_i}$ and $snapshot_{T_{i1}}$ are different.*

Claim 4. *quadboost is non-blocking.*

Proof. We have to prove that no process will execute loops infinitely without changing keys in a quadtree. First, we prove that $path$ is terminable. Next, we prove that $find(keys^k)$ starts from an active node in $physical_path(keys^k)$ in $snapshot_{T_i}$ between i_{th} iteration and $i + 1_{th}$ iteration. Finally, some *Leaf* nodes in $snapshot_{T_{i1}}$ at the returning of $find(keys)$ at i_{th} iteration are different from $snapshot_{T_{i2}}$ at the returning of $find(keys)$ at $i + 1_{th}$ iteration.

For the first part, we initially start from the root node. Therefore, $path$ is empty. Moreover, as Lemma 22 shows that $path^k$ consists of finite number of keys, we establish this part.

For the second part, the `continueFind` function and the `continueFindCommon` function pop all nodes with `Compress op` from $path$. For the insert operation and remove operation, since Lemma 13 shows that an *Internal* nodes whose op is not `Compress` is active, and Lemma 21 shows that nodes above the active node are also active, there is a snapshot in which the top node of $path$ is still in $physical_path(keys^k)$. For the move operation, if either $rFail$ or $iFail$ is true, it is equivalent with the prior case. If $cFail$ is true, Lemma 22 illustrates that if the LCA node is active, it is in $physical_path$ for both $oldKey$ and $newKey$. Thus, there is also a snapshot that the start node is in $physical_path$.

The third part is proved by contradiction, assuming that a quadtree is stabilized at T_i , and all invocations after T_i are looping infinitely without changing *Leaf* nodes.

For the insert operation and the remove operation, before the invocation of $find(keys)$ at the next iteration, they must execute the help function at line 118 and line 175 accordingly. In both cases, the help function changes the snapshot (Lemma 26).

For the move operation, different situations of the `continueFindCommon` function are considered. If $rFail$ or $iFail$ is true, two situations arise: (1) iOp or rOp is *Clean* but $mflag$ fails; or (2) iOp or rOp is not *Clean*. In both cases, before the invocation of $find(keys^k)$, the help function is performed at line 311 and line 287. Thus by Lemma 26, the snapshot is changed between two iterations. Now consider if $cFail$ is true. If ip and rp are the same, it could result from the difference between rOp and iOp . In this case, the snapshot might be changed between reading rOp and iOp . It could also result from the failure of $mflag$. For the above cases, the help function at line 332 would change the quadtree. If ip and rp are different, it results from either $iPath$ or $rPath$ that have popped the LCA node. We have proved the case in which either $iFail$ or $rFail$ is true and derives a contradiction.

From the above discussions, we prove that quadboost is non-blocking. \square

6 EVALUATION

We ran experiments on a machine with 64GB main memory and two 2.6GHZ Intel(R) Xeon(R) 8-core E5-2670 processors with hyper-threading enabled, rendering 32 hardware threads in total. We used the RedHat Enterprise Server 6.3 with Linux core 2.6.32, and all experiments were ran under Sun Java SE Runtime Environment (build 1.8.0_65). To avoid significant run-time garbage collection cost, we set the initial heap size to 6GB.

For each experiment, we ran eight 1-second cases, where the first 3 cases were performed to warm up JVM, and the median of the last 5 cases was used as the real performance. Before the start of each case, we inserted half keys from the key set into a quadtree to guarantee that the insert and the remove operation have equal success opportunity initially.

We apply uniformly distributed key sets that contain two-dimensional points within a square. We use the $range$ to denote the border of a square. Thus, points are located inside a $range * range$ square. In our experiments, we used two different key sets: 10^2 keys to measure the performance under high contention, and 10^6 keys to measure the performance under low contention. For simplicity, we let the $range$ of the first category experiment be 10, rendering $1 - 10^2$ consecutive keys for one-dimensional structure. For the second category experiment, we let the $range$ be 1000, generating $1 - 10^6$ consecutive keys.

TABLE 1: The concurrent quadtree algorithms with different optimization strategies.

type	insert	remove	move
qc	single CAS	single CAS	not support
qb-s	flag CAS, continuous find, stack	flag CAS, continuous find, decoupling stack, recursive compression	flag CAS, continuous find, decoupling stack, recursive compression
qb-o	flag CAS, continuous find	flag CAS, continuous find, decoupling compression	flag CAS, continuous find, decoupling compression

We evaluate quadboost algorithms via comparing with the state-of-the-art concurrent trees (*kary*, *ctrie*, and *patricia*) for throughput (Section 6.1) and presenting the incremental effects of the optimization strategies proposed in this work (Section 6.2). Table 1 lists the concurrent quadtree algorithms. **qb-o** (quadboost-one parent) is the one parent optimization based on **qb-s** (quadboost-stack) mentioned in Section 4.4. **qc** is the CAS quadtree introduced in Section 3.

6.1 Throughput

To the best of our knowledge, a formal concurrent quadtree has not been published yet. Hence, we compare our quadtrees with three one-dimensional non-blocking trees:

- **kary** is a non-blocking k -way search tree, where k represents the number of branches maintained by an internal node. Like the non-blocking BST [4], keys are kept in leaf nodes. When $k = 2$, the structure is similar as the non-blocking BST; when $k = 4$, each internal node has four children, it has a similar structure to quadtree. However, *kary*'s structure depends on the modification order, and it does not have a series of internal nodes representing the two-dimensional space hierarchy.
- **ctrie** is a concurrent hash trie, where each node can store up to 2^k children. We use $k = 2$ to make a

4-way hash trie that resembles quadtree. The hash trie also incorporates a compression mechanism to reduce unnecessary nodes. Different from quadtree, it uses a control node (INODE as the paper indicates) to coordinate concurrent updates. Hence, the search depth could be longer than quadtree.

- **patricia** is a binary search tree, which adopts Ellen’s BST techniques [4]. As the author points out, it can be used as quadtree by interleaving the bits of x and y . It also supports the move (replace) operation like quadboost. Unlike our LCA-based operation, it searches two positions separately without a continuous find mechanism.

Since the above structures only store one-dimensional keys, we had to transform a two-dimensional key into a one-dimensional key for comparison. Though **patricia** could store two-dimensional keys using the above method, **ctrie** and **kary** cannot use it. Thus, we devised a general formula: $key^1 = key_x^2 * range + key_y^2$. Given a two-dimensional key- key^2 , and $range$, we transformed it into a one-dimensional key- key^1 . To refrain from trivial transformations by floating numbers, we only considered integer numbers in this section.

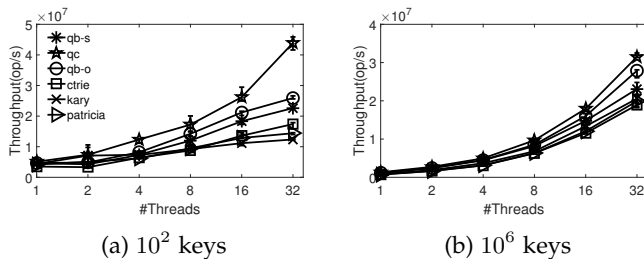


Fig. 15: Throughput of different concurrent trees under both high and low contention (50% insert, 50% remove).

Due to the lack of the move operation in CAS quadtree (**qc**), we compare throughput with/without the move operation respectively. Figure 15 plots throughput without any move operation for the concurrent algorithms. It is not surprising to observe that **qc** achieves the highest throughput. To some extent, **qc** represents an upper bound of throughput because it maintains the hierarchy without physical removal, i.e., its remove operation only applies a CAS on edges to change links, which leads to less contention than other practical concurrent algorithms. However, both **qb-s** and **qb-o** can achieve comparable throughput when the key set becomes larger. This phenomenon occurs due to: (i) given the large key set, fewer thread interventions results in a lesser number of CAS failures on nodes; and (ii) both algorithms compress nodes and use the continuous find mechanism to reduce the length of traverse path.

As a comparison, **ctrie**, **kary**, and **patricia** show lower performances with the increasing number of threads. For instance, in Figure 15a at 32 threads, **qb-o** outperforms **ctrie** by 49%, **patricia** by 79%, and **kary** by 109%. Note that **qb-o** and **qb-s** incorporate the continuous find mechanism to reduce the length of traverse path. Further, both **kary** and **patricia** flag the grandparent node in the remove operation, which allows less concurrency than **ctrie**, **qb-o** and **qb-**

s with the decoupling approach shown in Figure 7. **qb-s** is worse than **qb-o** due to its extra cost of recording elements and compressing nodes recursively. Figure 15b exhibits results from when the key set was large. There is less collision among threads but deeper depths of trees than the smaller key set. In the scenario, **qb-o** and **qb-s** show a similar performance as **qc** because of less number of CAS failures caused by thread interventions. **qb-o** is only 12% worse than **qc** at 32 threads, but 47% better than **ctrie**, 35% better than **kary**, and 39% better than **patricia** mainly for its shorter traversal paths caused by its static representation and the continuous find mechanism. As we discussed in the next section, **qb-s** and **qb-o** save a significant number of nodes as shown in Figure 18. It implies that **qb-s** and **qb-o** occupy less memory and result in a shorter path for traversal.

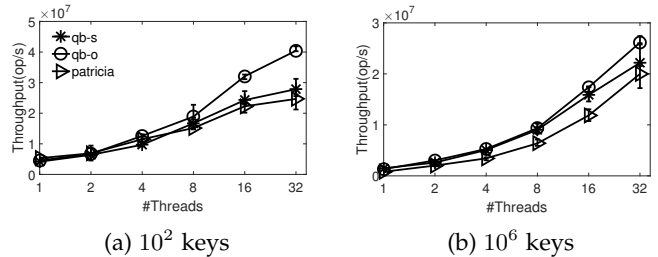


Fig. 16: Comparison of the move operation’s throughput between quadboost and patricia in both small and large ranges (10% insert, 10% remove, 80% move).

Figure 16b demonstrates that quadboost has an efficient move operation. Using the small key set, where the depth is not a significant impact, Figure 16a shows that **qb-o** is more efficient than **patricia** especially when contention is high. For example, it performs better than **patricia** by 47% at 32 threads because it adopts the continuous find mechanism to traverse less path and decouples physical adjustment for higher concurrency. However, **qb-s** is similar to **patricia** since it has to maintain a stack and recursively compress nodes in a quadtree. Figure 16b illustrates that **qb-s** and **qb-o** have a similar throughput for the large key set. **qb-o** outperforms **patricia** by 31% at 32 threads. When the key set is large, the depth becomes a more significant factor due to less contention. Since each *Internal* node in a quadtree maintains four children while **patricia** maintains two, the depth of **patricia** is deeper than quadboost. Further, the combination of the LCA node and the continuous find mechanism ensures that **qb-o** and **qb-s** do not need to restart from the root even if flags on two different nodes fail.

6.2 Analysis

To determine how quadboost algorithms improve the performance, we devised two algorithms that incrementally use parts of techniques in **qb-o**:

- **qb-f** flags the parent of a terminal node in the move operation, the insert operation, and the remove operation. It restarts from the root without a continuous find mechanism. Further, it adopts the traditional remove mechanism mentioned in Figure 7a.

