A Scalable Linearizable Multi-Index Table

Gali Sheffi
Yahoo research, Israel
gsheffi@oath.com

Guy Golan-Gueta
VMWare research, Israel
ggolangueta@vmware.com

Erez Petrank
Technion, Israel
erez@cs.technion.ac.il

***Abstract*—Concurrent data structures typically index data using a single primary key and provide fast atomic access to data associated with a given key value. However, it is often required to atomically access information via multiple primary and secondary keys, and even through additional properties that do not naturally represent keys for the given data. We present lock-free and lock-based algorithms of a table with multiple indexing, supporting linearizable inserts, deletes, and retrieve operations. We have implemented Java versions of our algorithms and evaluated their performance on a multi-core machine. The results show that the proposed table implementations are scalable and more efficient than any existing available alternative for in-memory realizations of a multi-index table.**

I. Introduction

The rapid deployment of highly concurrent machines has resulted in the acute need for concurrent algorithms and their supporting concurrent data structures. Concurrent data structures are designed to utilize all available cores in order to achieve faster performance. The literature contains plenty of designs for concurrent data structures that implement sets (or maps). Typically, these data structures consider a key and data associated with it as an item. Such an item can be inserted into the data structure, deleted from it, and the item can be quickly retrieved using its key. However, in practice, data sometimes contains various different keys [1], [2]. For example, a student may have an identity number (such as an S.S.N.), a driving license number, a name, a phone, an address, etc., and it is often useful to be able to access the student's record using any of his properties. If a student record is kept in a data structure using the identity number as the key, then it would be costly (if at all possible) to locate a student record according to his phone number. There are two common solutions for this problem (i.e, solutions that provide concurrent multi-indexing): (1) STM (usually not used in practice), and (2) ad hoc combinations of several concurrent maps (in many cases, such combinations do not provide the wanted result. see [3]).

We present constructions of a concurrent table. The table has entries that can be quickly accessed via their primary key but also via secondary keys and properties that would not normally be considered as keys. For example, we may ask if there is a student in the class who lives on 5th Ave. Our constructions are linearizable, meaning that an item is inserted to (or removed from) the table simultaneously through all its indices. We propose lock-free and lock-based constructions and compare them against each other, against a simple construction that employs coarse-grained locking and against a software transactional memory (STM) table.

In a sequential setting, the solution to this problem is simple: employ several data structures, one to index each relevant property. This way, an item can be quickly found using any of the indexed fields. While such a solution is easy to implement in a sequential environment, it is not simple to make it correct, i.e., linearizable [4], when operations are executed on the table concurrently. Linearizability means, for example, that the existence of an item in the table does not depend on which index is used to search for it. The operation of inserting (or removing) elements simultaneously and instantly into (or from) several concurrent data structures brings up interesting algorithmic challenges.

In the context of a table, it is useful to deal with standard unique keys but also with non-unique properties of the item. For example, if we try to insert an item to the table with an identity number that already exists in the table, then the insertion should fail. The item already exists in the table. But inserting an item with an address that already exists in the table is fine. There could be two items in the table that represent two students who live in the same building. Interestingly, we found out that even the *contains* operation is not trivial to implement for a property that is non-unique.

On the practical level, while using several indices to organize data is often needed, existing solutions are either very heavy (in the sense that they provide much additional functionality at a cost) or very slow (because they do not support secondary keys). Databases often allow multiple indexing into the data, but their optimizations are mainly focused on disk operations. In particular, they are not designed to provide efficient in-memory concurrency for multi-core machines (as in concurrent data structures). The concurrent table proposed in this paper is a simple concurrent structure that supports multiple indices, but is meant to hold a fast and scalable in-memory table, which quickly locates an item given any of its keys or properties.

A different practical alternative is to use a concurrent table that is indexed only by a single primary key. Many concurrent data structures that implement a key-value store can be used to implement such a table (e.g. [5]). However, finding an item given a secondary key requires traversing the table. This is not only inefficient, but it is not simple to implement a linearizable *contains* operation without holding a coarse-grained lock on the table during the entire traversal. One alternative to using a coarse-grained lock is to employ a snapshot mechanism, (e.g., [6], [7]). But that would still require a costly traversal of the table.

II. Preliminaries

Our model for concurrent multi-threaded computation follows the linearizability model of [4]. In particular, we assume an asynchronous shared memory system where a finite set of deterministic threads communicate by executing atomic operations on some finite number of shared variables.

A. Table Terminology

A *table* is a set of n-tuples $\langle e_1, \ldots, e_n \rangle$ where all tuples have the same number of elements. A table T has $n = N_T$ fields (given T, N_T is a constant), denoted by $f_1, \ldots, f_{N_T}$, where each field represents a different element of the tuples. Each field f_i can be either *unique* or *non-unique*. If f_i is unique, then any two different tuples $\langle e_1, \ldots, e_n \rangle$ and $\langle e'_1, \ldots, e'_n \rangle$ in T satisfy $e_i \neq e'_i$. Otherwise (i.e., f_i is non-unique), T may contain two different tuples $\langle e_1, \ldots, e_n \rangle$ and $\langle e'_1, \ldots, e'_n \rangle$ such that $e_i = e'_i$.

Each table field may be of a specific type (such as Integer, Float or String). We assume that there exists a total order $<$ of all possible elements such that for any two elements $e_i \neq e'_i$ either $e_i < e'_i$ or $e'_i < e_i$. We also assume that there exist two special elements $-\infty$ and ∞. The elements $-\infty$ and ∞ represent the minimal and the maximal possible element that can be stored in a tuple.

B. Table Operations

The proposed table supports the *add*, *remove* and *retrieve* operations. The *add* operation receives a tuple $\langle e_1, \ldots, e_n \rangle$ and adds it to the table if and only if for every unique field number i the table does not contain a tuple $\langle e'_1, \ldots, e'_n \rangle$ such that $e_i = e'_i$. The *remove* operation receives a value val and a unique field number i, and tries to remove a tuple $\langle e_1, \ldots, e_n \rangle$ such that $e_i = val$ from the table. It will fail is no such tuple exists in the table. The *retrieve* operation receives a value val and a field number i and returns all tuples $\langle e_1, \ldots e_n \rangle$ that satisfy $e_i = val$. A simpler *contains* operation can be implemented by calling the *retrieve* operation, and returning *true* if *retrieve* returns at least one tuple, and *false* otherwise.

III. The Algorithm

We chose to implement multiple indexing of a concurrent table by maintaining multiple data structures, each indexing a different key (or property) of an item. In order to simplify the design, we chose simple concurrent (ordered) linked-lists to maintain the indexed set (or multi-set). Next, in order to speed up the access time, we built skip-lists on top of the linked-lists. The linked-lists are carefully designed to maintain correctness and linearizability and make sure that items appear to be inserted to or removed from all indices at once. Being a member of the table means being a member of all linked-lists. Thus, an item can either be in all linked-lists, or in none. The skip-lists are implemented in a more relaxed manner and are only used to provide fast indexing. They do not need to always properly hold an updated view of all items in the table.

In this section we describe the algorithm. We start by defining some variants of the CAS operation, which are widely used in our algorithms (in Section III-A). We then present the data structure we use (in Section III-B), and continue with a detailed description of the lock-free table algorithm (in Section III-D), and a short description of the lock-based variant (in Section III-E). Finally, we explain the additional fast skip-list indexing (in Section III-F). We put the formal proof of the lock-free variant in Appendix A, and the full description and formal proof of the lock-based variant in Appendix B.

A. Extending the CAS Operation

The *compareAndSwap* (CAS) operation is a standard synchronization instruction widely available by hardware. It receives three inputs: $addr$, exp and new, and it executes the following atomically. If the memory content at location $addr$ equals the input value exp, then it assigns new as the new content of memory location $addr$. Otherwise, the memory is not changed. The CAS operation returns true if the content was indeed replaced and false otherwise. The literature contains many CAS extensions (see [8]), enabling the comparison and swap of two or more memory locations.

In our lock-free implementation, we often use the CAS operation as described above (for example, see line 31 or 46 in Figure 3) but also in an extended manner in which a pointer has an additional associated flag by which it can be atomically marked. Marked atomic pointers were first used in [9] and later in many other papers (see [5], [10]). The extended CAS atomically modifies a reference and an associated boolean flag. It receives five input parameters, $addr$, $expPtr$, $newPtr$, $expFlag$ and $newFlag$. It compares the contents of the pointer and boolean flag of the reference object in $addr$ to $expPtr$ and $expFlag$ (respectively), and if both are equal, it modifies them to $newPtr$ and $newFlag$. These two comparisons and two assignments are executed atomically, returning true when successful and false otherwise. For example, this appears in line 48 or 49 in Figure 3. This extended CAS is supported by existing programming environments such as the Java run-time library.

Practically, this can be done in C by using the least-significant-bit of the pointer (which is always zeroed in practice). Thus, one needs to modify a single word, which can be reduced to the simple CAS. Or, in Java, this can be done by allocating a new object that holds the two fields, and modifying the pointer to it in a single atomic swap in order to execute a modification of the two fields simultaneously.

B. Internal Data Structure

Our table organizes the stored tuples as a collection of sorted linked-lists. In each list L_i the tuples are sorted according to the i-th field.

We call the object that represents a tuple *a record*. Each record consists of (1) an array $data$ with N_T entries, (2) an array $next$ with N_T entries, and (3) a $status$ flag. The $data$ array stores the represented tuple. Specifically, if record R represents the tuple $\langle e_1, \ldots, e_n \rangle$ then $R.data[i] = e_i$ for any $1 \leq i \leq n$. The $next$ array contains pointers that maintain the internal lists of the table. if $L_i = R_1 \rightarrow R_2 \rightarrow \ldots \rightarrow R_{k+1}$

then $R_j.next[i]$ points to R_{j+1} for every $1 \leq j \leq k$. In addition to the pointer, each $next[i]$ field contains a $marked$ boolean flag. The pointer and boolean flag can be read and updated simultaneously and atomically, using a single CAS execution (for more details, see Section III-A).

In order to implement a simultaneous insertion (or deletion) of a record into all the linked-lists at once, we use the *status* field (a more general *status* field is being used in [11]). The status of a record is set to *Pending* while it is being inserted into all the lists. While the status is *Pending*, the algorithm does not consider this record as part of the table, even if it has already been inserted into some of the lists. Upon completion of insertions to all linked-lists, the record's status is modified to *InTable*, and it is exactly at this point that the record becomes a member of the table (this is considered as the logical insertion of the record into the table). To remove a record from all lists simultaneously, it is enough to mark the status field as *Removed* and from that point and on the algorithm considers it deleted. Deletion proceeds by unlinking the record from all the linked-lists.

If the table has a unique field, then a thread may fail to insert a record because its relevant property already appears in the list of that unique field. This happens also in the standard case of a single-key list, and the failed attempt is not even visible to the other threads. But in the case of a table we need to handle the case of a successful insertion into one list and then a failed insertion to a second list. For example, a student may have changed a driving license (e.g., due to moving to a different state), but still have the same identity number. In this case, it may happen that we attempt to insert a record for this student while an old record representing him is present in the table. Our algorithm will succeed to insert the record using the first index (the driving license number) because the old record of this student has a different driving license number. But when it attempts to insert this record into the second index (the identity number), it will discover that the student already appears in the table and then the insertion would return failure. When this happens, we mark the status of the inserted record as *Failed* and execute a protocol that eventually removes this record from the list.

We use two special records, *head* and *tail*, which represent the tuples $\langle -\infty, \ldots, -\infty \rangle$ and $\langle \infty, \ldots, \infty \rangle$, respectively. Each list L_i starts with the *head* record and ends with the *tail* record. The records *head* and *tail* are not considered table items, and therefore they are never added to or removed from the table. Both *head* and *tail*'s statuses are always set to *InTable*.

In Figure 1 we present an example of a table. The arrows represent the next[] references, and in the i-th list, records are sorted according to their i-th field. Notice that the table does not contain two tuples with the same unique field, and that if a record's status is not *InTable*, it is not necessarily fully linked.

C. Algorithmic Challenges

We propose a lock-free and fine-grained locked-based implementation of the concurrent table using the ideas raised in Section III-B. Let us highlight some algorithmic challenges that must be handled.

Interestingly, for the lock-free algorithm, the *retrieve* (or even a *contains*) operation is not trivial when one searches for a property that is non-unique. Note that it suffices to retrieve one record with the required property, and this is still non-trivial. Of course, if we find a record with the required property in the table, then we simply retrieve this record. The problem is that if we do not find a record with this property, we cannot determine in a linearizable manner that it is not in the table. During a search, it is possible that a record with a given property is continuously present in the table, but the *retrieve* method does not find any record that matches this property. This can happen, for example, in the following scenario. When the search finds the first record with the given property, that record has a pending status. It has not yet been added to the table, but it is in the process. Next, the search goes to the next record and discovers that its status is removed. If there are no more records with this property, the search may deduce erroneously that this property is not in the table. However, it is possible that before the second record was removed, the first record has been completely added to the table. And so a record with this property has been part of the table for the entire execution of the search.

The problem is that the traversal of records with the required property is not atomic. This problem resembles the challenge of obtaining a snapshot of a lock-free data structure (which is challenging) and there exist in the literature various ways to deal with it in a lock-free or even wait-free manner (e.g., [7]). We chose a simple and efficient lock-free solution for this problem. First, we maintain an invariant that a new record with a given property is always added before all existing records with that property. Given this invariant, it is possible to traverse the sublist (of records with the given property) repeatedly until we find two equal views of it. And if these two views do not contain a record that is properly in the table, then we can determine that there is a point in the execution in which no record with this property was present. Whenever we do not obtain the same view, another operation has made progress and so lock-freedom is maintained.

Moving to the proposed fine-grained *lock-based* design, we would like to lock a record (or two) in each list in order to execute an insertion (or a deletion) of a record. A natural locking strategy would be to keep one lock for each record and lock the needed records in a disciplined order one by one starting from the one needed for the first linked-list until the one needed for the last linked-list. Interestingly, this creates a potential deadlock.

To see that, one needs to notice that the order of records in the different linked-lists may differ. So a record A can precede record B in the first list, but record B can precede A in the second list. Therefore, when two insertions are executed

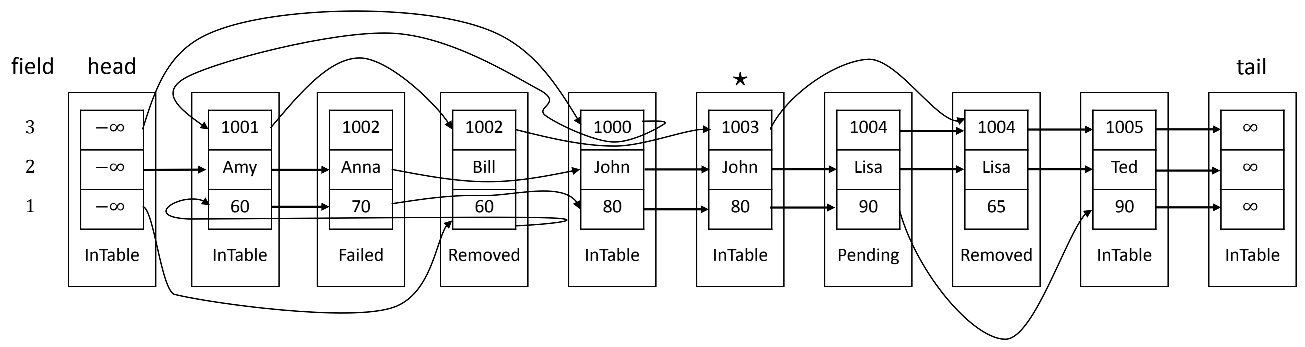

Fig. 1. A table with three fields ($N_T = 3$). The first and second fields are non-unique and the third field is unique. Besides the *head* and *tail* sentinels, there are four records with an *InTable* status (i.e. the table contains four tuples)

concurrently, we can see the following problem. The first thread acquires a lock to record A in order to execute the insertion to the first linked-list, and then the first thread waits to acquire the lock of record B in order to execute the insert to the second linked list. At the same time, a second thread locks B in order to insert a different record to the first linked-list and then waits for the lock of record A in order to insert a record to the second linked list, and we get a deadlock.

To avoid such deadlock, we allocate a lock per record per property. Now, we obtain an order on all locks that we can use to maintain discipline in the order of acquiring locks. In this order, locks of a preceding linked-list precede all locks of a subsequent linked-list and locks in between a linked-list are ordered according to the order of their records in that list. Using this ordering, a thread that waits for a lock in the second linked-list is independent of other threads acquiring locks in a the first linked-list or in any other linked-list, and dead-lock is prevented.

D. The Lock-Free Table

In this section we describe the lock-free table design. While all operations present interesting algorithmic challenges, one unexpected challenge was to obtain linearizability for *retrieve* (at the end of this section). We say that a record is logically in the table if it is physically in all internal linked-lists and its status is *InTable* (for the formal definition, see Definition **??** in Section **??**). This means that the successful insertion of a new record into the table includes adding it to all lists and then changing its status from *Pending* to *InTable*. The removal of a record includes changing its status (does not necessarily include its physical removal from the lists) from *InTable* to *Removed*.

The find method

Before describing the *add*, *remove* and *retrieve* operations, we are going to describe the *find* auxiliary method to be used by *add* and *remove* (shown in Figure 2). The *find* method receives a value val and a field number i and returns two records, $pred$ and $curr$, such that (1) $pred.data[i] < val$, (2) $curr.data[i] \geq val$ and (3) $curr$ is $pred$'s successor in L_i.

```
1:  find(val,i)
2:      retry:                                        ▷ Starting a new search
3:      pred ← head
4:      curr ← pred.next[i]
5:      while (true) do                 ▷ While desired records not yet found
6:          succ, marked ← curr.next[i]
7:          while (marked) do                  ▷ While curr is logically removed
8:              snip ← pred.next[i].CAS(curr, succ, false, false)
9:              if (!snip)                    ▷ If the physical removal failed
10:                 goto retry
11:             curr ← pred.next[i]
12:             succ, marked ← curr.next[i]
13:         if (curr.data[i] ≥ val)
14:             return pred, curr                  ▷ Return the current records
15:         pred ← curr
16:         curr ← succ
```

Fig. 2. The lock-free *find* method

The method starts its traversal from *head*. The $pred$ and $curr$ references are advanced throughout L_i, until the relevant two records are found and returned (line 14).

In addition to finding the described records, the *find* method also helps physically removing records that have been logically removed (see [9]). Records are removed from the table during unsuccessful *add* and successful *remove* executions.

A record R is physically removed from L_i only after its $next[i]$ pointer is marked. The *find* method is the only one in charge of physical removals, and it executes them in the following way: Each time $curr$ is advanced, the method checks if $curr$'s $next[i]$ pointer is marked. If $curr$'s $next[i]$ pointer is not marked, the traversal continues. Otherwise, the method attempts to physically remove $curr$ from L_i using a CAS operation (line 8). It does it by setting $pred$'s successor in L_i to $succ$. If the execution is successful, then $pred$'s successor in L_i becomes $succ$, meaning $curr$ is no longer in L_i. Otherwise, The traversal starts again from *head*.

The add operation

The *add* operation (shown in Figure 3) receives a tuple tup, creates a new corresponding record R with the data of tup and a *Pending* status and tries to add R to the table. In order to enable lock-freedom, the adding procedure is divided into smaller steps, allowing other executing threads to help

```
17: add(tup)
18:     newRecord ← makerecord(tup)                    ▷ Creating the new record
19:     helpAdding(newRecord,1)                        ▷ 1 stands for the first list
20:     if (newRecord.status == Failed)                ▷ Insertion was unsuccessful
21:         for (i=1 to $N_T$) do
22:             succ, marked ← newRecord.next[i]
23:             while (!marked) do
24:                 newRecord.next[i].CAS(succ, succ, false, true)
25:                 succ, marked ← newRecord.next[i]
26:         return false
27:     else   return true                             ▷ Insertion was successful

28: helpAdding(newRecord, m)
29:     for (i=m to $N_T$) do
30:         helpAddingField(newRecord, i)
31:     newRecord.status.CAS(Pending, InTable)         ▷ Logical insertion

32: helpAddingField(newRecord, i)
33:     while (newRecord.status == Pending) do
34:         succ ← newRecord.next[i]                   ▷ For a later CAS
35:         pred, curr ← find(newRecord.data[i], i)
36:         tmp ← curr
37:         while (tmp.data[i] == newRecord.data[i]) do
38:             if (tmp == newRecord)
39:                 return                             ▷ The record is already in the list
40:             tmp ← tmp.next[i]
41:         if (Unique(i))
42:             if (curr.data[i] == newRecord.data[i])
43:                 if (curr.status == Pending)
44:                     helpAdding(curr, i+1)          ▷ Help adding curr
45:                 if (curr.status == InTable)        ▷ Insertion failed
46:                     newRecord.status.CAS(Pending, Failed)
47:                     break
48:         if (newRecord.next[i].CAS(succ, curr, false, false))
49:             if (pred.next[i].CAS(curr, newRecord, false, false))
50:                 return                             ▷ The record was added to the list
```

Fig. 3. The lock-free *add* operation

the insertion (if it prevents them from finishing their own insertions). After creating R, it is added to each list (according to their order) using the *helpAdding* method. When returning from the *helpAdding* method, if the insertion was successful, R's status is either *InTable* or *Removed* (The later is possible if another thread has removed it after its insertion and before this check of its status). In this case, the whole operation is considered successful and it returns $true$ (line 27). If the insertion was unsuccessful, R's status is set to *Failed*, and all of R's $next$ pointers are marked (lines 21-25), for a physical removal during a later *find* execution.

The *helpAdding* method (line 28) receives a record R and a field number $1 \leq m \leq N_T$, and it tries to insert R into $L_m, L_{m+1} \ldots, L_{N_T}$, according to their order. This allows helping to insert a record given that it already has been inserted into the first $m-1$ lists. After finishing the physical adding procedure (the loop in lines 29-30), the executing thread tries to change R's status from *Pending* to *InTable* using a CAS operation (line 31).

The *helpAddingField* method is in charge of the insertion of the new record R into a list L_i, where i and R are provided as parameters. The method returns immediately when finding out that R's status is not *Pending* anymore, and keeps trying to add it otherwise.

Every insertion trial starts by saving R's current successor in L_i in the local variable $succ$ for a later use (line 34). Next, The *find* method is called in order to locate R in L_i (line 35). Our next concern is making sure that R has not been already inserted into L_i, before trying to physically insert it. In order to make sure that R has not been already inserted into L_i, we go over all records R' satisfying $R'.data[i] = R.data[i]$ that follow $curr$ in L_i (lines 36-40), and stop the current insertion procedure if one of them is R (line 39).

Now, if i is a unique field number, adding R to L_i may cause a violation of the uniqueness property. Therefore, before we try to add R to L_i, we have to make sure that there does not exist any other record with R's i-th property, which is logically in the table. If $curr$'s i-th property is not equal to R's i-th property (checked in line 42), then we can continue in the adding procedure. Otherwise, we need to make sure that either $curr$ is not logically in the table (and we can continue in our adding procedure) or is logically in the table (and the adding procedure must fail).

If $curr$'s status is *Pending*, then $curr$ has not been fully added to all lists, and we are not yet able to determine whether its insertion procedure is going to succeed or fail. In this case, we help adding $curr$ to $L_{i+1}, \ldots, L_{N_T}$ (line 44), since it is guaranteed that it has already been inserted into $L_1, \ldots, L_i$. Now, if $curr$'s insertion has been successful, and its status has been set to *InTable*, R's insertion would violate the table's uniqueness property. Therefore, in this case, we set R's status to *Failed*, which notifies all helping threads that its insertion is no longer relevant (line 46). Otherwise, if either $curr$'s insertion has not been successful (its status is *Failed*) or it had been successful but $curr$ has already been logically removed thereafter (its status is *Removed*), we can proceed with the adding procedure.

At this stage, regardless of i being a unique or a non-unique field number, we try to insert R into L_i. We do it by making $curr$ be its successor (line 48) and make $pred$ be its predecessor (line 49) in L_i. If both CAS executions are successful then R is successfully inserted into L_i. Otherwise, we start a new insertion trial.

```
51: remove(val, i)
52:     pred, victim ← find(val, i)                    ▷ We do not need pred
53:     if (victim.data[i] ≠ val)
54:         return false                               ▷ There is no such item
55:     else if (!victim.status.CAS(InTable, Removed))
56:         return false                 ▷ The record has already been removed
57:     for (j=1 to $N_T$) do
58:         succ, marked ← victim.next[j]
59:         while (!marked) do
60:             victim.next[j].CAS(succ, succ, false, true)
61:             succ, marked ← victim.next[j]
62:     return true                        ▷ The record was logically removed
```

Fig. 4. The lock-free *remove* operation

The remove operation

The *remove* operation (shown in Figure 4) receives a value val and a unique field number i, and tries to logically remove a record R such that $R.data[i] = val$. In line 52, the *remove* operation calls the *find* method in order to determine whether there exists a record R which is logically in the table and for which $R.data[i] = val$. The potential victim for removal is assigned into the $victim$ local variable. If $victim$'s i-th property is not val, then it is guaranteed that there is no other relevant record, and the operation fails (line 54). Otherwise, if the attempt to logically remove $victim$ from the table (by setting $victim$'s status to *Removed*) fails, the operation is considered to be a failure, and it returns in line 56.

