Non-blocking Patricia tries with replace operations

Niloufar Shafiei

Received: 29 November 2016 / Accepted: 13 January 2019 / Published online: 4 February 2019
© Springer-Verlag GmbH Germany, part of Springer Nature 2019

Abstract
This paper presents a non-blocking Patricia trie implementation for an asynchronous shared-memory system using Compare&Swap. The trie is a linearizable implementation of a set and supports three update operations: insert adds an element to the trie, delete removes an element from the trie and replace replaces one element by another. The replace operation is interesting because it changes two different locations of a trie. We design a mechanism that allows the two changes of a replace operation to appear to be executed atomically. If all update operations modify different parts of the trie, they run completely concurrently. The implementation also supports a wait-free find operation, which only reads shared memory and never changes the data structure. Our implementation and its correctness proof are modular and can be adapted for other data structures. Empirically, we compare our algorithms to some existing tree-based set implementations and our results show that our trie performs consistently well in different scenarios.

Keywords Patricia trie · Non-blocking · Shared memory · Lock-free · Concurrent data structure · Dictionary · Set

1 Introduction
A Patricia trie [25] is a tree that stores a set of keys, which are encoded as \(\ell\)-bit binary strings (see Fig. 1). The trie is structured so that the path from the root to a key is determined by the sequence of characters in the key. So, the length of this path is at most the length of the key (and will often be shorter). Thus, if key strings are short, the height of the trie remains small without requiring any complicated balancing. The simplicity of the data structure makes it a good candidate for concurrent implementations. Patricia tries are widely used in practice. They have applications in routing systems [20], data mining [29], machine learning [18], bioinformatics [5], etc. Allowing concurrent access is essential in some applications and can boost efficiency in multicore systems.

In a Patricia trie, each internal node has exactly two children (see Fig. 1). The elements of the set are stored as labels of the leaves of the trie. Each internal node has a label that is the longest common prefix of its children’s labels. If a node’s label has length \(k - 1\), then the \(k\)th bit of the node’s left and right children’s labels are 0 and 1, respectively. Since keys are \(\ell\)-bit binary strings, the height of the trie is at most \(\ell\).

We present a new concurrent implementation of Patricia tries using single-word Compare&Swap (CAS). The operations on the trie are linearizable, meaning they appear to take place atomically [23]. They are also non-blocking (lock-free): some process completes its operation in a finite number of steps even if other processes fail. Wait-free algorithms satisfy the stronger guarantee that every process completes its operation in a finite number of steps.

Our implementation provides wait-free find operations and non-blocking insertions and deletions. We also provide a non-blocking replace operation that makes two changes to the trie that appear to take place atomically: it deletes one key and inserts another. If all pending updates are at disjoint parts of the trie, they do not interfere with one another.

A Patricia trie can be used to store a set of points in \(\mathbb{R}^d\). For example, a point in \(\mathbb{R}^2\) whose coordinates are \((x, y)\) can be represented by a key formed by interleaving the bits of \(x\) and \(y\) (this yields a data structure very similar to a quadtree). Then, the replace operation can be used to move a point from one location to another atomically. This operation has applications in Geographic Information Systems [19]. The trie can store the locations of the moving objects. When an object moves from an old location \((x, y)\) to a new location \((x', y')\), the replace operation can atomically remove \((x, y)\) and add \((x', y')\).
from the trie and adds \((x', y')\) to the trie. The replace operation would also be useful if the Patricia trie were adapted to implement a priority queue, so that one could change the priority of an element in the queue.

Search trees are another class of data structures that are commonly used to represent sets. When keys are not uniformly distributed, balanced search trees generally outperform unbalanced ones. The reverse is often true when keys are uniformly distributed due to the simplicity of unbalanced search trees. Our empirical results show that the performance of our trie is consistently good in both scenarios. This is because our trie implementation is as simple as an unbalanced search tree but also keeps trees short. For simplicity, we rely on a garbage collector (such as the one provided in Java implementations) that deallocates objects when they are no longer accessible. We assume the garbage collector is safe: if a thread can access an object by following a pointer (or a chain of pointers), the garbage collector does not recycle the object.

For our Patricia trie algorithms, we extend the scheme used in [15] for binary search trees to coordinate processes. Thus, we show that the scheme is more widely applicable. In particular, we extend the scheme so that it can handle update operations that make more than one change to the tree structure. When a process \(p\) performs an update, it first creates a descriptor object that contains enough information about the update, so that other processes can help complete the update by reading the descriptor object. As in [15], before \(p\) changes the tree, it flags a small number of nodes to avoid interference with other concurrent updates. A node is flagged by setting a pointer in the node to point to the update’s descriptor object using a CAS. The CAS fails if the node is already flagged by another update; in this case, \(p\) helps the other update before retrying its own update (this helping mechanism is used to guarantee the non-blocking property). When an update is complete, nodes that are still in the tree are unflagged by removing the pointers to the update’s descriptor object. Searches do not need to check for flags and can therefore traverse the tree very efficiently simply by reading child pointers. Searches in our Patricia trie are wait-free, unlike the searches in [15], because the length of a search path in a Patricia trie is bounded by the length of the key.

There are several novel features of this work. In our implementation, we design one fairly simple routine that is called to perform the real work of all update operations. In contrast, insert and delete operations in [15] are handled by totally separate routines. This makes our proof of correctness more modular than the proof of [15]. It also makes it easy to add new types of update operations to our implementation. Our techniques and correctness proof can be generalized to other tree-based data structures.

In [15], modifications were made only at the bottom of the search tree. Our new Patricia trie implementation also copes with modifications that can occur anywhere in the trie. This requires proving that changes in the middle of the trie do not cause concurrent search operations passing through the modified nodes to go down the wrong branch. Howley and Jones [24] introduced changes in the middle of a search tree but only to keys stored in internal nodes, not the structure of the tree itself.

In [15], atomic changes had to be done by changing a single pointer. Cederman and Tsigas [11] proposed a non-blocking replace operation for a tree-based data structure, but they require double-CAS, which modifies two non-adjacent locations. Our replace operation makes two changes to the trie appear atomic, but uses only single-word CAS. The insertion and then the deletion of the replace operation are performed by two CAS steps on child pointers. To ensure that both of these changes appear as a single atomic operation on the trie, we design the implementation so that all operations can observe both changes as soon as the first of these two CAS steps has occurred. Prior to that first CAS step that inserts the new key, the replace operation flags the leaf to be removed by storing a pointer to the replace operation’s descriptor object in that leaf. That flagged leaf becomes logically removed when the insertion CAS step occurs: any operation that reaches the leaf between the two CAS steps determines that the leaf is already removed from the trie using the information stored in the replace operation’s descriptor object. This new scheme can be generalized to make several changes to the trie atomically by making all changes visible at a single linearization point.

To summarize our contributions:

- We present a non-blocking linearizable Patricia trie.
- Searches in our Patricia trie are wait-free.
- We employ one routine to implement the real work of any update operation.
- We present a non-blocking update operation that changes two child pointers using single-word CAS.
2 Related work

Most concurrent data structures are lock-based. However, locks may cause priority inversion: A high-priority process might have to wait for a low-priority process to release a lock. Locks may cause convoys: suppose two or more processes repeatedly attempt to obtain the same sequence of locks. While one process acquires a lock, others wait to acquire the same lock. Each time a process attempts to acquire the lock and fails, it has to do a context switch and does not use the rest of its scheduling quantum. In non-blocking algorithms, operations do not wait for one another. If an operation is blocked by another operation, it first helps the operation to complete and restarts. So, non-blocking algorithms do not cause priority-inversion and convoys.

Two state of the art examples of lock-based implementations of set data structures are the AVL tree by Bronson et al. [7], which maintains an approximately balanced tree, and the self-adjusting binary search tree by Afek et al. [1], which moves frequently accessed nodes closer to the root. Aref and Ilyas [3] described how lock-based implementations could be designed for a class of space-partitioning trees that includes Patricia tries. Lock-coupling can also be applied to implement a concurrent Patricia trie [37].

Here, we focus on non-blocking algorithms, which do not use locks. There are two general techniques for obtaining non-blocking data structures: universal constructions (see the related work section of [14] for a recent survey of work on this) and transactional memory [34] (see [21] for a survey). Such general techniques are usually not as efficient as algorithms that are designed for specific data structures.

Tsai and Li [36] gave a general wait-free construction for tree-based data structures. To access a node, a process makes a local copy of the path from the root to the node, performs computations on the local copy, and then atomically replaces the entire path by its local copy. Since this approach copies many nodes and causes high contention at the root, their approach is not very efficient. Barnes [4] presented another general technique to obtain non-blocking implementations of data structures in which processes cooperate to complete operations.

Ellen et al. [15] presented a non-blocking binary search tree data structure from CAS operations. Their approach has some similarity to the cooperative technique of [4]. In [15], modifications were only made at the leaves of the search tree. As discussed in Sect. 1, our Patricia trie implementation extends the approach used in [15]. Our new Patricia trie implementation also copies with modifications that can occur anywhere in the trie. This requires proving that changes in the middle of the trie do not cause concurrent search operations passing through the modified nodes to go down the wrong branch. The main difference between our implementation and [15] is that our replace operation applies two changes to the data structure that appear to take place atomically. This requires us to treat some keys that are physically in the tree as logically removed.

Brown and Helga [10] generalized the binary search trees of [15] to non-blocking k-ary search trees and compared the non-blocking search trees with the lock-based search tree of Bronson et al. [7] empirically. Brown et al. [8,9] also extended the approach used in [15] to propose a new set of primitive operations to implement non-blocking data structures, and then used these primitives to implement a class of non-blocking trees [9]. Their approach simplifies designing non-blocking data structures but can only handle one change to the data structure atomically. Since our replace operation must change two different parts of the tree and the set of primitive operations in [8] can apply a conditional update only to a single memory location, the set of primitive operations cannot be used directly to implement our replace operation. However, extending the approach in [8,9] to handle two changes to a tree atomically is an interesting direction for future work. Our technique is the first one that extends the approach used in [15] to handle multiple changes to a data structure atomically. This makes generalizing our approach in the manner of [8,9] harder, although we believe our approach can be applied to many other data structures.

Howley and Jones [24] presented a non-blocking search tree from CAS operations using a cooperative technique similar to [15]. Their tree stores keys in both leaf and internal nodes. However, search operations sometimes help update operations by performing CAS operations. Braginsky and Petrak [6] proposed a non-blocking balanced B+tree from CAS operations.

Prokop et al. [30] described a non-blocking hash trie that uses CAS operations. Their approach is very different from our implementation. Unlike Patricia tries, in their trie implementation, an internal node might have only one child.
In their implementation, nodes have up to $2^k$ children (where $k$ is a parameter) and extra intermediate nodes are inserted between the actual nodes of the trie. With $k = 5$, the height of their trie is very small, making their implementation very fast when contention is low. However, our experiments suggest that it is not very scalable under high contention. Unlike our implementation, their search operation may perform CAS steps.