- **qb-d** decouples the physical adjustment in the remove operation based on **qb-f**.

range here was set to $2^{32} - 1$, and both key_x and key_y could be floating numbers. We used an insert dominated experiment and a remove dominated experiment to demonstrate the effects of different techniques. We used a remove dominated experiment to show the effect of decoupling, where *insert:remove* ratio was 1:9. In the insert dominated experiment, the *insert:remove* ratio was 9:1; hence, there were far more insert operations. Since fewer compress operations were induced, the experiment showed the effect of the continuous find mechanism. Figure 17a illustrates

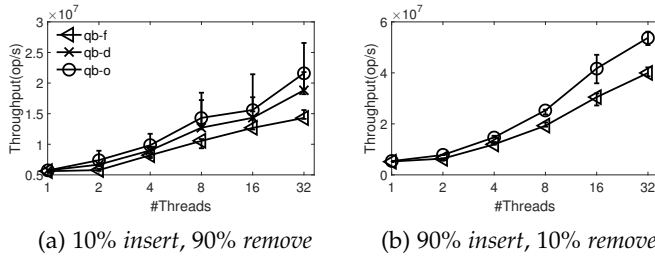


Fig. 17: Throughput comparison in insert dominated and remove dominated cases (10^2 keys).

that quadtrees with decoupling exhibit a higher throughput than **qb-f**, the basic flag concurrent quadtree. Besides, **qb-o** which incorporates the continuous find is more efficient than **qb-d**. Specifically, at 32 threads, **qb-o** performs 15% better than **qb-d** and 51% better than **qb-f**. From Figure 17b, we determine that **qb-o** outperforms **qb-f** by up to 35%. Therefore, it demonstrates that the continuous find mechanism and the decoupling approach play a significant role in our algorithm.

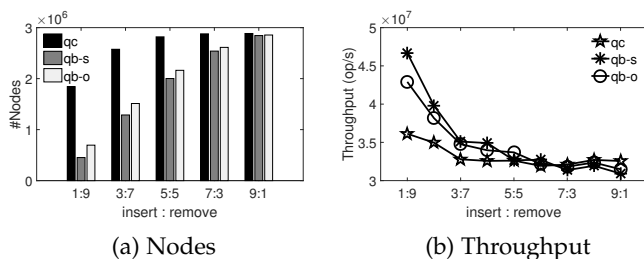


Fig. 18: Number of nodes left and throughput under different ratios of *insert:remove* from 9:1 to 1:9, with fixed 90% contain, under 32 threads and 10^6 keys.

Another advantage of quadboost results from the compression technique, which reduces the search path for each operation and the memory consumption. Figure 18² plots the number of nodes left and the throughput of each quadtree at different *insert:remove* ratios. As **qc** only replaces a terminal node with an *Empty* node without compression, it results in the greatest number of nodes in the memory (Figure 18a). In contrast, **qb-s** and **qb-o** compress a quadtree if necessary. When **qc** contains much more nodes than other quadtrees, the remove operation dominates (the first groups

2. Unlike previous experiments, we run eight 3-second cases in the experiment to ensure stable amount of modifications

bars to the left) and has three times the amount of nodes of **qb-s**. The result also indicates that **qb-o** contains a similar number of nodes to **qb-s** despite it only compresses one layer of nodes. Figure 18b illustrates the effectiveness of compression in the face of tremendous contain operations. **qb-s** outperforms **qc** by 30% at 9:1 *insert:remove* ratio because **qb-s** adjusts the quadtree's structure by compression to reduce the length of the search path. With the increment of the insert ratio, **qb-s** performs similar to **qc** due to the extra cost of maintaining a stack and the recursive compression. However, **qb-o** achieves good balance between **qb-s** and **qc**, which compresses one layer of nodes without recording the whole traverse path.

7 RELATED WORKS

Because there are few formal works related to concurrent quadtrees, we present a roadmap to show the development of state-of-the-art concurrent trees in Figure 19.

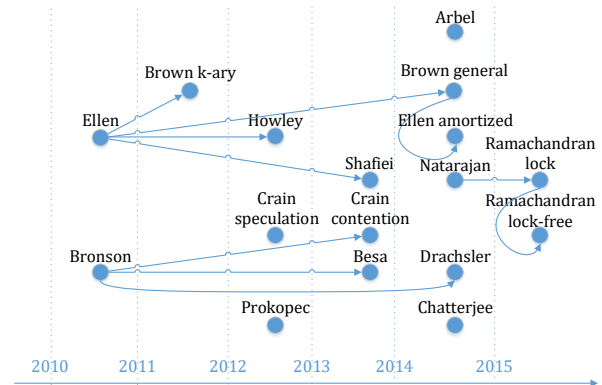


Fig. 19: Concurrent trees roadmap

Ellen [4] provided the first non-blocking BST and proved it correct. This work is based on the cooperative method described in Turek [17] and Barnes [18]. Brown [19] used a similar approach for the concurrent k-ary tree. Shafiei [20] also applied the method for the concurrent patricia trie. It also showed how to design a concurrent operation where two pointers need to be changed. The above concurrent trees have an external structure, where only leaf nodes contain actual keys. Howley [5] designed the first internal BST by the cooperative method. Recently, Brown [21] presented a generalized template for all concurrent down-trees. Ellen [22] exhibited how to incorporate a stack to reduce the original complexity from $O(ch)$ to $O(h+c)$. Our quadboost is a hybrid of these techniques. It uses a cooperative method for concurrent coordination, changes two different positions with atomicity, and devises a continuous find mechanism to reduce restart cost.

Different from the previously mentioned methods that apply flags on nodes, Natarajan [6] illustrated how to apply flags on edges for a non-blocking external BST. Ramachandran [7] adopted CAS locks on edges to design a concurrent internal BST and later extended the work to non-blocking [8]. Their experiments showed that internal trees are more scalable than external trees with a large key range. On one hand, internal trees take up less memory; on the other hand, the remove operation is more complicated than

that in external trees. Chatterjee [9] provided a threaded-BST with edge flags and claimed it has a low theoretical complexity. Unlike the above trees that have to flag their edges before removal, our CAS quadtree uses a single CAS in both the insert operation and the remove operation.

The first balanced concurrent BST was proposed by Bronson [23] in which they used an optimistic and relaxed balance method to build an AVL tree. Besa [24] employed a similar method for a red-black tree. Crain [25] proposed a method that decouples physical adjustment from logical removal by a background thread. Drachler [10] mentioned an alternative technique called logical ordering, which uses the key order of the BST to optimize the contain operation. All of these works were constructed on fine-grained locks and are deadlock free. Based on the special properties of quadtree, we also decoupled physical adjustment from logical removal and achieved a high throughput.

There are numerous studies on concurrent trees in addition to ours and the ones mentioned in this work. Crain [26] designed a concurrent AVL tree based on STM. Prokopec [27] used a control node, which is similar to our *Operation* object to develop a concurrent trie. Arbel [28] provided a balanced BST with RCU and fine-grained locks.

8 CONCLUSIONS

In this paper, we present a set of concurrent quadtree algorithms—*quadboost*, which support concurrent insert, remove, contain, and move operations. In the remove operation, we decouple physical updates from the logical removal to improve concurrency. The continuous find mechanism analyses flags on nodes to decide whether to move down or up. Further, our LCA-based move operation can modify two pointers with atomicity. The experimental results demonstrate that quadboost outperforms existing one-dimensional tree structures while maintaining a two-dimensional hierarchy. The quadboost algorithms are scalable with a variety of workloads and thread counts.

REFERENCES

- [1] M. Moir and N. Shavit, "Concurrent data structures," *Handbook of Data Structures and Applications*, pp. 47–14, 2007.
- [2] N. Shavit, "Data structures in the multicore age," *Communications of the ACM*, vol. 54, no. 3, pp. 76–84, 2011.
- [3] T. David, R. Guerraoui, and V. Trigonakis, "Asynchronized concurrency: The secret to scaling concurrent search data structures," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2015, pp. 631–644.
- [4] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel, "Non-blocking binary search trees," in *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*. ACM, 2010, pp. 131–140.
- [5] S. V. Howley and J. Jones, "A non-blocking internal binary search tree," in *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2012, pp. 161–171.
- [6] A. Natarajan and N. Mittal, "Fast concurrent lock-free binary search trees," in *ACM SIGPLAN Notices*, vol. 49, no. 8. ACM, 2014, pp. 317–328.
- [7] A. Ramachandran and N. Mittal, "Castle: fast concurrent internal binary search tree using edge-based locking," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 2015, pp. 281–282.
- [8] —, "A fast lock-free internal binary search tree," in *Proceedings of the 2015 International Conference on Distributed Computing and Networking*. ACM, 2015, p. 37.
- [9] B. Chatterjee, N. Nguyen, and P. Tsigas, "Efficient lock-free binary search trees," in *Proceedings of the 2014 ACM symposium on Principles of distributed computing*. ACM, 2014, pp. 322–331.
- [10] D. Drachler, M. Vechev, and E. Yahav, "Practical concurrent binary search trees via logical ordering," in *ACM SIGPLAN Notices*, vol. 49, no. 8. ACM, 2014, pp. 343–356.
- [11] V. Ng and T. Kameda, "Concurrent accesses to r-trees," in *Advances in Spatial Databases*. Springer, 1993, pp. 142–161.
- [12] J. Chen, Y.-F. Huang, and Y.-H. Chin, "A study of concurrent operations on r-trees," *Information Sciences*, vol. 98, no. 1, pp. 263–300, 1997.
- [13] R. Obe and L. Hsu, *PostGIS in action*. Manning Publications Co., 2011.
- [14] A. C. Tassio Knop, "Qollide - Quadtrees and Collisions," <https://graphics.ethz.ch/~achapiro/gc.html>, 2010, [Online; accessed 29-August-2015].
- [15] G. J. Sullivan and R. L. Baker, "Efficient quadtree coding of images and video," *Image Processing, IEEE Transactions on*, vol. 3, no. 3, pp. 327–331, 1994.
- [16] D. P. Mehta and S. Sahni, *Handbook of data structures and applications*. CRC Press, 2004.
- [17] J. Turek, D. Shasha, and S. Prakash, "Locking without blocking: making lock based concurrent data structure algorithms nonblocking," in *Proceedings of the eleventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. ACM, 1992, pp. 212–222.
- [18] G. Barnes, "A method for implementing lock-free shared-data structures," in *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*. ACM, 1993, pp. 261–270.
- [19] T. Brown and J. Helga, "Non-blocking k-ary search trees," in *Principles of Distributed Systems*. Springer, 2011, pp. 207–221.
- [20] N. Shafiei, "Non-blocking patricia tries with replace operations," in *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*. IEEE, 2013, pp. 216–225.
- [21] T. Brown, F. Ellen, and E. Ruppert, "A general technique for non-blocking trees," in *ACM SIGPLAN Notices*, vol. 49, no. 8. ACM, 2014, pp. 329–342.
- [22] F. Ellen, P. Fatourou, J. Helga, and E. Ruppert, "The amortized complexity of non-blocking binary search trees," in *Proceedings of the 2014 ACM symposium on Principles of distributed computing*. ACM, 2014, pp. 332–340.
- [23] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun, "A practical concurrent binary search tree," in *ACM Sigplan Notices*, vol. 45, no. 5. ACM, 2010, pp. 257–268.
- [24] J. Besa and Y. Eterovic, "A concurrent red-black tree," *Journal of Parallel and Distributed Computing*, vol. 73, no. 4, pp. 434–449, 2013.
- [25] T. Crain, V. Gramoli, and M. Raynal, "A contention-friendly binary search tree," in *Euro-Par 2013 Parallel Processing*. Springer, 2013, pp. 229–240.
- [26] —, "A speculation-friendly binary search tree," *Acm Sigplan Notices*, vol. 47, no. 8, pp. 161–170, 2012.
- [27] A. Prokopec, N. G. Bronson, P. Bagwell, and M. Odersky, "Concurrent tries with efficient non-blocking snapshots," in *Acm Sigplan Notices*, vol. 47, no. 8. ACM, 2012, pp. 151–160.
- [28] M. Arbel and H. Attiya, "Concurrent updates with rcu: Search tree as an example," in *Proceedings of the 2014 ACM symposium on Principles of distributed computing*. ACM, 2014, pp. 196–205.