If the trial to logically remove $victim$ from the table is successful, then the operation is considered successful. Before returning $true$ in line 62, we mark all of the victim's $next$ pointers, for its future physical removals by *find* executions.

```
63: retrieve(val, i)
64:     while (true) do
65:         tmpSet ← ∅            ▷ For saving pointers to the relevant records
66:         curr ← head                                    ▷ First traversal
67:         while (curr.data[i] < val) do
68:             curr ← curr.next[i]
69:         if (curr.data[i] > val)
70:             return tmpSet                  ▷ There are no relevant records
71:         first ← curr                   ▷ Saving the first relevant record
72:         while (curr.data[i] == val) do
73:             status ← curr.status
74:             if ((status == InTable) or (status == Pending))
75:                 tmpSet ← tmpSet ∪ {⟨curr, status⟩}
76:             curr ← curr.next[i]
77:         curr ← head                                   ▷ Second traversal
78:         while (curr.data[i] < val) do
79:             curr ← curr.next[i]
80:         if (first == curr)                  ▷ No new records were added
81:             tuples ← ∅                        ▷ For returning the tuples
82:             valid ← true
83:             for (⟨R, status⟩ in tmpSet) do
84:                 if (R.status == status)
85:                     if (status == InTable)
86:                         tuples ← tuples ∪ {R.data}
87:                 else
88:                     valid ← false
89:                     break
90:             if (valid)
91:                 return tuples
```

Fig. 5. The lock-free *retrieve* operation

The retrieve operation

The *retrieve* operation (shown in Figure 5) receives a value val and a field number i, and returns the set of all tuples $\langle e_1, \ldots e_n \rangle$ satisfying $e_i = val$ (meaning all arrays $R.data$ for which R is logically in the table and $R.data[i] = val$).

Interestingly, for the lock-free algorithm, the *retrieve* (or even a *contains*) operation is not trivial when one searches for a property that is non-unique. Note that it remains non-trivial even if we only need to retrieve one record with the required property. The problem is that if we do not find a record with the required property, we cannot determine in a linearizable manner that it is not in the table, because a traversal may miss it due to lack of atomicity. This can happen, for example, in the following scenario. The search first encounters a record with the given property and a *Pending* status. Namely, this record is in the process of being added to the table. Next, the search finds that the next record's status is *Removed*. If there are no more records with this property, the search may deduce erroneously that this property is not in the table. However, it is possible that before the second record was removed, the insertion of the first record has been completed. Namely, a record with this property has been in the table for the entire execution of the search.

The problem is that the traversal of records with the required property is not atomic. This problem resembles the challenge of obtaining a snapshot of a lock-free data structure (which is challenging) and there exist in the literature various ways to deal with it in a lock-free or even wait-free manner (e.g., [7]). We chose a simple lock-free solution for this problem. First, we maintain an invariant that a new record with a given property is always added before all existing records with that property (notice that when a new record is inserted into the i-th list in line 49 in Figure 3, its predecessor's i-th property is smaller). Given this invariant, it is possible to traverse the sublist (of records with the given property) repeatedly until we find two equal views of this list. Now, if these two views do not contain a record that is properly in the table, then we can determine that there is a point in the execution in which no record with this property was present. Whenever we do not obtain the same view, another operation has made progress and so lock-freedom is maintained.

The operation starts by traversing L_i to find the first record for which $data[i] \geq val$. If there does not exist a record with val as its i-th property, meaning the first record found satisfies $curr.data[i] > val$, then the set returned is empty (line 70). If a relevant record is found during this traversal, a pointer to the first relevant record is saved in the local variable $first$ (line 71). After saving it, pointers to all of the following relevant records are also saved, together with their statuses (lines 72-76), in the $tmpSet$ set. Relevant records are records R such that $R.data[i] = val$ and whose status is either *Pending* or *InTable*.

After all the relevant seen records are saved, the operation starts a second traversal (line 77) and checks whether the first record R satisfying $R.data[i] = val$ is the same one as in the first traversal (line 80), saved in the $first$ local variable. If it is not, then the whole procedure starts again (from line 64). Otherwise, it is guaranteed that the snapshot saved in $tmpSet$ is a correct snapshot of all records in the table that have val in their i-th field, and this snapshot can be linearized when the second traversal starts.

After making sure that there are no new relevant records in L_i, the operation goes over the saved pairs of records and statuses $\langle R, status \rangle$, and checks whether $R.status$ is still $status$ (lines 83-89). If all pairs satisfy this condition, all the relevant tuples (represented by saved records with an *InTable* status) are added to the $tuples$ set (line 86), and returned as output (line 91).

E. The Lock-Based Table

Moving to the proposed fine-grained *lock-based* design, we use the same overall strategy of maintaining a collection of linked-lists to provide the multiple indexing, but we use lists with fine-grained locking. We relegate the detailed description and proof of the lock-based table variant to [12]. But let us highlight one potential pitfall. To execute an *add* or a *remove* of a record, we would like to lock a record (or two) in each list. A natural locking strategy would be to keep one lock for each record and lock the needed records in a disciplined order one by one starting from those needed for the first linked-list until those needed for the last linked-list. Interestingly, this creates a potential deadlock.

To see that, one needs to notice that the order of records in the different linked-lists may differ. So a record A can precede record B in the first list, but record B can precede A in the second list. Therefore, when two insertions are executed concurrently, we can see the following problem. The first thread acquires a lock to record A in order to execute the insertion to the first linked-list, and then the first thread waits to acquire the lock of record B in order to execute the insert to the second linked list. At the same time, a second thread locks B in order to insert a different record to the first linked-list and then waits for the lock of record A in order to insert a record to the second linked list, and we get a deadlock.

To avoid such deadlock, we allocate a lock per record per property. Now, we obtain an order on all locks that we can use to maintain discipline in the order of acquiring locks. In this order, locks of a preceding linked-list precede all locks of a subsequent linked-list and locks in between a linked-list are ordered according to the order of their records in that list.

F. Adding a Fast Skip-List Indexing

We improve access to the table by using a skip-list index [13]. In the above sections, each table operation traverses at least one linked-list until reaching the desired record. Such linked-list traversals may yield poor performance. Moreover, in the lock-free version, traversals might restart due to unsuccessful physical removals of marked records. To improve performance, we employ an index mechanism which enables getting to a record's predecessor in a list faster (without traversing from the head of the list). For example, in an execution of *retrieve(*$1005, 3$*)* with the table of Figure 1, instead of starting from the *head* record, our index can be used to start traversing from the record which represents $\langle 80, \text{John}, 1003\rangle$ (marked with $\star$).

For each list L_i we use a linearizable skip-list S_i that serves as an index to the records in L_i. S_i represents a sequence of record pointers that are sorted by their i-th field. S_i supports operations *insert(R)* and *remove(R)* which are used to insert and remove record pointers; their implementation ensure that each pointer may appear at most once in S_i (i.e., S_i is a set). It also supports operation *getPrev(v)* that returns the latest record pointer R in S_i that satisfies $R.data[i] < v$. As noted in [13], such S_i can be realized by a standard linearizable skip list; in our work it is implemented as simple wrapper of Java's *ConcurrentSkipListSet*.

Using the index

In operations *find*(v, i) and *retrieve*(v, i), instead of starting the traversal from *head* we start from the record R returned by $S_i.getPrev(v)$. As described below, the content of S_i does not necessarily represent the exact content of the table. In particular, the returned record R may have already been removed from the table. In such a case we invoke $S_i.getPrev(R.data[i])$ and continue in the same manner until *getPrev* returns a valid record (i.e., *R.status* = *InTable*). This process will necessarily stop after a finite number of steps, since *head* cannot be removed from S_i.

Updating the index

Each S_i is updated in a way that attempts to (but does not always) ensure that S_i points to the valid records in the table. When the table is created, *head* is added to all S_i indexes. After completing a successful insertion (or deletion) of a R into (or from) the table, we add (or remove) R to all S_i indexes. Thus, S_i may point to a record that has already been deleted from the table. Furthermore, a removal of record R from the table can be fully executed before R is added to all indexes - in such a case, $S_i.remove(R)$ might be executed before $S_i.add(R)$, resulting in leaving R in S_i forever (even though it is not in the table anymore). To prevent such cases, after adding R to the indexes we check the status of R. If its status is not *InTable* then we explicitly remove R from all the indexes.

IV. Measurements

We compared the performance of our lock-free and lock-based table against each other, and against a global lock-based table and an STM table. All implementations use the skip-list optimization of Section III-F.

A. Methodology

Each experiment consists of 10 trials. A trial is a run in which each thread executes randomly chosen operations drawn from the workload distribution, on a table consisting of two unique fields and three non-unique fields. We ran four experiments.

In the first three experiments (presented in the first row of Figure 6), we used a workload distribution with 50% *retrieve*, 25% *add* and 25% *remove* operations. In the last three experiments, we used a workload distribution with 90% *retrieve*, 5% *add* and 5% *remove* operations.

For each workload distribution, we ran the same three tests, which differed in the key range size. In the first experiment, unique fields were selected uniformly at random from the range [0,255], in the second one, from the range [0,$10^4 - 1$], and in the third one, from the range [0,$10^6 - 1$]. For creating repetitions in the non-unique fields (which yields a realistic table content), they were selected uniformly at random from the range [0,63] for the first experiment, from the range

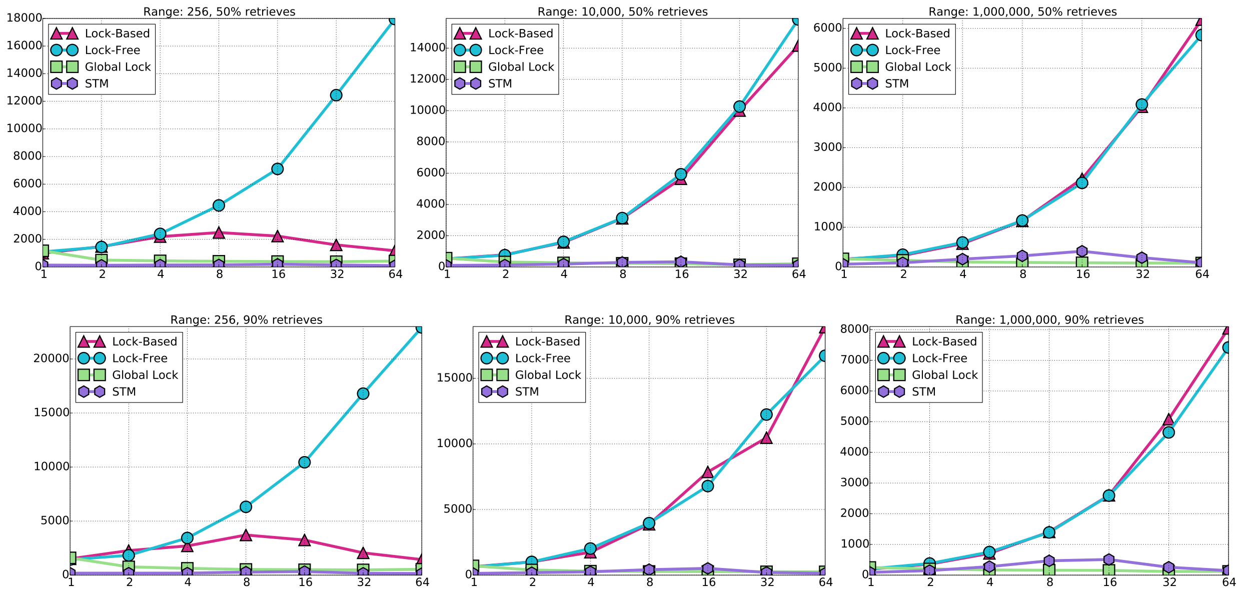

Fig. 6. Throughput graphs. The x-axis is the number of threads. The y-axis is the throughput operations per millisecond.

$[0,25 \cdot 10^2 - 1]$ for the second experiment, and from the range $[0,25 \cdot 10^4 - 1]$ for the third experiment. The table was initiated with 128 records in the first experiment, with $5 \cdot 10^3$ records in the second experiment, and with $5 \cdot 10^5$ records in the third experiment. The graphs present the average throughput (operations executed per millisecond) over all trials.

2 Intel Xeon E5-2686 v4 processors, each with 16 cores. Each core has 2 hardware threads (hyper-threading is enabled). All implementations are in Java. We used Java's *AtomicMarkableReference* for the lock-free version and Java's $ConcurrentSkipListSet$ (e.g. [14]) for the skip-list optimization. For our STM table, we adopt the TL2 [15] implementation from Synchrobench [16].

B. Results

As shown in Figure 6, for a small range of fields (when contention is high), our lock-free table is far-superior to the other versions and scales well. For large fields ranges, our lock-free and lock-based versions present a similar performance, and are far superior to the other two alternatives.

V. Related Work

Many sophisticated concurrent data structures (e.g.,see [10], [17]–[21]) have been developed and used in modern software systems. The *concurrent Map* is a notable data structure which is widely used in real life concurrent programs [3]. In this work, we extend the Map's capabilities by adding support for efficient thread-safe multiple indexes. We hope that our new data structure will practically extend the Map's usability in concurrent software; especially in programs where the same information is indexed and accessed via several different properties (e.g.,see [22], [23]).

Many (relational) database systems have built-in support for multiple primary and secondary indexes [1], [24]. Typically, database indexes can be correctly used in the presence of concurrency (e.g., by using transactions [24]). Our table data structure can be seen as an in-memory representation of the well-known database table. Our data structure is much more restricted than a standard database table (e.g., it does not support common database operations like *join* [1] and *snapshot* [6], and is not required to access hard disks). However, its restricted functionality enables creating efficient in-memory implementations which are optimized for modern multi-core machines.

Recently, several multi-index implementations were developed for NOSQL datastores (e.g., see [2], [25]–[29]). However, these implementations have been designed to work on several distributed machines, and are based on complex and expensive mechanisms for handling persistency and inter-machine synchronization. These multi-index implementations utilize a simple in-memory concurrency control which resembles global lock synchronization, and it is not clear how to convert them into effective in-memory data structures that can be used by multi-threaded applications.

A table with multi-index support can be simply realized via general-purpose software transactions [13], [16]. However, in contrast to our specialized table implementations, general transactions provide limited performance due to their high runtime overhead [16], [30], and their excessive thread contention [13].

References

[1] R. Ramakrishnan and J. Gehrke, *Database management systems (3. ed.)*. McGraw-Hill, 2003.

[2] A. Tai, M. Wei, M. J. Freedman, I. Abraham, and D. Malkhi, "Replex: A scalable, highly available multi-index data store," in *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016.*, A. Gulati and H. Weatherspoon, Eds. USENIX Association, 2016, pp. 337–350. [Online]. Available: https://www.usenix.org/conference/atc16/technical-sessions/presentation/tai

[3] O. Shacham, *Verifying atomicity of composed concurrent operations*. University of Tel-Aviv, 2012.

[4] M. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, 1990. [Online]. Available: http://doi.acm.org/10.1145/78969.78972

[5] J. Vu, "The art of multiprocessor programming by maurice herlihy and nir shavit," *ACM SIGSOFT Software Engineering Notes*, vol. 36, no. 5, pp. 52–53, 2011. [Online]. Available: http://doi.acm.org/10.1145/2020976.2021006

[6] G. Golan-Gueta, E. Bortnikov, E. Hillel, and I. Keidar, "Scaling concurrent log-structured data stores," in *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*, L. Réveillère, T. Harris, and M. Herlihy, Eds. ACM, 2015, pp. 32:1–32:14. [Online]. Available: http://doi.acm.org/10.1145/2741948.2741973

[7] E. Petrank and S. Timnat, "Lock-free data-structure iterators," in *Distributed Computing - 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings*, ser. Lecture Notes in Computer Science, Y. Afek, Ed., vol. 8205. Springer, 2013, pp. 224–238. [Online]. Available: https://doi.org/10.1007/978-3-642-41527-2_16

[8] T. L. Harris, K. Fraser, and I. A. Pratt, "A practical multi-word compare-and-swap operation," in *Distributed Computing, 16th International Conference, DISC 2002, Toulouse, France, October 28-30, 2002 Proceedings*, ser. Lecture Notes in Computer Science, D. Malkhi, Ed., vol. 2508. Springer, 2002, pp. 265–279. [Online]. Available: https://doi.org/10.1007/3-540-36108-1_18

[9] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *Distributed Computing, 15th International Conference, DISC 2001, Lisbon, Portugal, October 3-5, 2001, Proceedings*, ser. Lecture Notes in Computer Science, J. L. Welch, Ed., vol. 2180. Springer, 2001, pp. 300–314. [Online]. Available: https://doi.org/10.1007/3-540-45414-4_21

[10] S. Timnat, A. Braginsky, A. Kogan, and E. Petrank, "Wait-free linked-lists," in *Principles of Distributed Systems, 16th International Conference, OPODIS 2012, Rome, Italy, December 18-20, 2012. Proceedings*, ser. Lecture Notes in Computer Science, R. Baldoni, P. Flocchini, and B. Ravindran, Eds., vol. 7702. Springer, 2012, pp. 330–344. [Online]. Available: https://doi.org/10.1007/978-3-642-35476-2_23

[11] K. Fraser and T. L. Harris, "Concurrent programming without locks," *ACM Trans. Comput. Syst.*, vol. 25, no. 2, p. 5, 2007. [Online]. Available: http://doi.acm.org/10.1145/1233307.1233309

[12] "A scalable linearizable multi-index table." [Online]. Available: http://www.cs.technion.ac.il/~erez/Papers/table.pdf

[13] A. Spiegelman, G. Golan-Gueta, and I. Keidar, "Transactional data structure libraries," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, C. Krintz and E. Berger, Eds. ACM, 2016, pp. 682–696. [Online]. Available: http://doi.acm.org/10.1145/2908080.2908112

[14] "Java language home page," http://java.sun.com/. [Online]. Available: http://java.sun.com/

[15] D. Dice, O. Shalev, and N. Shavit, "Transactional locking II," in *Distributed Computing, 20th International Symposium, DISC 2006, Stockholm, Sweden, September 18-20, 2006, Proceedings*, ser. Lecture Notes in Computer Science, S. Dolev, Ed., vol. 4167. Springer, 2006, pp. 194–208. [Online]. Available: https://doi.org/10.1007/11864219_14

[16] V. Gramoli, "More than you ever wanted to know about synchronization: synchrobench, measuring the impact of the synchronization on concurrent algorithms," in *ACM SIGPLAN Notices*, vol. 50, no. 8. ACM, 2015, pp. 1–10.

[17] A. Braginsky, N. Cohen, and E. Petrank, "CBPQ: high performance lock-free priority queue," in *Euro-Par 2016: Parallel Processing - 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings*, ser. Lecture Notes in Computer Science, P. Dutot and D. Trystram, Eds., vol. 9833. Springer, 2016, pp. 460–474. [Online]. Available: https://doi.org/10.1007/978-3-319-43659-3_34

[18] A. Braginsky and E. Petrank, "A lock-free b+tree," in *24th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '12, Pittsburgh, PA, USA, June 25-27, 2012*, G. E. Blelloch and M. Herlihy, Eds. ACM, 2012, pp. 58–67. [Online]. Available: http://doi.acm.org/10.1145/2312005.2312016

[19] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel, "Non-blocking binary search trees," in *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010*, A. W. Richa and R. Guerraoui, Eds. ACM, 2010, pp. 131–140. [Online]. Available: http://doi.acm.org/10.1145/1835698.1835736

[20] A. Kogan and E. Petrank, "Wait-free queues with multiple enqueuers and dequeuers," in *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011*, C. Cascaval and P. Yew, Eds. ACM, 2011, pp. 223–234. [Online]. Available: http://doi.acm.org/10.1145/1941553.1941585

[21] S. Timnat and E. Petrank, "A practical wait-free simulation for lock-free data structures," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*, J. E. Moreira and J. R. Larus, Eds. ACM, 2014, pp. 357–368. [Online]. Available: http://doi.acm.org/10.1145/2555243.2555261

[22] S. Benchmarks, "Standard performance evaluation corporation," 2000.

[23] M. McCandless, E. Hatcher, and O. Gospodnetic, *Lucene in Action: Covers Apache Lucene 3.0*. Manning Publications Co., 2010.

[24] G. Weikum and G. Vossen, *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

[25] J. C. Anderson, J. Lehnardt, and N. Slater, *CouchDB - The Definitive Guide: Time to Relax*. O'Reilly, 2010. [Online]. Available: http://www.oreilly.de/catalog/9780596155896/index.html

[26] A. Cassandra, "Apache cassandra," *Website. Available online at http://planetcassandra. org/what-is-apache-cassandra*, p. 13, 2014.

[27] K. Chodorow and M. Dirolf, *MongoDB - The Definitive Guide: Powerful and Scalable Data Storage*. O'Reilly, 2010. [Online]. Available: http://www.oreilly.de/catalog/9781449381561/index.html

[28] A. Khetrapal and V. Ganesh, "Hbase and hypertable for large scale distributed storage systems," *Dept. of Computer Science, Purdue University*, pp. 22–28, 2006.

[29] R. Klophaus, "Riak core: Building distributed applications without shared state," in *ACM SIGPLAN Commercial Users of Functional Programming*. ACM, 2010, p. 14.

[30] C. Cascaval, C. Blundell, M. M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee, "Software transactional memory: why is it only a research toy?" *Commun. ACM*, vol. 51, no. 11, pp. 40–46, 2008. [Online]. Available: http://doi.acm.org/10.1145/1400214.1400228

Appendix A
Correctness

This section contains the proofs of our table's invariants (in Section A-A), a linearizability proof (in Section A-B) and a proof that our table implementation is lock-free (in Section A-C). First, we define some basic terms we are going to use throughout the proof.

Definition 1: We say that a record R is i-marked if the $marked$ boolean flag of R's $next[i]$ field is $true$.

Definition 2: Given two records, R_1 and R_2 , we say that R_1 is an i-predecessor of R_2 (or that R_2 is the i-successor of R_1) if $R_1.next[i]$ points to R_2.

Definition 3: Given two records, R_1 and R_2 , we say that R_2 i-follows R_1 (or that R_1 i-precedes R_2) if there exist records $R_{i_0}, R_{i_1}, \ldots, R_{i_k}$ such that $R_{i_0} = R_1$, $R_{i_k} = R_2$ and for every $0 < j \leq k$, R_j is the i-successor R_{j-1}.

Definition 4: A record R is i-reachable if R i-follows *head*.

Definition 5: A record R is logically in the table if its status is *InTable* and for every $1 \leq i \leq N_T$, R is i-reachable.

A. Table Invariants

In this section we show that the following invariants hold at any time during the execution:

1) For every $1 \leq i \leq N_T$, *head* and *tail* are always i-reachable.
2) A record R is logically in the table if and only if its status is *InTable* and it is i-reachable for every $1 \leq i \leq N_T$.
3) For every two records R_1, R_2 and $1 \leq i \leq N_T$, if R_2 i-follows R_1 then $R_1.data[i] \leq R_2.data[i]$.
4) For every unique field number i and any two records R_1, R_2, if both R_1 and R_2 are logically in the table then $R_1.data[i] \neq R_2.data[i]$.

Our basic assumption is that no thread ever sends the $-\infty$ or ∞ values as input to one of the table operations. Based on that, we are going to prove the above invariants. In order to prove that the above invariants hold throughout every legal execution, we still need to define a legal execution. Before doing so, we can already prove a basic claim regarding the relationship between every two i-reachable ($1 \leq i \leq N_T$) records.

Claim 6: Let $1 \leq i \leq N_T$ and let R_1, R_2 be two records. If both R_1 and R_2 are i-reachable then either R_1 i-follows R_2 or R_2 i-follows R_1.

Proof 1: By definition, there exist records $R_{i_0}, \ldots, R_{i_m}$ and $R_{j_0}, \ldots, R_{j_n}$ for which *(1)* $head = R_{i_0}$, *(2)* $head = R_{j_0}$, *(3)* $R_{i_m} = R_1$, *(4)* $R_{j_n} = R_2$, *(5)* for every $0 \leq t < m$, R_{i_t}'s i-successor is $R_{i_{t+1}}$, and *(6)* for every $0 \leq t < n$, R_{j_t}'s i-successor is $R_{j_{t+1}}$.

For simplicity, we assume that $m \leq n$ (the case when $m > n$ is symmetric). If for every $0 \leq t \leq m$ it holds that $R_{i_t} = R_{j_t}$, then $R_{j_m} = R_1$ and R_2 i-follows R_1. Otherwise, assume by contradiction that there exists $0 \leq t \leq m$ for which $R_{i_t} \neq R_{j_t}$, and let t be the minimal number for which $R_{i_t} \neq R_{j_t}$. Since $R_{i_0} = R_{j_0} = head$, $t \geq 1$ and thus, $R_{i_{t-1}}$'s i-successor is not $R_{j_{t-1}}$'s i-successor, meaning $R_{i_{t-1}} \neq R_{j_{t-1}}$, in contradiction to t's minimality.

a) Configurations, steps and executions: A configuration C is an instantaneous snapshot of the system describing the state of all local and shared variables as well as the program counter and invocation stack of each thread. A step can be either a shared-memory access by a thread (including the result of the access) or a local step that simply updates a threads own local variables. For local steps, a step corresponds to executing one line of code. In particular, an invocation of a method and the return from a method are each considered to be a step.

We assume each step is atomic, so an execution consists of an alternating sequence of configurations and steps, starting with the initial configuration where the shared memory is initialized with *head* and *tail*, meaning the table is empty. An execution is legal if every thread follows its algorithm in the subsequence consisting of the steps that it performs on the configuration that precedes it, and if every shared object behaves according to its sequential specification in the subsequence of steps that access it.

After defining a configuration, a step and an execution, we can use the terms of "before" and "after" when relating to different executed instructions. This way, it is easier to prove that certain invariants hold throughout the execution. Before diving into the correctness proof, we define a basic operator to be used in the proof:

Definition 7: We use $p \rightsquigarrow R$ to denote that the pointer p points to the record R. p can either be a local variable or a record field.

The next claim asserts some basic observations regarding local changes of record statuses and $next$ fields.