Oshman and Shavit [28] sketched a non-blocking trie implementation that also maintains a doubly-linked-list and skip list at the bottom of the trie. Their implementation supports insertions and deletions. The focus of their paper is the analysis of the amortized step complexity of the data structure. In their implementation, they used CAS and Double-Compare&Single-Swap (DCSS) primitive. DCSS($X$, $x_{old}$, $x_{new}$, $Y$, $y_{old}$) atomically changes $X$ to $x_{new}$ if $X$ is equal to $x_{old}$ and $Y$ is equal to $y_{old}$. Otherwise, DCSS fails. Unlike CAS, DCSS is not supported as a hardware primitive.

Natarajan and Mittal [26] presented a new non-blocking binary search tree. In their implementation, an insertion is performed by only one CAS step and does not require helping. A delete operation first flags edges between nodes and then removes the leaf node. Other process might help to perform a deletion and multiple deletions that interfere with each other might be combined and performed by changing only one edge. There are also other non-blocking implementations of search trees that have been proposed in recent years [12,13,27,31].

Non-blocking implementations of set data structures have also been proposed based on skip lists using CAS operations [16,17,35]. However, these do not provide a replace operation. A non-blocking skip list (ConcurrentSkipListMap) has been implemented in the Java class library by Doug Lea.

### 3 Algorithm description

We first give the sequential specification of the operations. The trie stores a set $D$ of keys from a finite universe $U$. If $v \notin D$, insert$(v)$ changes $D$ to $D \cup \{v\}$ and returns true; otherwise, it returns false. If $v \in D$, delete$(v)$ changes $D$ to $D \setminus \{v\}$ and returns true; otherwise, it returns false. If $v \in D$ and $v' \notin D$, replace$(v, v')$ changes $D$ to $D \setminus \{v\} \cup \{v'\}$ and returns true; otherwise, it returns false. If $v \in D$, find$(v)$ returns true; otherwise, it returns false. In either case, find$(v)$ does not change $D$. We assume keys are encoded as $\ell$-bit binary strings (in Sect. 6, we describe how to handle keys with unbounded length).

We assume an asynchronous shared-memory system with single-word CAS operations. The correctness of algorithms using CAS often depends on the fact that, if the CAS succeeds, the value has not been changed since the preceding read. So, the ABA problem can sometimes arise when using CAS: Suppose the value of a variable $x$ might be changed from the expected value $A$ to another value $B$, and then set back to the expected value $A$ just before the CAS occurs. Then, a CAS can succeed to change $x$ from $A$ to $C$ when it is not supposed to. In our implementation, we avoid this scenario (We later explain in detail how).

### 3.1 Data structures

We now describe the objects that are used in the implementation (Fig. 2). The Patricia trie is represented using Leaf and Internal objects which are subtypes of Node objects. Each Node object has a label field, which is never changed after initialization and stores the node’s label. An Internal object has an array child of two Node objects that stores pointers to the children of the node.

For simplicity, our trie initially contains just two Leaf Nodes with labels $0^\ell$ and $1^\ell$ and a root node whose label is the empty string (see Fig. 3). We assume the keys $0^\ell$ and $1^\ell$ cannot be elements of $D$. This ensures that the trie always has at least two Leaf Nodes and avoids special cases of update operations that would occur if the root were a Leaf. This assumption simplifies the code and the correctness proof.

![Initialization](Fig. 3) Initialization of the Patricia trie

```plaintext
1. Leaf: (subtype of Node) descriptor of update
2. label: String
3. info: Info
4. Internal: (subtype of Node) descriptor of update
5. label: String
6. child: Node[2] left and right child
7. info: Info
8. Flag: (subtype of Info) descriptor of update
9. flag: Internal[4] nodes to be flagged
10. old: Info[4] expected values of CASs that flag
11. unflag: Internal[2] nodes to be unflagged
12. par: Internal[2] nodes with children to be changed
13. old: Node[2] expected children of nodes par
14. new: Node[2] children of par to be changed to
15. rmvLeaf: Leaf Leaf to be flagged
16. flagDone: Boolean set to true if flagging succeeds
17. Unflag: (subtype of Info) has no fields
```

![Data types used in the implementation](Fig. 2) Data types used in the implementation

- **Initialization**
  1. lChild ← new Leaf($0^\ell$, new Unflag())
  2. rChild ← new Leaf($1^\ell$, new Unflag())
  3. Root ← new Internal($\varepsilon$, [lChild, rChild], new Unflag())
Each Node object also has an info field that stores a pointer to an Info object that serves as the descriptor of the update operation that is in progress at the node (if any). An Info object contains enough information to allow other processes to help the update to complete. Info objects have two subtypes: Flag and Unflag. An Unflag object is used to indicate that no update is in progress at a node. Initially, the info field of each Node object is an Unflag object. When the info field of a Node is changed, the Node’s info field is updated to point to a newly created Unflag or Flag object. Hence, the ABA problem on the info fields is avoided (using a safe garbage collector). We say a node is flagged or unflagged, depending on whether its info field stores a Flag or Unflag object. The info and child fields of an Internal Node are changed using CAS steps. However, a Leaf Node gets flagged by writing a Flag object into its info field.

Since a Flag object describes how to perform an update operation, before describing the fields of a Flag object, we briefly describe the steps of an update operation. To perform an update operation, first some Internal Nodes get flagged, then some child fields of flagged Internal Nodes are changed and then nodes that are still in the trie get unflagged. The nodes that must be flagged to perform an update operation are the Internal Nodes whose child fields will be changed by the update or that will be removed from the trie by the update. Flagging nodes is similar to locking nodes: it avoids having other operations change the part of the trie that would be changed by the update.

A Flag object has a number of fields. The flag field stores a pointer to an Info object that serves as the descriptor of the update that node will be affected by the update before reading that node’s child field. This value of the info field is stored in the Flag’s old field, and is used as the expected value by the CAS that flags the node. This ensures that if the node is successfully flagged, the node’s child field has not changed since the node’s children were read. The boolean flagDone field is set to true when the flagging for the update has been completed. The flagDone field is the only field of an Info object that is not immutable. In the case of a replace operation, the rmvLeaf field points to the Leaf to be removed by the update after flagging is complete. The actual changes to be made to the trie are described in three more array fields of the Flag object: par, old, and new. For each i, the update should CAS the appropriate child pointer of par[i] from old[i] to new[i].

As we shall see, once all nodes are successfully flagged, the CAS on each child pointer will be guaranteed to succeed because that pointer cannot have changed since the old value was read from it. Thus, like locks, the info field of a node is used to give an operation exclusive permission to change the child field of that node.

### 3.2 Update operations

The implementation has three update operations: **insert**, **delete** and **replace**. All three have the same overall structure. The pseudo-code for our implementation is given in Figs. 4, 5, 6 and 7. An update op uses the search routine to find the location(s) in the trie to be changed. It then creates a new Flag object I containing all the information required to complete the update by calling newFlag. If newFlag sees that some node that must be flagged is already flagged with a different Flag I’, it calls help(I’) at line 112 to try complet-

```plaintext
21. insert(v : U)
22. while(true)
23. I ← null
24. (p, node, pI, keyInTrie) ← search(v)  # search for a location to insert key
25. if keyInTrie then return false
26. nodeI ← node.info
27. copy ← new copy of node  # create new copy of node located at where the new node should be inserted
28. new ← createNode(copy, new Leaf containing v, nodeI)  # if no interference detected, create new internal node new
29. if new ≠ null then
30. if node is Internal then
31. I ← newFlag(flag ← [p, node], oldI ← [pI, nodeI], unflag ← [p], par ← [p], old ← [node], new ← [new], rmvLeaf ← null)  # if node new is successfully created
32. else I ← newFlag(flag ← [p], oldI ← [pI], unflag ← [p], par ← [p], old ← [node], new ← [new], rmvLeaf ← null)  # if no interference detected, create insertion’s descriptor
33. if I ≠ null and help(I) then return true  # if no interference detected, try to perform the insertion

34. delete(v : U)
35. while(true)
36. I ← null
37. (gp, p, node, gpI, pl, keyInTrie) ← search(v)  # search for key v to be deleted
38. if keyInTrie then return false  # v is not in the trie
39. sibling ← p.child[1 - (p.label + 1)th bit of v]  # read the sibling of the Node with key v
40. if gp ≠ null then
41. I ← newFlag(flag ← [gp, p], oldI ← [gpI, pl], unflag ← [gp], par ← [gp], old ← [p], new ← [sibling], rmvLeaf ← null)  # if no interference detected, try to perform the deletion
42. if I ≠ null and help(I) then return true
```

**Fig. 4** The insert and delete operations
ing the update described by \( I \)', and then \( op \) retries its update from scratch. Otherwise, \( op \) calls \( \text{help}(I) \) to try to complete its own update.

As mentioned earlier, flagging nodes ensures exclusive access for changing \( \text{child} \) pointers. Thus, an update operation flags the nodes whose \( \text{child} \) pointers it wishes to change and permanently flags any Internal node that is removed from the trie to ensure update operations are not applied to a deleted portion of the trie.

Unlike locks, the Flag objects store enough information, so that if a process performing an operation crashes while nodes are flagged for the operation, other processes can attempt to complete the operation and remove the flags. This ensures that a failed operation cannot prevent others from progressing. To avoid deadlock, if an update must flag more than one Internal Node, we order the Internal Nodes by their labels.

The \( \text{help}(I) \) routine carries out the real work of an update operation using the information stored in the Flag object \( I \).

In the following discussion, we identify certain key steps of the \( \text{help} \) routine: an execution of line 92 is called a \( \text{flag} \) CAS, line 100 is called a \( \text{child} \) CAS, line 103 is called an \( \text{unflag} \) CAS and line 107 is called a \( \text{backtrack} \) CAS. The \( \text{help} \) routine first uses flag CAS steps (line 92) to flag some nodes by setting their \( \text{info} \) fields to \( I \). If all nodes are flagged successfully, \( \text{help}(I) \) uses child CAS steps (line 100) to change the \( \text{child} \) fields of the Internal Nodes in \( I, \text{par} \) to perform the update. A successful child CAS changes the \( \text{child} \) pointer of an Internal Node to execute the physical change to the trie. Then, \( \text{help}(I) \) uses unflag CAS steps (line 103) to unflag nodes that were flagged earlier, except those that were removed from the trie, by setting their \( \text{info} \) fields to new Unflag objects. In this case, any nodes deleted by the update remain flagged forever. If any node is not flagged successfully, the attempt to perform the update has failed and backtrack CAS steps (line 107) are used to unflag any nodes that were already flagged.