APPENDIX

.1 Basic Invariants

We provide a detailed proof of quadboost in this section. We follow the same naming convention in the paper. There are four kinds of *basic operations* in quadboost, i.e. the insert operation, the remove operation, the move operation, and the contain operation. Other functions are called *subroutines*, which are invoked by *basic operations*. The insert operation and the remove operation only operate on one terminal node, whereas the the move operation operates two different terminals—one for inserting a node with *newKey*, the other is for removing a node with *oldKey*. We call them *newKey*'s terminal and *oldKey*'s terminal, and we call their parents *newKey*'s parent and *oldKey*'s parent accordingly. Moreover, we name a CAS that changes a node's *op* a *flag* operation and a CAS that changes a node's child a *replace* operation. We define *snapshot_{T_i}* as the state of our quadtree at some time T_i .

.1.1 Subroutines

To begin with, we have following observations from quadboost.

Observation 1. *The key field of a Leaf node is never changed. The op field of a Leaf node is initially null.*

Observation 2. *The space information— $\langle x, y, w, h \rangle$ of an Internal node is never changed.*

Observation 3. *The root node is never changed.*

Based on Observation 3, we derive a Corollary as follows:

Corollary 1. *Two nodes in quadtree must share a common search path starting from the root.*

Using these observations, we prove that each subroutine satisfies their specific pre-conditions and post-conditions. Because all basic operations invoke the find function at line 134 that returns $\langle l, pOp, path \rangle$ and the findCommon function at line 270 that returns $\langle il, iOp, iPath, rl, rOp, rPath \rangle$, we prove that the two functions satisfy their pre-conditions and post-conditions beforehand. We suppose that they execute from a *snapshot_{T_i}* to derive these conditions.

Definition 1. *The pre-condition of find($l, pOp, path, keyX, keyY$) invoked at T_i :*

- 1) *If path is not empty, l was on the direction $d \in \{nw, ne, sw, se\}$ of the top node in path at $T_{i1} \leq T_i$.*

The post-conditions of find($l, pOp, path, keyX, keyY$) that returns a tuple $\langle l, pOp, path \rangle$ at T_i :

- 1) *l is a Leaf node or an Empty node.*
- 2) *At some $T_{i1} < T_i$, the top node in path has contained pOp.*
- 3) *At some $T_{i1} < T_i$, the top node in path has contained l.*
- 4) *If pOp was read at T_{i1} , and l was read at T_{i2} , then $T_{i1} < T_{i2} < T_i$.*
- 5) *For each node n in path, $size(path) \geq 2$, n_t is on the top of n_{t-1} , and n_t was on the direction $d \in \{nw, ne, sw, se\}$ of n_{t-1} at $T_{i1} \leq T_i$.*

Definition 2. *The pre-conditions of findCommon($il, rl, lca, rOp, iOp, iPath, rPath, oldKeyX, oldKeyY, newKeyX, newKeyY$) invoked at T_i :*

- 1) *If iPath is not empty, il was on the direction $d \in \{nw, ne, sw, se\}$ of the top node in iPath at $T_{i1} \leq T_i$.*
- 2) *If rPath is not empty, rl was on the direction $d \in \{nw, ne, sw, se\}$ of the top node in rPath at $T_{i1} \leq T_i$.*

The post-conditions of findCommon($il, rl, lca, rOp, iOp, iPath, rPath, oldKeyX, oldKeyY, newKeyX, newKeyY$) that returns two tuples $\langle il, iOp, iPath \rangle$ and $\langle rl, rOp, rPath \rangle$ at T_i :

- 1) *rl was a Leaf or an Empty node.*
- 2) *il was a Leaf or an Empty node.*
- 3) *At some $T_{i1} < T_i$, the top node in rPath has contained rOp.*
- 4) *At some $T_{i1} < T_i$, the top node in iPath has contained iOp.*
- 5) *At some $T_{i1} < T_i$, the top node in rPath has contained rl.*
- 6) *At some $T_{i1} < T_i$, the top node in iPath has contained il.*
- 7) *If iOp was read at T_{i1} , and il was read at T_{i2} , then $T_{i1} < T_{i2} < T_i$.*
- 8) *If rOp was read at T_{i1} , and rl was read at T_{i2} , then $T_{i1} < T_{i2} < T_i$.*
- 9) *For each node n in rPath, $size(rPath) \geq 2$, n_t is on the top of n_{t-1} , and n_t was on the direction $d \in \{nw, ne, sw, se\}$ of n_{t-1} at $T_{i1} \leq T_i$.*
- 10) *For each node n in iPath, $size(iPath) \geq 2$, n_t is on the top of n_{t-1} , and n_t was on the direction $d \in \{nw, ne, sw, se\}$ of n_{t-1} at $T_{i1} \leq T_i$.*

By observing the find function and the findCommon function, we have following lemmas.

Lemma 1. *At the first time calling the find function, the contain operation, the insert operation, and the remove function start with l as an Internal node and an empty path.*

Proof. The contain operation at line 97, the insert operation at line 105 and the remove operation at line 161 start with the root node by $l = root$, an Internal node that is never changed (Observation 3).

Besides, the contain operation at line 95, the insert operation at line 103, and the remove operation at line 160 start with an empty stack to record nodes. \square

Lemma 2. *At the first time the move operation calls the findCommon function, it starts with rl as an Internal node and rPath and iPath are empty.*

Proof. The move operation begins with the root node at line by $rl = root$ that is never changed (Observation 3) at line 204. Both rPath and iPath are initialized as empty stacks at line 199. \square

Lemma 3. *All nodes pushed by the find function (line 134) and the continueFind function (line 122) are Internal nodes.*

Proof. Before pushing into the stack, the find function at line 135 first check the class of a node. Also, the continueFind function calls the find function at line 132 to push nodes into path. Thus, nodes other than Internal cannot be pushed. \square

Lemma 4. All nodes pushed by the `findCommon` function (line 270) the `continueFindCommon` function (line 285) are *Internal*.

Proof. Before pushing into the stack, the `findCommon` function first checks the class of a node at line 271. Also, the `continueFindCommon` function calls the `findCommon` function at line 342. Or it calls the `find` function at line 323 and line 302 to push *Internal* nodes according to Lemma 3. Thus, nodes other than *Internal* cannot be pushed. \square

Lemma 5. For the loop in the `find` function at line 134 and the `findCommon` function at line 270, we suppose that $path(rPath)$ is empty and refer $l(rl)$ and $pOp(rOp)$ to each field updated by the loop. If the loop executes at least once and breaks at T_i , $l(rl)$, $pOp(rOp)$, and $path(rPath)$ satisfy following conditions:

- 1) $l(rl)$ is a *Leaf* node or an *Empty* node.
- 2) At some $T_{i1} < T_i$, the top node in $path(rPath)$ has contained $pOp(rOp)$.
- 3) At some $T_{i1} < T_i$, the top node in $path(rPath)$ has contained $l(rl)$.
- 4) If $pOp(rOp)$ was read at T_{i1} , and $l(rl)$ was read at T_{i2} , then $T_{i1} < T_{i2} < T_i$.
- 5) For each node n in $path(rPath)$, $size(path) \geq 2$, n_t is on the top of n_{t-1} , and n_t was on the direction $d \in \{nw, ne, sw, se\}$ of n_{t-1} at $T_{i1} \leq T_i$.

Proof. Apart from the terminate condition that the `findCommon` function judges whether two directions are the same or not, it performs in the same pattern as the `find` function. Both functions first push the previous *Internal* node l' into $path$, then get its op , and read a child pointer l finally.

The first condition always holds because by Lemma 4 and Lemma 3 *Leaf* nodes and *Empty* nodes will never be pushed into the $path(rPath)$.

We then prove part 2-4. At the last iteration, $pOp(rOp)$ and $l(rl)$ are read from l' , which has already been pushed into the $path(rPath)$. Hence, part 2 and part 3 are correct. Besides, $pOp(rOp)$ is read before $l(rl)$ so that part 4 holds.

For the last part, we assume that at T_{i1} the lemma holds, so at $T_{i2} > T_{i1}$ l is read from l' which has already been pushed into the $path(rPath)$. As $path(rPath)$ is initially empty, and T_i is at the end of the last iteration, l which on the top of l' has been a child of it. \square

Lemma 5 proves loop within the `find` function and the `findCommon` function satisfy post-conditions in Definition 1 and Definition 2. Next parts are going to show that other statements do not change these properties. We consider the first invocation as the base case, and prove lemmas by induction.

Lemma 6. Every call to the `find` function satisfies its pre-condition and post-conditions.

Proof. Now we prove the base case. We prove that initially the lemma holds.

The pre-condition:

By Lemma 1, the contain operation, the insert operation, and the remove operation start with an empty stack to record nodes.

For the move operation, by Lemma 2 it starts with rl as an *Internal* node and empty paths. Therefore it executes the

loop more than once to push *Internal* nodes into $path$. In the last iteration rl is pushed at line 272, and the parent of it has already been push at the prior iteration. Then, rl is read from at line 278. Hence, rl at line 279 is a child of the top node in $rPath$ if it is not empty. $iPath$ is empty since it starts with an empty stack.

Therefore, initially the pre-conditions are satisfied.

The post-conditions:

The first condition always satisfies as the class of a node is checked at line 135.

By Lemma 1, the contain operation, the insert operation, and the remove operation start with an empty stack. Thu Lemma 5 indicates that post-conditions are satisfied.

For the `find` function within the `findCommon` function, by Lemma 5 it satisfies post-conditions when it breaks out. Hence, at line 282, post-conditions are satisfied because $iPath$ is empty. At line 279, if rl starts as a *Leaf* node, the post-conditions are also satisfied. Otherwise, there are two parts of $rPath$ at the time the loop breaks out, where we have proved each of them satisfy post-conditions by Lemma 5. Moreover, by the pre-condition, rl has been a child of the top node in $rPath$ of the `find` function. Thus, all nodes pushed later also satisfy the post-conditions.

We have proved the base case. Then we assume that for invocations $find_0, find_1, find_2 \dots find_k$, the first k invocations satisfy the pre-conditions and post-conditions. We should prove that $find_{k+1}$ also satisfy the conditions. We have to consider all places where the `find` function is invoked.

The pre-condition:

First we consider the contain operation, the insert operation, and the move operation.

At line 98, line 106, line 163, the invocations follow the base case which we have proved satisfy the pre-condition.

At line 132, there are two scenarios l is read. l could either be read from the $path$ at line 126 or be assigned to its parent node at line 123. In the latter case, p was at the top of $path$ at $T_{i1} < T_i$ (line 109 and line 166) by the hypothesis. Otherwise l is assigned to a node in $path$. As there's no other push operations, n_t on the top of n_{t-1} must be its child at some $T_{i1} < T_i$ by the hypothesis.

Then, we discuss the move operation.

At line 279, the `find` function is wrapped by the `findCommon` function. Since Lemma 5 shows that the loop set rl as a child of the top node in $rPath$, we have to prove that either it is an *Internal* node or an *Leaf* node that was a child of the top node before entering the loop. At line 212, the `findCommon` function follows the base case. At line 342, Corollary 1 shows that the root node will always be a LCA node. Hence, after setting $rPath$ to its lca index at line 334, it contains at least one node. By Lemma 3 and Lemma 4, rl is an *Internal* node popped from $rPath$ at line 337.

At line 282, the `find` function is also wrapped by the `findCommon` function. We could prove that $iPath$ is empty. At line 212, the `findCommon` function follows the base case. At line 342, $iPath$ is set to empty before the invocation (line 335).

At line 302, rl is either assigned to rp (line 288) or a node popped from $rPath$ (line 295). If it is popped from $rPath$, our hypothesis ensures that it was a child of the top node from $rPath$. If it is read from rp , which is an *Internal* node

popped from $rPath$ at line 214, line 344, or line 305, it also satisfies the claim.

At line 323, likewise, il is either assigned to ip (line 312) or a node popped from $iPath$ (line 315). If it is popped from $iPath$, our hypothesis ensures that il was a child node of the top node from $iPath$ if it is not empty. If it is read from ip , which is an *Internal* node popped from $iPath$ at line 213, line 345, or line 326, it also satisfies the claim.