Claim 8: Let $\alpha = C_0 \cdot s_1 \cdot C_1 \cdot s_2 \cdot C_2 \cdot \ldots \cdot s_k \cdot C_k$ be a legal execution, let $0 \leq j \leq k$, and let R be a record, then:

1) If there exists $0 \leq j' < j$ and $1 \leq i \leq N_T$ such that R is i-marked and is an i-predecessor of a record R' at $C_{j'}$, then R is also i-marked and is an i-predecessor of R' at C_j.
2) If there exists $0 \leq j' < j$ such that $R.status \in \{Removed, Failed\}$ at $C_{j'}$ then R's status at C_j is equal to R's status at $C_{j'}$.
3) If there exists $0 \leq j' < j$ such that $R.status = InTable$ at $C_{j'}$ then $R.status \in \{InTable, Removed\}$ at C_j.
4) If there exists $0 \leq j' < j$ such that R's status at $C_{j'}$ is *Pending* and R's status at C_j is *Removed* then there exists $j' < j'' < j$ for which R's status at $C_{j''}$ is *InTable*.
5) If there exists $1 \leq i \leq N_T$ such that R is i-marked at C_j then $R.status \in \{Removed, Failed\}$ at C_j.

Proof 2: *head* and *tail* are created with an *InTable* status, and are not i-marked at initialization (for every $1 \leq i \leq N_T$). Therefore, the claim is vacuously true at C_0. Assume that the invariants hold throughout $\alpha' = C_0 \cdot s_1 \cdot C_1 \cdot \ldots \cdot C_{j-1}$ of the execution. We show that each of the parts of the claim holds throughout $\alpha = \alpha' \cdot s_j \cdot C_j$.

1) Assume that there exists $0 \leq j' < j$ for which R is i-marked and is an i-predecessor of a record R' at $C_{j'}$. If $j' < j - 1$ then by the induction assumption, R is also i-marked and is the i-predecessor of R' at C_{j-1} (if $j' = j - 1$, this obviously holds). The only instructions that can change a record's $next[i]$ field are successful cas executions which assume that $R.next[i]$'s marked flag is false. Therefore, R is also i-marked and is the i-predecessor of R' at C_j.
2) Assume that there exists $0 \leq j' < j$ for which $R.status \in \{Removed, Failed\}$ at $C_{j'}$. By the induction assumption, even if $j' \neq j - 1$, R's status at C_{j-1} is equal to R's status at $C_{j'}$. The only instructions that can change a record's $status$ field are successful cas executions which assume that R's status is either *Pending* or *InTable*. Therefore, R's status at C_j is also equal to R's status at $C_{j'}$.
3) Assume that there exists $0 \leq j' < j$ for which $R.status = InTable$ at $C_{j'}$. By the induction assumption, even if $j' \neq j-1$, $R.status \in \{InTable, Removed\}$ at C_{j-1}. If R's status at C_{j-1} is *Removed*, then from

(2), R's status at C_j is also *Removed*. Otherwise, since the only instruction that can change an *InTable* status is a successful cas execution in line 55 (which changes the status to *Removed*), R's status is also *InTable* or *Removed* at C_j.

4) Assume that there exists $0 \leq j' < j$ for which $R.status = \textit{Pending}$ at $C_{j'}$ and $R.status = \textit{Removed}$ at C_j. If R's status at C_{j-1} is also *Removed*, then by the induction assumption, there exists $j' < j'' < j-1 < j$ for which R's status at $C_{j''}$ is *InTable*. Otherwise, since the only instruction that can change a status into *Removed* is a successful cas execution in line 55 (which only changes an *InTable* status), R's status at C_{j-1} must be *InTable*. Therefore, in this case there also exists $j' < j'' = j-1 < j$ for which R's status at $C_{j''}$ is *InTable*.
5) If R is also i-marked at C_{j-1} then by the induction assumption, $R.status \in \{\textit{Removed}, \textit{Failed}\}$ at C_{j-1}. In this case, from (2), it also holds that $R.status \in \{\textit{Removed}, \textit{Failed}\}$ at C_j. Otherwise, s_j is either the execution of line 24 during an *add* operation, or of line 60 during a *remove* operation. In the first case, there exists $j'' < j$ for which R's status is *Failed* at $C_{j''}$ (the condition in line 20 holds) and from (2), R's status at C_j is $\textit{Failed} \in \{\textit{Removed}, \textit{Failed}\}$. In the second case, there exists $j'' < j$ for which R's status is *Removed* at $C_{j''}$ (after the successful cas execution in line 55), and from (2), R's status at C_j is $\textit{Removed} \in \{\textit{Removed}, \textit{Failed}\}$.

The *head* and *tail* records are the only records that are i-reachable at the initial configuration. In order to show later that *head* and *tail* remain the first and last members of every internal linked-list (respectively), using the fact that a record must be i-marked before it is removed from L_i, we prove the following claim. Notice that by definition, *head* is always i-reachable, for every $1 \leq i \leq N_T$. Therefore, there is no need to show it later.

Claim 9: Let $\alpha = C_0 \cdot s_1 \cdot C_1 \cdot s_2 \cdot C_2 \cdot \ldots \cdot s_k \cdot C_k$ be a legal execution and let $0 \leq j \leq k$, then:

1) $\textit{head.status} = \textit{tail.status} = \textit{InTable}$ at C_j.
2) For every $i \in \{1, \ldots, N_T\}$, *head* and *tail* are not i-marked at C_j.

Proof 3: At initialization, *head* and *tail* are created with an *InTable* status. We assume that for every *remove*(val, i) invocation, $val \notin \{-\infty, \infty\}$. Since a record's *InTable* status can only be changed during a *remove*(val, i) execution, and only if its i-th data element is val, *head* and *tail*s' statuses never change during the execution. Therefore, (1) always holds, and (2) holds consequentially, from claim 8.

Definitions 10 and 11 are made in order to give the best description of the set of records that currently i-follow a certain record, and which are relevant in the sense of being logically in the table.

Definition 10: Let $\alpha = C_0 \cdot s_1 \cdot C_1 \cdot s_2 \cdot C_2 \cdot \ldots \cdot s_k \cdot C_k$ be a legal execution. Given $0 \leq j \leq k$, the i-followers set of R at C_j is the set of all records R' satisfying *(1)* $R' \neq R$, *(2)* R' i-follows R at C_j, and *(3)* R' is not i-marked at C_j.

Definition 11: Let $\alpha = C_0 \cdot s_1 \cdot C_1 \cdot s_2 \cdot C_2 \cdot \ldots \cdot s_k \cdot C_k$ be a legal execution. Given $0 \leq j \leq k$, the i-inclusive followers set of R at C_j is the set of all records R' satisfying *(1)* R' i-follows R at C_j, and *(2)* R' is not i-marked at C_j.

Notice that, given $1 \leq i \leq N_T$, the difference between a record's i-followers set and a record's i-inclusive followers set is at most the record itself. When the record is i-marked, its i-followers set and i-inclusive followers set are equal.

We use the notion of i-infancy in order to describe a record which has not been inserted into L_i yet. We will show later that an i-infant cannot i-follow any other record and therefore, we can update its i-successor during the insertion procedure without causing any damage.

Definition 12: Let $\alpha = C_0 \cdot s_1 \cdot C_1 \cdot s_2 \cdot C_2 \cdot \ldots \cdot s_k \cdot C_k$ be a legal execution, let $0 \leq j \leq k$ and let $R \notin \{\textit{head}, \textit{tail}\}$ be a record. If there does not exist $j' \leq j$ for which $s_{j'}$ is a successful cas execution in line 49, satisfying $newRecord \rightsquigarrow R$, we say that R is an i-infant at C_j.

We allow multiple i-reachable records with the same i-th property, even when i is a unique field number, but at most one such record can be logically in the table. When i is not a unique field number, there can be multiple records with the same i-th property which are logically in the table. Therefore, we would like to define an order on the records that are eventually added into a certain linked-list. Intuitively, we order records by the order of their relevant properties, and records with the same property by the order they were added to the list (reversed).

Definition 13: Let $\alpha = C_0 \cdot s_1 \cdot C_1 \cdot s_2 \cdot C_2 \cdot \ldots \cdot s_k \cdot C_k$ be a legal execution, let $0 \leq j \leq k$, let $i \in \{1, \ldots, N_T\}$ and let R_1, R_2 be records. We denote $R_1 <_i R_2$ if either $R_1.data[i] < R_2.data[i]$, or $R_1.data[i] = R_2.data[i]$ and there exists $0 \leq j' \leq j$ for which R_1 is an i-infant at $C_{j'}$ but R_2 is not an i-infant at $C_{j'}$.

Notice that the $<_i$ relation (Definition 13) is not a total order on all records, but it is a total order on the set of all records which are eventually inserted into L_i. This means that, given two records R_1 and R_2, it is clear whether R_1 should i-follow R_2 or R_2 should i-follow R_1. We later show that indeed R_2 only i-follows R_1 if $R_1 <_i R_2$. Claim 14 shows that $<_i$ defines a total order on all records which are eventually inserted into L_i (meaning, stop being i-infants), and even on some of the records which remain i-infants until the execution ends.

Claim 14: Let $\alpha = C_0 \cdot s_1 \cdot C_1 \cdot s_2 \cdot C_2 \cdot \ldots \cdot s_k \cdot C_k$ be a legal execution, let $0 \leq j \leq k$, let $i \in \{1, \ldots, N_T\}$ and let R_1, R_2 be two different records. If R_1 is not an i-infant at C_j then either $R_1 <_i R_2$ or $R_2 <_i R_1$.

Proof 4: If $R_1.data[i] \neq R_2.data[i]$ the by definition, we are done. Otherwise, Since a step is defined as executing one instruction, R_1 and R_2 cannot stop being i-infants at the same configuration. Therefore, either $R_1 <_i R_2$ or $R_2 <_i R_1$. Notice that no thread ever tries to insert a tuple containing a property from $\{-\infty, \infty\}$ into the table and thus, *head* $<_i R <_i$ *tail* for every record R.

In order to show that the table invariants hold throughout any legal execution, we define a set of operations (definitions 15-18). As we later show (in claim 21), any change to the internal linked-lists can be made only by those operations. Therefore, we do not need to take into consideration any other operation in our later proofs.

Definition 15: Let $\alpha = C_0 \cdot s_1 \cdot C_1 \cdot s_2 \cdot C_2 \cdot \ldots \cdot s_k \cdot C_k$ be a legal execution, and let $0 \leq j \leq k$. We say that s_j is an i-marking if there exist a record R_1 and $i \in \{1, \ldots, N_T\}$ such that R_1 is not i-marked at C_{j-1} and is i-marked at C_j.

Definition 16: Let $\alpha = C_0 \cdot s_1 \cdot C_1 \cdot s_2 \cdot C_2 \cdot \ldots \cdot s_k \cdot C_k$ be a legal execution, and let s_j be the execution of line 8, having $pred \rightsquigarrow R_1$, $curr \rightsquigarrow R_2$ and $succ \rightsquigarrow R_3$. We say that s_j is an i-snipping if *(1)* R_1 is i-reachable and not i-marked at C_{j-1}, *(2)* R_2 is i-marked at C_{j-1}, *(3)* R_2 is the i-successor of R_1 at C_{j-1}, *(4)* R_3 is the i-successor of R_2 at C_{j-1}, and *(5)* s_j assigns $\langle R_3, false\rangle$ to the $R_1.next[i]$ field.

Definition 17: Let $\alpha = C_0 \cdot s_1 \cdot C_1 \cdot s_2 \cdot C_2 \cdot \ldots \cdot s_k \cdot C_k$ be a legal execution, and let s_j be the execution of line 48 , having $newRecord \rightsquigarrow R_1$ and $curr \rightsquigarrow R_2$, or of line 49, having $pred \rightsquigarrow R_1$ and $newRecord \rightsquigarrow R_2$. We say that s_j is an i-redirection if *(1)* R_2 is not an i-infant at C_{j-1}, *(2)* $R_1 <_i R_2$, *(3)* R_1's i-followers set at C_{j-1} is a subset of R_2's i-inclusive followers set at C_{j-1}, and *(4)* s_j assigns $\langle R_2, false\rangle$ to the $R_1.next[i]$ field.

Definition 18: Let $\alpha = C_0 \cdot s_1 \cdot C_1 \cdot s_2 \cdot C_2 \cdot \ldots \cdot s_k \cdot C_k$ be a legal execution, and let s_j be the execution of line 49, having $pred \rightsquigarrow R_1$ and $newRecord \rightsquigarrow R_2$. We say that s_j is an i-insertion if *(1)* R_1 is not an i-infant and not i-marked at C_{j-1}, *(2)* $R_1 <_i R_2$, *(3)* R_2 is an i-infant at C_{j-1} and not an i-infant at C_j, *(4)* R_1's i-followers set at C_{j-1} is a subset of R_2's i-followers set at C_{j-1}, and *(5)* s_j assigns $\langle R_2, false\rangle$ to the $R_1.next[i]$ field.

After defining our basic terms, we can now start with our correctness proof. As mentioned before, we assume $-\infty$ and ∞ are never sent as input to the table operations. Claim 19 shows that they are never sent as input to the *find* method as well.

Claim 19: Let val be the first input parameter of a *find* call. Then $val \notin \{-\infty, \infty\}$.

Proof 5: The *find* method is called only from lines 35 and 52. In line 35, val is a property of a tuple, sent as input to the *add* operation. We assume that no thread is trying to insert a tuple with a property from $\{-\infty, \infty\}$ and thus, $val \notin \{-\infty, \infty\}$ in this case.

In line 35, val is the property sent as input to the *remove* operation. We assume that no thread is trying to remove a tuple with a property from $\{-\infty, \infty\}$ and thus, $val \notin \{-\infty, \infty\}$ in this case as well.

Assuming that the *find* method indeed returns two records as output, claim 20 proves that those records are the relevant ones. Meaning, if val and i are the input parameters, then $pred$'s i-th property is smaller than val, and $curr$'s i-th property is at least val. Claim 20 only deals with properties, that do not change throughout the execution. We will relate to $pred$ and $curr$s' relevance as the potential i-predecessor and i-successor of searched records later.

Claim 20: Let R_1 and R_2 be two records. If $pred \rightsquigarrow R_1$ and $curr \rightsquigarrow R_2$, when $\langle pred, curr\rangle$ is the output of a *find*(val, i) execution, then $R_1.data[i] < val \leq R_2.data[i]$.

Proof 6: If the last assignment into $pred$ was in line 3 then R_1 is *head* and, since $-\infty$ is never sent as input to the *find* method (claim 19), $R_1.data[i] < val$. Otherwise, the last assignment into $pred$ was in line 15, right after the condition in line 13 did not hold for R_1. Therefore, $R_1.data[i] < val$ in this case also. Regarding R_2: since R_2 was returned as $curr$ in line 14, the condition in line 13 held for R_2. Therefore, $val \leq R_2.data[i]$.

As mentioned, claim 20 assumes that the *find* method indeed returns two records. In our next claim, which proves that some invariants hold throughout any legal execution, we also assume that. We can do so because invariant 1 ensures that all relevant local variables ($pred$ and $curr$) point to actual records (and not nulls) throughout the method execution (and in particular, when returning).

Claim 21: Let $\alpha = C_0 \cdot s_1 \cdot C_1 \cdot s_2 \cdot C_2 \cdot \ldots \cdot s_k \cdot C_k$ be a legal execution, let $0 \leq j \leq k$, $1 \leq i \leq N_T$ and let R_1, R_2 be two different records, then:

1) If $pred \rightsquigarrow X$ or $curr \rightsquigarrow X$ at C_j, during a *find*(val, i) execution, then X is a record. In particular, if $pred \rightsquigarrow X$ then $X.data[i] < val$.
2) If R_2 is the i-successor of R_1 at C_j then R_2 is not an i-infant at C_j.
3) If R_1 is not an i-infant at C_j then there exists $j' \leq j$ for which R_1 is i-reachable at $C_{j'}$.
4) If $pred \rightsquigarrow R_1$ and $curr \rightsquigarrow R_2$ at C_j, during a *find*(val, i) execution, then R_1 and R_2 are not i-infants at C_j.
5) If $pred \rightsquigarrow R_1$ and $curr \rightsquigarrow R_2$ at C_j, during a *find*(val, i) execution, then there exists $j_1 \leq j$ for which R_1 is i-reachable and R_2 is the i-successor of R_1 at C_{j_1}.
6) If $R_1.data[i] = R_2.data[i]$, $R_1 <_i R_2$, both R_1 and R_2 are not i-infants at C_{j-1} and R_2 is not i-marked at C_j, then R_2 belongs to R_1's i-followers set at C_{j-1}.
7) If s_j is a successful cas execution in line 48, for which $newRecord \rightsquigarrow R_1$ and $curr \rightsquigarrow R_2$, then $R_1 <_i R_2$.
8) If R_2 is the i-successor of R_1 at C_j then $R_1 <_i R_2$.
9) Suppose there exists $j_1 \leq j-1$ for which *(1)* R_2 is not an i-infant at C_{j-1}, *(2)* R_2's i-inclusive followers set at C_{j-1} is a subset of R_1's i-inclusive followers set at C_{j-1}.
   Then R_2's i-inclusive followers set at C_{j-1} is a subset of R_1's i-followers set at C_{j-1}.
10) If R_1 is not i-marked and is an i-predecessor of R_2 at C_{j-1}, and either is i-marked or is not an i-predecessor of R_2 at C_j, then s_j is either an i-marking, an i-snipping, an i-redirection or an i-insertion.
11) If $R_1 <_i R_2$ then R_1 does not i-follow R_2 at C_j.
12) If R_1 is i-reachable and is not i-marked at C_{j-1}, then R_1 is i-reachable at C_j.

13) If R_1 is not an i-infant and is not i-marked at C_j, then R_1 is i-reachable at C_j.
14) If R_2 belongs to R_1's i-inclusive followers set at C_{j-1} and is not i-marked at C_j then it belongs to R_1's i-inclusive followers set at C_j.

Proof 7: Most claim invariants are vacuously true at C_0. In addition, invariant 2 is true because *tail* cannot be an i-infant at C_0 by definition, invariant 3 is true because both *head* and *tail* are i-reachable at C_0, invariant 8 is true because *head* $<_i$ *tail*, invariant 11 is true because *head* does not i-follow *tail* at C_0, and invariant 13 is true because both *head* and *tail* are i-reachable at C_0.

Assume that the invariants hold throughout $\alpha' = C_0 \cdot s_1 \cdot C_1 \cdot \ldots \cdot C_{j-1}$ of the execution. We show that each of the parts of the claim hold throughout $\alpha = \alpha' \cdot s_j \cdot C_j$.

1) The steps that change the $pred$ and $curr$ local variables during the execution are the ones in lines 3, 4, 11, 15 and 16.
Suppose s_j is the execution of line 3, and that $pred \rightsquigarrow X$ at C_j. Then X is the *head* record. In addition, since $-\infty < val$ (by claim 19), $X.data[i] < val$.
Suppose s_j is the execution of line 4, and that $curr \rightsquigarrow X$ at C_j. By definition, *tail* is not an i-infant at C_{j-1}, and from claim 9, *tail* is not i-marked at C_{j-1}. By the induction assumption (invariant 13), *tail* is i-reachable at C_{j-1}. Since s_j does not change L_i, *tail* is still i-reachable at C_j. By definition, *tail* i-follows *head* at C_j. Therefore, *head* must have an i-successor at C_j. Conclusion: X is a record.
Suppose s_j is the execution of line 11, and that $curr \rightsquigarrow X$ at C_j. Let R be the record for which $pred \rightsquigarrow R$ at C_{j-1}. By the induction assumption, $R.data[i] < val$. Now, let s_{j_1} be the last execution of line 8 before s_j. By the induction assumption (invariant 4), R is not an i-infant at C_{j-1}. In addition, since the cas execution must be a successful one, R is not i-marked at C_{j_1}. By the induction assumption (invariant 13), R is i-reachable at C_{j_1}. As already explained, *tail* also must be i-reachable at C_{j_1} and therefore, by claim 6, either R i-follows *tail* or *tail* i-follows R at C_{j_1}. By claim 19, $val <$ *tail*.$data[i]$ and therefore, $R.data[i] <$ *tail*.$data[i]$. By definition, $R <_i$ *tail*. By the induction assumption (invariant 11), R does not i-follow *tail* at C_{j_1} and thus, *tail* i-follows R at C_{j_1}. By definition, *tail* belongs to R's i-inclusive followers set at C_{j-1}. Since *tail* is never i-marked (claim 9), by the induction assumption (invariant 14), *tail* belongs to R's i-inclusive followers set at C_{j-1}. Since R's $next[i]$ pointer field points to X at C_{j-1}, by the definition of i-following, X is a record.
Suppose s_j is the execution of line 15, and that $pred \rightsquigarrow X$ at C_j. Since $curr \rightsquigarrow X$ at C_{j-1}, by the induction assumption, X is a record. In addition, since tha condition checked in line 13 did not hold, $X.data[i] < i$.
Suppose s_j is the execution of line 16, and that $curr \rightsquigarrow X$ at C_j. Let R be the record for which $pred \rightsquigarrow R$ at C_{j-1} and let s_{j_1} be the step assigning X into the $succ$ local variable, either in line 6 or 12. By the induction assumption (invariant 4), R is not an i-infant at C_{j-1}. In addition, since s_{j_1} is the last assignment into $succ$, R is not i-marked at C_{j_1} (the condition checked in line 7 did not hold). Therefore, by the induction assumption (invariant 13), R is i-reachable at C_{j_1}. In addition, since $pred \rightsquigarrow R$ at C_{j-1}, by the induction assumption, $R.data[i] < val$. As in a former case (the one for which s_j is the execution of line 11), *tail* i-follows R at C_{j_1}. Since R's $next[i]$ pointer field points to X at C_{j_1}, by the definition of i-following, X is a record.
For all other cases, the invariant holds by the induction assumption.
2) If R_2 is the i-successor of R_1 at C_{j-1} then by the induction assumption, R_2 is not an i-infant at C_{j-1}. By definition, R_2 is not an i-infant at C_j in this case.
Otherwise, s_j is either a successful cas execution in line 8 for which $pred \rightsquigarrow R_1$ and $succ \rightsquigarrow R_2$, a successful cas execution in line 48 for which $newRecord \rightsquigarrow R_1$ and $curr \rightsquigarrow R_2$, or a successful cas execution in line 49 for which $pred \rightsquigarrow R_1$ and $newRecord \rightsquigarrow R_2$.
If s_j is a successful cas execution in line 8 for which $pred \rightsquigarrow R_1$ and $succ \rightsquigarrow R_2$, then there exists $j' < j$ for which R_1's i-successor is an i-predecessor of R_2 at $C_{j'}$. By the induction assumption, R_2 is not an i-infant at $C_{j'}$. By definition, R_2 is not an i-infant at C_j as well.
If s_j is a successful cas execution in line 48 for which $newRecord \rightsquigarrow R_1$ and $curr \rightsquigarrow R_2$, then by the induction assumption (invariant 4), there exists $j' < j$ for which R_2 is not an i-infant at $C_{j'}$. By definition, R_2 is not an i-infant at C_j as well.
If s_j is a successful cas execution in line 49 for which $pred \rightsquigarrow R_1$ and $newRecord \rightsquigarrow R_2$, then by definition, if R_2 is an i-infant at C_{j-1} then it is not an i-infant at C_j (otherwise, by definition it is not an i-infant at C_j as well).
3) If R_1 is also not an i-infant at C_{j-1} then by the induction assumption, there exists $j' \le j-1 < j$ for which R_1 is i-reachable at $C_{j'}$. Otherwise, s_j is a successful cas execution in line 49 for which $newRecord \rightsquigarrow R_1$. Let R_3 be the record for which $pred \rightsquigarrow R_3$ at C_j. Since the cas is successful, R_3 is not i-marked at C_j. From claim 8, R_3 is not i-marked at C_{j-1} as well. In addition, by the induction assumption (invariant 4), R_3 is not an i-infant at C_{j-1}. By the induction assumption (invariant 13), R_3 is i-reachable at C_{j-1}. Since s_j only affects R_3's i-successor, it is still i-reachable at C_j. Finally, since R_1 is R_3's i-successor at C_j, it is i-reachable at C_j. Since $j \le j$, the invariant holds.
4) If $pred \rightsquigarrow R_1$ and $curr \rightsquigarrow R_2$ at C_{j-1}, then by the induction assumption, R_1 and R_2 are not i-infants at C_{j-1}. By definition, R_1 and R_2 are also not i-infants at C_j in this case.
Otherwise, s_j is either the execution of line 3, 4, 11, 15 or 16.

If s_j is the execution of line 3 then R_1 is *head* and by definition, not an i-infant at C_j. In addition, since $curr \rightsquigarrow R_2$ at C_j, $curr \rightsquigarrow R_2$ at C_{j-1} as well. Therefore, by the induction assumption, R_2 is not an i-infant at C_{j-1}. By definition, R_2 is not an i-infant at C_j in this case.

If s_j is the execution of line 4 or 11 then $pred \rightsquigarrow R_1$ at C_{j-1} and like in former cases, R_1 is not an i-infant at C_j. Since R_2 is R_1's i-successor at C_j, by invariant 2, R_2 is not an i-infant at C_j.

If s_j is the execution of line 15 then $R_1 = R_2$ - a contradiction to our initial assumption.

If s_j is the execution of line 16 then $pred \rightsquigarrow R_1$ at C_{j-1} and like in former cases, R_1 is not an i-infant at C_j. Now, let $s_{j'}$ be the step which assigns R_2 into the $succ$ variable (either in line 6 or 12). R_2 is R_1's i-successor at $C_{j'}$ and by invariant 2, not an i-infant at $C_{j'}$. By definition, R_2 is not an i-infant at C_j.

5) If $pred \rightsquigarrow R_1$ and $curr \rightsquigarrow R_2$ at C_{j-1} then by the induction assumption, $j_1 \leq j-1 \leq j$ exists. Otherwise, s_j is either the execution of line 4, 11, 15 or 16.

If s_j is the execution of line 4 then since R_1 is the *head* record, by definition it is i-reachable at C_j. Since R_2 is R_1's i-successor at C_j, the invariant holds for $j_1 = j$ in this case.