If any child CAS step is executed inside \( \text{help}(I) \), the update operation is successful and it is linearized at the first such child CAS. If a \( \text{replace} \) operation performs two different child CAS steps, it first executes a child CAS to insert the new key \( v_i \), and then a child CAS to delete the old key \( v_d \). In this case, the \( \text{replace} \) operation also flags the Leaf Node, \( n_d \).
73. \textbf{find}(v : U) \hspace{1cm} \triangleright \text{search for key } v \text{ in the trie}
74.  \{ \sim \text{-}, \sim \text{-}, \sim \text{-} \} \text{keyInTrie} \leftarrow \textbf{search}(v) \hspace{1cm} \triangleright \text{return } \text{keyInTrie}
75.  return \text{keyInTrie}
76. \textbf{search}(v : U) \hspace{1cm} \triangleright \text{start from root}
77.  \langle p, pl \rangle \leftarrow (\text{null, null}) \hspace{1cm} \triangleright \text{v might be in subtree whose root is node}
78.  \text{node} \leftarrow \text{root} \hspace{1cm} \triangleright \text{go down the trie}
79.  \text{while } (\text{node} \text{ is Internal and } \text{node.label} \text{ is prefix of } v) \hspace{1cm} \triangleright \text{detect if Leaf is already replaced by another key}
80.  \langle gp, gpI \rangle \leftarrow (p, pl) \hspace{1cm} \triangleright \text{try to flag nodes in }
81.  \langle p, pl \rangle \leftarrow (\text{node.node.info}) \hspace{1cm} \triangleright \text{help an update to complete}
82.  \text{node} \leftarrow p.\text{child}[(p.label) + 1] \text{th bit of } v \hspace{1cm} \triangleright \text{return } \text{node.label} = v \text{ and rmvd } = \text{false}
83.  \text{if node is Leaf then} \hspace{1cm} \triangleright \text{if all Nodes in I.flag are successfully flagged}
84.  \text{rmvd } \leftarrow \textbf{logicallyRemoved}(\text{node.info}) \hspace{1cm} \triangleright \text{flag } \text{CAS}
85.  \text{keyInTrie } \leftarrow (\text{node.label} = v \text{ and rmvd } = \text{false}) \hspace{1cm} \triangleright \text{flag the Leaf Node}
86.  \text{else } \text{keyInTrie } \leftarrow \text{false} \hspace{1cm} \triangleright \text{update the trie}
87.  \text{return } (gp, p, \text{node}, gpI, pI, \text{keyInTrie}) \hspace{1cm} \triangleright \text{if update is completed}
88. \textbf{help}(I: \text{Flag}) \hspace{1cm} \triangleright \text{if update is completed}
89.  i \leftarrow 0 \hspace{1cm} \triangleright \text{if update is completed}
90.  \text{doChildCAS } \leftarrow \text{true} \hspace{1cm} \triangleright \text{try to flag nodes in I.flag}
91.  \text{while } (i < |I.flag| \text{ and doChildCAS}) \hspace{1cm} \triangleright \text{try to flag nodes in I.flag}
92.  \text{CAS}(I.\text{flag}[i].\text{info}, I.\text{old}[i], I) \hspace{1cm} \triangleright \text{try to flag nodes in I.flag}
93.  \text{doChildCAS } \leftarrow (I.\text{flag}[i].\text{info} = I) \hspace{1cm} \triangleright \text{flag } \text{CAS}
94.  i \leftarrow i + 1 \hspace{1cm} \triangleright \text{if all Nodes in I.flag are successfully flagged}
95.  \text{if doChildCAS then} \hspace{1cm} \triangleright \text{flag the Leaf Node}
96.  I.\text{flagDone } \leftarrow \text{true} \hspace{1cm} \triangleright \text{flag the Leaf Node}
97.  \text{if } I.\text{rmvLeaf} \neq \text{null then } I.\text{rmvLeaf.info } \leftarrow I \hspace{1cm} \triangleright \text{update the trie}
98.  \text{for } i \leftarrow 0 \text{ to } ((|I.par| - 1)) \hspace{1cm} \triangleright \text{child } \text{CAS}
99.  k \leftarrow (I.\text{par}[i].\text{label}) + 1 \text{th bit of } I.\text{new}[i].\text{label} \hspace{1cm} \triangleright \text{unflag } \text{CAS}
100. \text{CAS}(I.\text{par}[i].\text{child}[k], I.\text{old}[i], I.\text{new}[i]) \hspace{1cm} \triangleright \text{unflag } \text{CAS}
101. \text{if I.\text{flagDone} then} \hspace{1cm} \triangleright \text{if the update failed}
102.  \text{for } i \leftarrow (I.\text{unflag} - 1) \text{ down to } 0 \hspace{1cm} \triangleright \text{backtrack } \text{CAS}
103.  \text{CAS}(I.\text{unflag}[i].\text{info}, I, \text{new Unflag}) \hspace{1cm} \triangleright \text{backtrack } \text{CAS}
104.  \text{return } \text{true} \hspace{1cm} \triangleright \text{return } \text{true}
105.  \text{else} \hspace{1cm} \triangleright \text{return } \text{true}
106.  \text{for } i \leftarrow (I.\text{flag} - 1) \text{ down to } 0 \hspace{1cm} \triangleright \text{return } \text{false}
107.  \text{CAS}(I.\text{flag}[i].\text{info}, I, \text{new Unflag}) \hspace{1cm} \triangleright \text{return } \text{false}
108.  \text{return } \text{false} \hspace{1cm} \triangleright \text{return } \text{false}

109. \textbf{newFlag}(flag: \text{array of Internal, oldI: array of Info, unflag: array of Internal, par: array of Internal, old: array of Node, new: array of Node, rmvLeaf: Leaf}) \hspace{1cm} \triangleright \text{create the update's descriptor}
110.  \text{for } i \leftarrow 0 \text{ to } (|oldI| - 1), \hspace{1cm} \triangleright \text{detect if other updates in progress}
111.  \text{if oldI[i] is Flag then} \hspace{1cm} \triangleright \text{detect if other updates in progress}
112.  \textbf{help}(\text{oldI}[i]) \hspace{1cm} \triangleright \text{help an update to complete}
113.  \text{return } \text{null} \hspace{1cm} \triangleright \text{help an update to complete}
114.  \text{if flag has duplicates with different values in oldI then return null} \hspace{1cm} \triangleright \text{retry my update}
115.  \text{else remove duplicates in flag and unflag (and corresponding entries of oldI)} \hspace{1cm} \triangleright \text{retry my update}
116.  \text{sort elements of flag and permute elements of oldI accordingly} \hspace{1cm} \triangleright \text{retry my update}
117.  \text{return } \text{new Info(flag, oldI, unflag, par, old, new, rmvLeaf, false)} \hspace{1cm} \triangleright \text{create the update's descriptor}

\textbf{Fig. 6} The find operation and search, help and newFlag subroutine

118. \textbf{createNode}(node1: Node, node2: Node, info: Info) \hspace{1cm} \triangleright \text{create the update's descriptor}
119.  \text{if node1.label is prefix of node2.label or node2.label is prefix of node1.label then} \hspace{1cm} \triangleright \text{create a new Internal Node}
120.  \text{if info is Flag then help(info)} \hspace{1cm} \triangleright \text{create a new Internal Node}
121.  \text{return } \text{null} \hspace{1cm} \triangleright \text{create a new Internal Node}
122.  \text{else return new Internal whose children are node1 and node2} \hspace{1cm} \triangleright \text{create a new Internal Node}

123. \textbf{logicallyRemoved}(I: Info) \hspace{1cm} \triangleright \text{create the replace described by } I \text{ is linearized}
124.  \text{if } I \text{ is Unflag then return } \text{false} \hspace{1cm} \triangleright \text{create the replace described by } I \text{ is linearized}
125.  \text{return } (I.\text{old}[0] \text{ not in } I.\text{par}[0].\text{child})
of the old key \( v_d \) before the first child CAS step. We say the Leaf is logically removed from the trie at all times after the first child CAS step occurs. When this first child CAS occurs, \( n_d \) is flagged for the operation, so \( n_d.info \) contains the Flag object \( I \) that describes the replace operation, and the field \( n_d.info \) will never change thereafter. The first child CAS of the replace operation changes a child pointer of \( I.par[0] \) from \( I.old[0] \) to another node. We prove below that child fields never experience an ABA problem, so \( I.old[0] \) will never become a child of \( I.par[0] \) again afterwards. Thus the Leaf \( n_d \) is logically removed if and only if (for some Flag object \( I \)), \( n_d.info = I \) and \( I.old[0] \) is not a child of \( I.par[0] \). Any other operation that reaches the Leaf \( n_d \) before it is physically removed from the tree (by the second Child CAS of the replace operation) tests at line 84 whether \( n_d \) is logically removed using subroutine logicallyRemoved. Because this test examines several memory locations and therefore cannot be carried out atomically, it requires some proof (in Sect. 4.3) that it accurately tests whether a Leaf is logically removed at some time during the search that calls it. Any operation that finds a Leaf that has been logically removed behaves as though the key contained in the Leaf has already been removed. This allows the replace operation to be linearized at the first child CAS, which inserts the new key \( v_1 \) and logically removes the old key \( v_d \).

Note only for the replace operation that is performed by two child CAS steps, a Leaf Node is first logically removed from the trie and then physically (by the second child CAS step of the replace). For other update operations that are performed by one child CAS, Leaf Nodes are only physically removed from the trie (by the only child CAS step of the update). The decision of whether a Leaf Node is logically removed from the trie is done by reading the info of the Leaf Node. If it is of type Unflag, it is not logically removed indicating the Leaf Node is in the trie (line 124). E.g., new Leaf Nodes added to the trie have Unflag as their info, indicating they are in the trie. If the info is Flag, then the Node is logically removed if the Node at info.old[0] is not a child of info.par[0] (line 125).

We say a node is reachable at time \( T \) if there is a path from the root to the node at time \( T \). We say a Leaf Node is logically in the trie at time \( T \) if the node is reachable and not logically removed at time \( T \). The leaves that are logically in the trie contain exactly the set of keys in the set \( D \).

Whenever a child pointer is changed, the old child is permanently flagged and it is removed from the trie to avoid the ABA problem (in some cases, this requires the update to add a new copy of the old child to the trie). When a call to help(\( I \)) performs a child CAS on \( I.par[i] \) (for some \( i \)), it uses \( I.old[i] \) as the old value. Since there is no ABA problem, only the first such CAS on \( I.par[i] \) can succeed. Moreover, we prove that the flagging mechanism ensures that this first CAS does succeed. Since processes might call help(\( I \)) to help one another complete their operations, there might be a group of child CAS operations on each node. However, the child pointer is changed exactly once for the operation.

### 3.3 Detailed description of algorithms

First, we explain the routines that operations call. A search(\( v \)) is used by updates and find operations to locate the key \( v \) within the trie. A search(\( v \)) starts from the root node and traverses down the trie. At each step of the traversal, search(\( v \)) chooses the child according to the appropriate bit of \( v \) (line 82). The search(\( v \)) stops if it reaches an Internal Node whose label is not a prefix of \( v \). We show later that each node visited by the search was reachable at some time during the search. If the search(\( v \)) does not return a Leaf containing \( v \), there was a time during the search when no Leaf containing \( v \) was reachable. Moreover, the node that is returned is the location where an insert would have to put \( v \). If search(\( v \)) reaches a Leaf Node and the Leaf Node is logically in the trie, search(\( v \)) sets keyInTrie to true (line 85).