The post-conditions:

The first condition always holds. Because by Lemma 3, if l is an *Internal* node, it will be pushed into $path$.

We then prove other conditions.

First we consider the contain operation, the insert operation, and the move operation.

At line 98, line 106, line 163, the invocations follow the base case which we have proved satisfy the post-conditions.

At line 132, there are two scenarios l is read. l could either be read from the $path$ at line 126 or be assigned to its parent node at line 123. In both cases, l was an *Internal* node in $path$. Therefore Lemma 5 indicates that all nodes followed by l satisfy the post-conditions.

Then, we discuss the move operation.

At line 279, the find function is wrapped by the findCommon function. The pre-condition indicates rl was a child of $rPath$ if it is not empty. Hence, it suffices to show that the post-conditions are satisfied by Lemma 5.

At line 282, the find function is also wrapped by the findCommon function. The pre-condition indicates $iPath$ is an empty. In this way, by Lemma 5 we prove the post-conditions.

At line 302, likewise, the pre-condition suggests that rl was an *Internal* node in $rPath$. By Lemma 5 we also prove the post-conditions.

At line 323, the pre-condition suggests that il was an *Internal* node in $iPath$. Therefore, by Lemma 5 we prove the post-conditions. \square

After showing the pre-condition and post-conditions of the find function, we then prove that the findCommon function that returns $\langle il, iOp, iPath, rl, rOp, rPath \rangle$ satisfies its pre-conditions and post-conditions.

Lemma 7. *Every call to the findCommon function satisfies its pre-conditions and post-conditions.*

Proof. The pre-condition:

We prove the lemma by induction. First we prove the base case.

At line 212, the findCommon function is initially invoked, and both $iPath$ and $rPath$ are empty (line 199).

Then we assume that for invocations $findCommon_0, findCommon_1, findCommon_2 \dots findCommon_k$, the first k invocations satisfy the pre-conditions. We prove that $findCommon_{k+1}$ also satisfies the conditions.

There are two places the findCommon function is invoked. At line 212, the base case proves the post-conditions. At line 342, Corollary 1 shows that the root node will always be a LCA node. Hence, after setting $rPath$ to its lca index at line 334, it contains at least one node. By Lemma 3 and Lemma 4, rl is an *Internal* node popped from $rPath$ at line 337. Hence, we prove the pre-condition.

The post-conditions:

The findCommon function wraps two find functions at line 282 and line 279. Because $\langle rl, rOp, rPath \rangle$ that passed from the find function for rl and $\langle il, iOp, iPath \rangle$ that passed from the find function for il hold the post-conditions, the lemma is true. \square

After proving the pre-conditions and post-conditions of the find function and the findCommon function, we shall prove that the continuous find mechanism (i.e. the continueFind function and the continueFindCommon function) satisfies its pre-conditions and post-conditions.

Definition 3. *The pre-conditions of continueFind($pOp, path, l, p$):*

- 1) l was child of p .
- 2) p was child of the top node in $path$ if it is not empty.

The post-conditions are the same as the find function.

Definition 4. *The pre-conditions of continueFindCommon($il, rl, lca, rOp, iOp, iPath, rPath, ip, rp, oldKeyX, oldKeyY, newKeyX, newKeyY, iFail, rFail, cFail$):*

- 1) At least one of $iFail, rFail$ or $cFail$ is true.
- 2) il was a child of ip .
- 3) ip was a child of the top node in $iPath$ if it is not empty.
- 4) rl was a child of rp .
- 5) rp was a child of the top node in $rPath$ if it is not empty.

The post-conditions are the same as the findCommon function.

We now prove that the continueFind function satisfies its pre-conditions and post-conditions.

Lemma 8. *Every call to the continueFind function satisfies its pre-condition and post-conditions.*

Proof. The post-conditions:

At line 132 of the continueFind function, it executes the find function. Because the find operation satisfies its post-conditions by Lemma 6, the continueFind function also satisfies the same conditions.

The pre-condition:

The continueFind function is invoked at line 119 and line 176.

Initially, the continueFind function follows the find function at line 106 and line 163. For the first part, the post-condition of the find function shows that l was a child of p which popped at line 109 or line 166 (Lemma 5). For the second part, after popping p , it becomes a child of the top node in $path$ if it is not empty.

Otherwise, it reads results from the continueFind invocation at the prior iteration. As the post-conditions of the continueFind function is the same as the find function, we prove the lemma. \square

We now prove that the continueFindCommon function satisfies its pre-conditions and post-conditions.

Lemma 9. *Every call to the continueFindCommon function satisfies its pre-conditions and post-conditions*

Proof. The post-conditions:

There are three cases, the first part of the pre-condition shows that it must enter one of the loop. We consider each case by the program execution order.

- 1) If $rFail$ is true and $cFail$ is false, it executes the case starts from line 236. If $cFail$ is not set to true, it executes the find function at line 302. Because the find function satisfies its post-conditions (Lemma 6), the lemma holds. If $iFail$ is true, it comes to the second part of the proof. If $cFail$ is set to true, it comes to the third part of the proof.
- 2) If $iFail$ is true and $cFail$ is false, it executes the case starts from line 235. If $cFail$ is not set to true, it executes the find function at line 323. Because the find function satisfies its post-conditions, part 1 and Lemma 6 indicate the Lemma holds. If $cFail$ is true, it comes to the third part of the proof.
- 3) If $cFail$ is true, it executes the case starts from line 240. Because the findCommon function satisfies its post-conditions (Lemma 7), it establishes this part of the lemma.

The pre-condition:

The continueFindCommon function is invoked at line 243.

For the first condition, line 216, line 217, line 218, line 232, line 235, line 236, line 240 indicate that at least on of three flags is set to true.

For the next conditions, initially the continueFindCommon follows the findCommon function at line 212. As ip and rp are popped at line 213 and line 214, il was a child of ip and rl was a child of rp . Further, ip and rp was a child of the top node in $iPath$ and $rPath$. All these claims are based on Lemma 8. Otherwise, the continueFindCommon function reads results from its invocation at the prior iteration. In such a case, our post-conditions prove the pre-conditions. \square

We observe that the find function, the continueFind function, the findCommon function, and the continueFindCommon function return a similar pattern of results. The find function and the continueFind function return a tuple $\langle l, pOp, path \rangle$. The findCommon function and the continueFindCommon function return two such tuples. Hence We use $find(keys)$ to denote such a set of find operations for searching keys. Further, we number $find(keys)$ by their invocation orders. Invocations at line 106, line 163, and line 212 are at $iteration_0$. Invocations at line 106, line 163, and line 212 are at $iteration_i$, $0 < i$.

Definition 5. For $compress(path, p)$ invoked at T_i , it has following pre-conditions:

- 1) p is an Internal node.
- 2) $path$ is read from the result of $find(keys)$ at the prior iteration.
- 3) If $path$ is not empty, p was a child node of the top node in $path$ at $T_{i1} < T_i$.

Lemma 10. Every call to the compress function satisfies its pre-conditions.

Proof. The compress function is called at line 171 or line 229. At line 171, p is a node popped from $path$ (line 166). At line 229, rp is an Internal node popped from $rPath$ (line 214, line 305 or line 344). This proves the first part.

For last two parts, since the compress function reads result following $find(keys)$, it suffices to prove the post-conditions of $find(keys)$ satisfy the preconditions. By Lemma 8, Lemma 9, Lemma 6, and Lemma 7, the post-conditions of the find function, the findCommon function, the continueFind function, and the continueFindCommon function establish the lemma. \square

Corollary 2. Every call to the check function satisfies its pre-conditions that p is an Internal node.

Definition 6. For $moved(node)$ invoked at T_i , it has following the pre-conditions that $node$ is a Leaf node or an Empty node.

Lemma 11. Every call to the moved function satisfies its pre-condition.

Proof. The moved function is called at line 108, line 165, line 283, line 280, line 324, line 303.

At line 108, it reads l from line 106 or line 119. Hence, according to Lemma 6, l is a Leaf node or an Empty node.

At line 165, it reads l from line 163 or line 176. Hence, according to Lemma 6, l is a Leaf node or an Empty node.

At line 283 and line 280, it reads il from line 282 and rl from line 279. Therefore, according to Lemma 6, il and rl are Leaf node or Empty nodes.

At line 324, it reads il from line 323. Therefore, according to Lemma 6, il is a Leaf node or an Empty node.

At line 303, it reads rl from line 302. Therefore, according to Lemma 6, rl is a Leaf node or an Empty node. \square

Based on these post-conditions, we show that each op created at T_i store their corresponding information.

Lemma 12. For op created at T_i :

- 1) If op is a Substitute object, $op.parent$ has contained $op.oldChild$, a Leaf node, at $T_{i1} < T_i$ from the result of $find(keys)$ at the previous iteration.
- 2) If op is a Compress object, $op.grandparent$ has contained $op.parent$, an Internal node, at $T_{i1} < T_i$ from the result of $find(keys)$ at the previous iteration.
- 3) If op is a Move object, $op.iParent$ has contained $op.oldICChild$, a Leaf node, $op.rParent$ has contained $op.oldRChild$, a Leaf node, $op.iParent$ has contained $op.oldIOp$, and $op.rParent$ has contained $op.oldROp$ before T_i from the result of $find(keys)$ at the previous iteration.

Proof. 1) A Substitute object is created at line 112 or line 168 by assigning $\langle p, l, newNode \rangle$. Since p read from the top node in $path$ at line 109 or line 166, and l is returned by the find function or findCommon function, the post-conditions of them establish the claim according to Lemma 6, Lemma 7, Lemma 9, and Lemma 8.

2) A Compress object is created at line 185 by assigning $\langle path, p \rangle$. p is passed from the top of $path$ at line 166 or line 214, the post-conditions of $find(keys)$ and the property of $path$ establishes the claim (Lemma 6, Lemma 7, Lemma 9, Lemma 8, Lemma 3, and Lemma 4).

3) A Move object is created at line 224 by assigning $\langle ip, rp, il, rl, newNode, iOldOp, rOldOp \rangle$. ip is assigned to the top node of $iPath$ at line 213, rp is

assigned to the top node of $rPath$ at line 214, il was a child of the top node of $iPath$, rl was a child of the top node of $rPath$, and iOp and rOp are read from ip and rp by the post-conditions of the findCommon function and the continueFindCommon function (Lemma 9 and Lemma 7). \square

Then we prove the pre-conditions of other functions which use the result of above functions to modify quadtree.

The following lemmas satisfy a basic pre-condition that their arguments are the same type as indicated in algorithms.

- Lemma 13.** 1) Every call to the help function satisfies its pre-conditions.
 2) Every call to the helpSubstitute function satisfies its pre-conditions.
 3) Every call to the helpCompress function satisfies its pre-conditions.
 4) Every call to the helpMove function satisfies its pre-conditions.

Proof. 1) The help function is called at line 118, line 175, line 311, line 287, line 333, and line 332.

At line 118 and line 175, pOp might be obtained from the result of the find function and the continueFind function. Hence, the post-conditions of them ensures that pOp is an Operation object read from p . Otherwise, pOp might be obtained from line 116, or line 173 by reading a node's op .

At line 311 and line 287, rOp and iOp are passed the move operation. The post-conditions of the continueFindCommon function and the findCommon function (Lemma 7 and Lemma 9), with line 235, line 236, and line 240 guarantee that arguments' type are the same.

At line 333 and line 332, rOp and iOp could be passed from the move function or read from ip and rp in the continueFindCommon function at lines that have been mentioned.