If s_j is the execution of line 11, then let j_1 be the maximal value for which $pred$ is not i-marked before (and not including) C_j. Since R_1 is not i-marked when executing line 8, C_{j_1} is during the period for which $pred \rightsquigarrow R_1$. Therefore, from the induction assumption (invariant 4), R_1 is not an i-infant at C_{j_1}. By the induction assumption (invariant 13), R_1 is i-reachable at C_{j_1}. Since either R_1 is i-marked at C_{j_1+1} or $j_1 = j-1$, R_2 is R_1's i-successor at C_{j_1}, either by claim 8 or because s_j does not change R_1's i-successor (respectively). Therefore, the invariant holds for j_1 in this case.

If s_j is the execution of line 15 then $R_1 = R_2$ and we have a contradiction to our initial assumption.

If s_j is the execution of line 16 then let s_{j_1} be the assignment of R_2 into the $succ$ variable, either in line 6 or 12. In both cases, R_2 is R_1's i-successor at C_{j_1}. We still need to show that R_1 is i-reachable at C_{j_1}. By the induction assumption (invariant 4), R_1 is not an i-infant at C_{j_1}. In addition, since s_j is the execution of line 16, the condition checked in line 7 does not hold for R_1. Therefore, by claim 8, R_1 is not i-marked at C_{j_1}. By the induction assumption (invariant 13), R_1 is i-reachable at C_{j_1}.

6) If R_2 belongs to R_1's i-followers set at C_{j-2} then, since R_2 is not i-marked at C_j (and by claim 8, at C_{j-1} as well), by invariant 14, R_2 belongs to R_1's i-followers set at C_{j-1}.

If R_2 does not belong to R_1's i-followers set at C_{j-2} then by the induction assumption, either R_1 is an i-infant at C_{j-2} or R_2 is an i-infant at C_{j-2}. Since $R_1 <_i R_2$ and $R_1.data[i] = R_2.data[i]$, and since both are not i-infant at C_{j-1}, by definition R_2 is not an i-infant at C_{j-2}. Therefore, R_1 is an i-infant at C_{j-2}. In this case, s_{j-1} is a successful cas execution in line 49, having $newRecord \rightsquigarrow R_1$.

Let R_3 be the record for which $pred \rightsquigarrow R_3$ at C_{j-1}. By invariant 4, R_3 is not an i-infant at C_{j-2}. In addition, since the cas execution is successful, R_3 is not i-marked at C_{j-2}. By the induction assumption (invariant 13), R_3 is i-reachable at C_{j-2}.

Since R_2 is not an i-infant and is not i-marked at C_{j-2} (we know that it is not i-marked at C_j + claim 8), By the induction assumption (invariant 13), R_2 is i-reachable at C_{j-2}.

From claim 6, either R_3 i-follows R_2 or R_2 i-follows R_3 at C_{j-2}. From claim 20, $R_3.data[i] < R_1.data[i]$ and therefore, by definition, $R_3 <_i R_2$. By the induction assumption (invariant 11), R_3 does not i-follow R_2 at C_{j-2} and thus, R_2 i-follows R_3 at C_{j-2}. By definition, R_2 is in R_3's i-followers set at C_{j-2} and by the induction assumption (invariant 14), R_2 is in R_3's i-followers set at C_{j-1}. Since R_1 is R_3's i-successor at C_{j-1}, R_2 is also in R_1's i-followers set at C_{j-1}.

7) If $R_1.data[i] < R_2.data[i]$ then by definition, $R_1 <_i R_2$ and we are done. Otherwise, since $R_1.data[i]$ is sent as the first input parameter to the *find* call in line 35, from claim 20 it holds that $R_1.data[i] = R_2.data[i]$.

We denote the $tmp \leftarrow curr$ assignment in line 36 with s_{j_1} and the last time the instruction in line 40 was executed before C_j with s_{j_2}. Notice that since $R_1.data[i] = R_2.data[i]$ and the method did not return in line 39 (s_j is the execution of line 48), s_{j_2} is well-defined. Since R_2 is a *find* output parameter, from the induction assumption (invariant 4), R_2 is not an i-infant at C_{j_1}.

Assume by contradiction that $R_1 <_i R_2$ does not hold. From claim 14, $R_2 <_i R_1$. Therefore, there exist j_3, j_4 satisfying *(1)* $j_3 < j_4 < j_1$, *(2)* j_3 is the minimal value for which R_1 is not an i-infant at C_{j_3}, and *(3)* j_4 is the minimal value for which R_2 is not an i-infant at C_{j_4}. Since the cas execution in line 48 is successful, R_1 is not i-marked at C_j. From claim 8, R_1 is also not i-marked at for every earlier configuration. Therefore, by the induction assumption (invariant 6), R_1 i-follows R_2 at C_{j_4}. In addition, by invariant 14, R_1 i-follows R_2 at every configuration between C_{j_1} and C_{j_2}.

Let R_3 be the last record satisfying $tmp \rightsquigarrow R_3$ and $R_3 <_i R_1$ between C_{j_1} and C_{j_2} (R_3 must exists, because R_2 satisfies those conditions). Notice that R_3 must satisfy $R_3.data[i] = R_1.data[i]$, and that by the induction assumption (invariant 2), R_3 is not an i-infant while $tmp \rightsquigarrow R_3$.

By the induction assumption (invariant 6), R_1 belongs to R_3's i-followers set at the assignment of R_3 into tmp (either in line 36 or in line 40). In addition, by the induction assumption (invariant 14), it belongs to R_3's i-followers set until C_{j_2} (including C_{j_2}).

Since $R_3.data[i] = R_1.data[i]$, R_3 is not the last assignment into tmp before C_{j_2}. Now, let R_4 be the next record satisfying $tmp \rightsquigarrow R_4$. If R_4 is null then R_1 does not belong to R_3's i-followers set when this assignment is executed, and before C_{j_2} - a contradiction. Otherwise, when the assignment $tmp \leftarrow R_4$ is made, R_3 is an i-predecessor of R_4, and by the induction assumption (invariant 2), R_4 is not an i-infant at this point. Since $R_4 <_i R_1$ does not hold (R_3 was chosen as the last record for which it holds), from claim 14, $R_1 <_i R_4$. By the induction assumption (invariant 11), R_1 does not i-follow R_4 when the assignment is done, and thus, also does not i-follow R_3. Therefore, R_1 does not belong to R_3's i-followers set before C_{j_2} - a contradiction. Our assumption leads to a contradiction for every case and therefore, it is wrong. Conclusion: $R_1 <_i R_2$ does hold.

8) If R_2 is also the i-successor of R_1 at C_{j-1}, then by the induction assumption, $R_1 <_i R_2$ and we are done.

If s_j is a successful cas execution in line 8, then $pred \rightsquigarrow R_1$ and $succ \rightsquigarrow R_2$. Let R_3 be the record for which $curr \rightsquigarrow R_3$ at C_j. Since the cas execution is successful, R_3 is the i-successor of R_1 at C_{j-1} and by the induction assumption, $R_1 <_i R_3$. In addition, R_3 is i-marked and R_2 is the i-successor of R_3 when the condition in line 7 is checked. From claim 8, it is also the i-successor of R_3 at C_{j-1}. By the induction assumption, $R_3 <_i R_2$. Since the $<_i$ relation (as described in definition 13) is transitive, $R_1 <_i R_2$.

If s_j is a successful cas execution in line 48 then $newRecord \rightsquigarrow R_1$ and $curr \rightsquigarrow R_2$. From invariant 7, $R_1 <_i R_2$.

If s_j is a successful cas execution in line 49 then $pred \rightsquigarrow R_1$ and $newRecord \rightsquigarrow R_2$. $R_2.data[i]$ is sent as the first input parameter to the *find* execution returning R_1 (in line 35). From claim 20, $R_1.data[i] < R_2.data[i]$ and by definition 13, $R_1 <_i R_2$.

Since all other instructions do not change the record referenced by $R_1.next[i]$, the invariant holds for them as well.

9) If $j_1 = j - 1$, then R_2's i-inclusive followers set at C_{j-1} is a subset of R_1's i-followers set at C_{j-1} and we are done. Otherwise, by the induction assumption, R_2's i-inclusive followers set at C_{j-2} is a subset of R_1's i-followers set at C_{j-2}.

If s_{j-1} does not change any $next[i]$ pointer, then by definition 10, R_1's i-followers set at C_{j-2} equals R_1's i-followers set at C_{j-1} and R_2's i-inclusive followers set at C_{j-2} equals R_2's i-inclusive followers set at C_{j-1}. Therefore, in this case R_2's i-inclusive followers set at C_{j-1} is a subset of R_1's i-followers set at C_{j-1}.

If s_{j-1} does change a $next[i]$ pointer then by the induction assumption (invariant 10), s_{j-1} is either an i-marking, an i-snipping, an i-redirection or an i-insertion. Now, let R be a record in R_2's i-inclusive followers set at C_{j-1}. By definition 10, R i-follows R_2 at C_{j-1} and is not i-marked at C_{j-1}.

Suppose s_{j-1} is an i-marking. Since an i-marking cannot affect any pointer (by definition 15), it can only affect a $marked$ flag, and from claim 8, R also i-follows R_2 and is not i-marked at C_{j-2}. By definition 10, R is in R_2's i-inclusive followers set at C_{j-2} and thus, in R_1's i-followers set at C_{j-2}. Since R is not i-marked at C_{j-1}, by definition 11 it is in R_1's i-followers set at C_{j-1} as well.

Suppose s_{j-1} is an i-snipping. Assume by contradiction that R does not i-follow R_2 at C_{j-2}. By definition, there exist records $R_{i_0}, \ldots, R_{i_m}$ for which $R_2 = R_{i_0}$, $R = R_{i_m}$ and for every $0 < t \le m$, R_{i_t} is $R_{i_{t-1}}$'s i-successor at C_{j-1}. Since R does not i-follow R_2 at C_{j-2}, there exists a maximal t among $0, \ldots, m-1$ for which R does not i-follow R_{i_t} at C_{j-2}. This means that $R_{i_{t+1}}$ does not i-follow R_{i_t} at C_{j-2}, but it is R_{i_t}'s i-successor at C_{j-1}. Since s_{j-1} is an i-snipping, from definition 16 it must hold that there exists a record R' which is the i-successor of R_{i_t} and an i-predecessor of $R_{i_{t+1}}$ at C_{j-2} - a contradiction to the fact that $R_{i_{t+1}}$ does not i-follow R_{i_t} at C_{j-2}. Therefore, R does i-follow R_2 at C_{j-2} and by definition, belongs to R_2's i-inclusive followers set at C_{j-2}. Since R_2's i-inclusive followers set at C_{j-2} is a subset of R_1's i-followers set at C_{j-2}, R belongs to R_1's i-followers set at C_{j-2}. By the induction assumption (invariant 14), R belongs to R_1's i-followers set at C_{j-1}.

Suppose s_{j-1} is an i-redirection or an i-insertion. If R also i-follows R_2 at C_{j-2} then (since it is not i-marked), it belongs to R_2's i-inclusive followers set at C_{j-1}. Therefore, it also belongs to R_1's i-followers set at C_{j-2}, and from the induction assumption (invariant 14), it belongs to R_1's i-followers set at C_{j-1}. Otherwise (R does not i-follow R_2 at C_{j-2}), since R does i-follow R_2 at C_{j-1}, there exist records $R_{i_0}, \ldots, R_{i_m}$ for which $R_2 = R_{i_0}$, $R = R_{i_m}$ and for every $0 < t \le m$, R_{i_t} is $R_{i_{t-1}}$'s i-successor at C_{j-1}. Notice that we can assume that this path does not contain circles (otherwise, they can be removed, and we are still left with a proper path). Since R does not i-follow R_2 at C_{j-2}, there exists a maximal t among $0, \ldots, m-1$ for which R_{i_t} i-follows R_2 at C_{j-2}. This means that $R_{i_{t+1}}$ does not i-follow R_{i_t} at C_{j-2}, but it is R_{i_t}'s i-successor at C_{j-1}. Since s_{j-1} changes R_{i_t}'s i-successor, R_{i_t} is not i-marked at C_{j-2} (by claim 8). By definition 11, R_{i_t} is in R_2's i-inclusive followers set at C_{j-2} and thus, in R_1's i-followers set at C_{j-2}. By the induction assumption (invariant 14), R_{i_t} is in R_1's i-followers set at C_{j-1}. Since R i-follows R_{i_t} at C_{j-1}, it also i-follows R_1 at C_{j-1}. In addition, since it is not i-marked at C_{j-1}, it belongs to R_1's i-followers set at C_{j-1}. Notice that R_{i_t} and R must be two different records. Assume by contradiction that $R_1 = R$, then R_1 i-follows R_{i_t} and R_{i_t} i-follows R_1 at C_{j-1}, in contradiction to the combination of claim 14 and invariant 11. Therefore, $R_1 \neq R$ and R indeed belongs to its i-followers set at C_{j-1} (and not just its

i-inclusive followers set at C_{j-1}).

10) The only instruction that can change R_1's i-th $mraked$ flag or i-successor are successful cas executions in lines 8, 24, 48, 50 and 60.

If s_j is a successful cas execution in line 24 or 60, then by definition 15, s_j is an i-marking.

If s_j is a successful cas execution in line 8, then $pred \rightsquigarrow R_1$ and $curr \rightsquigarrow R_2$. Let R_3 be the record for which $succ \rightsquigarrow R_3$. We are going to show that s_j is an i-snipping, by definition 16:

(1) By the induction assumption (invariant 4), R_1 is not an i-infant at C_{j-1}. Since the cas execution at s_j is successful, R_1 is also not i-marked at C_{j-1}. By the induction assumption (invariant 13), R_1 is i-reachable at C_{j-1}. *(2)* The condition in line 7 held. Therefore, there exists $j' < j-1$ for which R_2 is i-marked at $C_{j'}$. From claim 8, R_2 is also i-marked at C_{j-1}. *(3)* The cas execution at s_j is successful and thus, R_2 is the i-successor of R_1 at C_{j-1}. *(4)* R_3 is the i-successor of R_2 when the condition in line 7 is checked. From claim 8, it is also the i-successor of R_2 at C_{j-1}. *(5)* s_j assigns $\langle R_3, false\rangle$ to the $R_1.next[i]$ field.

If s_j is a successful cas execution in line 48 then $newRcord \rightsquigarrow R_1$ and $succ \rightsquigarrow R_2$. Let R_3 be the record for which $curr \rightsquigarrow R_3$. We are going to show that s_j is an i-redirection. Since R_3 is not an i-infant at C_{j-1} (by the induction assumption, invariant 4), $R_1 <_i R_3$ (by the induction assumption, invariant 7) and s_j assigns $\langle R_3, false\rangle$ into R_1's $next[i]$ field, we need to show that R_1's i-followers set at C_{j-1} is a subset of R_3's i-inclusive followers set at C_{j-1}. Since R_2 is R_1's i-successor at C_{j-1}, it suffices to show that R_2's i-inclusive followers set at C_{j-1} is a subset of R_3's i-inclusive followers set at C_{j-1}. We are going to show that invariant 9 holds for R_3 and R_2.

Let R_4 be the record for which $pred \rightsquigarrow R_4$ at C_j and let C_{j_1} the configuration which is guaranteed by invariant 5. By definition, C_{j_1} is during the *find* execution, invoked in line 35, R_4 is i-reachable at C_{j_1} and R_3 is R_4's i-successor at C_{j_1}. In addition, since R_2 is R_1's i-successor before this *find* execution, by the induction assumption (invariant 2), R_2 is not an i-infant at C_{j_1}.

We still need to show that R_2's i-inclusive followers set at C_{j-1} is a subset of R_3's i-inclusive followers set at C_{j_1}. Let R be a record in R_2's i-inclusive followers set at C_{j_1}. By definition, R is not i-marked at C_{j_1} and R i-follows R_2 at C_{j_1}. By the induction assumption (invariant 2), R is not an i-infant at C_{j_1} and therefore, by the induction assumption (invariant 13), R is i-reachable at C_{j_1}. By claim 6, either R i-follows R_4 or R_4 i-follows R at C_{j_1}.

Since R i-follows R_2 at C_{j_1}, by the induction assumption (invariant 11), either $R = R_2$ or $R_2 <_i R$. Since R_2 i-follows R_1 when executing line 34, by the induction assumption (invariant 8), $R_1 <_i R_2$ and therefore, $R_1 <_i R$, meaning $R_1.data[i] \le R.data[i]$. From claim 20, $R_4.data[i] < R_1.data[i]$ and thus, by definition, $R_4 <_i R$. Therefore, since either R i-follows R_4 or R_4 i-follows R at C_{j_1}, by the induction assumption (invariant 11), R i-follows R_4 at C_{j_1}. Since R_3 is R_4's i-successor at C_{j_1}, R also i-follows R_3 at C_{j_1}. Therefore, R is in R_3's i-inclusive followers set at C_{j_1}.

After showing that R_2's i-inclusive followers set at C_{j_1} is a subset of R_3's i-inclusive followers set at C_{j_1}, we can use invariant 9. R_2's i-inclusive followers set at C_{j-1} is indeed a subset of R_3's i-inclusive followers set at C_{j-1}. Conclusion: s_j is an i-redirection.

If s_j is a successful cas execution in line 49 then $pred \rightsquigarrow R_1$ and $curr \rightsquigarrow R_2$. Let R_3 be the record for which $newRecord \rightsquigarrow R_3$. s_j assigns $\langle R_3, false\rangle$ into the $R_1.next[i]$ field, and from claim 20, $R_1.data[i] < R_3.data[i]$. Therefore, by definition 13 $R_1 <_i R_3$.

From invariant 4, R_1 is not an i-infant at C_{j-1}. In addition, since the cas execution is successful, R_1 is not i-marked at C_{j-1}. In order to show that s_j is an i-redirection or an i-insertion, we still need to show that R_1's i-followers set at C_{j-1} is a subset of R_3's i-followers set at C_{j-1}.

If R_3's i-successor at C_{j-1} is R_2, then, since R_2 is R_1's i-successor at C_{j-1} (the cas execution is successful), R_1's i-followers set at C_{j-1} is a subset of R_3's i-followers set at C_{j-1}.

Otherwise (R_3's i-successor at C_{j-1} is not R_2), there exists $j' < j-1$ for which R_2 is R_3's i-successor at $C_{j'}$ (right after the successful cas execution in line 48). From the induction assumption (invariant 4), R_2 is not an i-infant at $C_{j'}$. In addition, by definitions 10-11, R_2's i-inclusive followers set at $C_{j'}$ is a subset of R_3's i-followers set at $C_{j'}$. By the induction assumption, R_2's i-inclusive followers set at C_{j-1} is a subset of R_3's i-followers set at C_{j-1}. Since R_2 is R_1's i-successor at C_{j-1}, R_1's i-followers set at C_{j-1} is a subset of R_3's i-followers set at C_{j-1} in this case also.

Therefore, when s_j is a successful cas execution in line 49, if R_3 is an i-infant at C_{j-1} then s_j is an i-insertion and otherwise, s_j is an i-redirection.

There are no other instructions that can change R_1's i-th $mraked$ flag or i-successor and thus, the invariant holds.

11) Suppose $R_1 <_i R_2$. Assume by contradiction that R_1 i-follows R_2 at C_j. By definition, there exist records $R_{i_0}, \ldots, R_{i_m}$ for which $R_2 = R_{i_0}$, $R_1 = R_{i_m}$ and for every $0 < t \le m$, $R_{i_{t-1}} \ne R_{i_t}$ and R_{i_t} is $R_{i_{t-1}}$'s i-successor at C_j. By the induction assumption (invariant 2), for every $0 < t \le m$, R_{i_t} is also not an i-infant at C_j and therefore, from claim 14, either $R_{i_t} <_i R_{i_{t-1}}$ or $R_{i_{t-1}} <_i R_{i_t}$. Now, let $0 < t \le m$ be the minimal value for which $R_{i_t} <_i R_{i_{t-1}}$ (since $R_1 <_i R_2$, it must exist). By the induction assumption, R_{i_t} does not i-follow $R_{i_{t-1}}$. Therefore, by invariant 10, s_j is either an i-snipping, an i-redirection or an i-insertion, changing $R_{i_{t-1}}$'s i-successor (it cannot be an i-marking, since it

does not change any record's i-successor).
Suppose s_j is an i-snipping. Then there must exist a record R_3 which is $R_{i_{t-1}}$'s i-successor and an i-predecessor of R_{i_t} at C_{j-1}. By definition, R_{i_t} i-follows $R_{i_{t-1}}$ at C_{j-1} - a contradiction. Suppose s_j is an i-redirection. Then by definition 17, $R_{i_{t-1}} <_i R_{i_t}$ - a contradiction. Suppose s_j is an i-insertion. Then by definition 18, $R_{i_{t-1}} <_i R_{i_t}$ - a contradiction. Therefore, R_1 does not i-follow R_2 at C_j.

12) Suppose R_1 is i-reachable and is not i-marked at C_{j-1}. By definition, there exist records $R_{i_0}, \ldots, R_{i_m}$ for which *head* $= R_{i_0}$, $R_1 = R_{i_m}$ and for every $0 < t \le m$, R_{i_t} is $R_{i_{t-1}}$'s i-successor at C_{j-1}.
Assume by contradiction that R_1 is not i-reachable at C_j. If $R_{i_{m-1}}$ is still i-reachable at C_j, then R_1 is not its i-successor at C_j. Therefore, by invariant 10, s_j is either an i-snipping, an i-redirection or an i-insertion, changing $R_{i_{m-1}}$'s i-successor (it cannot be an i-marking, since it does not change any record's i-successor). Suppose s_j is an i-snipping. Then R_1 is i-marked at C_{j-1} - a contradiction. Suppose s_j is an i-redirection or an i-insertion. Since R_1 is not i-marked at C_{j-1}, it belongs to $R_{i_{m-1}}$'s i-followers set at C_{j-1}. By definition 17, R_1 belongs to $R_{i_{m-1}}$'s i-followers set at C_j and thus, it is i-reachable at C_j - a contradiction.
Therefor, $R_{i_{m-1}}$ is not i-reachable at C_j. Let $0 \le t < m-1$ be the maximal value for which R_{i_t} is i-reachable at C_j (t must exist, since *head* is still i-reachable at C_j). By definition, $R_{i_{t+1}}$ is not R_{i_t}'s i-successor at C_j and therefore, by invariant 10, s_j is either an i-snipping, an i-redirection or an i-insertion, changing R_{i_t}'s i-successor. Suppose s_j is an i-snipping. Then $R_{i_{t+2}}$ is i-reachable at C_j - a contradiction. Suppose s_j is an i-redirection or an i-insertion. Then R_1 i-follows R_{i_t} at C_j, and therefore, is i-reachable at C_j - a contradiction.
Therefore, R_1 is i-reachable at C_j.

13) Suppose R_1 is not an i-infant at C_{j-1} and is not i-marked at C_j. From claim 8, R_1 is also not i-marked at C_{j-1}.
If R_1 is an i-infant at C_{j-1} then s_j is the successful cas execution in line 49, having $newRecord \rightsquigarrow R_1$. By invariant 10, s_j is either an i-insertion or an i-redirection. Since R_1 is an i-infant at C_{j-1}, s_j is an i-insertion. Let R_3 be the record for which $pred \rightsquigarrow R_3$ at C_j. By definition, R_3 is not an i-infant at C_{j-1} and is not i-marked at C_{j-1}. By the induction assumption, R_3 is i-reachable at C_{j-1}. Since s_j only affects R_3's i-successor, R_3 is still i-reachable at C_j. Therefore, since R_1 is R_3's i-successor at C_j, it is also i-reachable at C_j.
If R_1 is not an i-infant at C_{j-1} then by the induction assumption, it is i-reachable at C_{j-1}. From invariant 12, R_1 is i-reachable at C_j in this case as well.

14) Suppose R_2 belongs to R_1's i-inclusive followers set at C_{j-1} and is not i-marked at C_j. By definition 11 it i-follows R_1 at C_{j-1}. By definition, there exist records $R_{i_0}, \ldots, R_{i_m}$ for which $R_1 = R_{i_0}$, $R_2 = R_{i_m}$ and for every $0 < t \le m$, R_{i_t} is $R_{i_{t-1}}$'s i-successor at C_{j-1}.
Assume by contradiction that R_2 does not belong to R_1's i-inclusive followers set at C_j. From definition 11, it does not i-follow R_1 at C_j. If $R_{i_{m-1}}$ still i-follows R_1 at C_j, then R_2 is not its i-successor at C_j. Therefore, by invariant 10, s_j is either an i-snipping, an i-redirection or an i-insertion, changing $R_{i_{m-1}}$'s i-successor (it cannot be an i-marking, since it does not change any record's i-successor). Suppose s_j is an i-snipping. Then R_2 is i-marked at C_{j-1} - a contradiction. Suppose s_j is an i-redirection or an i-insertion. Since R_2 is not i-marked at C_{j-1}, it belongs to $R_{i_{m-1}}$'s i-followers set at C_{j-1}. By definition 17, R_2 belongs to $R_{i_{m-1}}$'s i-followers set at C_j and thus, it i-follows R_1 at C_j - a contradiction.
Therefor, $R_{i_{m-1}}$ does not i-follow R_1 at C_j. Let $0 \le t < m-1$ be the maximal value for which R_{i_t} i-follows R_1 at C_j (t must exist, since by definition, R_1 i-follows R_1 at C_j). By definition, $R_{i_{t+1}}$ is not R_{i_t}'s i-successor at C_j and therefore, by invariant 10, s_j is either an i-snipping, an i-redirection or an i-insertion, changing R_{i_t}'s i-successor. Suppose s_j is an i-snipping. Then $R_{i_{t+2}}$ i-follows R_1 at C_j - a contradiction. Suppose s_j is an i-redirection or an i-insertion. Then R_2 i-follows R_{i_t} at C_j, and therefore, i-follows R_1 at C_j - a contradiction.
Therefore, R_2 belongs to R_1's i-inclusive followers set at C_j.