As we shall see, update operations must change the child pointers of the parent or grandparent of the node returned by search. The search operation therefore returns \( gp \), \( p \), and node, the last three nodes reached (where \( p \) stands for parent and \( gp \) stands for grandparent). A search also returns the values \( gpI \) and \( pI \) that it read from the info fields of \( gp \) and \( p \) before reading their child pointers. More formally, if search(\( v \)) returns \( gp \), \( p \), node, \( gpI \), \( pI \), \( keyInTrie \), it satisfies the following postconditions.

1. At some time during the search, \( gp.info = gpI \) (if \( gp \) is not null).
2. Later during the search, \( p \) was a child of \( gp \) (if \( gp \) is not null).
3. Later during the search, \( p.info = pI \).
4. Later during the search, \( p.child[i] = node \) for some \( i \), and \( (p.label) \cdot i \) is a prefix of \( v \).
5. If node is internal, node.label is not a prefix of \( v \).
6. If keyInTrie is true, the node whose label is \( v \) is logically in the trie at some time during search(\( v \)).
7. If keyInTrie is false, then at some time during the search, no node containing \( v \) is logically in the trie.

Postconditions (1) to (5) follow easily from the pseudocode. We show in Sect. 4 that (6) and (7) are satisfied. After an update calls search, it determines whether the desired update is possible. If not, it returns false (For example, an insert(\( v \)) operation returns false if \( v \) is already in the trie). If the update is possible, it continues. The insert and replace operations call the createNode routine if the update needs to create an new Internal Node. First, createNode checks at line 119 if there is some other incomplete operation in progress on any of the nodes that are to become children of the
new Internal Node. If so, **createNode** tries to complete that incomplete update and returns null causing the update that called **createNode** to start over (line 120–121). Otherwise, **createNode** creates and returns the required new Internal Node (line 122).

After that, the update calls **newFlag** to create a Flag object. Because **newFlag** takes 7 arguments, each call to **newFlag** in the pseudocode (e.g., on line 31) includes the parameter names as well as the values assigned to those parameters, in order to remind the reader which role each parameter plays. For each node that the update must flag, a value read from the info field during search of the node is passed to **newFlag** as the old value to be used in the flag CAS step. The old value for a flag CAS was read before the old value for the corresponding child CAS, so if the flag CAS succeeds, then the node’s child field has not been changed since the last time its old value was read. The **newFlag** routine checks if all old values for info fields are Unflag objects (line 111). If some info field is not an Unflag object, then there is some other incomplete update operating on that node. The **newFlag** routine tries to help complete the incomplete update (line 112), and then returns null, which causes the update to restart. The **replace** must perform an insertion and deletion. These two changes each requires a set of nodes to be flagged. These two sets might contain some common nodes. So, the list of nodes to be flagged by the **replace** might contain duplicates. If the duplicate elements do not have the same old values, their child fields might have changed since the operation read them, so **newFlag** returns null and the operation starts over (line 114). Otherwise, only one copy of each duplicate element is kept (line 115). The **newFlag** routine sorts the nodes to be flagged (to avoid deadlock) and returns the new Flag object (line 116–117).

After an update u creates a Flag object I, it calls **help(I)**. This routine attempts to complete the update. First, it uses CAS steps to put the Flag object I in the info fields of the nodes to be flagged (line 92). We call these steps flag CAS steps. If all nodes are flagged successfully, the flagDone field of the Flag object is set to true (line 96). The value of the flagDone field is used to coordinate processes that help the update. Suppose a process p is executing **help(I)**. After p performs a flag CAS on a node x, if it sees a value different from I in x’s info field, there are two possible cases. The first case is when all nodes were already successfully flagged for I by other processes running **help(I)**, and then x was unflagged before p tries to flag x (prior to this unflagging, some helper performed the child CAS steps of I successfully). The second case is when no process flags x successfully for I. Since the flagDone field of I is only set to true after all nodes are flagged successfully, p checks the value of the flagDone field on line 101 to determine which case happened. If flagDone is true, the modifications to the trie for update u have been made. If flagDone is false, the update operation cannot be successfully completed, so all Internal Nodes that got flagged earlier are unflagged by the backtrack CAS steps at line 107 and the update u will have to start over.

After flagging all nodes successfully and setting I.flag Done, if I.rmvLeaf is non-null, the info field of the specified Leaf is set to I (line 97). A Leaf might be flagged only by a replace operation that is removing that Leaf from the trie. Then, **help(I)** changes the child fields of nodes in I.par using child CAS steps (line 98–100). Finally, **help(I)** uses unflag CAS steps to unflag the nodes in I.unflag and returns true (line 101–104).

We now explain how each of the updates uses the general scheme described above. An **insert(v)** operation first calls **search(v)**. Let (¬p, node, ¬v, keyInTrie) be the result returned by **search(v)**. If keyInTrie is true, **insert(v)** returns false since the trie already contains v (line 25). Otherwise, the insertion attempts to replace node with a node created at line 122, whose children are a new Leaf Node containing v and a new copy of node (see Fig. 8). Thus, the parent p of node must be flagged. A new copy of node is used to avoid the ABA problem. If node is an Internal Node, to ensure no other operation changes node’s children, **insert(v)** must flag node (line 31). Since node is replaced by a new copy, node is flagged permanently.

---

**Fig. 8** Different cases of **insert(v)** and **delete(v)**. Triangles are either a leaf node or a subtree. The grey circles are flagged nodes. The dotted lines are the new child pointers that replace the old child pointers (solid lines) and the dotted circles and squares are newly created nodes.
A delete(v) operation first calls search(v). Let ⟨gp, p, node, -, -, keyInTrie⟩ be the result returned by the search(v). If keyInTrie is false, delete(v) returns false since the trie does not contain v (line 38). Then, delete(v) replaces p by the sibling of node (see Fig. 8). So, delete(v) must flag the grandparent gp and the parent p of node (line 41). Since p is removed from the trie, only gp must be unflagged after the deletion is completed.

A replace(v_d, v_i) operation first calls search(v_d) and search(v_i), which return ⟨gp_d, p_d, node_d, -, -, keyInTrie_d⟩ and ⟨-, p_i, node_i, -, -, keyInTrie_i⟩. The replace checks that v_d is in the trie and v_i is not, as in the insert and delete operations (line 46–49). If either test fails, the replace operation returns false.

If insert(v_i) and delete(v_d), as described in Fig. 8, would not overlap, replace(v_d, v_i) is accomplished by two child CAS steps and is linearized at the first of these two changes (see Fig. 9). This is called the general case of replace. Situations when the insertion and deletion would occur in overlapping portions of the trie are handled as special cases as shown in Fig. 10. In the special cases, the replace operation changes the trie with just one child CAS.

In the general case of the replace operation (line 52–58), we create a Flag object which instructs the help routine to perform the following actions. The replace flags the same nodes that an insert(v_i) and a delete(v_d) would flag. There are three steps (b1–b3) of the general case of the replace operation between flagging the Internal nodes and unflagging them. Figure 9 shows these steps. (b1) The Leaf node_d gets flagged. Since all Internal nodes are successfully flagged, it is guaranteed that flagging node_d succeeds. So, we simply write the flag object into info field of node_d since CAS is not necessary. (b2) v_i is added to the trie, as in insert(v_i). When the new Leaf Node is added, the Leaf node_d, which contains v_d, becomes logically removed, but is not physically removed yet. Step b2 is the linearization point of the replace operation: Between step b2 and b3, the trie contains v_i and logically does not contain v_d. So, if a process accesses node_d between step b2 and b3, it checks if the trie logically contains v_d (line 84) and it behaves as if v_d is removed from the trie. (b3) node_d is physically deleted as in delete(v_d). After node_d is flagged (step b1), any search that reaches node_d checks (line 84) if p_i is a parent of the old child of p_i using node_d.info (line 125). If it is not, it means the new Leaf containing v_i is already inserted (step b2) and the operation behaves as if v_d is already removed.

There are four special cases of replace(v_d, v_i) where the changes required by the insertion and deletion are on overlapping portions of the trie and the replace operation is done using a single child CAS step. Although the pseudo-code for
these cases looks somewhat complicated, it simply implements the actions described in Fig. 10 by creating a Flag object and calling help. The insertion of \( v \) replaces \( node_i \) by a new node. The cases when the deletion must remove \( node_i \) or change \( node_i.child \) are handled as special cases. So, the case that \( node_d = node_i \) is one special case (line 59–60). In the deletion, \( p_d \) is removed, so the case that \( p_d = node_i \) or \( p_d = p_i \) are also handled as a special case (line 61–64). In the deletion, \( gp_d.child \) is changed. So, the last special case is when \( gp_d = node_i \) (line 65–71). In all special cases, \( node_i \) is replaced by a new node. Here, we explain one special case in detail. The others are handled in a similar way. In case 2 of Fig. 10, \( p_d = node_i \) and \( gp_d = p_i \) (line 61). So, replace\((v_d, \quad v_i)\) creates an Info object that contains instructions to flag \( gp_d \) and \( node_i \), replace \( node_i \) with a new Internal Node whose non-empty children are a new Leaf Node containing \( v_i \) and the sibling of \( node_d \), and then unflag \( gp_d \) (line 61–64).

We remark that logical removal of a Leaf is only required for replace operations that are performed by two child CAS steps. In the special cases where the replace can be performed by a single child CAS, the Leaf containing \( v_d \) is physically removed by the same child CAS that inserts the new Leaf containing \( v_i \). In these special cases, there is therefore no need to flag the Leaf containing \( v_d \).

4 Algorithm correctness

A detailed proof of correctness is provided in [32]. It is very lengthy, so we provide a higher-level proof here.

A configuration of an implementation at some time consists of the values of all parts of the shared memory and of the local variables of all processes at that time. A step of an implementation is some part of the code of the implementation that is performed by one process and includes at most one access to the shared memory. The model of computation is asynchronous shared memory. Any number of processes may crash.

First, we define a linearization point for each operation to preserve the invariant that the set of keys that are logically in the trie form the set \( D \) represented by the data structure. We define a linearization point for searches as well as for all data structure operations. Let \( \langle \text{keyInTrie} \rangle \) be the result returned by search\((v)\). If \text{keyInTrie} is true, postcondition (6) of the search says there is a time during the search when node is logically in the trie and the search is linearized at that time. Otherwise, postcondition (7) of the search ensures there is a time during the search when no Leaf containing \( v \) is logically in the trie and the search is linearized at that time. Each complete find operation is linearized at the linearization point of its search. If an update returns false, it is linearized at the linearization point of the search that caused the update to fail. Let \( I \) be a Flag object created by an update. If a child CAS performed by any call to help\((I)\) is executed, the update is linearized at the first such child CAS.