- 2) The helpSubstitute function is called at line 114, line 170, and line 150. At line 114 and line 170, op is created at line 112 or line 168 respectively. At line 150, it checks whether op is *Substitute* before calling the helpSubstitute function.
 3) The helpCompress function is called at line 185, line 317, line 297, and line 339. For each invocation, it checks whether op is *Compress* beforehand.
 4) The helpMove function is invoked at line 228, line 239, and line 151. At line 228 and line 239, op is created at line 224. At line 151, it checks whether op is *Move* before the invocation. \square

Corollary 3. For $help(op)$, op is read from an Internal node.

Definition 7. For $helpFlag(p, oldOp, newOp)$ invoked at T_i , it has the pre-conditions:

- 1) p is an Internal node.
- 2) For $cflag$, p has contained pOp at $T_{i1} < T_i$ after reading p ; otherwise, p has contained pOp at $T_{i1} < T_i$ from the result of $find(keys)$ at the prior iteration.

Lemma 14. Every call to the helpFlag function satisfies its pre-conditions.

Proof. The helpFlag function is invoked at line 113, line 169, line 186, line 226, line 248-line 250, and line 261-line 267.

At line 113 and line 169, the post-conditions of the find function and the continueFind function (Lemma 6 and Lemma 8 imply that p is an Internal node popped at line 109 and has contained pOp .

At line 186, the pre-condition of the compress function (Lemma 10 shows that p and nodes from $path$ are read from $find(keys)$ at the prior iteration, and pOp is read at line 181. Thus, p has pOp at $T_{i1} < T_i$ after reading p .

At line 226, ip and rp are popped from $iPath$ at line 213 and line 214 or read from the continueFindCommon function. By Lemma 7 and Lemma 9, either $op.iParent$ has contained $op.iOp$ or $op.rParent$ has contained $op.rOp$.

At line 248-line 250 and line 261-line 267, by the pre-condition of the helpMove function 13, op is a Move object. Therefore, according to Lemma 12, either $op.iParent$ has contained $op.iOp$ or $op.rParent$ has contained $op.rOp$. \square

Lemma 15. Every call to the hasChild function satisfies its pre-conditions that parent is an Internal node.

Proof. According to Lemma 11, $node$ is a Leaf node or an Empty node. According to Lemma 14, flag operations only perform on Internal nodes. Therefore line is the only place that set a Move op on a Leaf node. By Observation 1, op is initially null. After checking the condition, op is assigned to a Leaf node where $op.iParent$ is an Internal node by Lemma 12 \square

Definition 8. For $helpReplace(p, oldChild, newChild)$ invoked at T_i , it has the pre-conditions:

- 1) p , $oldChild$, and $newChild$ are read from the same op .
- 2) $newChild$ is a node that has not been in quadtree.
- 3) p is an Internal node that has contained $oldChild$ at $T_{i1} < T_i$.

Lemma 16. Every call to the helpReplace function satisfies its pre-conditions.

Proof. For the first condition:

The helpReplace function is invoked at line 145, line 193, line 254-259.

By Lemma 13, at line 193, it reads a Compress object; at line 145, it reads a Substitute object; at line 254-259. it reads a Move object.

For the second condition:

Since part 1 illustrates that p , $oldChild$, and $newChild$ are read from the same op , if op is a Substitute object, $op.newNode$ is created at line 111 or line 161; if op is a Compress object, each time the helpCompress function creates a new Empty node at line 193; if op is a Move object, $op.newIChild$ is created at line 220, and an empty node is created at line 145.

For the third condition:

Since part 1 illustrates that p , $oldChild$, and $newChild$ are read from the same op , Lemma 12 shows that p has contained $oldChild$. \square

Definition 9. For $createNode(l, p, newKeyX, newKeyY, value)$ that returns a newNode invoked at T_i , it has the pre-conditions:

- 1) p is an Internal node.
- 2) p has contained l at $T_{i1} < T_i$ from the result of $find(keys)$ at the prior iteration.

post-conditions:

The `newNode` returned is either a Leaf node with `newKey` and `value`, or a sub-tree that contains both `l.key` node and `newKey` node with the same parent.

Lemma 17. Every call to the `createNode` function satisfies its pre-conditions and post-condition.

Proof. The `createNode` function is invoked at line 111 and line 221.

At line 111, the post-conditions of the `find` function and the `continueFind` function ensure that l has been a child of p that popped from `path` at line 109 (Lemma 6 and Lemma 8).

At line 221, the post-conditions of the `findCommon` function and the `continueFindCommon` function ensure that l has been a child of p popped at line 213. (Lemma 7, Lemma 9).

Though we have not presented the details of the `createNode` function, our implementation guarantee that the post-condition must be satisfied. \square

.1.2 Flag and replace operations

We argue that each successful CAS operation occurs in a correct order. At outset, we argue that CAS behaviours for each `helpFlag` function. A flag operation involves three arguments: `node`, `oldOp`, `newOp`. A replace operation involves three arguments: `p`, `oldChild`, `newChild`. From Figure 9, we denote each flag operation accordingly. *iflag* occurs in the insert operation, *rflag* occurs in the remove operation, and *mflag* occurs in the move operation. Moreover, we specify a flag operation which attaches a *Clean op* on a node as an unflag operation. We call *ireplace* as the replace operation occurs in the insert operation, *rreplace* as the replace operation occurs in the remove operation, *mreplace* as the replace operation occurs in the move operation, and *creplace* as the replace operation occurs in the compress operation. The next lemmas describe the behaviours of the `helpFlag` function according to the state transition diagram.

Observation 4. *op* is only attached on an Internal node.

Lemma 18. For a new node n :

- 1) It is created with a *Clean op*.
- 2) *rflag*, *iflag*, *mflag* or *cflag* succeeds only if n 's *op* is *Clean*.
- 3) *unflag* succeeds only if n 's *op* is *Substitute* or *Move*.
- 4) Once n 's *op* is *Compress*, its *op* will never be changed.

Proof. 1) As shown at line 71, Internal nodes are assigned a *Clean op* when they are created.

- 2) Before *iflag*, *op* is checked at line 110; before *rflag*, *op* is checked at line 167; before *cflag*, *op* is checked at line 182. For the move operation, before *mflag* on `oldKey`'s parent and `newKey`'s parent, they are checked at line 217 and line 216.
- 3) *unflag* is called at line 146 and line 261-line 267, where the pre-conditions specify that *op* is *Move* or *Substitute* according to Lemma 13.
- 4) By part 3, there's no unflag operation happens on an Internal node with a *Compress op*.

\square

Next we illustrate that there's no ABA problem on any *op*. That is to say, in terms of an Internal node n , its op_i at T_i has not appeared at $T_{i1} < T_i$, $op_{i1} \neq op_i$. We prove the following lemma:

Lemma 19. For an Internal node n , it never reuse an *op* that has been set previously.

Proof. If a node's *op* is set to op_i at T_i , it has never been appeared before. We prove the lemma by discussing different types of *flag* operations.

For *iflag*, a *Substitute op* is created at line 112 before its invocation. For *rflag*, a *Substitute op* is created at line 168 before its invocation. For a *cflag*, a *Compress op* is created at line 185 before its invocation. For *mflag*, a *Move op* is created at line 224 before its invocation. Because each *newOp* is newly created, it has not been set before. For *unflag*, each time it creates a new *Clean op* to replace the prior *Move op* or *Substitute op* by Lemma 18. Therefore, the new *Clean op* has never appeared before. \square

We have shown that for each successful flag operation, it never set an *op* that has been used before. Every Internal node is initialized with *Clean op* by Lemma 18 and we use a flag operation to change *op* at the beginning of each CAS transition. We say successive replace operations and unflag operations that read the *op* belongs to it. According to Figure 9, there are three categories of successful CAS transitions. We denote them as *flag* \rightarrow *unflag*, *flag* \rightarrow *replace*, and *flag* \rightarrow *replace* \rightarrow *unflag*.

Let $flag_0, flag_1, \dots, flag_n$ be a sequence of successful flag operations; let $unflag_0, unflag_1, \dots, unflag_n$ be a sequence of successful unflag operations. $flag_i$ attaches op_i , and $replace_i$ and $unflag_i$ read op_i and come after it. Therefore we say $flag_i, replace_i$, and $unflag_i$ belong to the same *op*. In addition, if there are more than one replace operations belong to the same op_i , we denote them as $replace_i^0, replace_i^1, \dots, replace_i^k$ ordered by their successful sequence. Similarly, if there are more than one flag operations that belong to op_i on different nodes, we denote them as $flag_i^0, flag_i^1, \dots, flag_i^k$. A similar notation is used for $unflag_i$. The following lemmas prove the correct ordering of three different transitions.

Lemma 20. 1) *ireplace* will not be done before the success of *iflag* which belongs to the same *op*.
2) *rreplace* will not be done before the success of *rflag* which belongs to the same *op*.

Proof. For the insert operation and the remove operation, at line 145 the `helpReplace` function is wrapped by the `helpSubstitute` function. The `helpSubstitute` function is called at line 114, line 170, and line 150. At line 114, it is called after the success of *iflag* at line 113. At line 170, it is called after the success of *rflag* at line 169. At line 150, by Corollary 3 and Lemma 18, there must have been *iflag* or *rflag* that change the node's *op* from *Clean* to *Substitute*. \square

Lemma 21. *creplace* will not be done before the success of *cflag* which belongs to the same *op*.

Proof. The `helpCompress` function which wraps *creplace* is called at line 185, line 317, line 297, and line 339, and

line 149. At line 185, the `helpCompress` function follows the successful `helpFlag` function at line 186. At line 317, line 297, and line 339, a `pOp` is read from `Internal` nodes. At line 149, Lemma 3 shows that `op` is read from an `Internal` node. Hence, by Lemma 18, successful `cflag` must have changed the node's `op` from `Clean` to `Compress`. \square

Lemma 22. *`mreplace` will not be done before the success of `mflag` which belongs to the same `op`.*

Proof. The `helpMove` function which wraps two `mreplaces` is called at line 228, line 239, and line 151. At line 228 and line 239, the `Move op` is created at line 267. Otherwise, at line 151, by Corollary 3, `op` is read from an `Internal` node. Hence, Lemma 18 demonstrates that a successful `mflag` must have changed the node's `op` from `Clean` to `Move`. \square

From the above lemmas, we have the following corollary:

Corollary 4. *A successful `replace` operation which belongs to an `op` will not succeed before its `flag` operations have been done.*

We discuss each CAS transition accordingly. We begin by discussing the `flag` \rightarrow `replace` \rightarrow `unflag` transition. We refer $\langle \text{oldIOp}, \text{oldIChild}, \text{newIChild} \rangle$ and $\langle \text{oldROp}, \text{oldRChild}, \text{newRChild} \rangle$ to different tuples of $\langle \text{oldOp}, \text{oldChild}, \text{newChild} \rangle$.

Lemma 23. *`flag` \rightarrow `replace` \rightarrow `unflag` occurs when `rflagi`, `iflagi`, or `mflagi` succeeds, and it has following properties:*

- 1) *`replacei` never occurs before `flagi`.*
- 2) *`flagik`, $0 \leq k < |\text{flag}_i|$ is the first successful flag operation on `opi.parentk` after T_{i1} when `pOpi` is read.*
- 3) *`replaceik`, $0 \leq k < |\text{replace}_i|$ is the first successful replace operation on `opi.parentk` after T_{i2} when `opi.oldChildk` is read.*
- 4) *`replaceik`, $0 \leq k < |\text{replace}_i|$ is the first successful replace operation on `opi.parentk` that belongs to `opi`.*
- 5) *`unflagik`, $0 \leq k < |\text{unflag}_i|$ is the first successful unflag operation on `opi.parentk` after `flagik`.*
- 6) *There is no successful unflag operation occurs before `replacei`.*
- 7) *The first replace operation on `opi.parentk` that belongs to `opi` must succeed.*

Proof. 1) Corollary 4 proves the claims.