Using claim 21, we can now easily prove that most invariants hold for our table (lemmas 22, 23 and 24). Notice that by the combination of lemma 22 and lemma 23, *head* and *tail* are always the first and last records of every internal linked-list (respectively).

Lemma 22: Let $\alpha = C_0 \cdot s_1 \cdot C_1 \cdot s_2 \cdot C_2 \cdot \ldots \cdot s_k \cdot C_k$ be a legal execution, let $0 \le j \le k$ and let $1 \le i \le N_T$. Then *head* and *tail* are i-reachable at C_j.

Proof 8: By definition, *head* is i-reachable at C_j. Therefore, we only need to show that *tail* is i-reachable at C_j. By claim 9, *tail* is not i-marked at C_j. In addition, by definition, *tail* is not an i-infant at C_j. By claim 21 (invariant 13), *tail* is i-reachable at C_j.

Lemma 23: Let $\alpha = C_0 \cdot s_1 \cdot C_1 \cdot s_2 \cdot C_2 \cdot \ldots \cdot s_k \cdot C_k$ be a legal execution, let $0 \le j \le k$, $1 \le i \le N_T$ and let R_1 and R_2 be two different records. If R_2 i-follows R_1 at C_j then $R_1.data[i] \le R_2.data[i]$.

Proof 9: Assume by contradiction that $R_1.data[i] > R_2.data[i]$. By definition, $R_2 <_i R_1$. By claim 21 (invariant 11), R_2 does not i-follow R_1 at C_j - a contradiction. Therefore, if R_2 i-follows R_1 at C_j then $R_1.data[i] \le R_2.data[i]$.

Lemma 24: Let $\alpha = C_0 \cdot s_1 \cdot C_1 \cdot s_2 \cdot C_2 \cdot \ldots \cdot s_k \cdot C_k$ be a legal execution, let $0 \le j \le k$ and let R be a record. R's status at C_j is *InTable* if and only if R is logically in the table at C_j.

Proof 10: If R is logically in the table at C_j then by definition, R's status at C_j is *InTable*. Now, suppose R's status at C_j is *InTable*. In order to show that R is logically in the table at C_j, we need to show that for every $1 \leq i \leq N_T$, R is i-reachable at C_j.

Since both *head* and *tail* are i-reachable at C_0, and are the only records at C_0, it holds for C_0. Assume that it holds throughout $\alpha' = C_0 \cdot s_1 \cdot C_1 \cdot \ldots \cdot C_{j-1}$ of the execution. We show that it holds throughout $\alpha = \alpha' \cdot s_j \cdot C_j$.

If R's status at C_{j-1} is also *InTable*, then by the induction assumption, R is i-reachable at C_{j-1}. If R is *head* then by definition, R is still i-reachable at C_j. Otherwise, again by definition, R has an i-predecessor at C_{j-1} and thus, by claim 21 (invariant 2), R is not an i-infant at C_{j-1}. By definition, R is also not an i-infant at C_j. From claim 8, R is not i-marked at C_j and therefore, from claim 21 (invariant 13), R is i-reachable at C_j.

If R's status at C_{j-1} is not *InTable*, then by claim 8, it is *Pending*. Therefore, s_j is a successful cas execution in line 31, satisfying $newRecord \rightsquigarrow R$. when the method returns from the *helpAddingField(*$newRecord, i$*)* call in line 30, R's status is *Pending* (again, by claim 8). Therefore, it either returned in line 39 or 50.

In both cases, R had an i-predecessor before returning and by claim 21 (invariant 2), R is not an i-infant at C_j. From claim 8, R is not i-marked at C_j and therefore, from claim 21 (invariant 13), R is i-reachable at C_j in this case as well. After showing that most of our table invariants hold throughout any legal execution, we still need to show that the uniqueness property is never violated. Before showing that in lemma 26, we prove the following claim:

Claim 25: Let $\alpha = C_0 \cdot s_1 \cdot C_1 \cdot s_2 \cdot C_2 \cdot \ldots \cdot s_k \cdot C_k$ be a legal execution, let $0 \leq j \leq k$, $1 \leq i \leq N_T$ such that i is a unique field number, and let R_1, R_2 be two different records for which $R_1.data[i] = R_2.data[i]$, then if R_1 is an i-predecessor of R_2 at C_j, R_2's status at C_j is either *Removed* or *Failed*.

Proof 11: Since $head.data[i] < tail.data[i]$, the claim is vacuously true at C_0. Assume that the lemma holds throughout $\alpha' = C_0 \cdot s_1 \cdot C_1 \cdot \ldots \cdot C_{j-1}$ of the execution. We show that it holds throughout $\alpha = \alpha' \cdot s_j \cdot C_j$.

If R_1 is also an i-predecessor of R_2 at C_{j-1}, then by the induction assumption, R_2's status at C_{j-1} is either *Removed* or *Failed*. From claim 8, R_2's status at C_j is also *Removed* or *Failed*. Otherwise, s_j updates R_1's i-successor, by executing a successful cas in line 8, 48 or 49.

If s_j is a successful cas execution in line 8, then $pred \rightsquigarrow R_1$ and $succ \rightsquigarrow R_2$. Let R_3 be the record satisfying $curr \rightsquigarrow R_3$ at C_j. Since R_1's i-successor at C_{j-1} is R_3 (the cas is successful) and R_3's i-successor at an earlier point (when assigning R_2 into $succ$) is R_2, by claim 21 (invariant 8), $R_1.data[i] \leq R_3.data[i] \leq R_2.data[i]$. Since $R_1.data[i] = R_2.data[i]$, $R_3.data[i] = R_2.data[i]$ and therefore (by the induction assumption), there exists an earlier point for which R_2's status is either *Removed* or *Failed*. From claim 8, R_2's status at C_j is also *Removed* or *Failed*.

If s_j is a successful cas execution in line 48, then $newRecord \rightsquigarrow R_1$ and $curr \rightsquigarrow R_2$. Assume by contradiction that R_2's status at C_j is *Pending*. Since i is a unique field number, the *helpAdding(curr,i+1)* (from claim 8, R_2's status is *Pending* at every earlier configuration). Before returning from that method, the cas execution in line 31 was either successful or unsuccessful. For both cases, from claim 8, R_2's status when execution line 45 is not *Pending* and therefore, it is not *Pending* at C_j - a contrdiction. Now, assume by contrdiction that R_2's status at C_j is *InTable*. If R_2's staus when executing line 45 is *InTable* then line 48 is not executed. Since it also cannot be *Pending*, we get a contradiction here as well. Therefore, when executing line 45, R_2's status is either *Removed* or *Failed*. From claim 8, R_2's status at C_j is also *Removed* or *Failed*.

If s_j is a successful cas execution in line 49, then $pred \rightsquigarrow R_1$ and $newRecord \rightsquigarrow R_2$. By claim 20), $R_1.data[i] < R_2.data[i]$ - a contradiction.

Lemma 26: Let $\alpha = C_0 \cdot s_1 \cdot C_1 \cdot s_2 \cdot C_2 \cdot \ldots \cdot s_k \cdot C_k$ be a legal execution, let $0 \leq j \leq k$, $1 \leq i \leq N_T$ such that i is a unique field number, and let R_1, R_2 be two different records. If both R_1 and R_2 are logically in the table at C_j then $R_1.data[i] \neq R_2.data[i]$.

Proof 12: Assume by contradiction that both R_1 and R_2 are logically in the table at C_j and that $R_1.data[i] = R_2.data[i]$. By lemma 24, both R_1 and R_2 are i-reachable at C_j and therefore, from claim 6, either R_1 i-follows R_2 or R_2 i-follows R_1 at C_j. Assume without loss of generality that R_2 i-follows R_1 at C_j. By definition, there exist records $R_{i_0}, \ldots, R_{i_m}$ for which $R_1 = R_{i_0}$, $R_2 = R_{i_m}$ and for every $0 < t \leq m$, R_{i_t} is the i-successor of $R_{i_{t-1}}$ at C_j. From claim 21 (invariant 8), for every $0 \leq t < m$, $R_{i_t}.data[i] \leq R_{i_{t+1}}.data[i]$. Since $R_1.data[i] = R_2.data[i]$, $R_{i_{m-1}}.data[i] = R_2.data[i]$. From claim 25, R_2's status at C_j is either *Failed* or *Removed* and thus, R_2 is not logically in the table at C_j - a contradiction. Conclusion: If both R_1 and R_2 are logically in the table at C_j then $R_1.data[i] \neq R_2.data[i]$.

B. *Linearizability proof*

Our table implementation is linearizable. This means that, for every execution, one can assign a linearization point to each completed operation and some of the uncompleted operations so that the linearization point of each operation occurs after the operation starts and before it ends, and the results of these operations are the same as if they had been performed sequentially, in the order of their linearization points.

Let $\alpha = C_0 \cdot s_1 \cdot C_1 \cdot s_2 \cdot C_2 \cdot \ldots \cdot s_k \cdot C_k$ be a legal execution. We shall define linearization points for each of the *add*, *remove* and *retrieve* operations, terminated before C_k. In addition, it will be convenient to define linearization points for calls to the *find* method, terminated before C_k. The first goal is to show that at any time, the set of tuples represented by records which are currently logically in the table is exactly the set of tuples for which there exists an already linearized *add* call inserting them into the table, and there does not exist an already linearized *remove* call removing them from the table. The second goal is to show that the output of each operation matches the table status, as will be explained next.

a) find: First, we define linearization points for the *find(val, i)* method. We wish to choose a linearization point for the *find(val, i)* so the two records $pred, curr$, returned as output, satisfy *(1)* $pred.data[i] < val \leq curr.data[i]$, *(2)* $pred$ and $curr$ are both i-reachable at the linearization point, and *(3)* $curr$ is $pred$'s i-successor at the linearization point. Notice that by the time the method returns, $pred$ and $curr$ may no longer satisfy those conditions.

Claim 27: If a *find(val, i)* method was invoked at s_{j_1}, and returned at s_{j_2} ($0 < j_1 < j_2 \leq k$) with R_1 and R_2 as the output parameters ($pred \rightsquigarrow R_1$, $curr \rightsquigarrow R_2$), then there exists j for which:

1) $j_1 < j < j_2$.
2) R_2 is R_1's i-successor at C_j.
3) Both R_1 and R_2 are i-reachable at C_j.
4) For every record R_3 which is i-reachable at C_j and satisfying $R_3.data[i] < val$, R_3 i-precedes R_1 at C_j.
5) For every record R_3 which is i-reachable at C_j and satisfying $R_3.data[i] \geq val$, R_3 i-follows R_2 at C_j.

Proof 13: Suppose we have a number j satisfying 1-3. We can now prove 4 and 5 using j: Suppose R_3 is i-reachable at C_j and $R_3.data[i] < val$. Since R_1 is i-reachable at C_j, from claim 6, R_3 either i-follows or i-precedes R_1 at C_j. Assume by contradiction that R_3 does not i-precede R_1 at C_j. Then R_3 i-follows R_1 at C_j and $R_1 \neq R_3$. Therefore, R_3 i-follows R_2 at C_j. From claim 20, $R_2.data[i] \geq val$. Since $R_3.data[i] < val \leq R_2.data[i]$, by definition 13, $R_3 <_i R_2$. From claim 21 (invariant 11), R_3 does not i-follow R_2 at C_j - a contradiction. Therefore, every record R_3 which is i-reachable at C_j and satisfying $R_3.data[i] < val$, R_3 i-precedes R_1 at C_j. Now Suppose R_3 is i-reachable at C_j and $R_3.data[i] \geq val$. Since R_2 is i-reachable at C_j, from claim 6, R_3 either i-follows or i-precedes R_2 at C_j. Assume by contradiction that R_3 does not i-follow R_2 at C_j. Then R_3 i-precedes R_2 at C_j and $R_2 \neq R_3$. Since R_1 is also i-reachable at C_j, R_3 either i-follows or i-precedes R_1 at C_j. If it i-follows R_1, since it does not i-follow R_2 and R_2 is R_1's i-successor at C_j, R_3 must be R_1 and from claim 20, $R_3.data[i] < val$ - a contradiction. Otherwise, R_3 must i-precede R_1 at C_j, and $R_1 \neq R_3$ as well. Since R_1 is not an i-infant at C_j (claim 21, invariant 4), either $R_1 <_i R_3$ or $R_3 <_i R_1$. Since R_3 i-precedes R_1 at C_j, by claim 21 (invariant 11), $R_3 <_i R_1$. By definition 13, $R_3.data[i] \leq R_1.data[i]$. Therefore, $R_3.data[i] < val$ - a contradiction. Since every possible case results in a contradiction, our assumption was wrong. Therefore, for every record R_3 which is i-reachable at C_j and satisfying $R_3.data[i] \geq val$, R_3 i-follows R_2 at C_j.
We are now going to show that j indeed exists, and that it satisfies 1-3.
If the last assignment of R_2 into the $curr$ variable during the method execution is in line 4, we choose s_j to be this assignment.

1) $j_1 < j < j_2$: Since s_j is an execution of an instruction during the method execution, obviously $j_1 < j < j_2$ holds.
2) R_2 is R_1's i-successor at C_j: By definition.
3) R_1 and R_2 are i-reachable at C_j: If the last assignment of R_2 into the $curr$ variable during the method execution is in line 4, then R_1 is the *head* record. By definition, R_1 is i-reachable at C_j. Since R_2 is R_1's i-successor at C_j, it is also i-reachable at C_j.

If the last assignment of R_2 into the $curr$ variable during the method execution is in line 11, let s_{j_4} be this assignment and let s_{j_3} be the last execution of line 8 before s_{j_4}. If R_2 is R_1's i-successor at s_{j_3}, we choose $j = j_3$.

1) $j_1 < j < j_2$: Since s_{j_3} is an execution of an instruction during the method execution and $j = j_3$, obviously $j_1 < j < j_2$ holds.
2) R_2 is R_1's i-successor at C_j: j is chosen to be j_3 because R_2 is R_1's i-successor at s_{j_3}.
3) R_1 and R_2 are i-reachable at C_j: The last execution of line 8 before s_{j_4} must be a successful cas execution (otherwise, we go back to line 2). Therefore, R_1 is not i-marked at C_{j_3} and from claim 8, it is not i-marked at $C_{j'}$ for every $j' \leq j_3$. From claim 21 (invariant 4), R_1 is not an i-infant at C_{j_3} and therefore, from claim 21 (invariant 13), R_1 is i-reachable at C_j ($= C_{j_3}$). Since R_2 is R_1's i-successor at C_j, it is also i-reachable at C_j.

Otherwise, if R_2 is not R_1's i-successor at s_{j_3}, since R_2 is R_1's i-successor at C_{j_4}, there exists $j_3 < j' < j_4$ for which $s_{j'}$ makes R_1's $next[i]$ field point to R_2. In this case, we choose $j = j'$.

1) $j_1 < j < j_2$: Holds since $j_1 < j_3 < j' < j_4 < j_2$.
2) R_2 is R_1's i-successor at C_j: Holds since s_j makes R_1's $next[i]$ field point to R_2.
3) R_1 and R_2 are i-reachable at C_j: Since R_1's i-successor at C_j is different from R_1's i-successor at C_{j-1}, by claim 8, R_1 is not i-marked at C_j. As explained in the former case, R_1 is i-reachable at C_j, and so is R_2.

If the last assignment of R_2 into the $curr$ variable during the method execution is in line 16, let s_{j_6} be this assignment. There exists $j_5 < j_6$ for which s_{j_5} is the last assignment of R_2 into the $succ$ variable, either in line 6 or 12. In this case, we choose $j = j_5$. Notice that $curr \rightsquigarrow R_1$ at C_{j_5}.

1) $j_1 < j < j_2$: Since s_{j_5} is an execution of an instruction during the method execution, obviously $j_1 < j < j_2$ holds.
2) R_2 is R_1's i-successor at C_j: Holds both possibilities, since $curr \rightsquigarrow R_1$, $succ \rightsquigarrow R_2$ and $curr.next[i] \rightsquigarrow succ$ at C_{j_5}.
3) R_1 and R_2 are i-reachable at C_j: From claim 21 (invariant 4), R_1 is not an i-infant at C_{j_5}. Line 7 must be executed between s_{j_5} and s_{j_6}. Since s_{j_5} is the last assignment of R_2 into the $succ$ variable before s_{j_6}, the condition checked must hold and therefore, R_1 is not i-marked at this point. From claim 8, R_1 is also not i-marked at C_{j_5}. As explained in the former case, R_1 is i-reachable at C_j ($= C_{j_5}$), and so is R_2.

Since the last assignment into the $curr$ variable must be one of the above, the claim holds.

We define the linearization point of a *find*(val, i) execution that has terminated to be C_j, described in claim 27. Note that there may be several j's satisfying the conditions of the claim and any of them can be chosen. From claim 20, the records $pred, curr$, returned as output, indeed satisfy $pred.data[i] < val \leq curr.data[i]$. In addition, from claim 27, $pred$ and $curr$ are both i-reachable at the linearization point, and $curr$ is $pred$'s i-successor.

b) add: After defining the linearization points of each *find*(val, i) execution which terminated, we can define the linearization points of each *add(tup)* execution which terminated. We define a successful *add* execution as an execution which returned $true$, and an unsuccessful *add* execution as an execution which returned $false$.

Given a successful *add(tup)* and the record $newRecord$, created in line 18, we want to set a linearization point which is between the invocation and the termination of the operation, and for which $newRecord$ is logically in the table.

Claim 28: Given an *add(tup)* operation which was invoked at s_{j_1} and terminated at s_{j_2} ($0 < j_1 < j_2 \leq k$), let R be the record created in line 18 (meaning, $newRecord \rightsquigarrow R$ during the execution). If the operation returned $true$ then there exists $j_1 < j < j_2$ for which R is logically in the table at C_j.

Proof 14: Using lemma 24, it suffices to show that there exists $j_1 < j < j_2$ for which R's status at C_j is *InTable*.

Before returning from the *helpAdding*($newRecord, 1$) call in line 19, the operation must have tried to change R's status vis a cas execution in line 31. Let $s_{j'}$ be this cas execution. Obviously, $j_1 < j' < j_2$. If R's status at $C_{j'-1}$ is *Pending*, then this cas execution is successful and R's status at $C_{j'}$ is *InTable*. If the cas execution is unsuccessful because R's status is already *InTable* at $C_{j'-1}$, then R's status at $C_{j'}$ is also *InTable*. For both cases, we choose $j = j'$ and we are done. If R's status at $C_{j'-1}$ is *Removed* then, from claim 8, and since R was created with a *Pending* status in line 18, there exists $j_1 < j'' < j'-1$ for which R's status at $C_{j''}$ is *InTable*. In this case, we choose $j = j''$. Since the operation returns true in line 27, then R's status when executing line 20 is not *Failed*. From claim 8, it cannot be *Failed* at $C_{j'}$. Therefore, C_j is well-defined.

We define the linearization point of a successful *add(tup)* execution that has terminated to be C_j, described in claim 28. From claim 28, it is guaranteed that the new record created during the operation, and which represents the tuple tup, has been added into the table during the execution.

Given an unsuccessful *add(tup)* operation, we want to set a linearization point which is between the invocation and the termination of the operation, for which a record representing tup cannot be inserted into the table. Failure can only happen if a record with an equal value of a unique field is logically in the list. Formally, let $tup = \langle e_1, \ldots, e_{N_T} \rangle$ and let $i \in \{1, \ldots, N_T\}$ be a unique field number. We want to show that at the operation linearization point there exists a record R' which is not the record created in line 18 during the operation, satisfying $R'.data[i] = e_i$, and is logically in the table during that linearization point.

Claim 29: Given an *add(tup)* operation which was invoked at s_{j_1} and terminated at s_{j_2} ($0 < j_1 < j_2 \leq k$), let R be the record created in line 18 (meaning, $newRecord \rightsquigarrow R$ during the execution). If the operation returned $false$ then there exist *(1)* $j_1 < j < j_2$, *(2)* $i \in \{1, \ldots, N_T\}$ for which i is a unique field number, and *(3)* a record $R' \neq R$ whose status is *InTable* at C_j and for which $R'.data[i] = R.data[i]$.

Proof 15: If the operation returned $false$, then R's status when executing line 20 was *Failed*. Since new records are created with a *Pending* status, R's status was changed in a successful cas execution in line 46, during a *helpAddingField*($newRecord, i$) for which $newRecord \rightsquigarrow R$. Let R' be the record for which $curr \rightsquigarrow R'$ when the cas is executed, and let s_j be the reading of the status of R' in line 45. Since s_j was after R was created and before the execution of line 20, $j_1 < j < j_2$. In addition, since line 46 was executed, i is a unique field number, $R'.data[i] = R.data[i]$, and the status of R' at C_j was *InTable*.

We still need to show that $R \neq R'$. Notice that, since line 46 was executed, the condition checked in line 38 did not hold for $tmp \rightsquigarrow R'$. Therefore, $R \neq R'$.

We define the linearization point of an unsuccessful *add(tup)* execution that has terminated to be C_j, described in claim 29. From claim 29, it is guaranteed that inserting the new record into the table at C_j would break the uniqueness property of our table.

c) remove: We define a successful *remove* execution as an execution which returned $true$, and an unsuccessful *remove* execution as an execution which returned $false$. Assume a successful *remove*(val, i) was invoked at s_{j_1} and returned at s_{j_2}. Let R be the record found in line 52 (meaning, $victim \rightsquigarrow R$). We want to set a linearization point s_j for which $j_1 < j < j_2$, and s_j is an operation step that logically removes R from the table. Notice that, since the operation is successful, we can choose s_j to be the successful cas execution in line 55 changing R's status from *InTable* to *Removed*. Obviously, s_j is executed by the operation, and $j_1 < j < j_2$. In addition, by lemma 24, s_j indeed logically removes R from the table.

Now, assume an unsuccessful *remove*. We want to set a linearization point between the invocation and termination of the operation, for which there does not exist a record that can be removed by the operation (meaning, the operation failure is justified).

Claim 30: Given a *remove*(val, i) operation, invoked at s_{j_1} and terminated at s_{j_2} ($0 < j_1 < j_2 \leq k$), let R be the record found in line 52 (meaning, $victim \rightsquigarrow R$ during the execution). If the operation returned $false$ then there exists $j_1 < j < j_2$ for which there does not exist a record R_1 satisfying $R_1.data[i] = val$ that is logically in the table at C_j.

Proof 16: Let C_{j_3} be the linearization point of the *find*(val, i) call in line 52, and let R_2 be the $pred$ record, returned as output from that call. Obviously, $j_1 < j_3 < j_2$. From claim 27, both R_2 and R are i-reachable and R is

R_2's i-successor at C_{j_3}. From claim 20, $R_2.data[i] < val \leq R.data[i]$.

Now, let $R_1 \neq R$ be a record for which $R_1.data[i] = val$. If R_1 is i-reachable at C_{j_3} then by claim 6, it either i-precedes or i-follows R_2 at C_{j_3} and either i-precedes or i-follows R at C_{j_3}. Since $R_2 <_i R_1$, from claim 21 (invariant 11), R_1 cannot i-precede R_2. Therefore, since $R_1 \neq R$, R_1 i-follows R at C_j. Using claim 21 again, it must hold that $R <_i R_1$ and thus, $R.data[i] \leq val$. Therefore, $R.data[i] = val$. Since we assume i is a unique field number (it was sent as input to a *remove* call), by claim 25, R_1's status at C_{j_3} is not *InTable* (meaning, it is not logically in the table). Conclusion: for every record $R_1 \neq R$ such that $R_1.data[i] = val$, R_1 is not logically in the table at C_{j_3}.

If the operation returned $false$ in line 54, we choose $j = j_3$. Obviously, $j_1 < j < j_2$. From claim 20, $R_2.data[i] < val \leq R.data[i]$. Since the operation returned in line 54, $R.data[i] \neq val$ and therefore, $val < R.data[i]$. Since $R.data[i] \neq val$ and there does not exist any other record R_1 for which $R_1.data[i] = val$, and that is logically in the table at C_j, j satisfies all conditions for that case, and we are done.

If the operation returned $false$ in line 56 and R's status at C_{j_3} is not *InTable* then R is not logically in the table at C_{j_3}. In this case, we choose $j = j_3$. As explained above, there also does not exist any other record R_1 for which $R_1.data[i] = val$, and that is logically in the table at C_j. Therefore, j satisfies all conditions for that case as well.

If the operation returned $false$ in line 56, but R's status at C_{j_3} is *InTable*, let s_{j_4} be the unsuccessful *cas* execution in line 55. Since the *cas* execution in line 55 is unsuccessful, R's status at C_{j_4-1} is not *InTable*. Let C_j be the first configuration between C_{j_3} and C_{j_4-1} for which R's status is not *InTable*. Obviously, $j_1 < j < j_2$. Since R is logically in the table at C_{j-1} and $R.data[i] = val$ (the condition checked in line 53 does not hold), from lemma 26, there does not exist $R_1 \neq R$ which is logically in the table at C_{j-1} and for which $R_1.data[i] = val$. Since s_j just changes R's status, there also does not exist $R_1 \neq R$ which is logically in the table at C_j and for which $R_1.data[i] = val$. Since R is also not logically in the list at C_j (its status is not *InTable* at C_j), there does not exist a record R_1 which is logically in the table at C_j and for which $R_1.data[i] = val$. Therefore, j satisfies all conditions for that case as well.

We define the linearization point of an unsuccessful *remove(val, i)* execution that has terminated to be C_j, described in claim 30. From claim 30, it is guaranteed that indeed there does not exist any record that can be removed at C_j, according to the operation input parameters.

d) retrieve: For each *retrieve(val, i)* execution that has terminated, we want to set a linearization point for which the set of tuples returned by the operation is the set of all tuples $t = \langle e_i, \ldots, e_{N_T} \rangle$ which are currently in the table and satisfy $t.e_i = val$. Formally, assume a *retrieve(val, i)* execution, invoked at s_{j_1} and terminated at s_{j_2} (returning a set S of tuples). We want to set a linearization point C_j for which *(1)* $j_1 < j < j_2$, *(2)* for every record R satisfying $R.data[i] = val$, it holds that $R.data \in S$ if and only if R is logically in the table at C_j.