Next, we have the correctness proof in the following five subsections. Sections 4.1 and 4.2 prove that the successful CAS steps performed by all calls to help\((I)\) proceed in the expected order (see Fig. 11).

4.1 Flagging

This section is the heart of the proof. Consider the flag object \( I \) created by some update operation. Our goal is to prove that the successful CAS steps that flag or unflag nodes performed by all calls to help\((I)\) proceed in the expected order (see Fig. 11). More precisely, we prove that, first, the flag CAS steps are performed on nodes \( I.flag \), ordered according to the nodes’ labels. We prove that only the first flag CAS (by any of the helpers of \( I \)) on each node can succeed. If one of these flag CAS steps fails, then the nodes that have been flagged are unflagged by backtrack CAS steps and all calls to help\((I)\) return false, indicating that the attempt at performing the update has failed. Otherwise, the child CAS steps are performed, and then the unflag CAS steps remove flags from nodes in \( I.unflag \). If several helpers perform one of these CAS steps, we prove that the first helper succeeds and no others do. In this case, all calls to help\((I)\) return true.

To do this, we first have some straightforward properties of search that follow directly from the pseudo-code (line 77–82).

Lemma 1 The search satisfies its postconditions (1) to (5), described in Sect. 3.3.
Invariant 2. If \( x.child[i] = y \), then \((x.label) \cdot i\) is a prefix of \( y.label \).

**Proof.** We show that whenever an Internal Node is created and whenever the child field of an Internal Node is changed, the invariant is preserved if the invariant was true at all earlier configurations. The invariant holds when the trie is initialized on line 20. It is trivial to check that when a new copy of a Node is made on line 27 or 53 or a new Internal Node is created on line 122, the invariant is preserved.

Next, we need to show when a child CAS (line 100) of \( I \) succeeds, the invariant is preserved. The child CAS changes \( I.par[j].child[i] \) from \( I.old[j] \) to \( I.new[j] \) (for some \( j \)) where \( i \) is the \((I.par[j].label)+1\)th bit of \( I.new[j].label \). Thus, it suffices to show that \( I.par[j].label \) is a proper prefix of \( I.new[j].label \). \( I \) can be created on line 31, 32, 41, 56, 58, 60, 64 or 71. We consider different cases according to what line creates \( I \).

Case 1: \( I \) is created at line 31 or 32. Let \((-, p, node, -, -, -)\) be the result returned by the call to \( \text{search}(v) \) on line 24 that precedes the creation of \( I \). Then, new is the new Node that is created at line 28 and whose children are a new copy of \( node \) and a new Leaf Node whose label is \( v \). In this case, \( I.par[0] = p \) and \( I.new[0] = new \). By Lemma 1, \( p.child[k] = node \) for some \( k \) at an earlier configuration. Since the invariant is true at all earlier configurations, \((p.label) \cdot k\) is a prefix of \( node.label \). By Lemma 1, \((p.label) \cdot k\) is a prefix of \( v \). Since \( new.label \) is the longest common prefix of \( v \) and \( node.label \), \((p.label) \cdot k\) is a prefix of \( new.label \).

Case 2: \( I \) is created at line 41. Let \((gp, p, node, -, -, -)\) be the result returned by the call to \( \text{search}(v) \) on line 37 that precedes the creation of \( I \). Let sibling be the child of \( p \) that is read at line 39. Since the invariant is true at all earlier configurations, \( p.label \) is a prefix of \( sibling.label \). In this case, \( I.par[0] = gp \), \( I.old[0] = p \), and \( I.new[0] = sibling \). Since the child CAS of \( I \) succeeds, \( gp.child[i] \) was \( p \) just before the child CAS. Since the invariant is true at all earlier configurations, \((gp.label) \cdot i\) is a prefix of \( p.label \). Since \( p.label \) is a prefix of \( sibling.label \), \((gp.label) \cdot i\) is a prefix of \( sibling.label \).

When \( I \) is created on line 56 or 58, the invariant can be proved by arguments similar to both case 1 and 2. When \( I \) is created on line 60, 64 or 71, the invariant can be proved by an argument similar to case 1.

The following lemma describes how the info field of a Node is changed. It follows easily from lines 92, 103, 107 and 111–113 of the pseudo-code.

Lemma 3. Let \( x \) be an Internal Node. Then, \( x.info \) is initially set to a new Unflag object and the only changes to \( x.info \) that can occur are (1) a flag CAS at line 92 that changes \( x.info \) from an Unflag object to a Flag object, or (2) an unflag CAS at line 103 or a backtrack CAS at line 107 that changes \( x.info \) from a Flag object to a newly created Unflag object.

Using Lemma 3, we can prove that the ABA problem on the info fields is avoided because whenever an info field is changed, it is set to a newly created Flag or Unflag object. Recall that we assume the garbage collector does not recycle an object if a thread can access the object by following a pointer (or a chain of pointers).

Lemma 4. If a flag, unflag or backtrack CAS on the info field of a Node succeeds, the value of the info field has not been changed between the preceding read and the CAS.

Thus, if several helpers of an Info object try to perform a flag CAS, backtrack CAS or unflag CAS on a node, only the first one can succeed. It follows from the pseudo-code (line 92, 103 and 107) that these CAS steps proceed in the order shown in Fig. 11. If \( \text{doChildCAS} \) is true at line 95, then the info fields of all Nodes in \( I.flag \) were set to \( I \) earlier. So, we have the following lemma.

Lemma 5. Let \( I \) be a Flag object. A childCAS step of \( I \) is preceded by flagging all Nodes in \( I.flag \) using flag CAS steps of \( I \).

**Proof.** By the pseudo-code, a child CAS step of \( I \) can be executed only when \( \text{doChildCAS} \) is true on line 95. That line can be reached only if all nodes in \( I.flag \) have already been flagged by flag CAS steps of \( I \).

We want to ensure the info field of a node gives an operation exclusive access to update the node’s child field. The following lemma guarantees that the node is still flagged with \( I \) when a helper of \( I \) first tries to update the node’s child field.

Lemma 6. If there is any child CAS of an Info object \( I \), then there is no unflag or backtrack CAS of \( I \) before the first child CAS step of \( I \) on each Node in \( I.par \).

**Proof.** To derive a contradiction, assume, for some \( j \), an unflag or backtrack CAS of \( I \) occurs before the first child CAS of \( I \) on \( I.par[j] \). Let \( S \) be that child CAS and \( S' \) be the first unflag or backtrack CAS of \( I \) that occurs before \( S \). Let \( H \) be the invocation of \( \text{help} \) that executes \( S' \). So, \( H \) does not execute any child CAS of \( I \) on \( I.par[j] \) before \( S \) and the \( \text{doChildCAS} \) variable is false when \( H \) performs line 95 (before \( S' \)). Thus, just after \( H \) tries to flag some Node \( y \) by a flag CAS of \( I \), \( H \) sets the \( \text{doChildCAS} \) variable to false at an execution of line 93 when \( y.info \neq I \). By Lemma 5, \( y.info \) is set to \( I \) before \( S \). Since only the first flag CAS of \( I \) on \( y.info \) succeeds and a flag CAS of \( I \) on \( y.info \) is performed
just before that execution of line 93, \( y.info \) is set to \( I \) before \( H \) reads \( y.info \) on line 93. Then, \( y.info \) is changed from \( I \) to another value before \( H \) reads \( y.info \) at line 93, contradicting the fact that \( S' \) is the first unflag or backtrack CAS of \( I \). \( \Box \)

By Lemmas 5 and 6, we have the following corollary.

**Corollary 7** Each Node in \( I \).flag is flagged by \( I \) when the first child CAS of \( I \) on each Node in \( I.par \) occurs.

### 4.2 Child CAS steps

Let \( gp, p \) and \( node \) be the three Nodes that the last call to \( \text{search}(v) \) inside an update returns. Let \( I \) be the Flag object that the update creates after that call to \( \text{search} \). In this section, we show that successful flagging ensures that \( gp, p \) and \( node \) are three consecutive Nodes in the trie just before the first child CAS of \( I \), and that the first child CAS of \( I \) on each Node in \( I.par \) succeeds (and no other child CAS steps of \( I \) do).

**Lemma 8** Let \( I \) be a Flag object.

1. The first child CAS performed by a helper of \( I \) on each Node in \( I.par \) succeeds.
2. If all Nodes in \( I \).flag are successfully flagged with \( I \) and \( x = I \).flag[i] for some \( i \), then no child CAS of any other Flag object \( I' \neq I \) changes \( x.child \) between the time when \( x.info \) is set to \( I \) and the first child CAS of \( I \) on \( I.par[j] \) for some \( j \).
3. A child field of a Node is never set to a Node that has been stored there before.
4. If a Node becomes unreachable, the Node never becomes reachable after that.
5. If an Internal Node \( y \) becomes unreachable by a successful child CAS of \( I \), \( x.info = I \) at all configurations after the child CAS.

**Proof** Lemma 8 is proved by induction on the length of the execution. It is trivial to show that all statements are true just after the initialization of the trie.

The induction step requires reasoning about the way flags act like locks. To prove Statements 1 and 2, consider the first child CAS by some helper of \( I \) on a Node \( x \) in \( I.par \). By Corollary 7, \( x \) is flagged with a pointer to \( I \) when that first child CAS is performed. Since the flag CAS of \( I \) on \( x \) succeeded, \( x \) has not been flagged by any other update between when the update that created \( I \) read \( x.info \) during its \( \text{search} \) and the flag CAS of \( I \) on \( x \) (by Lemma 4). It follows that no other update has flagged \( x \) between that read and the child CAS, and hence the child field of \( x \) has not changed during that interval by Corollary 7. It follows from Lemma 1 that the old value used by the child CAS is still in \( x.child \) when the child CAS occurs, so it succeeds. This proves Statement 1 and 2.

The ABA problem on the \( child \) fields is avoided because whenever a child pointer is changed, the old child is permanently removed from the trie as indicated in Figs. 8 and 10. It follows that if a Node \( y \) becomes reachable by a child CAS of \( I \), \( y \) is newly created by the update that created \( I \). This proves Statement 3 and 4.

If \( y \) becomes unreachable by a child CAS of \( I \), \( y.info = I \) when the child CAS occurs (by Corollary 7). Since \( I.flagDone \) is set to true before the child CAS, no backtrack CAS of \( I \) is executed after the child CAS. As indicated in Figs. 8 and 10 and following from the pseudo-code (line 31, 32, 41, 56, 58, 60, 64 or 71), \( y \) is not in \( I.unflag \). So, \( y.info \) cannot be changed from \( I \) to an Unflag object after the child CAS of \( I \). This proves Statement 5. \( \Box \)

The arguments used in Sects. 4.1 and 4.2 are mostly focused on the structure of the \( \text{help} \) routine. So, any new update that preserves the main invariants of the trie can be added with minor changes to these parts of the correctness proof.