- 2) By Lemma 14, for each flag operation, `p` has contained `pOp` at some time T_{i1} . Suppose that there's another `flagk` that occurs after T_{i1} but before `flagik`, then `flagik` fails because Lemma 19 shows that `op` is not reused. Therefore, it contradicts to the definition that `flagik` is a successful flag operation.
- 3) By Lemma 16, for each replace operation, `p` has contained `oldChild` at some time T_{i1} . Suppose that there's another `replace` that occurs after T_{i1} but before `replaceik`, then `replaceik` fails because Lemma 16 shows that `newChild` is never a node in the tree. Therefore, it contradicts to the definition that `replaceik` is a successful flag operation.
- 4) Suppose there's `replacek` that belongs to `opi` and occurs before `creplaceik`. It must follow `flagk` according to Corollary 4. If `flagk` happens before `flagik`, it fails because `failik` will fail by consequence

(Lemma 19). If `flagk` happens after `flagik`, it contradicts Lemma 18 that a flag operation starts because `flagk` is done after reading `opi.oldChildk`, `replacek` is also after reading it. Therefore, if `flagk` succeeds, it contradicts part 3 that `replaceik` is the first successful replace operation after reading `opi.oldChildk`.

- 5) Based on Lemma 14, if there is another `unflag` changes `p.op`, `unflagik` that use `opi` as the `oldOp` will fail. This contradicts to the definition that `unflagik` is a successful unflag operation that reads `opi`.

- 6) We consider each operation separately. For the insert operation and the remove operation, `unflag` follows `replace` immediately at line 146. Assuming that `replaceik` occurs after `unflagik`, there must be another `replace` reads `opi` and changes the link. It contradicts part 3 of the proof that `replaceik` is the first successful replace operation that belongs to `opi`.

For the move operation, because we assume that replace operations exist, `doCAS` is set to true before performing replace operations. An unflag operation follows replace operations from line 261-267. Consider if a `replaceik`, $0 \leq k < |\text{replace}_i|$ happens after any unflag operation, then another `replace` which use the same `op` must have succeed as the program order specifies that `replace` execute before `unflag` for a process. Hence, `replaceik` fails and contradicts the definition.

Therefore all successful unflag operations occur after `replace`.

- 7) According to Corollary 4, `replaceik` follows `flagik` belongs to `opi`. Suppose there exists `replace` after reading `opi.parentk` and before `replaceik`. Then, there must be `flag` that precedes `replace` by Corollary 4. By part 1, `flagi` is the first successful flag operation after reading `opi`. By Lemma 14, `opi` is read before `opi.l`. Hence, there does not exist any replace operation between T_{i1} reading `opi.oldChildk` and `flagik`. If `flag` happens between `flagik` and `replaceik`, it will also fail because `opi.parentk.op` is not `Clean`. Then, the first replace operation that belongs to `opi` must succeed. \square

We then discuss the ordering of the `flag` \rightarrow `replace` transition.

Lemma 24. *`flag` \rightarrow `replace` occurs only when `cflagi` succeeds, it has following properties:*

- 1) *`creplacei` never occurs before `cflagi`.*
- 2) *`cflagi` is the first successful flag operation on `opi.parent` after T_{i1} when `rOpi` is read.*
- 3) *`creplacei` is the first successful replace operation on `opi.grandparent` after T_{i1} when `opi.parent` is read.*
- 4) *`creplacei` is the first successful replace operation on `opi.grandparent` that belongs to `opi`.*
- 5) *There is no unflag operation after `creplacei`.*
- 6) *The first replace operation that belongs to `opi` must succeed.*

Proof. 1) Corollary 4 proves the claims.

- 2) By Lemma 14, for each flag operation, p has contained pOp at some time T_{i1} . Suppose that there's another $cflag$ that occurs after T_{i1} but before $cflag_i$, then $cflag_i$ fails because Lemma 19 shows that op is not reused. Therefore, it contradicts to the definition that $cflag_i$ is a successful flag operation.
- 3) By Lemma 16, for each replace operation, p has contained $oldChild$ at some time T_{i1} . Suppose that there's another $creplace$ that occurs after T_{i1} but before $replace_i$, then $creplace_i$ fails because $oldChild$ remains the same but the link has been changed to another node. Therefore, it contradicts to the definition that $creplace_i$ is a successful flag operation.
- 4) Suppose there's a replace operation that belongs to op_i and occurs before $creplace_i$. It must follow $cflag$ according to Corollary 4. If $cflag$ happens before $cflag_i$, it fails because $cfail_i$ will fail by consequence (Lemma 19). If $cflag$ happens after $cflag_i$, it contradicts Lemma 18 that a flag operation starts with a *Clean op*. Therefore, $cflag$ is the same as $cflag_i$. Because $cflag$ is done after reading $op_i.parent$, $creplace$ is also after reading it. Therefore, if $cflag$ succeeds, it contradicts part 3 that $creplace_i$ is the first successful replace operation after reading $op_i.parent$.
- 5) By Lemma 18, once a node's op is *Compress*, it will never be changed.
- 6) Suppose the first replace operation that belongs to op_i fails, there must exist some successful $replace$ after reading $op_i.parent$ and before $creplace_i$. Further, there is $flag$ that precedes $replace$ by Corollary 4. If $flag$ happens between $creplace_i$ and $cflag_i$, $flag$ will fail by Lemma 18. If $flag$ happens between reading op_i and $cflag_i$, it contradicts part 1 that $cflag_i$ is the first successful flag operation after reading op_i . Because op_i is read after $op_i.parent$, we also have to consider if $flag$ happens before reading op_i . Consider if $cflag$ happens, $flag_i$ will fail because $op_i.parent.op$ will not be set back to *Clean* according to the fourth part. Consider if $mflag$, i flag or r flag happens, it will not change the link from $op_i.grandparent$ to $op_i.p$ according to Lemma 23. \square

Corollary 5. *The first replace operation belongs to op_i must succeed.*

Next we discuss the $flag \rightarrow unflag$ transition, where replace operation occurs between a successful flag operation and an unflag operation that belong to the same op . We suppose that $op.iFirst$ is true. The claim for $op.iFirst = false$ is symmetric.

Observation 5. *$allFlag$ and $iFirst$ in a Move op are initially false, and they will never be set false after assigning to true.*

Lemma 25. *For $flag \rightarrow unflag$ circle, it only results from a Move object op_i such that:*

- 1) $unflag_i$ is the first successful unflag operation on $op_i.rParent$.
- 2) The first flag operation on $op_i.iParent$ must fail, and no later flag operation will succeed.

- 3) $op_i.iParent$ and $op_i.rParent$ are different.

Proof.

- 1) We prove it by contradiction. Suppose there's another unflag operation that reads op_i and succeeds before $unflag_i$ on $op_i.rParent$. It must have changed $op_i.rParent.op$ to a new op , by Lemma 19 $unflag_i$ will fail. This contradicts the definition of $unflag_i$.
- 2) For the former part of the claim, we prove it by contradiction. Assuming that $op_i.iParent$ is flagged on op_i , and will be unflagged without performing replace operations. However, unflag operations occur after the judgement at line 254. The first process must set $doCAS$ to true. In the next step, it executes line 254-259. According to Corollary 5, the first $replace_i^t$ must succeed. Therefore it contradicts the definition that there's no replace operation before an unflag operation. For the latter part, by Lemma 12 $op_i.iParent$ has contained $op_i.oldIOp$. Because the first flag operation on $op_i.iParent$ fails, latter flag operations read the $op_i.oldIOp$ also fail.
- 3) Assuming $op_i.iParent$ and $op_i.rParent$ are the same. By part 2, flag operations on $op_i.iParent$ must fail, hence $doCAS$ will never be true. $op_i.allFlag$ is always false by Observation 5. In this way, line 261-267 prevents any unflag operation. This derives a contradiction. \square

1.3 Quadtree properties

In the Section, we use above lemmas to show that quadtree's properties are maintained during concurrent modifications.

Definition 10. *Our quadtree has these properties:*

- 1) *Two layers of dummy Internal nodes are never changed.*
- 2) *An Internal node n has four children, which locate in the direction $d \in \{nw, ne, sw, se\}$ respectively according to their $\langle x, y, w, h \rangle$, or $\langle keyX, keyY \rangle$.*

For Internal nodes reside on four directions:

- $n.nw.x = n.x, n.nw.y = n.y;$
- $n.ne.x = n.x + w/2, n.ne.y = n.y;$
- $n.sw.x = n.x, n.sw.y = n.y + n.h/2;$
- $n.se.x = n.x + w/2, n.se.y = n.y + h/2,$

and all children have their $w' = n.w/2, h' = n.h/2$.

For Leaf nodes reside on four directions:

- $n.x \leq n.nw.keyX < n.x + n.w/2, n.y \leq n.nw.keyY < n.y + n.h/2;$
- $n.x + n.w/2 \leq n.ne.keyX < n.x + n.w, n.y \leq n.ne.keyY < n.y + n.h/2;$
- $n.x \leq n.sw.keyX < n.x + n.w/2, n.y + n.h/2 \leq n.sw.keyY < n.y + n.h;$
- $n.x + n.w/2 \leq n.se.keyX < n.x + n.w, n.y + n.h/2 \leq n.se.keyY < n.y + n.h.$

Lemma 26. *Two layers of dummy nodes are never changed.*

Proof. We prove the lemma by induction. Only replace operations will affect the structure of quadtree as we define in Definition 10.

Initially the property holds as we initialize quadtree with two layers of dummy nodes and a layer of *Empty* nodes.

Suppose that after $replace_i$ the lemma holds, we shall prove that after $replace_{i+1}$ the lemma is still true. For $ireplace$, $rreplace$, and $mreplace$, the pre-condition 16 ensures that three nodes are from the same op . Thus, as Lemma 12 further ensures that they only swing *Leaf* nodes and *Empty* node, dummy node are not changed. For $creplace$, since at $replace_i$ the property is true, the root is connected to a layer of dummy nodes. Lemma 12 shows that $op.oldChild$ is an *Internal* node, and line 186 shows that the root node's child pointer will never be changed. Therefore, $replace_{i+1}$ will not occur on the root node. \square

Lemma 27. *Children of a node with Compress op will not be changed.*

Proof. Based on Lemma 18, the node's op will never be changed after being set to *Compress*. Lemma 24 and Lemma 23 indicate that for a node flagged with *Compress op*, only itself will be unlinked. This establishes the Lemma. \square

Lemma 28. *Only an Internal node with all children Empty could be attached with a Compress object.*

Proof. Corollary 2 ensures that the pre-conditions of the check function is satisfied. It checks whether all children are *Empty* before calling the $helpFlag$ function at line 186. This establishes the Lemma. \square

Lemma 29. *An Internal node whose op is not Compress is reachable from the root.*

Proof. We have to consider all kinds of replace operations.

For $ireplace$, $rreplace$, and $mreplace$, by Lemma 16 and Lemma 12 they use a new node to replace a *Leaf* node or an *Empty* node. Thus, *Internal* nodes are still reachable from the root.

For $creplace$, by Lemma 16 and Lemma 12, it replaces *Internal* nodes. Lemma 24 shows that it is preceded by $cflag$. Lemma 28 implies that all children are *Empty* nodes. Therefore, an *Internal* node with a *Compress op* and its children are not reachable from the root only after the success of $creplace$. \square

Next we define *active* set and *inactive* set for different kinds of nodes. We say a node is *moved* if the $moved$ function turns true. For an *Internal* node or an *Empty* node, if it is reachable from the root in $snapshot_{T_i}$, it is *active*; otherwise, it is *inactive*. For a *Leaf* node, if it is reachable from the root in $snapshot_{T_i}$ and not *moved*, it is *active*; otherwise, it is *inactive*. We denote $path(keys^k)$, $0 \leq k < |replace_i|$ as a stack of nodes pushed by $find(keys)$ in a snapshot. We define $physical_path(keys^k)$ to be the path for $keys^k$ in $snapshot_{T_i}$, consisting of a sequence of *Internal* nodes with a *Leaf* node or an *Empty* node at the end, which is the actual path in the snapshot. We say a subpath of $path(keys^k)$ is an *active_path* if all nodes from the root to node $n \in path(keys^k)$ are *active* due to their pushing order. Hence a physical path with a *Leaf* node is *active* only if $path(keys^k)$ is an *active path* and the *Leaf* node is *active*.