Claim 31: Let $\alpha = C_0 \cdot s_1 \cdot C_1 \cdot s_2 \cdot C_2 \cdot \ldots \cdot s_k \cdot C_k$ be a legal execution, and let $1 \leq i \leq N_T$. If there exist records $R_1, R_2, \ldots, R_m$ and $j_1 < j_2 < \ldots < j_{m-1}$ such that for every $1 \leq t \leq m-1$, R_t is the i-predecessor of R_{t+1} at C_{j_t}, then there exists $j_1 \leq j \leq j_{m-1}$ for which R_{m-1} i-follows R_1 and R_m is the i-successor of R_{m-1} at C_j.

Proof 17: We are going to prove the claim by induction on m.

Base Case ($m = 2$): the claim obviously holds for j_1.

Induction Step: Assume that the claim holds for $m-1$, and let $R_1, R_2, \ldots, R_m$ and $j_1 < j_2 < \ldots < j_{m-1}$ be the described records and numbers. By the induction assumption, there exists $j_1 \leq j' \leq j_{m-2}$ for which R_{m-2} i-follows R_1 and R_{m-1} is the i-successor of R_{m-2} at $C_{j'}$. By definition, R_{m-1} i-follows R_1 at $C_{j'}$. If R_m is the i-successor of R_{m-1} at $C_{j'}$ then the claim holds for $j = j'$.

If R_m is not the i-successor of R_{m-1} at $C_{j'}$ then there exists $j' < j \leq j_{m-1}$ for which R_m is not the i-successor of R_{m-1} at C_{j-1} and R_m is the i-successor of R_{m-1} at C_j. From claim 8, R_{m-1} is not i-marked at C_j and therefore, not i-marked at $C_{j_{m-1}}$. By definition 11, R_{m-1} belongs to R_1's i-inclusive followers set at $C_{j_{m-1}}$. Since R_{m-1} is not i-marked at C_j, from claim 21 (invariant 14), it also belongs to R_1's i-inclusive followers set at $C_{j_{m-1}}$. Therefore, R_{m-1} i-follows R_1 at C_j. Since R_m is the i-successor of R_{m-1} at C_j, the claim holds for j.

Let $R_1, \ldots, R_m$ be the records assigned into the $curr$ variable during the last execution of lines 66-68 before the operation returns, and for every $0 \leq t < m$, let s_{j_t} be the step assigning R_{t+1} into $curr$. Obviously, $j_0 < j_1 < \ldots < j_{m-1}$ and for every $1 \leq t \leq m-1$, R_t is an i-predecessor of R_{t+1} at C_{j_t}. From claim 31, there exists $j_1 \leq j' \leq j_{m-1}$ for which R_{m-1} i-follows R_1 and R_m is R_{m-1}'s i-successor at $C_{j'}$. Since R_1 is the *head* record, by definition, both R_{m-1} and R_m are i-reachable at $C_{j'}$.

Now, if the operation returns in line 70, then S is empty. In this case, we want to prove that there does not exist any record R for which $R.data[i] = val$, and that is logically in the table at $C_{j'}$ (obviously satisfying $j_1 < j' < j_2$).

Claim 32: Let $j = j'$. If the operation returns in line 70, then there does not exist any record R for which $R.data[i] = val$, and that is logically in the table at C_j.

Proof 18: Assume by contradiction that there exists such record R. Since R is logically in the table at C_j, by definition it is i-reachable at C_j. Since the operation returns in line 70, $R_{m-1}.data[i] < val < R_m.data[i]$. Therefore, by definition 13, $R_{m-1} <_i R <_i R_m$. from claim 21 (invariant 11), R does not i-precede R_{m-1} and does not i-follow R_m at C_j. Since R_m is R_{m-1}'s i-successor at C_j, R does not i-follows R_{m-1} at C_j as well - a contradiction to claim 6. Therefore, there does not exist any record R for which $R.data[i] = val$, and that is logically in the table at C_j.

If the operation returns in line 91, then S is the $tuples$ set, returned in this line. Let $R'_1, \ldots, R'_n$ be the records assigned into the $curr$ variable during the last execution of lines 77-79 before the operation returns, and for every $0 \leq t < n$, let $s_{j'_t}$ be the step assigning R'_{t+1} into $curr$. Obviously, $j'_0 < j'_1 < \ldots < j'_{n-1}$ and for every $1 \leq t \leq n-1$, R'_t is an i-predecessor of R'_{t+1} at $C_{j'_t}$. From claim 31, there exists $j'_1 \leq j'' \leq j'_{n-1}$ for which R'_{n-1} i-follows R'_1 and R'_n is R'_{n-1}'s i-successor at $C_{j''}$. Since R'_1 is the *head* record, both R'_{n-1} and R'_n are i-reachable at $C_{j''}$. In addition, since the condition checked in line 80 must hold, $R_m = R'_n$.

We are going to show that for every record R it holds that $R.data \in S$ if and only if $R.data[i] = val$ and R is logically in the table at $C_{j''}$. Since $j_1 < j'' < j_2$, we can use $C_{j''}$ as our linearization point when the operation returns in line 91.

Claim 33: Let $j = j''$. For every record R it holds that $R.data \in S$ if and only if $R.data[i] = val$ and R is logically in the table at C_j.

Proof 19: Let R be a record for which $R.data \in S$. Then R was inserted, with its status, into $tmpSet$ in line 75 (and therefore, $R.data[i] = val$), and $R.data$ was inserted into S in line 86. Since the insertion into $tmpSet$ happened before C_j, and R's status was read in line 84, and in both cases, R's status was *InTable* (otherwise, $R.data$ would not have beeen inserted into S), from claim 8, R's status at C_j was also *InTable*. From lemma 24, R was logically in the table at C_j.

Let R be a record for which $R.data[i] = val$ and that is logically in the table at C_j. In particular, R is i-reachable at C_j. Assume by contradiction that R is an i-infant at $C_{j'}$. Since R'_n $(= R_m)$ has an i-predecessor at $C_{j'}$ (R_{m-1}), by claim 21 (invariant 2), R'_n is not an i-infant at $C_{j'}$. Since the operation did not return in line 70, $R'_n.data[i] = val$. Therefore, by definition, $R <_i R'_n$. In addition, it must hold that $R'_{n-1}.data[i] < val$ and thus, $R'_{n-1} <_i R$. From claim 21 (invariant 11), R does not i-precede R'_{n-1} and does not i-follow R'_n at C_j. Since R'_n is R'_{n-1}'s i-successor at C_j, R also does not i-follow R'_{n-1} at C_j. Since both R'_{n-1} and R are i-reachable at C_j, we get a contradiction to claim 6. Therefore, R is not an i-infant at $C_{j'}$.

Since R's status at C_j is *InTable*, by claim 8, it is not i-marked at C_j and also at any earlier configuration. Since R is not an i-infant at $C_{j'}$, from claim 21 (invariant 13), R is i-reachable at $C_{j'}$, at C_j, and at every configuration between them.

If R is inserted, with its current status, into $tmpSet$, in line 75, then since this line is executed after $C_{j'}$, the status inserted is *InTable*. In addition, since the operation returns in line 91, when reading R's status in line 84, it must be equal to the inserted status (which is *InTable*). Therefore, $R.data$ is inserted into S in line 86.

Now, assume by contradiction that R and its current status are not inserted into $tmpSet$ in line 75. Let $R_{m+1}, \ldots, R_{m+k}$ be the records assigned into the $curr$ variable in lines 72-76. In addition, for every $0 \leq t \leq k-1$, let $s_{j_{m+t}}$ the step assigning R_{m+t-1} into the $curr$ variable in line 76 (notice that $s_{j_{m-1}}$ is the assignment of R_m into $curr$ in line 68).

If there exists $0 \leq t < k$ for which R_{m+t} is R, then since these lines are executed between $C_{j'}$ and C_j, when executing lines 73-75, having $curr \rightsquigarrow R$, R is inserted with its *InTable* status into $tmpSet$ in line 75 - a contradiction. Therefore, there does not exist $0 \leq t < k$ for which R_{m+t} is R.

Notice that for every $0 \leq t < k$, $R_{m+t}.data[i] = val$. In addition, since at some configuration during the execution of lines 72-76, R_{m+k} is the i-successor of R_{m_k-1}, from claim 21 (invariant 8), $R_{m_k-1} <_i R_{m+k}$. Therefore, since $R_{m+k}.data[i] \neq val$ (the condition checked in line 72 did not hold for it), $R_{m+k}.data[i] > val$.

Since R and R_m are two different records which are i-reachable at $C_{j'}$, from claim 6, either R i-follows R_m or R_m i-follows R at $C_{j'}$. In addition, since R_{m-1} is also i-reachable at $C_{j'}$, either R i-follows R_{m-1} or R_{m-1} i-follows R at $C_{j'}$. Since $R_{m-1} <_i R$ $(R_{m-1}.data[i] < val)$, from claim 21 (invariant 11), R_{m-1} does not i-follow R at $C_{j'}$ and thus, R i-follows R_{m-1} at $C_{j'}$. Since R_m is R_{m-1}'s i-successor at $C_{j'}$, R i-follows R_m at $C_{j'}$.

Since R is not an i-infant at $C_{j'}$, from claim 14, for every record $0 \leq t < k$, either $R_{m+t} <_i R$ or $R <_i R_{m+t}$. Since R i-follows R_m at $C_{j'}$, from claim 21 (invariant 11), $R_m <_i R$.

Now, let t be the maximal number among $0, 1, \ldots, k$ for which $R_{m+t} <_i R$. Notice that since $R_m <_i R$, t is well defined. In addition, since $R_{m+k} > val$, $R <_i R_{m+k}$ and thus, $t < k$. Since $j_{m+t} > j'$, R is not an i-infant at $C_{j_{m+t}}$. In addition, since R_{m+t} has an i-predecessor at $C_{j_{m+t-1}}$, from claim 21 (invariants 2), it is also not an i-infant at $C_{j_{m+t}}$. Therefore, from claim 21 (invariant 6), R belongs to R_{m+t}'s i-followers set at $C_{j_{m+t}}$. Since $R <_i R_{m+t+1}$, from claim 21 (invariant 11) R does not belong to R_{m+t+1}'s i-followers set at $C_{j_{m+t}}$. Since R_{m+t+1} is R_{m+t}'s i-successor at $C_{j_{m+t}}$, we have a contradiction. Therefore, for every record R, if $R.data[i] = val$ and R is logically in the table at C_j, then $R.data \in S$.

For each *retrieve*(val, i) execution that has terminated, we define its linearization point at C_j, defined in claims 32 and 33. From claims 32 and 33 it is indeed guaranteed that the set of tuples returned by the operation is the set of all tuples $t = \langle e_i, \ldots, e_{N_T} \rangle$ which are in the table at C_j and satisfy $t.e_i = val$.

After defining linearization points for every *add*, *remove*, *retrieve* and *find* execution that has terminated, we end our linearization proof with the following theorem:

Theorem 34: The implementation given in Figure 2, 3, 4 and 5 is a linearizable implementation of a table.

C. Progress

We are now going to prove that our table implementation is lock-free. Meaning, if all executing operations are run sufficiently long, at least one of the operations terminates.

To derive a contradiction, assume there is some execution $\alpha = C_0 \cdot s_1 \cdot C_1 \cdot s_2 \cdot C_2 \cdot \ldots$, and $k > 0$ for which no executing operation terminates after C_k, and no operation is invoked after C_k. We denote this suffix of α with $\alpha' = C_k \cdot s_{k+1} \cdot C_{k+1} \cdot s_{k+2} \cdot C_{k+2} \cdot \ldots$

Let $S = \{O_1, \ldots, O_m\}$ be the set of operations that take infinitely many steps in α'. Notice that we can assume that S is finite, since there is a finite number of threads in the system, and a thread cannot invoke a new operation before returning from a former one.

Claim 35: There is a finite number of records with a *Pending* status at C_j.

Proof 20: Let R be a record whose status is *Pending* during α'. We are going to show that there exists $1 \leq t \leq m$ for which O_t is an *add* execution for which R is created in line 18.

Since R's status is *Pending*, from claim 9, R is not *head* or *tail*. Therefore, R was created in line 18 of an *add* execution. Assume by contradiction that this operation had terminated before C_k. Before returning from the *helpAdding* execution (invoked in line 19), the cas execution in line 31 either changes R's status to be *InTable*, or fails because R's status is not *Pending*. From claim 8, R's status during α' is never *Pending* - a contradiction. Therefore, there exists $1 \leq t \leq m$ for which O_t is an *add* execution such that R is created in line 18.

The set of all records whose status is *Pending* at some point during α' is smaller than m and therefore, it is finite.

Claim 36: There is a finite number of status changes during α'.

Proof 21: From claim 35, there is a finite number of records with a *Pending* status during α' and therefore, there is a finite number of status changes such that the original status is *Pending* during α'.

From claim 8, the only kind of status change for which the original status is not *Pending*, is a change from *InTable* to *Removed*. This change can only happen when executing line 55 of a running *remove* operation. Therefore, the number of such status changes is smaller than m. Conclusion: there is a finite number of status changes during α'.

Claim 37: There is a finite number of i-markings during α'.

Proof 22: An i-marking can be executed only in line 24 or 60. If the i-marking is done in line 24, then the marked pointer belongs to the record created in line 18. If the i-marking is done in line 60, then the marked pointer belongs to the record found in line 52. Since a marked pointer cannot be marked again, the total number of i-markings during α' is $N_T \cdot m$. Therefore, there is a finite number of i-markings during α'.

Claim 38: There is a finite number of i-insertions during α'.

Proof 23: By the definition of an i-insertion, a record cannot be i-inserted twice (since after its first i-insertion, it is no longer an i-infant). Therefore, the total number of i-insertions during α' is at most the sum of the number of records created before C_k (at most k) and the number of records created after C_k (at most m), multiplied by N_T. Therefore, there is a finite number of i-insertions during α'.

Definition 39: Let s_j be an i-redirection. We say that s_j is an observable i-redirection if R_1 (as chosen in definition 17) is not an i-infant at C_{j-1}.

Claim 40: Let $0 \leq j$, let $1 \leq i \leq N_T$ and let R be a record which is not an i-infant at C_j. Then *tail* belongs to R's i-inclusive followers set at C_j.

Proof 24: By definition, *tail* belongs to *head* and *tail*s' i-inclusive followers set at C_0 for every $1 \leq i \leq N_T$. Now, let $1 \leq i \leq N_T$ and assume that for every record R_1 and for every $j' \leq j-1$, if R_1 is not an i-infant at $C_{j'}$ then *tail* belongs to R_1's i-inclusive followers set at $C_{j'}$.

Let R be a record which is not an i-infant at C_j. If R is also not an i-infant at C_{j-1}, then by the induction assumption, *tail* belongs to R's i-inclusive followers set at C_{j-1}. By claim 9, *tail* is not i-marked at C_j and therefore, by claim 21 (invariant 14), *tail* also belongs to R's i-inclusive followers set at C_j.

If R is an i-infant at C_{j-1}, then by claim 21 (invariant 10) s_j is an i-insertion for which $newRecord \rightsquigarrow R$. Let R_1 be the record for which $pred \rightsquigarrow R_1$ at C_j. By claim 21 (invariant 4), R_1 is not an i-infant at C_{j-1} and therefore, as explained above, *tail* belongs to its i-inclusive followers set at C_j. Since R is R_1's i-successor at C_j, by definition *tail* also belongs to R's i-inclusive followers set at C_j.

Claim 41: Let R be the record for which $newRecord \rightsquigarrow R$ during a *helpAddingField(*$newRecord, i$*)* execution. Let s_{j_1} be the execution of line 34 and let s_{j_2} be the last execution of line 37 (during the same $while$ loop iteration). If R is not an i-infant at C_{j_1} and is not i-marked at C_{j_2} then $tmp \rightsquigarrow R$ at C_{j_2}.

Proof 25: From claim 8, R is not i-marked at C_j for every $j_1 \leq j \leq j_2$. In addition, by definition, R is not an i-infant at C_j for every $j_1 \leq j \leq j_2$. From claim 13, R is i-reachable at C_j for every $j_1 \leq j \leq j_2$.

If R was returned as the $curr$ output variable from the *find* call in line 35, then the claim obviously holds. Otherwise, let $C_{j_{i_0}}$ be that *find*'s linearization point, guaranteed by claim 27. Let $R_{i_0}, R_{i_1}, \ldots, R_{i_n}$ be the records that the tmp local variable references during the execution of lines 36-40 and for every $1 \leq t \leq n$, let $s_{j_{i_t}}$ be the step reading R_{i_t} into tmp. R_{i_0} is the record returned as the $curr$ output parameter for the *find* call and therefore (by claim 27), it is i-reachable at $C_{j_{i_0}}$. Since $j_1 < j_{i_0} < j_2$, R is also i-reachable at $C_{j_{i_0}}$ and from claim 6, either R i-follows R_{i_0} or R_{i_0} i-follows R at $C_{j_{i_0}}$.

From claim 27, R_{i_0}'s i-predecessor at $C_{j_{i_0}}$ (returned as the first output parameter from the *find* call) is also i-reachable at $C_{j_{i_0}}$ and thus, either i-follows or i-precedes R at $C_{j_{i_0}}$ (also from claim 6). Since its i-th property must be smaller than R's (by claim 20), by lemma 23, it i-precedes R at $C_{j_{i_0}}$. Meaning, R cannot i-precede R_{i_0} at $C_{j_{i_0}}$. Therefore, R i-follows R_{i_0} at $C_{j_{i_0}}$.

Assume by contradiction that R_{i_n} is not R. Let t be the minimal value among $0, \ldots, n$ for which R does not i-follow R_{i_t} at $C_{j_{i_t}}$. As shown above, $t > 0$. In addition, since R_{i_n} is not R, it must hold that $R_{i_n}.data[i] \neq R.data[i]$. Since $R_{i_{n-1}}.data[i] = R.data[i]$ and $R_{i_{n-1}}$ is an i-predecessor of R_{i_n} at $C_{j_{i_n}}$, by lemma 23, $R_{i_n}.data[i] > R.data[i]$. Therefore, by claim 21 (invariant 11), R cannot i-follow R_{i_n} at $C_{j_{i_n}}$ and thus, $t \leq n$.

Since R i-follows $R_{i_{t-1}}$ at $C_{j_{i_{t-1}}}$, by definition, R belongs to $R_{i_{t-1}}$'s i-inclusive followers set at $C_{j_{i_{t-1}}}$. From claim 21 (invariant 14), and since R is not i-marked before C_{j_2}, R belongs to $R_{i_{t-1}}$'s i-inclusive followers set at $C_{j_{i_t}}$. Since R_{i_t} is $R_{i_{t-1}}$'s i-successor at $C_{j_{i_t}}$ and R does not i-follow R_{i_t} at $C_{j_{i_t}}$, we get a contradiction. Conclusion: R_{i_n} is R. Therefore, $tmp \rightsquigarrow R$ at C_{j_2}.

Claim 42: There is a finite number of observable i-redirections during α'.

Proof 26: By definition, i-redirections are done only during *helpAddingField($newRecord, i$)* in a $while$ loop iteration for which the condition checked in line 33 holds. Assume by contradiction that there is an infinite number of observable i-redirections during α'. From claim 35, there is a finite number of records with a *Pending* status during α'. Therefore, there must exist a record R for which there is an infinite number of observable i-redirections, either in line 48 or 49, for which $newRecord \rightsquigarrow R$.

Since the cas in line 49 is executed only if the cas in line 48 is successful, it must hold that the cas in line 48 is successful infinitely many times. From claim 8, R is never i-marked.

Let s_j be the first observable i-redirections during α' for which $newRecord \rightsquigarrow R$. If s_j is the execution of line 48 then by definition 39, R is not an i-infant at C_j. If s_j is the execution of line 49, then if R is an i-infant at C_{j-1}, by claim 21 (invariant 10), s_j is an i-insertion and therefore, R is not an i-infant at C_j. If R is not an i-infant at C_{j-1}, then by definition it is not an i-infant at C_j as well. Therefore, for every case, R is not an i-infant at C_j.

Since there is an infinite number of observable i-redirections for which $newRecord \rightsquigarrow R$, there is an observable i-redirection that is executed during a $while$ iteration that starts after s_j. Let s_{j_1} be the execution of line 34 and let s_{j_2} be the last execution of line 37 during that iteration. Since R is not an i-infant at C_j, it is also not an i-infant at C_{j_1}. In addition, since R is never i-marked, it is not i-marked at C_{j_2}. From claim 41, the method returns in line 39 and thus, no observable i-redirections is executed during that iteration - a contradiction. Therefore, there is a finite number of observable i-redirections during α'.

By claims 36, 37, 38 and 42, the number of status changes, i-markings, i-insertions and observable i-redirections (for any $1 \leq i \leq N_T$) during α' is finite. Therefore, without loss of generality, we can set C_k to be a later configuration for which there are no status changes, i-markings, i-insertions and observable i-redirections (for any $1 \leq i \leq N_T$) during α' at all.

Claim 43: There is a finite number of i-snippings during α'.

Proof 27: Let $1 \leq i \leq N_T$. Since the only existing records can be *head*, *tail* and records created in line 18, there are at most $2 + k + m$ records. Now, assume by contradiction that there are infinitely many i-snippings during α'. Then there exists a record R_2 for which $curr \rightsquigarrow R_2$ when executing the i-snipping, infinitely many times during α'. Let $s_{j'}$ be such an i-snipping ($j' \geq k$). We are going to prove by induction that for every $j \geq j'$, R_2 is not i-reachable at C_j:
Base Case ($j = j'$): Assume by contradiction that R_2 is i-reachable at C_j. Let R_1 and R_3 be the records from definition 16. Since R_2 is no longer R_1's i-successor at C_j, there exist another record, R which is i-reachable at C_j, and whose i-successor at C_j is R_2. Since s_j only changes R_1's i-successor, R's i-successor at C_{j-1} is also R_2.

Since R_1 is i-reachable at C_{j-1} (by definition), and s_j only changes R_1's i-successor, R_1 is i-reachable at C_j. From claim 6, either R_1 i-follows R or R i-follows R_1 at C_j. If R_1 i-follows R at C_j then by definition (and since $R \neq R_1$), R_1 i-follows R_2 at C_j. Since R_2 is the i-successor of R_1 at C_{j-1}, by claim 21 (invariant 8), $R_1 <_i R_2$. Since R_1 i-follows R_2 at C_j, we get a contradiction to claim 21 (invariant 11). Therefore, R must i-follow R_1 at C_j. Since R_3 is R_1's i-successor at C_j (and since $R \neq R_1$), R also i-follows R_3 at C_j and thus, R_2 i-follows R_3 at C_j. Meaning, by claim 21 (invariant 11) and the fact that R_2 is not an i-infant at C_j, $R_3 <_i R_2$. Since R_3 is R_2's i-successor at C_j, we get a contradiction to claim 21 (invariant 8).
Therefore, R_2 is not i-reachable at C_j.
Induction Step: Assume that R_2 is not i-reachable at C_{j-1}, and assume by contradiction that R_2 is i-reachable at C_j. By definition, there exist records $R_{i_0}, \ldots, R_{i_n}$ such that $R_{i_0} =$ *head*, $R_{i_n} = R_2$ and for every $1 \leq t \leq n$, R_{i_t} is the i-successor of $R_{i_{t-1}}$ at C_j. Let t be the maximal value for which R_{i_t} is i-reachable at C_{j-1}. Notice that $0 \leq t < n$, since *head* is always i-reachable (by definition) and R_2 is not i-reachable at C_{j-1}. In addition, $R_{i_{t+1}}$ is not R_{i_t}'s i-successor at C_{j-1}.

Since R_{i_t}'s i-successor at C_{j-1} is different from its i-successor at C_j, by claim 21 (invariant 10), s_j is either an i-snipping, an i-redirection or an i-insertion. s_j cannot be an i-snipping, since $R_{i_{t+1}}$ is not i-reachable at C_{j-1}. It cannot be an i-insertion because there are no i-insertions during α'. Therefore, it is an i-redirection. Since R_{i_t} is either *head* or has an i-predecessor at C_{j-1}, by claim 21 (invariant 2), R_{i_t} is not an i-infant at C_{j-1}. Therefore, by definition 39, s_j is an observable i-redirection. Since there are no observable i-redirections during α', we get a contradiction. Therefore, R_2 is not i-reachable at C_j.
Now, from choosing R_2, there exists $j'' > j'$ for which $curr \rightsquigarrow R_2$ when executing $s_{j''}$, which is an i-snipping. By the definition of i-snipping and i-reachability, R_2 is i-reachable at $C_{j''-1}$ - a contradiction. Therefore, there is a finite number of i-snippings during α'.
Now, using claim 43, without loss of generality, we can set C_k to be a later configuration for which there are no status changes, i-markings, i-insertions, observable i-redirections and i-snippings (for any $1 \leq i \leq N_T$) during α' at all.

Claim 44: There exists $k' \geq k$ for which the condition checked in line 7 never holds after $C_{k'}$.

Proof 28: Since the only existing records can be *head*, *tail* and records created in line 18, there are at most $2 + k + m$ records. Assume by contradiction that the condition checked in line 7 holds infinitely many times after C_k. Then there exists a record R for which $curr \rightsquigarrow R$ when the condition holds infinitely many times after C_k.

Let s_{j_1} and s_{j_2} be two executions of line 7 for which this condition holds when $curr \rightsquigarrow R$ ($k < j_1 < j_2$). When executing line 8 after s_{j_1}, since there are no i-snippings after C_k, the cas execution fails and the traversal starts from *head* again, in line 3.