### 4.3 Postconditions of search operations

In this section, we show that postconditions (6) and (7) of the \( \text{search} \) operation are satisfied.

**Lemma 9** Each node a \( \text{search} \) visits was reachable at some configuration during the \( \text{search} \) before the \( \text{search} \) visited that node.

**Proof** We prove the lemma by induction on the number of steps that the \( \text{search} \) has done. Let \( T \) be a time that the \( \text{search} \) visits a node \( x \) on line 82. We assume the lemma is true for all nodes visited before \( T \). Now, we show that the lemma is true for \( x \). If \( x = \text{Root} \), the lemma is trivially true. Otherwise, a Node \( p \) is visited earlier at line 78 or at line 82 during the previous loop iteration before \( T \). Thus, \( p \) was reachable at some time \( T' \) after the \( \text{search} \) begins and before the \( \text{search} \) visited \( p \). If \( x \) was a child of \( p \) at \( T' \), the lemma is proved.

Otherwise, a child CAS of a Flag object \( I \) sets an element of \( p.child \) to \( x \) at some later time between \( T' \) and \( T \). By Lemma 8.1, the first child CAS of \( I \) on \( p \) succeeds between \( T' \) and \( T \). By Corollary 7, \( p.info = I \) at the configuration before the child CAS of \( I \) between \( T' \) and \( T \). Since \( p \) was reachable at \( T' \), by Lemma 8.5, \( p \) is still reachable at the configuration before the child CAS of \( I \). So, \( p \) is reachable and \( x \) is \( p \)'s child at the configuration after the child CAS of \( I \) between \( T' \) and \( T \). Thus, \( x \) is reachable at that configuration. \( \Box \)

The postconditions (6) and (7) state that the value of \( \text{keyInTrie} \) returned by a call to \( \text{search}(v) \) correctly indicates whether the trie logically contained label \( v \) or not at some time during the \( \text{search} \). To show these postconditions,
we first prove the following lemma regarding the updates that change the trie in two steps. Let \( I \) be a Flag object that is created at line 56 or 58 of a replace operation. Then, the number of elements of \( \mathit{I}.\mathit{par} \), \( \mathit{I}.\mathit{old} \) and \( \mathit{I}.\mathit{new} \) is two and \( \mathit{I}.\mathit{rmvLeaf} \) is set to a Leaf Node. The Leaf \( \mathit{I}.\mathit{rmvLeaf} \) is flagged at line 97 before the first child CAS of \( I \) at line 100. The following lemma shows that after a Leaf Node is flagged by a Flag object, the Leaf Node is never flagged by another Flag object. So, \( \mathit{I}.\mathit{rmvLeaf} \) remains flagged permanently.

**Lemma 10** Let \( I \) be a Flag object that is created by a replace operation on line 56 or 58 and \( x \) be \( \mathit{I}.\mathit{rmvLeaf} \). If \( x.\mathit{info} \) is set to \( I \) by line 97, then \( x.\mathit{info} \) is never changed after that.

**Proof** Suppose \( x \) is a Leaf Node whose \( \mathit{info} \) is set to a Flag object \( I \) on line 97. Since \( \mathit{I}.\mathit{rmvLeaf} \) is set to \( x \) on line 56 or 58, the replace operation does not return on line 47. Then, \( \mathit{keyInTrie}_d \) is true and the last call to search on line 46 before creating \( I \) executed line 85. So, \( x \) is a Leaf and the only line that can change \( x.\mathit{info} \) is line 97. Let \( s \) be the first child CAS of \( I \) on \( \mathit{I}.\mathit{par}[1] \). We first show \( x.\mathit{info} \) cannot be set to another Flag object before \( s \). Consider the last search that the replace operation called (at line 46) before creating \( I \). Let Node \( p \) be the parent of \( x \) read during that search. By the pseudo-code (line 56 and 58), \( p \) is in \( \mathit{I}.\mathit{flag} \). It follows from Lemma 4 and Corollary 7 that \( p \) is flagged by \( I \) at all configurations between setting \( x.\mathit{info} \) to \( I \) and \( s \). By Lemmas 8.5 and 9, \( p \) is reachable during that interval. By Lemma 8.2, \( p \) is the parent of \( x \) during that interval. So, when a Leaf Node is flagged by an Info object at line 97, the Leaf is reachable and the parent of the Leaf Node is also flagged with the same Info object. Thus, \( x.\mathit{info} \) cannot be set to another Flag object before \( s \).

Next, we show \( x.\mathit{info} \) cannot be set to another Flag object after \( s \). Let \( s' \) be the step that sets \( x.\mathit{info} \) to \( I \) for the first time. By similar argument to the paragraph above, at the configuration before \( s \), each Node in \( \mathit{I}.\mathit{flag} \) is reachable and flagged by \( I \) and the Nodes flagged by \( I \) are as shown as Fig. 8 or 10. It follows that \( s \) causes \( x \) to become unreachable. By Lemma 8.4, \( x \) is unreachable at all configurations after \( s \). By the argument in the paragraph above, when a Leaf is flagged at line 97, the Leaf is reachable. Since \( x \) is unreachable after \( s \), \( x.\mathit{info} \) cannot be set to another Flag object at line 97 after \( s \).

**Lemma 11** The search satisfies its postconditions (6) and (7), described in Sect. 3.3

**Proof** Suppose a call to search(\( v \)) returns \( \langle \cdot, \cdot, \cdot, \mathit{keyInTrie} \rangle \).

Case 1: \( \mathit{keyInTrie} \) is true. So, the search executed line 85 and \( \mathit{node} \) is a Leaf with label \( v \). We now show \( \mathit{node} \) is logically in the trie at some time during the search.

Case 1a: the value of \( \mathit{node}.\mathit{info} \) read on line 84 is an Unflag object. By Lemma 10, \( \mathit{node}.\mathit{info} \) is never set to a Flag object before that. By Lemma 9, \( \mathit{node} \) was reachable at some time during the search before the search visited \( \mathit{node} \). So, \( \mathit{node} \) was logically in the trie at some time during search.

Case 1b: the value of \( \mathit{node}.\mathit{info} \) read on line 84 is a Flag object \( I \). Let \( C \) be the configuration before reading the value of \( \mathit{I}.\mathit{par}[0].\mathit{child} \) on line 125 during the call to logicallyRemoved on line 84. Because logicallyRemoved returns false, \( \mathit{I}.\mathit{old}[0] \) is in \( \mathit{I}.\mathit{par}[0].\mathit{child} \) at the configuration \( C \). So, it follows from Lemma 8.3 that the first child CAS of \( I \) on \( \mathit{I}.\mathit{par}[0] \) has not occurred earlier than \( C \). By Lemma 10, \( \mathit{node} \) is not flagged at any time with any Flag object other than \( I \). Recall that when a child CAS of a Flag object logically removes \( \mathit{node} \), \( \mathit{node} \) must be flagged by the same Flag object. By definition, \( \mathit{node} \) is not logically removed from the trie earlier than \( C \). Thus, the configuration \( C \) is between step b1 and b2 of the replace operation that created \( I \) (Fig. 9).

Recall that b2 is the linearization point of the replace operation. By Lemma 9, \( \mathit{node} \) was reachable at some configuration \( C' \) during search earlier than \( C \). So, \( \mathit{node} \) is logically in the trie at \( C' \).

Case 2: \( \mathit{keyInTrie} \) is false. It follows from Lemma 9 that \( \mathit{node} \) was reachable at some configuration \( C'' \) during the search.

Case 2a: \( \mathit{node}.\mathit{label} \) is not \( v \). Since the condition on line 79 is not satisfied for \( \mathit{node} \), it follows from Invariant 2, no node containing \( v \) is reachable at the configuration \( C'' \).

Case 2b: \( \mathit{node}.\mathit{label} \) is \( v \). So, \( \mathit{node} \) is a Leaf and the call to logicallyRemoved returns true on line 84. So, the value of \( \mathit{node}.\mathit{info} \) read on line 84 is a Flag object \( I \). Let \( C^* \) be the configuration before reading the value of \( \mathit{I}.\mathit{par}[0].\mathit{child} \) on line 125 during the call to logicallyRemoved on line 84. Then, \( \mathit{I}.\mathit{old}[0] \) is not in \( \mathit{I}.\mathit{par}[0].\mathit{child} \) at the configuration \( C^* \). So, it follows from Lemma 8.3 that the first child CAS of \( I \) on \( \mathit{I}.\mathit{par}[0] \) has occurred earlier than \( C^* \). Since \( C'' \) is earlier than \( C^* \) (by Lemma 9) and \( \mathit{node} \) is reachable at \( C'' \), there is a configuration between \( C'' \) and \( C^* \) in which \( \mathit{node} \) is reachable but not logically in the trie. I.e., that configuration is between the step b2 and b3 of the replace operation that created \( I \). Recall b2 is the linearization point of the replace operation. So, at that configuration, no node containing \( v \) is logically in the trie (by Invariant 2).

### 4.4 Linearizability of update operations

This section proves that update operations are linearized correctly. Let \( T \) be the linearization point of a successful update operation, which is the first successful child CAS performed by any helper of the operation. We argue, using Lemma 1, that this first child CAS has the effect of implementing precisely
the change shown in Fig. 8 or 10 atomically. In the case of a replace operation including two successful child CAS steps, the linearization point of a successful replace adds a new Leaf to the trie. If another operation accesses the Leaf Node that would be deleted by the replace after that and before the second child CAS, the test performed by logicallyRemoved ensures that it behaves as if the Leaf is not in the trie. This is used to establish an invariant that proves all operations return correct results.

To establish linearizability, we define an auxiliary variable \( D \) that stores a set. Initially, it is empty. Each time an operation is linearized, the same operation is atomically applied to \( D \) according to the sequential specification.

**Invariant 12** The Leaf Nodes that are logically in the trie at a configuration (excluding the Leaf Nodes containing the dummy keys \( 0^C \) and \( V^I \)) contain exactly the keys in \( D \).

**Proof** We prove the invariant by induction. Initially, \( D \) is empty and the trie looks like Fig. 3. So, the invariant is true at the initial configuration.

Recall that a successful update operation is linearized at the first successful child CAS performed by any helper of the operation. So, the only steps that may change \( D \) or reachable Nodes of the trie are the successful child CAS steps and the linearization points of unsuccessful update operations. Let \( s \) be a successful child CAS or the linearization point of an unsuccessful update operation and \( C \) be the configuration before \( S \).

By induction hypothesis, the invariant is true at \( C \). Then, we show that \( s \) preserves the invariant.