Lemma 30. *There's at most a node with Compress op in the subpath of $path(keys^k)$ that is active, and it resides in the end of the path if exists.*

Proof. Assume that there are two nodes n and n' reside in the $active_path(keys^k)$ with *Compress op*. We prove the lemma by contradiction. Because n and n' are *active* with *Compress op*, all children of n and n' are *Empty* (Lemma 28). Since n and n' are *Internal* nodes in the same $path$, it derives a contradiction.

For the second part of the proof, if n resides in other places in $path$, its children should be *Empty* by Lemma 28. Therefore, only if n is on the end of path, conditions are satisfied. \square

Lemma 31. *For $path(keys^k)$, if n_t is active in $snapshot_{T_i}$, then n_0, \dots, n_{t-1} above n_t are active.*

Proof. By Lemma 29, nodes with op other than *Compress* are reachable from the root. Lemma 30 indicates that a node with a *Compress op* is on the end of the path. Therefore, other nodes do not have a *Compress op*, which establishes the lemma. \square

Lemma 32. *If in the $snapshot_{T_i}$, n is the LCA on $active_path$ for $oldKey$ and $newKey$. Then at $T_{i1}, T_{i1} > T_i$, n is still the LCA on $active_path$ for both $oldKey$ and $newKey$ if it is active.*

Proof. Nodes from root to the LCA node shares a common path (Observation 1). Because the LCA node is *active*, nodes above the LCA node are *active*. Hence, the subpath from the root to the LCA node is *active* by Lemma 31. \square

Lemma 33. *Nodes a with a Compress op will not be pushed into path more than once.*

Proof. Supposing that a node n is *active* in $snapshot_{T_i}$, and it becomes *inactive* at $T_{i1} > T_i$. Besides, suppose that op of n is *Compress* and pushed at $T_{i2}, T_i < T_{i2} < T_{i1}$. We prove that after T_{i2} , n will never be pushed again.

Lemma 24 shows that a node with a *Compress op* will never be unflagged. Hence, based on Lemma 6 and Lemma 6), we have to prove that nodes in $path$ with a *Compress op* will not be pushed into again. For the insert operation and the remove operation, pOp is checked at line 123 and line 127 before the next find operation. For the move operation, rOp is checked at line 312, line 316, line 339, and iOp is checked at line 296 and line 288 before calling the $findCommon$ function and the $find$ function. Hence, at $T_{i2} < T_{i1}$, op must be checked before $find(keys)$ so that nodes with a *Compress op* will not be pushed more than once. \square

Lemma 30 shows that for $active_path$, *active* nodes with a *Compress op* will only reside at the end. If other nodes pushed before n will become *inactive*, we can pop them beforehand until reaching the last node an op other than *Compress*.

Lemma 34. *After the invocation of $find(keys)$ which reads l^t , there is a snapshot, where the path from the root to l^t is $physical_path(keys^t)$.*

Proof. We prove the lemma by induction.

In the base case, where $path$ starts from the root node, the claim is true.

We consider the pushing sequence as n_0, n_1, \dots, n_k . Suppose that for first k nodes, the path from the root to n_k is

the *physical_path*. We shall prove that for n_{k+1} , the lemma is true. If there is no *replace* operation before reading n_{k+1} , it is obvious that we can linearized it as the same snapshot before pushing n_k .

Next, we assume that *replace* occurs on n_k before reading n_{k+1} , and results in *snapshot*. There are two cases:

- 1) Consider if *replace* occurs after reading n_{k+1} . We could have $snapshot_{k+1} = snapshot_k$ because n_{k+1} is connected to an active node in a snapshot so that n_{k+1} is also reachable from the root.
- 2) Consider if *replace* occurs before reading n_{k+1} . If the replace operation changes n_k , then n_k is flagged with a *Compress op* by Lemma 24. Then we have $snapshot_{k+1} = snapshot_k$ because Lemma 27 demonstrates that a node flagged with a *Compress* has no successive replace operations on its child pointers. Otherwise, if the replace operation changes n_{k+1} , n_k is not changed by Lemma 23. Hence, we have $snapshot_{k+1} = snapshot$, where n_{k+1} is reachable from the root in the *snapshot* just after a replace operation. □

Lemma 35. *After ireplace, rreplace, mreplace, and creplace, quadtree's properties remain.*

Proof. We shall prove that in any $snapshot_{T_i}$, quadtree's properties remain.

First, Lemma 10 shows that two layers of dummy nodes remain in the tree. We have to consider other layers of nodes which are changed by replace operations.

Consider *creplace* that replaces an *Internal* node by an *Empty* node. Because the *Internal* node has been flagged on a *Compress op* before *creplace* (Lemma 24), all of its children are *Empty* and not changed (Lemma 28 and Lemma 27). Thus, *creplace* does not affect the second claim of Definition 10.

Consider *ireplace*, *rreplace*, or *mreplace* that replaces a terminal node by an *Empty* node, a *Leaf* node, or a sub-tree. By Lemma 23, before $replace^k$, $op.parent^k$ is flagged with op such that no successful replace operation could happen on $op.parent^k$. Therefore, if the new node is *Empty*, it does not affect the tree property. If the new node is a *Leaf* node or a sub-tree, based on the post-conditions of the createNode function (Lemma 17), after replace operations the second claim of Definition 10 still holds. □

.2 Linearizability

In this Section, we define linearization points for *basic operations*. As the compress operation is included in the move operation and the remove operation that returns true, it does not affect the linearization points of them. If an algorithm is linearizable, it could be ordered equivalently as a sequential one by their linearization points. Since all the modifications depend on $find(keys)$, we first point out its linearization point.

To prove that our linearization points are correct, we shall demonstrate that for some time T_i , the key set in $snapshot_{T_i}$ is the same as the results of modifications linearized before T_i . For $find(keys)$, we should define the

linearization point at some $snapshot_{T_i}$ so that l^k returned is on the *physical_path* for key^k in $snapshot_{T_i}$.

Lemma 36. *For $find(keys)$ that returns tuples $\langle l^k, pOp^k, path^k \rangle$, there's a snapshot after its invocation and before reading l^k , such that $path(keys^k)$ returned with l^k at the end is a *physical_path* in $snapshot_{T_i}$.*

Proof. Since l^k is a *Leaf* node or an *Empty* node, by Lemma 34 there is a $snapshot_{T_i}$ that l^k as the last node from l_{i-1} , and l_0 to n_{i-1} in the $path(keys^k)$ are active. Therefore $path(keys^k)$ with l^k is *physical_path(keys^k)* in $snapshot_{T_i}$. We define $snapshot_{T_i}$ as $find(keys^k)$'s linearization point. □

In the next, we define linearization points for *basic operations*. For the insert operation, the remove operation, and the move operation that returns true, we define it to be the first $replace_i^k, 0 \leq k < |replace_i|$ belongs to op_i . To make a reasonable demonstration, we first show that the first $replace_i^k$ belongs the op_i created by each operation itself, and then illustrate that op is unique for each operation.

Lemma 37. *If the insert operation, the remove operation, or the move operation that returns true, the first $replace_i^k$ occurs before the returning, and it belongs to op_i created by the operation itself.*

Proof. First we prove that $replace_i^k$ belongs to op_i created by the operation itself.

For the insert operation, it returns true after successful *i*flag and *ireplace* using op created at line 112.

For the remove operation, it returns true after successful *r*flag and *rreplace* using op created at line 168.

For the move operation, it returns true after successful *m*flag and *mreplace* using op created at line 224.

Corollary 5 indicates that the first replace operation must succeed, therefore all successful replace operations happen before returning. □

Then, we define the linearization points of the insert operation, the remove operation, and the move operation that returns true are before returning. We should also point out that there's only one op lead to the linearization point for each operation.

Lemma 38. *For the insert operation, the remove operation, and the move operation that returns true, op is created at the last iteration in the while loop.*

Proof. Based on Lemma 23, all successful insert, remove and move operations follow the $flag \rightarrow replace \rightarrow unflag$ transition. *i*flag is the first successful flag operation after reading $op_i.parent$, so there is no other flag operation further. This establishes the claim. □

Lemma 39. *If the insert operation, the remove operation, and the move operation that return false, successful replace happens before returning.*

Proof. As Lemma 38 points out that op is created at the last iteration. Moreover, before returning true, *m*flag, *i*flag and *r*flag must succeed according to the program order. As the first replace operation that belongs to op_i must succeed (Lemma 5, *replace* happens before returning. □

Lemma 40. *If the insert operation, the remove operation, and the move operation that return false, there's no successful replace ever happened.*

Proof. Consider the execution of the insert operation, it returns false at line 108 where op created are not flagged on nodes. Consider the remove operation, it returns false at line 165. Also, op created are not flagged on nodes. For the move operation, it returns false by calling the `findCommon` function or the `continueFindCommon` function so that the `helpMove` function must fail or it is not called. If the `helpMove` function is not called, there is no replace operations. If the `helpMove` function returns false, by the ordering of $mflag$ and $unflag$ (Lemma 23 and Lemma 25), $op.iParent$ and $op.rParent$ are different. Moreover, the flag operation on the latter parent will fail such that $doCAS$ is false and no replace operation will perform. \square

In the next step, we show that how the algorithm could be ordered as a sequential execution. First we linearize the insert operation, the remove operation and the move operation that returns true, where $replace$ results in changing the key set. Let s_i be a set of *Leaf* nodes that are active in $snapshot_{T_i}$ after performing operations o_0, o_1, \dots, o_n , ordered by their linearization points $replace_0, replace_1, \dots, replace_n$ sequentially.

Observation 6. *If a Leaf node is moved, its op is set before the replace operation on $op.iParent$, which is before the replace operation on $op.rParent$.*

Lemma 41. *For the contain operation that returns true, there is a corresponding snapshot that l^k is active in $snapshot_{T_i}$. For the contain operation that returns false, there is a corresponding snapshot that l^k is inactive in $snapshot_{T_i}$.*

Proof. For the first part, we prove that in a snapshot, the end node l^k contains $keys^k$ and is not *moved*. Lemma 36 shows that there is a snapshot consists of $physical_path(keys^k)$ after calling, thus l^k that contains $keys^k$ is in $snapshot_{T_i}$ before judging whether the node is *moved*. We only have to discuss $mreplace$ in the next cases because other replace operations will not affect $l^k.op$. If $l^k.op$ is null at verifying, $mreplace$ must have not been done according to Observation 6. In this case, we could linearize it at $snapshot_{T_i}$. If $l^k.op$ is not null, but $mreplace$ occurs after verifying whether $l.op.iParent$ has $l.op.oldChild$, then it is also before the second $mreplace$ on $l.op.rParent$ (Observation 6). We could also linearize it $snapshot_{T_i}$.

For the second part, Lemma 36 shows that there is a snapshot consists $physical_path(keys^k)$. By Lemma 35, in every snapshot our quadtree's property maintains. Hence, if l^k does not contain $keys^k$, we could linearize it at $snapshot_{T_i}$. Or else, if l^k is in $physical_path$, but l^k is *moved* at verifying, we could linearize at $snapshot_{T_{i+1}}$ after $mreplace$. \square

Lemma 42. 1) *For the insert operation, it returns true when $key \notin s_{i-1}, key \in s_i$.*
 2) *For the remove operation, it returns true and $key \in s_{i-1}, key \notin s_i$.*
 3) *For the move operation, it returns true and $oldKey \in s_{i-1}, newKey \notin s_{i-1}, oldKey \notin s_i$, and $newKey \in s_i$.*

Proof. Lemma 39 shows that there is successful $replace$ occurs before the insert operation, the remove operation, and the move operation that returns true.