Let R_1 be the record for which $pred \rightsquigarrow R_1$ at C_{j_2}. If R_1 is *head*, then by claim 9, R_1 is not i-marked at C_{j_2}. Otherwise, it has been assigned into the $pred$ variable in line 15, during the current traversal. Let s_{j_3} be the last time for which line 7 was executed when $curr \rightsquigarrow R_1$ during the current traversal. Since the traversal is still ongoing at C_{j_2}, and that $curr \rightsquigarrow R_1$ when getting to line 15, the condition did not hold. Therefore, R_1 was not i-marked at C_{j_3}. Since there are no i-markings after C_k, and $k < j_1 < j_3$, R_1 is not i-marked at C_{j_2} in this case also.

Now, let s_{j_4} be the step assigning R into the $curr$ local variable. If s_{j_4} is the execution of line 4 or 11 then R is R_1's i-successor at C_{j_4}, and since there are no i-snippings, i-insertions and observable i-redirections after C_k, R is R_1's i-successor at C_{j_2} as well (by claim 21, invariant 4, R_1 is not an i-infant at C_{j_4}). If s_{j_4} is the execution of line 16 then there exists $k < j_1 < j_5$ for which R was assigned into the $succ$ variable, either in line 6 or 12 at s_{j_5}. Obviously, R was R_1's i-successor at C_{j_5}, and for the above reasons, it is also still R_1's i-successor at C_{j_2}.

Since R is R_1's i-successor at C_{j_2}, the fact that R_1 is not i-marked at C_{j_2}, and the fact that this cannot change until executing line 8, the execution of line 8 is an i-snipping, in contradiction to the fact that there are no i-snippings after C_k. Therefore there exists $k' \geq k$ for which the condition checked in line 7 never holds after $C_{k'}$.
Now, using claim 44, without loss of generality, we can set $k = k' + 1$. Now, there are no status changes, i-markings, i-insertions, observable i-redirections, i-snippings (for any $1 \leq i \leq N_T$), and the condition checked in line 7 never holds during α'. We are going to derive a contradiction by showing that some operation from S terminates during α'.

Claim 45: Every execution of the *find* method eventually terminates.

Proof 29: Suppose val and i are the input parameters to a *find* invocation. If the method terminates before C_k, then we are done. Otherwise, assume by contradiction that the execution never terminates. After C_k, the condition checked in line 7 never holds.

Now, let s_{k_1} be the first execution of line 13 after C_k. Notice that since the condition checked in line 7 never holds after C_k and that the execution never terminates, line 13 is executed after C_k. Let R be the record for which $curr \rightsquigarrow R$ at C_{k_1}. From claim 21 (invariant 4), R is not an i-infant at C_{k_1} and thus, from claim 40, *tail* belongs to R's i-inclusive followers set at C_{k_1}. In particular, *tail* i-follows R at C_{k_1}. By definition, there exist records $R_0, R_1, \ldots, R_n$ such that $R_0 = R$, $R_n =$ *tail* and for each $0 < t \leq n$, R_t is the i-successor of R_{t-1} at C_{k_1}.

From claim 21 (invariant 2), $R_1, \ldots, R_n$ are also not i-infants at C_{k_1}. Therefore, for every $0 \leq t < n$, the i-successor of R_t cannot be changed via an i-redirection which is not an observable one. In addition, since there are no i-snippings, i-insertions and observable i-redirections during α', for every $0 \leq t < n$, the i-successor of R_t cannot be changed at all. Therefore, for every $k' > k_1$ and for every $0 < t \leq n$, R_t is still the i-successor of R_{t-1} at $C_{k'}$.

Since the condition in line 7 never holds after C_{k_1}, $R_1, \ldots, R_n$ are assigned into the $succ$ variable in line 6 and then into the $curr$ variable in line 16, one by one. When $curr \rightsquigarrow R_n$, from claim 19, the condition checked in line 13 holds and the method returns in line 14 - a contradiction. Therefore, every execution of the *find* method eventually terminates.

Claim 46: Every execution of lines 37-40 eventually terminates.

Proof 30: Suppose $newRecord$ and i are the input parameters to a *helpAddingField* invocation, and that $newRecord \rightsquigarrow R$ throughout the execution. If the execution of lines 37-40 terminates before C_k, then we are done. Otherwise, assume by contradiction that the execution never terminates.

Let s_{k_1} be the first execution of line 37 after C_k, and let R_0 be the record for which $tmp \rightsquigarrow R_0$ at C_{k_1}. If $curr \rightsquigarrow R_0$ at C_{k_1} as well then from claim 21 (invariant 4), R_0 is not an i-infant at C_{k_1}. Otherwise, R_0 must have an i-predecessor when assigned into the tmp variable in line 40 and thus, by claim 21 (invariant 2), it is also not an i-infant at C_{k_1}.

From claim 40, *tail* belongs to R_0's i-inclusive followers set at C_{k_1}. By definition, there exist records $R_1, \ldots, R_n$ for which $R_n =$ *tail* and for every $0 < t \leq n$, R_t is the i-successor of R_{t-1} at C_{k-1}. Notice that since for every $0 < t \leq n$, R_t has an i-predecessor at C_{k_1} and thus, by claim 21 (invariant 2), it is also not an i-infant at C_{k_1}.

The only $next[i]$ change that can occur after C_k is an i-redirection which is not an observable one. Since $R_0, R_1, \ldots, R_n$ are not i-infants at C_{k_1}, for every $k' \geq k_1$ and for every $0 < t \leq n$, R_t is still the i-successor of R_{t-1} at $C_{k'}$. Therefore, eventually, *tail* is assigned into the tmp local variable in line 40. Since we assume $R.data[i] < \infty$, the condition checked in line 37 no longer holds and the execution of lines 37-40 terminates - a contradiction. Therefore, every execution of lines 37-40 eventually terminates.

Claim 47: Every execution of lines 66-68 or lines 77-79 eventually terminates.

Proof 31: Without loss of generality, we are going to prove the claim for the execution of lines 66-68. The same proof holds for lines 77-79.

Suppose val and i are the input parameters to a *retrieve* invocation. If the execution of lines 66-68 terminates before C_k, then we are done. Otherwise, assume by contradiction that the execution never terminates.

Let s_{k_1} be the first execution of line 67 after C_k, and let R_0 be the record for which $curr \rightsquigarrow R_0$ at C_{k_1}. If $R_0 =$ *head* then by definition, R_0 is not an i-infant at C_{k_1}. Otherwise, R_0 must have an i-predecessor when assigned into the $curr$

variable in line 68 and thus, by claim 21 (invariant 2), it is also not an i-infant at C_{k_1}.

From claim 40, *tail* belongs to R_0's i-inclusive followers set at C_{k_1}. By definition, there exist records $R_1, \ldots, R_n$ for which $R_n =$ *tail* and for every $0 < t \leq n$, R_t is the i-successor of R_{t-1} at C_{k-1}. Notice that since for every $0 < t \leq n$, R_t has an i-predecessor at C_{k_1} and thus, by claim 21 (invariant 2), it is also not an i-infant at C_{k_1}.

The only $next[i]$ change that can occur after C_k is an i-redirection which is not an observable one. Since $R_0, R_1, \ldots, R_n$ are not i-infants at C_{k_1}, for every $k' \geq k_1$ and for every $0 < t \leq n$, R_t is still the i-successor of R_{t-1} at $C_{k'}$. Therefore, eventually, *tail* is assigned into the $curr$ local variable in line 68. Since we assume $val < \infty$, the condition checked in line 67 no longer holds and the execution of lines 66-68 terminates - a contradiction. Therefore, every execution of lines 66-68 eventually terminates.

Claim 48: Every execution of lines 72-76 eventually terminates.

Proof 32: Suppose val and i are the input parameters to a *retrieve* invocation. If the execution of lines 72-76 terminates before C_k, then we are done. Otherwise, assume by contradiction that the execution never terminates.

Let s_{k_1} be the first execution of line 72 after C_k, and let R_0 be the record for which $curr \rightsquigarrow R_0$ at C_{k_1}. If $R_0 =$ *head* then by definition, R_0 is not an i-infant at C_{k_1}. Otherwise, R_0 must have an i-predecessor when assigned into the $curr$ variable in line 76 and thus, by claim 21 (invariant 2), it is also not an i-infant at C_{k_1}.

From claim 40, *tail* belongs to R_0's i-inclusive followers set at C_{k_1}. By definition, there exist records $R_1, \ldots, R_n$ for which $R_n =$ *tail* and for every $0 < t \leq n$, R_t is the i-successor of R_{t-1} at C_{k-1}. Notice that since for every $0 < t \leq n$, R_t has an i-predecessor at C_{k_1} and thus, by claim 21 (invariant 2), it is also not an i-infant at C_{k_1}.

The only $next[i]$ change that can occur after C_k is an i-redirection which is not an observable one. Since $R_0, R_1, \ldots, R_n$ are not i-infants at C_{k_1}, for every $k' \geq k_1$ and for every $0 < t \leq n$, R_t is still the i-successor of R_{t-1} at $C_{k'}$. Therefore, eventually, *tail* is assigned into the $curr$ local variable in line 76. Since we assume $val < \infty$, the condition checked in line 72 no longer holds and the execution of lines 72-76 terminates - a contradiction. Therefore, every execution of lines 72-76 eventually terminates.

Claim 49: Every execution of the *helpAddingField* method eventually terminates.

Proof 33: Suppose $newRecord$ and i are the input parameters. We are going to show by a backwards induction on i that the method eventually terminates.

Base Case ($i = N_T$): Assume by contradiction that the method does not terminate. By claims 45 and 46 (respectively), both the *find* call in line 35 and the traversal in lines 36-40 always terminate. Since the method does not return in line 39, from claim 41, the record R for which $newRecord \rightsquigarrow R$ is either an i-infant when executing line 36 or i-marked when executing line 37 for the last time.

Since the second input parameter to the *helpAdding* call in line 44 is $N_T + 1$, it always terminates. Therefore, the method does not terminate because at least one of the cas executions in lines 48-49 fails at every loop iteration, and R's status always remains *Pending*.

If R is i-marked at some point then by claim 8, its status is no longer *Pending* - a contradiction. Therefore, since the method never returns in line 39, R stays an i-infant at $C_{k'}$ for every $k' > k$.

From lemmas 22 and 23, there exist records $R_0, R_1, \ldots, R_n$ such that $R_0 =$ *head*, $R_n =$ *tail* and for every $0 < t \leq n$, R_t is the i-successor of R_{t-1} at C_k. By definition, R_0 and R_n are not i-infants at C_k. In addition, for every $0 < t < n$, R_t has an i-predecessor at C_k and thus, by claim 21 (invariant 2), it is also not an i-infant at C_k. Since there are no status changes, i-snippings, i-insertions and observable i-redirections at α', for every $0 < t \leq n$ and $k' \geq k$, R_t is still R_{t-1}'s i-successor at $C_{k'}$.

Now, every *find($newRecord.data[i], i$)* call invoked in line 35 after C_k, should return the same $\langle pred, curr \rangle$ window (by claim 27). This holds even for different invocations of the *helpAddingField($newRecord, i$)* method, as long as $newRecord \rightsquigarrow R$. Therefore, at some point, R's i-successor also becomes stable (since R is an i-infant at $C_{k'}$ for every $k' \geq k$, its i-successor is updated only by i-redirections which are not observable, in line 48). Therefore, the cas execution that keeps failing is the one in line 49.

Since the *find($newRecord.data[i], i$)* call is invoked after C_k (meaning, L_i is of the form $R_0 \rightarrow R_1 \rightarrow \ldots \rightarrow R_n$), by claim 27, it is guaranteed that there exist $k_1 \geq k$ and $0 \leq t < n$ for which $pred \rightsquigarrow R_t$, $curr \rightsquigarrow R_{t+1}$, R_t is not i-marked and R_{t+1} is R_t's i-successor at C_{k+1}. In addition, since R_t is not an i-infant at C_{k_1} (by claim 21, invariant 4), its i-successor remains R_{t+1} and it remains not i-marked for every $k' \geq k_1$. Therefore, the cas executed in line 49 should be successful - a contradiction. Conclusion: the method always terminates when $i = N_T$.

Induction Step: By the induction assumption, the call to *helpAdding($curr, i + 1$)* in line 44 always terminates. Therefore, in a similar way to the base case, the method always terminates when $i < N_T$ as well.

Claim 50: Every execution of the *helpAdding* method eventually terminates.

Proof 34: This derives directly by the guaranteed termination of the *helpAddingField* method (claim 49).

From the above claims, the *find* and *helpAdding* methods always terminate. In order to show that both the *add* and the *remove* operations always terminate, we still need to show that the loops in lines 21-25 and 57-61 always terminate.

Claim 51: Every execution of lines 21-25 eventually terminates.

Proof 35: Let R be the record that was created in line 18 and let $1 \leq i \leq N_T$. If lines 21-25 are executed then R's status is *Failed*. From claim 8, it remains *Failed* forever.

Since the only way R's i-successor can be changed after C_k is by an i-redirection which is not observable, it can only be changed during a *helpAddingField($newRecord, i$)* call for which $newRecord \rightsquigarrow R$ (in line 48). By claim 49, all *helpAddingField($newRecord, i$)* invocations eventually terminate. Therefore, at some point, R's i-successor can be changed only during *helpAddingField* calls that were invoked after R's status had become *Failed*. Since those calls must return immediately (the condition checked in line 33 does not hold for R), R's i-successor remains stable, starting from this point.

Therefore, after this point, the cas execution in line 24 can only fail because R is already i-marked. In this case, the condition checked in line 23 does not hold and thus, the loop in lines 21-25 eventually terminates.

Claim 52: Every execution of lines 57-61 eventually terminates.

Proof 36: Let R be the record that was found in line 52 and let $1 \leq i \leq N_T$. If lines 57-61 are executed then R's status is *Removed*. From claim 8, it remains *Removed* forever.

Since the only way R's i-successor can be changed after C_k is by an i-redirection which is not observable, it can only be changed during a *helpAddingField($newRecord, i$)* call for which $newRecord \rightsquigarrow R$ (in line 48). By claim 49, all *helpAddingField($newRecord, i$)* invocations eventually terminate. Therefore, at some point, R's i-successor can be changed only during *helpAddingField* calls that were invoked after R's status had become *Removed*. Since those calls must return immediately (the condition checked in line 33 does not hold for R), R's i-successor remains stable, starting from this point.

Therefore, after this point, the cas execution in line 60 can only fail because R is already i-marked. In this case, the condition checked in line 59 does not hold and thus, the loop in lines 57-61 eventually terminates.

By claims 51 and 52, the *add* and *remove* operations always terminate. We still need to show that the *retrieve* operation always terminates.

Claim 53: Every execution of the *retrieve* operation eventually terminates.

Proof 37: Assume by contradiction that there is a *retrieve(val, i)* execution that never terminates. In particular, the condition checked in line 69 never holds. By claims 47 and 48, the loops in lines 66-68, 72-76 and 77-79 always terminate. Therefore, every loop iteration (the loop in line 64) terminates, and there exists an iteration that starts after C_k.

By lemmas 22 and 23, there exist records $R_0, R_1, \ldots, R_n$ such that $R_0 = head$, $R_n = tail$, and for every $0 < t \leq n$, $R_{t-1}.data[i] \leq R_t.data[i]$ and R_t is the i-successor of R_{t-1} at C_k.

By definition, R_0 and R_n are not i-infants at C_k. In addition, for every $0 < t < n$, R_t has an i-predecessor at C_k and thus (by claim 21, invariant 2), it is also not an i-infant at C_k. Therefore, for every $0 \leq t < n$, R_t's i-successor cannot be changed using a non-observable i-redirection after C_k. Since there are no status changes, i-snippings, i-insertions and observable i-redirections after C_k, the described sequence of records does not change during the current loop of the *retrieve* execution.

From claim 47 the loops in lines 67-68 and 77-79 terminate. Since the sequence of records described above is stable after C_k, both traversals get to the same record and therefore, the condition in line 80 holds (notice that by claim 48, the loop in lines 72-76 terminates as well).

The $tmpSet$ set contains only records that were inserted into it in line 75 and tus, it is a finite set. Since there are no status changes after C_k, the condition, checked for every saved record, in line 84, holds for every such record. Therefore, $valid$'s value when checking it in line 90 is true, and the operation returns in line 91 - a contradiction. Conclusion: Every execution of the *retrieve* operation eventually terminates.

Since all *add*, *remove* and *retrieve* eventually terminate, there does not exist an execution $\alpha = C_0 \cdot s_1 \cdot C_1 \cdot s_2 \cdot C_2 \cdot \ldots$, and $k > 0$ for which no executing operation terminates after C_k. Therefore, our table implementation is lock-free, and we end our progress proof with the following theorem:

Theorem 54: The implementation given in Figure 2, 3, 4 and 5 is a lock-free implementation of a table.

Appendix B
Lock-Based Version

We now describe a lock based variant of the table data structure. As in the lock-free approach, this algorithm maintains a collection of linked-lists, and each record contains the $data$ and $next$ arrays as well as the $status$ flag. In contrast to the lock-free algorithm, here $next$ contains standard pointers (without any marking mechanism); and each record contains an additional array $locks$ that holds N_T locks.

In this algorithm, a thread may update $R.next[i]$ only if it owns the lock $R.lock[i]$. Moreover, it may update $R.status$ only if it owns all locks in $R.locks$. In contrast to the previous algorithm , here $R.status$ will never be *Failed* since a record's insertion cannot fail after it had started.

Here, the successful insertion of a new record into the table includes adding it to all lists and then changing its status from *Pending* to *InTable*, and the removal of a record includes changing its status from *InTable* to *Removed* and then removing it from all of the lists. Notice that, here, each operation is done entirely by its calling thread. Therefore, the *remove* operation is the one in charge of physically removing a record from the table.

```
1: find($val, i$)
2:     $pred \leftarrow head$
3:     $curr \leftarrow pred.next[i]$
4:     while $true$ do                    ▷ While not having the desired records
5:         if $curr.data[i] \geq val$
6:             return   $pred, curr$     ▷ Return the current pair of records
7:         $pred \leftarrow curr$
8:         $curr \leftarrow pred.next[i]$
```

Fig. 7. The lock-based *find* method

a) The find method: Like in the lock-free version, here the *find* auxiliary method (shown in Figure 7) locates a record in L_i, by finding its potential i-predecessor and i-successor in the list. However, since physical removals are done only during *remove* executions, the lock-based *find* method does not physically remove records from the lists. This method receives a value val and a field number i, and returns the $pred$ and $curr$ references. As in the lock-free version, it holds that (1) $pred.data[i] < val$, (2) $curr.data[i] \geq val$ and (3) $curr$ is $pred$'s i successor. Notice that, here, there is also no guarantee that the returned records will continue to satisfy the above three properties.

As in the lock-free version, the traversal starts by setting $pred$ to reference *head* (which is the first record in L_i) and setting $curr$ to reference *head*'s successor in L_i (lines 2-3). After this initialization, the $pred$ and $curr$ references are advanced throughout L_i, until the relevant two records are found and returned (line 6). Assuming that val is always bigger than $-\infty$ and smaller than ∞, the two relevant records are always found while traversing the list. Notice that there is a single traversal, which always ends by finding the relevant couple of records. However, when the calling operation gets the *find* output, $pred$ and $curr$ might not satisfy the above conditions anymore (either $pred$ or $curr$ have been meanwhile removed from the table, or a new record has been added between them). In such cases, the *find* method is called again in order to find a new couple of records.

```
 9: add(tup)
10:     $newRecord \leftarrow makerecord(tup)$          ▷ Creating the new record
11:     retry:                                          ▷ Starting a new adding attempt
12:     for $i = 1$ to $N_T$ do
13:         $pred_i, curr_i \leftarrow find(newRecord.data[i], i)$
14:         $pred_i.locks[i].lock()$
15:         $newRecord.locks[i].lock()$
16:         if $(pred_i.status \neq InTable) || (pred_i.next[i] \neq curr_i)$
17:             release all locks and goto retry
18:         else if $Uniqu(i)$                          ▷ If the $i$-th field is unique
19:             if $curr_i.data[i] == R.data[i]$
20:                 if $curr_i.status == InTable$
21:                     release all locks and return false
22:                 else release all locks and goto retry
23:     for $i = 1$ to $N_T$ do
24:         $newRecord.next[i] \leftarrow curr_i$
25:         $pred_i.next[i] \leftarrow newRecord$        ▷ Physical insertion
26:     $newRecord.status \leftarrow InTable$           ▷ Logical insertion
27:     release all locks and return true
```

Fig. 8. The lock-based *add* operation

b) The add operation: The lock-based *add* operation (shown in Figure 8) receives a tuple tup, creates a new corresponding record R with a *Pending* status and tries to add R to the table. It keeps trying until it either succeeds or fails (failing means there exists a unique field number i and a record R' such that R' is logically in the table and $R'.data[i] = R.data[i]$). Each trial consists of two phases. The first phase (lines 12-22) includes finding the designated predecessors and successors of R in each list, obtaining the relevant locks and making sure that inserting R into the table is not going to violate any uniqueness property. The second phase (lines 23-26) is executed only if the first phase is fully completed, and it cannot fail. Meaning, if the first phase was fully executed, the whole operation is already guaranteed to be successful, and the second phase just completes the physical and logical insertion of R into the table. We are now going to give a detailed description of the two phases.

In the first phase, we traverse each list L_i, using the *find* method, in order to find the designated i-predecessor and i-successor of R in L_i. We keep the i-predecessor in the $pred_i$ local variable and the i-successor in $curr_i$. After finding each pair of records, we obtain $pred_i$'s i-th lock (line 14), obtain $R.locks[i]$, and check whether $pred_i$ still satisfies the *find*'s output conditions. That includes checking whether $pred_i$ is indeed in the table (otherwise, we cannot assume that it is still physically in L_i after obtaining the lock) and whether $pred_i$'s i-successor is indeed $curr_i$ (line 16). If one of those conditions does not hold, we cannot use this couple of records. In this case, we release all obtained locks and start all over again. Notice that if the two conditions do hold after obtaining $pred_i$'s i-th lock, they will hold until we release this lock or change something ourselves. This is guaranteed because $next$ references are changed only after obtaining the relevant lock and a record's status is changed only after obtaining all of the record's locks.

After making sure that $pred_i$ and $curr_i$ satisfy the conditions, if i is a unique field number, we check whether $curr_i.data[i] = R.data[i]$. If they are equal and $curr_i$ is logically in the table, then R cannot be inserted into the table (inserting R would violate the uniqueness property). In this case, all locks obtained are released and we return $false$ (line 21). If $curr_i$ is not in the table (its status is not *InTable*), it means that its status is *Pending*. Therefore, we do not know yet if it is going to be inserted into the table or not (its insertion may fail). In this case we have to release all locks and start a new trial (line 22). Notice that its status cannot be *Removed*. If $curr_i$'s status is *Removed* then it has already been physically removed (otherwise, we would not have succeeded in obtaining its lock). Since $pred_i$ is in the table and still is $curr_i$'s i-predecessor, it is impossible that $curr_i$ has been physically removed. Therefore, if $curr_i$'s status is not *InTable*, it has to be *Pending*.

After obtaining all of the predecessors' locks and all of R's locks, we move to the second phase. In the second phase, we update all of the $next[i]$ references and update them to reference the $curr_i$s records found (line 24). After every such update, we perform the physical insertion of R into each list. We do it by updating the relevant $pred_i.next[i]$ to reference R (line 25). After the physical insertion is done, we complete the operation by setting R's status to *InTable* (line 26), releasing all locks and returning $true$.

c) The remove operation: The *remove* operation (shown in Figure 9) receives a value val and a field number i and tries to remove a record R such that $R.data[i] = val$. As in the lock-free version, here we also assume that i is a unique field number. We start by searching for a record R for which $R.data[i] = val$, using the *find* method, in line 29 (notice that

```
28: $remove(val, i)$
29:     $pred, victim \leftarrow find(val, i)$
30:     if $(victim.data[i] \neq val)||(victim.status \neq InTable)$
31:         return false
32:     retry:                                    ▷ Starting a new removing attempt
33:     for $m = 1$ to $N_T$ do
34:         $pred_m, curr_m \leftarrow find(victim.data[m], m)$
35:         while $(curr_m \neq victim) and (curr_m.data[m] == victim.data[m])$ do
36:             $curr_m \leftarrow curr_m.next[m]$
37:         if $curr_m \neq victim$
38:             release all locks and return false
39:         $pred_m.locks[m].lock()$
40:         if $(pred_m.status \neq InTable)||(pred_j.next[m] \neq victim)$
41:             release all locks and goto retry
42:         $victim.locks[m].lock()$
43:     if $victim.status \neq InTable$
44:         release all locks and return false
45:     $victim.status \leftarrow Removed$                         ▷ Logical removal
46:     for $m = 1$ to $N_T$ do
47:         $pred_m.next[j] \leftarrow victim.next[m]$               ▷ Physical removal
48:     release all locks and return true
```

Fig. 9. The lock-based *remove* operation

the $pred$ record returned from this call is not used during the operation). We use this search in order to find out whether the wanted record is currently in the table. After the call returns, the local variable $victim$ is a reference to the first record R in L_i such that $R.data[i] \geq val$. Since i is a unique field number, it is guaranteed that for every record R' which i-follows R, either $R'.data[i] > val$ or $R'.status \neq InTable$. Therefore, if $victim.data[i] > val$ or $victim.status \neq InTable$, it is guaranteed that there does not exist a relevant record in the table when $victim$ is found, and it is safe to return $false$ in line 31, which means that the operation is unsuccessful.

If $victim.data[i] = val$, and its status is *InTable*, then $victim$ becomes our removal objective, and we keep trying to remove it until we succeed or fail (after another *remove* operation has meanwhile removed it). Notice that if $victim$ is removed by another executing thread while we are trying to remove it, it is safe to stop the whole process and return $false$ (declaring the operation was unsuccessful). The reason is that right after $victim$'s status becomes *Removed*, it is guaranteed that there does not exist a record R such that $R.data[i] = val$ in the table. Since i is a unique field number, such a record can be added to the table only after $victim$ is removed. Therefore, there exists a point in time during the execution for which the operation can safely fail.