**Claim 1** The linearization point of a successful insert\((v)\) operation preserves the invariant.

**Proof of Claim 1** Suppose \( s \) is the successful child CAS of some Info object \( I \) created by a successful insert\((v)\). We show \( s \) changes \( D \) to \( D \cup \{ v \} \) and adds a reachable Leaf containing \( v \) to the trie. Let \( \langle -, p, node, -, -, keyInTrie \rangle \) be the result returned by the insert’s last call to search\((v)\) on line 24 before creating \( I \). By the pseudo-code (line 31 and 32), \( p \) is in \( I\).flag. It follows from Lemma 4 and Corollary 7 that \( p \) is flagged by \( I \) at all configurations between setting \( x\).info to \( I \) and \( s \). By Lemmas 8.5 and 9, \( p \) is reachable throughout that interval. By Lemma 8.2, \( p \) is the parent of node throughout that interval. So, node is also reachable throughout that interval. Since the delete does not return on line 38, keyInTrie is true. So, node.label = \( v \) by the pseudo-code (line 85). Since node is reachable at \( C \), by the induction hypothesis \( v \in D \). Then, \( s \) causes node to become unreachable and also changes \( D \) to \( D \setminus \{ v \} \). So, \( s \) preserves the invariant.

**Claim 2** The linearization point of a successful delete\((v)\) preserves the invariant.

**Proof of Claim 2** Suppose \( s \) is the successful child CAS of some Info object \( I \) created by a successful delete\((v)\). We show \( s \) changes \( D \) to \( D \setminus \{ v \} \) and removes a reachable Leaf containing \( v \) from the trie. Let \( \langle -, p, node, -, -, keyInTrie \rangle \) be the result returned by the delete’s last call to search\((v)\) on line 37 before creating \( I \). By the pseudo-code (line 41), \( p \) is in \( I\).flag. It follows from Lemma 4 and Corollary 7 that \( p \) is flagged by \( I \) at all configurations between setting \( x\).info to \( I \) and \( s \). By Lemmas 8.5 and 9, \( p \) is reachable throughout that interval. By Lemma 8.2, \( p \) is the parent of node throughout that interval. So, node is also reachable throughout that interval. Since the delete does not return on line 38, keyInTrie is true. So, node.label = \( v \) by the pseudo-code (line 85). Since node is reachable at \( C \), by the induction hypothesis \( v \in D \). Then, \( s \) causes node to become unreachable and also changes \( D \) to \( D \setminus \{ v \} \). So, \( s \) preserves the invariant.

**Claim 3** The linearization point of a successful replace\((v, v')\) preserves the invariant.

**Proof of Claim 3** Suppose \( s \) is the first successful child CAS of some Info object \( I \) created by a successful replace\((v, v')\). Let \( C' \) be the configuration after \( s \). To show \( s \) preserves the invariant, we show the following statements are true. (1) At the configuration \( C, v \in D \) and \( v' \notin D \). (2) At the configuration \( C \), a Leaf containing \( v \) is logically in the trie and no Leaf containing \( v' \) is reachable. (3) At the configuration \( C' \), a Leaf containing \( v' \) is logically in the trie. (4) At the configuration \( C' \), no Leaf containing \( v \) is logically in the trie.

If the replace requires only one successful child CAS to complete, the proofs of all statements above are similar to the arguments of the proofs of Claims 1 and 2. Otherwise, the replace requires two successful child CAS steps to complete. Then, the proof of Statement (1)–(3) are similar to the arguments of the proofs of Claim 1 and 2. Let \( \langle -, -, node_d, -, -, - \rangle \) be the result returned by the replace’s last call to search on line 46 before creating \( I \). To show Statement (4) is true, we need to show that the Leaf node_d containing \( v \) is reachable but not logically in the trie at \( C' \). The proof that node_d contains \( v \) and is reachable at \( C' \) is similar to the proof of Claim 2. By Lemma 10, node_d is flagged by a Flag object \( I \) created by the replace at \( C' \). By the definition of logically removed, node_d is not logically in the trie since the first child CAS of \( I \) on \( I.par[0] \) is the step before \( C' \). So, \( s \) preserves the lemma.
Claim 4  Suppose a successful replace operation requires two successful child CAS steps to complete. Then, that second successful child CAS preserves the invariant.

Proof of Claim 4  Suppose a successful replace(v, v') operation requires two successful child CAS steps to complete and s is the second successful child CAS of some Info object I created by the replace. Since s is not a linearization point of any operation, s does not change D. To show that s causes a Leaf node containing v to become unreachable is the same as the proof of Claim 2. By Lemma 10, node is flagged in all configurations after the first successful child CAS of I, which occurs earlier than s. By definition, node is logically in the trie at C. So, no Leaf containing v is logically in the trie at C. Thus, s does not change the Nodes that are logically in the trie.

Claim 5  The linearization point of an unsuccessful insert operation preserves the invariant.

Proof of Claim 5  Suppose s is the linearization point of an unsuccessful insert(v) operation. If a Flag object I is created by the insert, since not all Nodes in I.flag are flagged successfully with I, no child CAS of I ever occurs. So, s does not change the reachable Nodes of the trie. Now, we show s does not change D. Since the insert is unsuccessful, it returns false on line 25. Let (ṣ, ṣ, ṣ, ṣ, keyInTrie) be the result returned by the insert’s last call to search on line 24. So, keyInTrie is true. By Lemma 11, the Node whose label is v is logically in the trie at the configuration before s. By the induction hypothesis, v ∈ D at the configuration before s. So, s does not change D.

Claim 6  The linearization point of an unsuccessful delete operation preserves the invariant.

Proof of Claim 6  Suppose s is the linearization point of an unsuccessful delete(v) operation. By the same argument as in the proof of Claim 5, s does not change the reachable Nodes of the trie. Now, we show s does not change D. Since the delete is unsuccessful, it returns false on line 38. Let (ṣ, ṣ, ṣ, ṣ, keyInTrie) be the result returned by the delete’s last call to search on line 37. So, keyInTrie is false. By Lemma 11, no Node containing v is logically in the trie at the configuration before s. By the induction hypothesis, v /∈ D at the configuration before s. So, s does not change D.

Claim 7  The linearization point of an unsuccessful replace operation preserves the invariant.

Proof of Claim 7  Suppose s is the linearization point of an unsuccessful replace(v, v') operation. By the same argument as in proof of Claim 5, s does not change the reachable Nodes of the trie. Since the replace is unsuccessful, it returns false on line 47 or 49. If the replace returns on line 49, with the same argument as in proof of Claim 6, s does not change D. If the replace returns on line 47, with the same argument as in proof of Claim 5, s does not change D.

Lemma 13  If find(v) returns true, v ∈ D at the linearization point of the find. If find(v) returns false, v /∈ D at the linearization point of the find.

Proof  A find operation is linearized at the configuration that is defined by the postcondition (6) or (7) of search according to the result that its call to search returns. By Lemma 11 and Invariant 12, the claim is true.

4.5 Progress
Finally, this section of the proof establishes progress. First, we show that the search operation is wait-free.

Lemma 15  The search operation is wait-free.

Proof  Let ℓ be the length of the keys in U. By Invariant 2, length of node.label increases by at least one in each iteration of the loop in the search routine. Since labels of Nodes have length at most ℓ, there are at most ℓ iterations.

Theorem 16  The implementation is non-blocking.

Proof  To derive a contradiction, assume after some time T, no operation terminates or fails. Let I be a Flag object created by an update that is running after T. If a call to help(I) returns true, the update terminates, so after T, all calls to help(I) return false. Thus, after T, all calls to help(I) set doChildCAS to false because they failed to flag an Internal Node successfully. Consider the group of all calls to help(I). We say the group blames the Internal Node that is the first node that no call to help(I) could flag successfully. Let go, ..., gm be all these groups ordered by the labels of the nodes they blame. Since gm blames an Internal Node x, x is flagged by some other group gi where 0 ≤ i < m. Thus, gi blames some other node y whose label is less than x. So, gi flags x before attempting to flag y, contradicting the fact that gi flags Internal Nodes in order.
5 Empirical evaluation

We experimentally compared the performance of our implementation (PAT) with non-blocking binary search trees (BST) \[15\], non-blocking \(k\)-ary search trees (4-ST) \[10\], ConcurrentSkipListMap (SL) of the Java library, lock-based AVL trees (AVL) \[7\] and non-blocking hash tries (Ctrie) \[30\]. For the \(k\)-ary search trees, we use the value \(k = 4\), which was found to be optimal in \[10\]. Nodes in Ctrie have up to 32 children.

The experiments were executed on a Sun SPARC Enterprise T5240 with 32GB RAM. The machine had two Ultra-SPARC T2+ processors, each with eight 1.2GHz cores, for a total of 128 hardware threads. The experiments were run in Java. The Sun JVM version 1.7.0_3 was run in server mode. The heap size was set to 2G. This ensures the garbage collector would not be invoked too often, so that the measurements reflect the running time of the algorithms themselves. Using a smaller heap size affects the performance of BST, 4-ST and PAT more than AVL and SL since they create more objects.

We evaluated the algorithms in different scenarios. We ran most experiments using uniformly distributed random keys. We ran the algorithms using uniformly distributed keys in two different ranges: \((0, 10^2)\) to measure performance under high contention and \((0, 10^6)\) for low contention. In the range \((0, 10^2)\), the tree is very small and operations are more likely to access the same part of tree (we also ran the experiments for the key range of \((0, 10^3)\) and the results were very similar to the low contention case). We ran experiments with two different operation ratios: 5% inserts, 5% deletes and 90% finds (i5-d5-f90), and 50% inserts, 50% deletes and 0% finds (i50-d50-f0) (we also ran the experiments with ratio of 15% inserts, 15% deletes and 70% finds (i15-d15-f70). Since the results were similar to the experiments with the ratio of i5-d5-f90, we do not present them here).

Since the replace operation is not used in these sets of experiments, we made some minor optimizations to the pseudo-code. For example, we eliminated the rmvd variable in search operations.

Since the Java compiler optimizes its running code, before each experiment, we run the code for 10 s for each implementation. We start each experiment with a tree initialized to be half-full, created by running updates in the ratio i50-d50-f0 until the tree is approximately half-full. Each data point in our graphs is the average of eight 4-s trials (the error bars in the charts show the standard deviation).

For uniformly distributed keys, algorithms scale well under low contention \([\text{key range of } (0, 10^6)]\) (see Fig. 12). Under very high contention \([\text{key range of } (0, 10^2)]\), most scale reasonably well when the fraction of updates is low, but experience problems when all operation are updates (see Fig. 13). This key range was chosen to test performance where operations concurrently try to update the same part of the tree, even when the number of processes is fairly small. When the range is \((0, 10^6)\), Ctrie outperforms all others because the height of the Ctrie is very small by having nodes with 32 children. However when the range is \((0, 10^2)\) and the contention is very high, Ctrie does not scale. Excluding Ctrie, when the range is \((0, 10^6)\), PAT, 4-ST and BST outperform AVL and SL. Since updates are more expensive than finds, the throughput is greater for i5-d5-f90 than for i50-d50-f0.
To evaluate the replace operations, we ran an experiment with 10% inserts, 10% deletes and 80% replace operations (i10-d10-r80) and a key range of (0, 10^6) on uniformly random keys (see Fig. 14). We could not compare these results with other data structure since none provides atomic replace operations. As the chart shows, the replace operation scales well.