If a replace operation succeeds, $op.parent$ has been flagged with op other than *Compress* before $replace$ (Lemma 24 and Lemma 23). Lemma 29 implies that $op.parent$ is reachable from the root. Therefore, it is active. Besides, Corollary 31 shows that the path from the root to $op.parent$ is an active. Therefore a snapshot exists such that the path from the root to $op.oldChild$ is a $physical_path(key)$.

First we prove that $key \notin s_{i-1}$. By Lemma 41, in the snapshot l is inactive. Otherwise, replace operations will not execute. Hence, $key \notin s_{i-1}$.

Second, by the post-conditions of the `createNode` function 17, it creates a node or a sub-tree that contains $newKey$. According to Corollary 35, after $ireplace$ the node contains $newKey$ is inserted and quadtree's properties maintained. Hence, $key \in s_i$.

2) First we prove that $key \in s_i$. By Lemma 41, in the snapshot l is active. Otherwise, replace operations will not execute. Hence, $key \notin s_{i-1}$.

Second, the remove operation creates an *Empty* node. According to Corollary 35, after $rreplace$ the node contains $oldKey$ is removed so that $key \in s_i$.

3) First we prove that $oldKey \in s_{i-1}, newKey \notin s_{i-1}$. By Lemma 41, rl is active at $snapshot_{T_i}$, il is inactive at $snapshot_{T_{i+1}}, T_i < T_{i+1}$. We have to demonstrate that rl is still active at $snapshot_{T_{i+1}}$ so that we can use $snapshot_{T_{i+1}}$ as s_{i-1} .

We prove it by contradiction. Assuming that at $snapshot_{T_{i+1}}$ rl is inactive, hence there's some replace operation happens before T_{i+1} and after T_i . However, based on Lemma 23, $replace_i$ is the first successful replace operation that after reading rl . Therefore, it derives a contradiction. Hence we prove that in $snapshot_{T_{i+1}}$ rl is still active. \square

Finally we order linearization points for the insert operation, the remove operation and the move operation that returns false. We also order the contain operation. We consider they are linearized between $replace_{i-1}$ and $replace_i$ if exists.

Lemma 43. 1) *For the insert operation returns false, $key \in s_{i-1}$.*
 2) *For the remove operation returns false, $key \notin s_{i-1}$.*
 3) *For the move operation returns false, either $oldKey \notin s_{i-1}$ or $newKey \in s_{i-1}$.*

Proof. The first two parts are equivalent as the case in Lemma 41. \square

Lemma 44. *Our quadboost is a linearizable implementation.*

Proof. By lemma 42 and Lemma 43, our algorithm returns the same result as they are finished in the linearized order. Therefore we prove our algorithm is linearizable and our linearization points are correct. \square

.3 Progress Condition

We say an algorithm is non-blocking if the system as a whole is making progress even if some threads are starving. We prove our quadboost is non-blocking by following set of lemmas. We assume that there are finite number of *basic operations* invocations.

Observation 7. *There are finite number of basic operations.*

Lemma 45. *$path(key^k)$ returned by $find(keys)$ consists of finite number of keys.*

Proof. Observation 7 shows that the number of successful insert operation and move operation is limited. Lemma 40 illustrates that for *basic operations* return false, there is no successful replace operation. Hence, only successful insert and move operations add nodes into quadtree. The post-conditions of the createNode function (Lemma 17) and the effects of *replace* (Lemma 35) demonstrate that they will add finite number of nodes into quadtree. We then prove that $path^k$ is terminable. By Lemma 34, $find(keys)$ returns $path(key^k)$ which is a subpath of $physical_path(keys^k)$ in $snapshot_{T_i}$ which is terminable. Thus, there are finite number of nodes in $path^k$. \square

Corollary 6. *The compress function must terminate.*

Proof. By Lemma 45, $path^k$ is terminable. Since other functions called by the compress function do not consist loops, the compress function must terminate. \square

The we shall demonstrate that there are finite number of three different CAS transitions.

Lemma 46. *There is an unique spatial order among nodes in quadtree in $snapshot_{T_i}$*

Proof. As illustrated by the algorithm at line 223, the `getSpatialOrder` function compares ip with rp by the order: $x \rightarrow y \rightarrow w$. In our quadtree, we only consider square partitions on two-dimensional space. Hence, w is always equal to h . We prove the lemma by contradiction. Assuming there are two different *Internal* nodes with the same $\langle x, y, w \rangle$, they represent the same square starting with left corner $\langle x, y \rangle$ with width w . By Lemma 35, in every snapshot quadtree's properties remain. There cannot be two squares with the same left corner and w . Hence, in $snapshot_{T_i}$, a *Internal* node consists of a unique $\langle x, y, w \rangle$ tuple that can be ordered correctly. \square

Lemma 47. 1) *There are finite number of successful $flag \rightarrow replace \rightarrow unflag$ transitions.*

2) *There are finite number of successful $flag \rightarrow replace$ transitions.*

3) *There are finite number of successful $flag \rightarrow unflag$ transitions.*

Proof. 1) By Lemma 39, replace operations only occur in *basic operations* that return true. By Lemma 38, there's a unique *op* created for operations that return true. Hence, for the move operation, the insert operation and the remove operation, there are finite number of $flag \rightarrow replace \rightarrow circles$.

2) By Lemma 39, replace operations only occur in *basic operations* that return true. Lemma 24 shows

that the $flag \rightarrow replace$ transition happens only in the `compress` function which is called by the successful remove operation or the move operation. Hence, there are a finite number of $flag \rightarrow replace$ transitions, as Corollary 6 shows that the `compress` function is terminable.

3) The $flag \rightarrow unflag$ transition executes only in the `helpMove` function where $op.iParent$ cannot be flagged according to Lemma 25 (if $op.iFirst$ is true). The case that $op.iFirst$ is false can be proved symmetrically.

From first two parts of the lemma, there are finite number of replace operations that change quadtree's structure. If the $flag \rightarrow replace \rightarrow unflag$ transition happens simultaneously, it will reset a node's *op* to *Clean*. If the $flag \rightarrow replace \rightarrow unflag$ transition happens in the meantime. Lemma 33 indicates that a node with *Compress op* will not be pushed twice. Consider if the `helpMove` function detects $op.iParent.op$ as *Compress*, then $op.iParent$ is not reachable in the next time. Hence, quadtree is stabilized and only when $flag \rightarrow unflag$ transitions execute infinitely and never set `doCAS` as true.

According to Lemma 46, we order *spatial order* on both $op.iParent$ and $op.rParent$ as $ip_0 > ip_1 \dots > ip_n$ and $rp_0 > rp_1 \dots > rp_n$. We assume that there's a ring such that $move_i$ which flags rp_i but fails on ip_i , and $move'_i$ which flags ip_i but fails on rp_i . By assumption we order rp_i and ip_i as $rp_i > ip_i$ by $move_i$'s order, and $ip_i > rp_i$ by $move'_i$'s order. But by Lemma 46, all *Internal* nodes in a snapshot can be ordered uniquely. Therefore it derives a contradiction.

Then, we prove that there's no ring among all $move_i$ in the same snapshot, so at least one of $move_i$ could set `doCAS` to true, resulting in the $flag \rightarrow replace \rightarrow unflag$ transition. Let's consider the dependency among all $move_i$. If $move_i$ which fails on p_i has been flagged by $move'_i$, then there is a directed edge from $move_i$ to $move'_i$ by order. In this way, the head node in the graph must have no out edge as proved. It will set `doCAS` to true, all move operations that are directly connected to it will help it finish. After erase the former head node from the graph, there will be other nodes that have no out edge. Finally, all dependency edges are removed. Therefore, there are a finite number of $flag \rightarrow unflag$ transitions. \square

Observation 8. *The help function, the helpCompress function, the helpMove function, and the helpSubstitute function are not called in a mutual way. (If method A calls method B, and method B calls method A reciprocally, we say method A and B are called in a mutual way.)*

Lemma 48. *If help function returned at T_i , and $find(keys)$ at the prior iteration reads $p^0.op$ at $T_{i1} < T_i$, Leaf nodes in $snapshot_{T_i}$ and $snapshot_{T_{i1}}$ are different.*

Proof. We prove the lemma by contradiction, there are two

possible scenarios. First, the help function might be interminable. But based on Observation 8, the help function must terminate. Second, the help function might execute without changing quadtree. According to Lemma 47, there are finite number of CAS transitions. So if some help function does not change quadtree, the structure might be changed before T_{i+1} . For the move function, if ip and rp are the same but their ops are different, the help function might not change the structure. Nevertheless, $snapshot_{T_i}$ and $snapshot_{T_{i+1}}$ are different. Or else, there are some interval that all transitions are $flag \rightarrow unflag$. However, Lemma 47 also shows that all transitions form a acyclic graph that there is no interval that all transitions are $flag \rightarrow unflag$. Therefore, a $flag \rightarrow replace \rightarrow unflag$ transition or a $flag \rightarrow unflag$ transition is executed to change the snapshot. \square

Lemma 49. *Our quadboost algorithms are non-blocking.*

Proof. We have to prove that no process will execute loops infinitely without changing keys in quadtree. First, we prove that $path$ is terminable. Next, we prove that $find(keys^k)$ starts from an active node in $physical_path(keys^k)$ in $snapshot_{T_i}$ between i_{th} iteration and $i + 1_{th}$ iteration. Finally, $Leaf$ nodes in $snapshot_{T_{i+1}}$ at the returning of $find(keys)$ at i_{th} iteration is different from $snapshot_{T_{i+2}}$ that at the returning of $find(keys)$ at the $i + 1_{th}$ iteration.

For the first part, initially we start from the root node. Therefore, $path$ is empty. Moreover, as Lemma 45 shows that $path^k$ consists of finite number of keys, we establish this part.

For the second part, the `continueFind` function and the `continueFindCommon` function pop all nodes with a `Compress op` from $path$. For the insert operation and the remove operation, since Lemma 29 shows that an `Internal` nodes whose op is not `Compress` is active, and Lemma 31 shows that nodes above the active node are also active, there is a snapshot in which the top node of $path$ is still in $physical_path(keys^k)$. For the move operation, if either $rFail$ or $iFail$ is true, it is equivalent with the prior case. If $cFail$ is true, Lemma 32 illustrates that if the LCA node is active, it is in $physical_path$ for both $oldKey$ and $newKey$. Thus, there is also a snapshot in which the start node is in $physical_path$.

We prove the third part by contradiction, assuming that quadtree is stabilized at T_i , and all invocations after T_i are looping infinitely without changing `Leaf` nodes.

For the insert and remove operations, before the invocation of $find(keys)$ at the next iteration, they must execute the help function at line 118 and line 175 accordingly. In both cases, the help function changes the snapshot (Lemma 48).

For the move operation, consider different situations of the `continueFindCommon` function. If $rFail$ or $iFail$ is true, two situations arise: (1) iOp or rOp is `Clean` but $mflag$ fails. (2) iOp or rOp is not `Clean`. Consider the first case that $mflag$ fails, iOp and rOp are updated respectively at line 236 and line 235. If ip and rp are different, before its invocation of $find(keys^k)$, the help function is performed at line 311 and line 287. Thus by Lemma 48, the snapshot is changed between two iterations. Now consider if $cFail$ is true. If ip and rp are the same, it could result from the difference between rOp and iOp . In this case, the snapshot might be changed between reading rOp and iOp . It could

also result from the failure of $mflag$. For above cases, the help function at line 332 would change quadtree. If ip and rp are different, it results from that either $iPath$ or $rPath$ has popped the LCA node. We have proved the case in the former paragraph. Thus, we derives a contradiction.

From above discussions, we prove that quadboost is non-blocking. \square