We also performed some experiments on non-uniformly distributed random keys (see Fig. 15). To generate non-uniform keys, processes performed operations on sequence of 50 consecutive keys from the range (0, 10^6), starting from a randomly chosen key. In this experiment, since tries maintain a fixed height without doing expensive balancing operations, Ctrie outperforms all others and PAT greatly outperforms others except Ctrie. Since the results of these experiments for other operations ratios were similar, only the chart for the ratio i15-d15-f70 is presented here. Longer sequences of keys degrade the performance of BST and 4-ST even further.

6 Conclusion

Our algorithms can also be used to store unbounded length strings. One approach would be to append $ to the end of each string. To encode a binary string, 0, 1 and $ can be represented by 01, 10 and 11. Then, every encoded key is greater than 00 and smaller than 111, so 00 and 111 can be used as keys of the two dummy leaves (instead of 0\ell and 1\ell). Moreover, since labels of nodes never change, they need not fit in a single word.

The approach used in the replace operation can be used for operations on other data structures that must change several pointers atomically. Future work includes providing the general framework for doing this on any tree-based structure. Such a framework would have to guarantee that all changes become visible to query operations at the same time. Brown et al. [9] have proposed a general technique for non-blocking trees that support one change to the tree atomically.

Algorithms can be designed for more complicated non-blocking query operations on tries such as range queries, nearest neighbour searches and predecessor queries. Such a query can be done by reading a portion of the trie repeatedly until the same set of Nodes are obtained by two consecutive traversals. One must also check that the portion of the trie has not been removed. This can be done by checking that the Nodes are not permanently flagged. Since the implementation guarantees there is no ABA problem on the child field, the query operation that terminates can be linearized any time between the last two traversals. Such query operations do not interfere with any update operations.

In [8,9,15], nodes are unflagged by setting a bit in the descriptor object. However, in our trie, nodes are unflagged by setting their pointers to empty descriptor objects using CAS steps. There are trade-offs between these two approaches. The first approach unflags nodes in one step regardless of the number of nodes that must be unflagged. So, the first approach is more efficient when there are many nodes to be unflagged since the second approach creates an empty descriptor object and then performs a CAS step to unflag each node. However, the first approach might require a more complicated memory management technique since it can cause a chain of pointers from nodes in the data structure to nodes that are removed even when the operation that removed these nodes terminated. So, the second approach is more suitable for complex data structures. Since our trie supports more complex update operations, nodes are unflagged by setting their pointers to empty descriptor objects to simplify the correctness proof and memory management.

Since our algorithms create many Flag objects to avoid using locks, finding efficient memory management techniques is an important area for future work. This includes applying known memory management techniques to our implementations and empirically evaluating them. Hart et al. [22] evaluated different memory management techniques on some lock-free data structures. Their results show that there is no globally optimal scheme and the data structure, the workload, and the execution environment can dramatically affect memory reclamation performance.
Recently, Arbel and Brown [2] introduced a transformation for a non-blocking algorithm that may allocate many descriptors throughout an execution like our trie. It transforms such an algorithm to a non-blocking algorithm that initially allocates a fixed number of descriptors and reuses them. Their experimental results show that the transformed implementation performs at least as well as the original implementation, and significantly outperforms the original implementation in some workloads. In their transformation, each thread has a single descriptor that can be reused repeatedly and each descriptor has a version number. Each time an operation needs to create a new descriptor, it instead reuses its single descriptor and increments its version number. Whenever a thread tries to access another thread’s descriptor to help complete an operation, it must specify which version of that descriptor it is trying to access. If the version does not match the current version of the descriptor, then the thread fails to access the descriptor, indicating that operation no longer needs help.

Acknowledgements: I thank my supervisor, Eric Ruppert for his great guidance, advice and support. Trevor Brown for providing lots of help and code for the experiments, Michael L. Scott for giving us access to his multicore machines and the anonymous referees for their constructive comments.

References

1. Afek, Y., Kaplan, H., Korenfeld, B., Morrison, A., Tarjan, R.E.: CBTtree: a practical concurrent self-adjusting search tree. In: Proceedings of the 26th International Symposium on Distributed Computing, pp. 1–15 (2012)
2. Arbel-Raviv, M., Brown, T.: POSTER: reuse, don’t recyle: transforming algorithms that throw away descriptors. In: Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Austin, TX, USA, 4–8 Feb 2017, pp. 429–430 (2017)
3. Aref, W.G., Ilyas, I.F.: SP-GiST: an extensible database index for supporting space partitioning trees. J. Intell. Inf. Syst. 17(2–3), 215–240 (2001)
4. Barnes, G.: A method for implementing lock-free shared-data structures. In: Proceedings ACM Symposium on Parallel Algorithms and Architectures, pp. 261–270 (1993)
5. Bieganski, P., Riedl, J., Carlis, J.V., Retzel, E.F.: Generalized suffix trees for biological sequence data: applications and implementation. In: Proceedings of the 27th Hawaii International Conference on System Sciences, pp. 35–44 (1994)
6. Braginsky, A., Petrank, E.: A lock-free B+tree. In: Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures, pp. 58–67 (2012)
7. Bronson, N.G., Casper, J., Chafi, H., Olukotun, K.: A practical concurrent binary search tree. In: Proceedings ACM Symposium on Principles and Practice of Parallel Programming, pp. 257–268 (2010)
8. Brown, T., Ellen, F., Ruppert, E.: Pragmatic primitives for non-blocking data structures. In: Proceedings of the 32nd ACM Symposium on Principles of Distributed Computing, PODC’13, pp. 13–22 (2013)
9. Brown, T., Ellen, F., Ruppert, E.: A general technique for non-blocking trees. In: Proceedings of the 19th ACM Symposium on Principles and Practice of Parallel Programming, PPOPP’14, pp. 329–342 (2014)
10. Brown, T., Helga, J.: Non-blocking k-ary search trees. In: Proceedings International Conference on Principles of Distributed Systems, pp. 207–221 (2011)
11. Cederman, D., Tsigas, P.: Supporting lock-free composition of concurrent data objects: moving data between containers. IEEE Trans. Comput. 62(9), 1866–1878 (2013)
12. Chatterjee, B., Nguyen, N., Tsigas, P.: Efficient lock-free binary search trees. In: Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing, PODC’14, pp. 322–331 (2014)
13. Drachslser, D., Vecvev, M., Yahav, E.: Practical concurrent binary search trees via logical ordering. In: Proceedings of the 19th ACM Symposium on Principles and Practice of Parallel Programming, PPOPP’14, pp. 343–356 (2014)
14. Ellen, F., Fatourou, P., Kosmas, E., Milani, A., Travers, C.: Universal constructions that ensure disjoint-access parallelism and wait-freedom. Distrib. Comput. 29(4), 251–277 (2016)
15. Ellen, F., Fatourou, P., Ruppert, E., van Breugel, F.: Non-blocking binary search trees. In: Proceedings of the 29th ACM Symposium on Principles of Distributed Computing, pp. 131–140 (2010)
16. Fomitchev, M., Ruppert, E.: Lock-free linked lists and skip lists. In: Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing, pp. 50–59 (2004)
17. Fraser, K.: Practical lock freedom. Ph.D. thesis, Cambridge University Computer Laboratory (2003)
18. Gan, M., Zhang, M., Wang, S.: Extended negative association rules and the corresponding mining algorithm. In: Proceedings of the 4th International Conference on Advances in Machine Learning and Cybernetics, pp. 159–168 (2006)
19. Goodchild, M.F.: Geographic information systems and science: today and tomorrow. Ann. GIS 15(1), 3–9 (2009)
20. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: Mibench: a free, commercially representative embedded benchmark suite. In: Proceedings of the IEEE International Workshop Workload Characterization, pp. 3–14 (2001)
21. Harris, T., Larus, J., Rajwar, R.: Transactional Memory, 2nd edn. Morgan and Claypool Publishers, San Rafael (2010)
22. Hart, T.E., McKenney, P.E., Brown, A.D., Walpole, J.: Performance of memory reclamation for lockless synchronization. J. Parallel Distrib. Comput. 67(12), 1270–1285 (2007)
23. Herlihy, M.P., Wing, J.M.: Linearity: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (1990)
24. Howley, S.V., Jones, J.: A non-blocking internal binary search tree. In: Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures, pp. 161–171 (2012)
25. Morrison, D.R.: PATRICIA—practical algorithm to retrieve information coded in alphanumeric. J. ACM 15(4), 514–534 (1968)
26. Natarajan, A., Mittal, N.: Fast concurrent lock-free binary search trees. In: Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP’14, pp. 317–328 (2014)
27. Natarajan, A., Savoie, L., Mittal, N.: Concurrent wait-free red black trees. In: Proceedings of 15th International Symposium on Stabilization, Safety, and Security of Distributed Systems, SSS’13, pp. 45–60 (2013)
28. Oshman, R., Shavit, N.: The skiptrie: low-depth concurrent search without rebalancing. In: Proceedings of 32nd ACM Symposium on Principles of Distributed Computing, pp. 23–32 (2013)
29. Pietracaprina, A., Zandolin, D.: Mining frequent itemsets using Patricia tries. In: Proceedings of the ICDM 2003 Workshop on Frequent Itemset Mining Implementations (2003). (Available as
30. Prokopec, A., Bronson, N.G., Bagwell, P., Odersky, M.: Concurrent tries with efficient non-blocking snapshots. In: Proceedings ACM Symposium on Principles and Practice of Parallel Programming, pp. 151–160 (2012)

31. Ramachandran, A., Mittal, N.: A fast lock-free internal binary search tree. In: Proceedings of the 2015 International Conference on Distributed Computing and Networking, ICDCN’15, pp. 37:1–37:10 (2015)

32. Shafiei, N.: Non-blocking Patricia tries with replace operations. arXiv:1303.3626 (2012)

33. Shafiei, N.: Non-blocking Patricia tries with replace operations. In: Proceedings of the 33rd International Conference on Distributed Computing Systems, ICDCS’13, pp. 216–225 (2013)

34. Shavit, N., Touitou, D.: Software transactional memory. Distrib. Comput. 10(2), 99–116 (1997)

35. Sundell, H., Tsigas, P.: Scalable and lock-free concurrent dictionaries. In: Proceedings of the ACM Symposium on Applied Computing, pp. 1438–1445 (2004)

36. Tsay, J.-J., Li, H.-C.: Lock-free concurrent tree structures for multiprocessor systems. In: Proceedings of the International Conference on Parallel and Distributed Systems, pp. 544–549 (1994)

37. Zijlstra, P.: Concurrent pagecache. In: Linux Symposium, p. 311 (2007)

Publisher’s Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.