Each removing attempt starts by traversing the different lists L_m, trying to obtain $victim$'s m-predecessor's $locks[m]$ and $victim$'s $lock[m]$ (lines 33-42). For the m-th loop iteration, $victim$ is supposed to be saved in the $curr_m$ local variable. If $curr_m$ is not $victim$, it is guaranteed that $victim$ has already been removed from the table, and therefore we release all obtained locks and return $false$ in line 38. If $curr_m$ is $victim$, we obtain $pred_m.locks[m]$, check that $pred_m$ is indeed $victim$'s m-predecessor, and that $pred_m$'s status is *InTable*. If one of these conditions does not hold, then we need to search for $victim$'s m-predecessor again, and we start a new removal trial (line 41). As explained before, $pred_m$'s m-successor and $pred_m$'s status cannot change after we obtain $pred_m.lock[m]$ and thus, we only need to check it once in order to make sure that these conditions hold (until we release the lock or make any change ourselves). After making sure that both conditions hold, we can obtain $victim.locks[m]$ and continue to the next list.

After finishing obtaining all of the relevant locks, we check again whether $victim$'s status is still *InTable* (line 43). If it is not *InTable* then, as already mentioned, it is safe to release all locks and return $false$, as done in line 44. If $victim$'s status is still *InTable* then we are the only ones able to change it (since we hold all of $victim$'s locks). The next step is setting $victim$'s status to *Removed* (line 43), which represents its logical removal from the table. After that, we physically remove victim from all of the lists by setting $pred_m$'s m-successor to be $victim$'s m-successor (lines 46-47). After the logical and physical removals are done, we release all locks and return $true$.

```
49: $retrieve(val, i)$
50:     retry:                                          ▷ Starting a new search
51:     $tuples \leftarrow \emptyset$                              ▷ For returning the tuples
52:     $pred, curr \leftarrow find(val, i)$
53:     if $curr.data[i] > val$                 ▷ There are no relevant records
54:         return $tuples$
55:     $pred.locks[i].lock()$
56:     if $(pred.next[i] \neq curr)||(pred.status \neq InTable)$
57:         release all locks and goto retry                        ▷ Start again
58:     while $curr.data[i] == val$ do
59:         $curr.locks[i].lock()$
60:         if $curr.status \neq InTable$
61:             release all locks and goto retry                    ▷ Start again
62:         $tuples \leftarrow tuples \cup \{curr.data\}$
63:         $curr \leftarrow curr.next[i]$
64:     release all locks and return $tuples$
```

Fig. 10. The lock-based *retrieve* operation

d) The retrieve operation: Like in the lock-free version, the *retrieve* operation (shown in Figure 10) receives a value val and a field number i, and returns the set of all tuples $\langle e_1, \ldots e_n \rangle$ satisfying $e_i = val$ (meaning all arrays $R.data$ for which R is logically in the table and $R.data[i] = val$). In a similar way to the lock-free *retrieve*, it traverses L_i in order to find the first record R satisfying $R.data[i] = val$. Then, It obtains the i-th lock of R's i-predecessor, and for each record R' satisfying $R'.data[i] = val$, it obtains $R'.lock[i]$. After obtaining all of the relevant locks (making sure that no records with the given property are added to or removed from the table), it returns the set of all arrays $R.data$ for which $R.data[i] = val$. Notice that, since the operation includes obtaining the i-th lock of all of the records R satisfying $R.data[i] = val$, it is guaranteed that what is seen by the operation is always valid. Therefore, once all relevant locks are obtained, there is no need in a second check of the records' statuses (as in the lock-free version). The operation starts by calling the *find* method in order to find the first record R which is physically in L_i and which satisfies $R.data[i] = val$. If such a record does not exist ($curr.data[i] > val$, when $curr$ is the second record returned by the *find* call), then an empty set is returned as output (line 54). Otherwise, there exists at least one record R satisfying $R.data[i] = val$ (currently referenced by the $curr$ local variable). In line 55 we obtain $pred.locks[i]$ in order to make sure that no new records R', for which $R'.data[i] = val$, are inserted into L_i while we update our local $tuples$ set. This is guaranteed since a new record R' which satisfies $R'.data[i] = val$ can only be inserted into L_i between $pred$ and $curr$ (and only after obtaining $pred.locks[i]$). After obtaining the $pred.locks[i]$ lock, we make sure that $pred$ is still in the table (having an *InTable* status) and that it still holds that $pred.next[i] = curr$. Otherwise, we cannot assume that we have access to the first record R satisfying $R.data[i] = val$ (which is assumed to be $curr$). Therefore, if one of these two conditions does not hold, we release the $pred.locks[i]$ lock and start a new trial (from line 50).

After $pred.locks[i]$ is obtained and it is guaranteed that it is indeed the i-predecessor of the first record R satisfying $R.data[i] = val$, we go over all records whose $data[i]$ value is val (lines 58-63) and for each one, we obtain its i-th lock and add its $data$ array to our local $tuples$ set. Notice that before we obtain a record's lock, we always obtain its i-predecessor's lock. Therefore, it is guaranteed that each record we encounter is indeed in the table (in order to remove a record from L_i, one has to obtain its i-predecessor's i-th lock). After we are done updating our $tuples$ set, we release all locks and return it in line 64.

A. Correctness Proof

This section contains the proofs of our lock-based table's invariants. The lock-based version's linearizability proof appears in Secotion B-B and a proof that the implementation is deadlock-free appears in Section B-C.

As in the lock-free version's correctness proof, we are going to use the definitions of i-predecessors and i-successors (see definition 2), i-following and i-preceding (see Definition 3), and i-reachability (see Definition 4). The definitions of configurations, steps and executions remain the same as in section A. Notice that since the $next[i]$ pointers in the lock-based version are not marked, we no longer need the i-marked notation. Instead, we use some new definitions for describing the $locks$ arrays:

Definition 55: Let R be a record. We say that R is partially locked at C_j if there exist $j_1 \leq j$ and $1 \leq i \leq N_T$ for which s_{j_1} obtains R's i-th lock in line 14 or 39. In addition, there does not exist $j_1 < j_2 \leq j$ such that s_{j_2} releases this lock.

Definition 56: Let R be a record. We say that R is fully locked at C_j if all of R's locks were obtained by the same thread, either by executing line 15 or 42 for every $1 \leq i \leq N_T$, and have not been released before C_j.

Notice that by definition, a record cannot be both partially locked and fully locked at the same configuration.

Definition 57: Let R be a record. We say that R is i-locked if $R.locks[i] = true$.

Since the original i-infancy definition (see Definition 12) relies on the specific lock-free *add* implementation, we are going to use a slightly different definition here:

Definition 58: Let $\alpha = C_0 \cdot s_1 \cdot C_1 \cdot s_2 \cdot C_2 \cdot \ldots \cdot s_k \cdot C_k$ be a legal execution, let $0 \leq j \leq k$ and let $R \notin \{head, tail\}$ be a record. If there does not exist $j' \leq j$ for which $s_{j'}$ is the execution of line 25, for which $newRecord \rightsquigarrow R$, we say that R is an i-infant at C_j.

After defining the i-infancy term, we can use the $<_i$ partial order, defined in Section A (Definition 13). Now, before getting to the linearizability and deadlock-freedom proofs, we need to show that the following always hold:

1) For every $1 \leq i \leq N_T$, *head* and *tail* are always i-reachable.
2) A record R is logically in the table if and only if its status is *InTable* and it is i-reachable for every $1 \leq i \leq N_T$.
3) For every two records R_1, R_2 and $1 \leq i \leq N_T$, if R_2 i-follows R_1 then $R_1.data[i] \leq R_2.data[i]$.
4) For every $1 \leq i \leq N_T$ such that i is a unique field number, and two records R_1, R_2, if both R_1 and R_2 are logically in the table then $R_1.data[i] \neq R_2.data[i]$.

We start with some basic observations.

Claim 59: Let $1 \leq i \leq N_T$ and let R_1, R_2 be two records. If both R_1 and R_2 are i-reachable then either R_1 i-follows R_2 or R_2 i-follows R_1.

Proof 38: See the proof of claim 6 in Section A.

Claim 60: Let $\alpha = C_0 \cdot s_1 \cdot C_1 \cdot s_2 \cdot C_2 \cdot \ldots \cdot s_k \cdot C_k$ be a legal execution, let R be a record and let $0 \leq j_1 \leq j_2 \leq k$. If for every $j_1 \leq j \leq j_2$, R is partially locked, then R's status at C_{j_1} is equal to R's status at C_{j_2}.

Proof 39: Assume by contradiction that R's status at C_{j_1} is different from R's status at C_{j_2}. A record's status can only change when executing line 26 or 45. Therefore, there exists $j_1 \leq j \leq j_2$ for which either s_j is the execution of line 26 while $newRecord \rightsquigarrow R$ or the execution of line 45 while $victim \rightsquigarrow R$. In both cases, R must be fully locked, in contradiction to the fact that R is partially locked at C_j. Therefore, if for every $j_1 \leq j \leq j_2$, R is partially locked, then R's status at C_{j_1} is equal to R's status at C_{j_2}.

Claim 61: Let $\alpha = C_0 \cdot s_1 \cdot C_1 \cdot s_2 \cdot C_2 \cdot \ldots \cdot s_k \cdot C_k$ be a legal execution, let R be a record and let $0 < j \leq k$. If R's status at C_j is different from R's status at C_{j-1} then the thread that executes this change also holds all of R's locks at C_j.

Proof 40: R's status can only be changed when executing line 26 (and $newRecord \rightsquigarrow R$) or 45 (and $victim \rightsquigarrow R$). In both cases, all of R's locks are obtained (and not released), before executing the status change, by the executing thread.

Claim 62: Let $\alpha = C_0 \cdot s_1 \cdot C_1 \cdot s_2 \cdot C_2 \cdot \ldots \cdot s_k \cdot C_k$ be a legal execution and let $0 \leq j \leq k$, then both *head* and *tail*s' status at C_j is *InTable*.

Proof 41: At initialization, *head* and *tail* are created with an *InTable* status. We assume that for every *remove(*val, i*)* invocation, $val \notin \{-\infty, \infty\}$. Since a record's *InTable* status can only be changed during a *remove(*val, i*)* execution, and only if its i-th data element is val, *head* and *tail*s' statuses never change during the execution.

Notice that a record's status can also change during an *add(tup)* execution. Since the record whose status is changed, is created during this execution, and that its status becomes *InTable* anyway, this is not relevant for the case of *head* and *tail*.

Claim 63: Let val be the first input parameter of a *find* call. Then $val \notin \{-\infty, \infty\}$.

Proof 42: See the proof of claim 19 in Section A. Notice that in the lock-based table version, the *find* method is called during *retrieve* executions as well. Since we assume neither $-\infty$ nor ∞ are sent as input parameters to the *retrieve* operation, the proof of claim 19 still holds for the lock-based version.

Claim 64: Let R be a record and let $1 \leq i \leq N_T$. R's i-successor can be changed only by the thread holding $R.lock[i]$.

Proof 43: R's i-successor can only be changed in line 24 while $newRecord \rightsquigarrow R$, 25 while $pred_i \rightsquigarrow R$ or 47 while $pred_i \rightsquigarrow R$. In all three cases, R's i-th lock is held by the executing thread.

Claim 65: Let $\alpha = C_0 \cdot s_1 \cdot C_1 \cdot s_2 \cdot C_2 \cdot \ldots \cdot s_k \cdot C_k$ be a legal execution, let R be a record, let $1 \leq i \leq N_T$ and let $0 \leq j \leq k$. If s_j is the execution of line 25 for which $newRecord \rightsquigarrow R$, then R is an i-infant at C_{j-1}.

Proof 44: Assume by contradiction that R is not an i-infant at C_{j-1}. By definition, there exists $j' \leq j-1$ for which $s_{j'}$ is the execution of line 25, having $newRecord \rightsquigarrow R$. Since the $newRecord$ variable always points to the record created in line 10, $s_{j'}$ is a step done by the same executing thread (the one which executes s_j). Since the *add* operation returns after lines 23-25 are executed for the first time, we get a contradiction. Therefore, if s_j is the execution of line 25 for which $newRecord \rightsquigarrow R$, then R is an i-infant at C_{j-1}.

Claim 66: Let $\alpha = C_0 \cdot s_1 \cdot C_1 \cdot s_2 \cdot C_2 \cdot \ldots \cdot s_k \cdot C_k$ be a legal execution, let $0 \leq j < j' \leq k$, and let R be a record, then:

1) If $R.status = Removed$ at C_j then R's status at $C_{j'}$ is also *Removed*.
2) If $R.status = InTable$ at C_j then $R.status \in \{InTable, Removed\}$ at $C_{j'}$.
3) If R's status at C_j is *Pending* and R's status at $C_{j'}$ is *Removed* then there exists $j < j'' < j'$ for which R's status at $C_{j''}$ is *InTable*.

Proof 45: *head* and *tail* are created with an *InTable* status. Therefore, the claim is vacuously true at C_0. Assume that the invariants hold throughout $\alpha' = C_0 \cdot s_1 \cdot C_1 \cdot \ldots \cdot C_{j'-1}$ of the execution. We show that each part of the claim holds throughout $\alpha = \alpha' \cdot s_{j'} \cdot C_{j'}$.

1) Assume R's status is *Removed* at C_j. By the induction assumption, R's status at $C_{j'-1}$ is also *Removed*. The only instruction that can change a record's $status$ field to a status which is not *Removed* is the one in line 26. Assume by contradiction that $s_{j'}$ is indeed an execution of line 26 that changes R's status (meaning, $newRecord \rightsquigarrow R$). Since R's status at $C_{j'-1}$ is *Removed* (and records are created with a *Pending* status), by the induction assumption, there exists $j_1 < j'-1$ for which R's status at C_{j_1} is *InTable*. Since a record's status can only become *InTable* when executing the *add* operation during which it was created (line 10), s_{j_1} and $s_{j'}$ are executed during the same *add* execution. Since $j_1 < j'$ and the fact that the operation returns after executing line 26 for the first time, we get a contradiction. Therefore, R's status at $C_{j'}$ is *Removed*
2) Assume R's status is *InTable* at C_j. By the induction assumption, $R.status \in \{InTable, Removed\}$ at $C_{j'-1}$. If R's status at $C_{j'-1}$ is *Removed*, then from (1), R's status at $C_{j'}$ is also *Removed*. Otherwise, since the only instruction that can change an *InTable* status is a successful cas execution in line 45 in the *remove* operation (which changes the status to *Removed*), R's status is also *InTable* or *Removed* at $C_{j'}$.
3) Assume that R's status is *Pending* at C_j and R's status is *Removed* at $C_{j'}$. If R's status at $C_{j'-1}$ is also *Removed*, then by the induction assumption, there exists $j < j'' < j'$ for which R's status at $C_{j''}$ is *InTable*. If R's status at $C_{j'-1}$ is *InTable*, then there also exists $j < j'' < j'$ for which R's status at $C_{j''}$ is *InTable*.
   Assume by contradiction that R's status at $C_{j'-1}$ is *Pending*. Then $s_{j'}$ must be the instruction in line 45 during a *remove* operation, for which $victim \rightsquigarrow R$. Since the condition checked in line 43 must not hold for R, there exists $j_1 < j'-1$ for which R's status at C_{j_1} is *InTable*. By the induction assumption, R's status at $C_{j'-1}$ cannot be *Pending* - a contradiction. Therefore, there exists $j < j'' < j'$ for which R's status at $C_{j''}$ is *InTable*.

Claim 67: Let $\alpha = C_0 \cdot s_1 \cdot C_1 \cdot s_2 \cdot C_2 \cdot \ldots \cdot s_k \cdot C_k$ be a legal execution, let $0 \leq j \leq k$, let $i \in \{1, \ldots, N_T\}$ and let R_1, R_2 be two different records. If R_1 is not an i-infant at C_j then either $R_1 <_i R_2$ or $R_2 <_i R_1$.

Proof 46: See the proof of claim 14 in Section A.

Claim 68: Let R_1 and R_2 be two records. If $pred \rightsquigarrow R_1$ and $curr \rightsquigarrow R_2$, when $\langle pred, curr \rangle$ is the output of a *find*(val, i) execution, then $R_1.data[i] < val \leq R_2.data[i]$.

Proof 47: If the last assignment into $pred$ was in line 2 then R_1 is $head$ and, since $-\infty$ is never sent as input to the *find* method (claim 63), $R_1.data[i] < val$. Otherwise, the last assignment into $pred$ was in line 7, right after the condition in line 5 did not hold for R_1. Therefore, $R_1.data[i] < val$ in this case also. Regarding R_2: since R_2 was returned as $curr$ in line 6, the condition in line 5 held for R_2. Therefore, $val \leq R_2.data[i]$.

After stating our basic observations, we are now going to prove some small table invariants, in a similar way to the proof of claim 21 in Section A. Notice that here we also rely on the fact that during a *find* execution, $pred$ and $curr$ always reference actual records (and not just nulls). This is ensured by the first claim invariant.

Claim 69: Let $\alpha = C_0 \cdot s_1 \cdot C_1 \cdot s_2 \cdot C_2 \cdot \ldots \cdot s_k \cdot C_k$ be a legal execution, let $0 \leq j \leq k$, $1 \leq i \leq N_T$ and let R_1, R_2 be two different records, then:

1) If $pred \rightsquigarrow X$ or $curr \rightsquigarrow X$ at C_j, during a *find*(val, i) execution, then X is a record. In particular, if $pred \rightsquigarrow X$ then $X.data[i] < val$.
2) If R_2 is the i-successor of R_1 at C_j then R_2 is not an i-infant at C_j.
3) If $pred \rightsquigarrow R_1$ and $curr \rightsquigarrow R_2$ at C_j, during a *find*(val, i) execution, then R_1 and R_2 are not i-infants at C_j.
4) If R_1 is not an i-infant at C_j and its status is not *Removed* at C_j, then R_1 is i-reachable at C_j.
5) If $R_1 <_i R_2$ then R_1 does not i-follow R_2 at C_j.
6) If R_2 i-follows R_1 at C_{j-1} and its status is not *Removed* at C_j, then R_2 i-follows R_1 at C_j.
7) If R_1 is not an i-infant at C_j then *tail* i-follows R_1 at C_j.

Proof 48: Invariants 1, 3 and 6 are vacuously true at C_0. In addition, invariant 2 is true because *tail* cannot be an i-infant at C_0 by definition, invariant 4 is true because both *head* and *tail* are i-reachable at C_0, invariant 5 is true because *head* does not i-follow *tail* at C_0, and invariant 7 is true because *tail* i-follows both *head* and *tail* at C_0.

Assume that the invariants hold throughout $\alpha' = C_0 \cdot s_1 \cdot C_1 \cdot \ldots \cdot C_{j-1}$ of the execution. We show that each claim invariant holds throughout $\alpha = \alpha' \cdot s_j \cdot C_j$.

1) Suppose $pred \rightsquigarrow X$ at C_j. If $pred \rightsquigarrow X$ at C_{j-1} as well then by the induction assumption, X is a record and $X.data[i] < val$. Otherwise, s_j is either the execution of line 2 or 7.
   If s_j is the execution of line 2 then X is *head* and thus, it is a record. In addition, from claim 63, $X.data[i] = -\infty < val$. If s_j is the execution of line 7 then $curr \rightsquigarrow X$ at C_{j-1}. By the induction assumption, X is a record. In addition, since the condition in line 5 does not hold for X, $X.data[i] < val$.
   Now, suppose $curr \rightsquigarrow X$ at C_j. If $curr \rightsquigarrow X$ at C_{j-1} as well then by the induction assumption, X is a record. Otherwise, s_j is either the execution of line 3 or 8.
   If s_j is the execution of line 3, then X is *head*'s i-successor at C_{j-1}. By the induction assumption (invariant 7), *tail* i-follows *head* at C_{j-1}. Therefore, by definition, X is a record in this case. If s_j is the execution of line 8, let R be the record for which $pred \rightsquigarrow R$ at C_{j-1}. By the induction assumption, R is a record. In addition, by the induction assumption (invariant 3), R is not an i-infant at C_{j-1}, Therefore, by the induction assumption (invariant 7), *tail* i-follows R at C_{j-1}. Since X is R's i-successor at C_{j-1}, by definition X is a record.
2) If R_2 is also the i-successor of R_1 at C_{j-1} then by the induction, R_2 is not an i-infant at C_{j-1} and by definition, R_2 is not an i-infant at C_j. Otherwise, s_j must be the execution of line 24, 25 or 47.
   If s_j is the execution of line 24 then $newRecord \rightsquigarrow R_1$ and $curr_i \rightsquigarrow R_2$ at C_{j-1}. By the induction assumption (invariant 3), R_2 is not an i-infant at C_{j-1} and therefore, not an i-infant at C_j as well.
   If s_j is the execution of line 25 then $pred_i \rightsquigarrow R_1$ and $newRecord \rightsquigarrow R_2$. By definition 58, R_2 is not an i-infant at C_j.
   If s_j is the execution of line 47 then $pred_i \rightsquigarrow R_1$ at C_{j-1}. Let R_3 be the record for which $victim \rightsquigarrow R_3$ at C_{j-1}. Then R_2 is the i-successor of R_3 at C_{j-1}. Bu the induction assumption, R_2 is not an i-infant at C_{j-1} and thus, not an i-infant at C_j.
3) If $pred \rightsquigarrow R_1$ at C_{j-1} then by the induction assumption, R_1 is not an i-infant at C_{j-1} and by definition, it is not an i-infant at C_j as well. Otherwise, s_j is either the execution of line 2 or 7. If s_j is the execution of line 2 then $R_1 =$ *head* and by definition, R_1 is not an i-infant at C_j. If s_j is the execution of line 7 then $curr \rightsquigarrow R_1$ at C_{j-1} and by the induction assumption, it is not an i-infant at C_{j-1}. By definition, R_1 is not an i-infant at C_j.
   If $curr \rightsquigarrow R_2$ at C_{j-1} then by the induction assumption, R_2 is not an i-infant at C_{j-1} and by definition, it is not an i-infant at C_j as well. Otherwise, s_j is either the execution of line 3 or 8. In both cases, R_2 has an i-predecessor at C_{j-1} and by the induction assumption (invariant 2), it is not an i-infant at C_{j-1}. By definition, R_2 is not an i-infant at C_j in both cases.
4) Suppose R_1 is not i-reachable at C_{j-1}. By claim 66, R_1's status at C_{j-1} is not *Removed* and therefore, by the induction assumption, R_1 must be an i-infant at C_{j-1}. By definition, s_j is the execution of line 25 for which $newRecord \rightsquigarrow R_1$. Let R be the record for which $pred_i \rightsquigarrow R$ at C_{j-1}. By the induction assumption (invariant 3), R is not an i-infant at C_{j-1}.
   Let s_{j_1} be the execution of line 14 which obtains R's i-th lock, and let $j_2 = j - 1$. By definition, for every $j_1 \leq j' \leq j_2$, R is partially locked at $C_{j'}$. By claim 60, R's status at C_{j_1} is equal to R's status at C_{j_2}. Since there exists $j_1 \leq j' \leq j_2$ for which R's status at $C_{j'}$ is *InTable* (the condition checked in line 16 must not hold for R), by claim 66, R's status at C_{j-1} is *InTable*. By the induction assumption, R is i-reachable at C_{j-1}. Since s_j only affects R's i-successor, R is still i-reachable at C_j. By definition, R_1 is also i-reachable at C_j.
   Now, suppose R_1 is i-reachable at C_{j-1}. By definition, there exist records $R_{i_0}, R_{i_1}, \ldots, R_{i_n}$ for which $R_{i_0} =$ *head*, $R_{i_n} = R_1$ and for every $0 < t \leq n$, R_{i_t} is the i-successor of $R_{i_{t-1}}$ at C_{j-1}. By definition, R_{i_0} is not an i-infant at C_{j-1}. In addition, by the induction assumption (invariant 2), for every $0 < t \leq n$, R_{i_t} is also not an i-infant at C_{j-1}.
   Assume by contradiction that R_1 is not i-reachable at C_j. Let t be the minimal number among $0, \ldots, n$ for which R_{i_t} is not i-reachable at C_j. Since R_1 is not i-reachable at C_j and *head* is always i-reachable by definition, $0 < t \leq n$. Since $R_{i_{t-1}}$ is still i-reachable at C_j, R_{i_t} is no longer $R_{i_{t-1}}$'s i-successor at C_j.

Therefore, s_j is either the execution of line 25 or 47. Notice that s_j cannot be the execution of line 24, since by claim 65, this means that $R_{i_{t-1}}$ is an i-infant at C_{j-1}. If s_j is the execution of line 25, then $pred_i \rightsquigarrow R_{i_{t-1}}$ at C_j. Notice that since $R_{i_{t-1}}$'s i-th lock is obtained in line 14, by claim 64, its i-successor when checking the condition in line 16 is equal to its i-successor at C_{j-1}. Since $R_{i_{t-1}}$'s i-successor at C_{j-1} is R_{i_t}, $curr_i \rightsquigarrow R_{i_t}$ at C_{j-1} (as well as at C_j). Let R be the record for which $newRecord \rightsquigarrow R$ at C_j. Since R is $R_{i_{t-1}}$'s i-successor and R_{i_t} is R's i-successor at C_j (since we obtain R's i-th lock, by claim 64, R_{i_t} is R's i-successor after the execution of line 24), by definition R_{i_t} is i-reachable at C_j - a contradiction.

If s_j is the execution of line 47, then $pred_i \rightsquigarrow R_{i_{t-1}}$ at C_j. Notice that since $R_{i_{t-1}}$'s i-th lock is obtained in line 39, by claim 64, its i-successor when checking the condition in line 40 is equal to its i-successor at C_{j-1}. Since $R_{i_{t-1}}$'s i-successor at C_{j-1} is R_{i_t}, $victim \rightsquigarrow R_{i_t}$ at C_{j-1}. Since R_{i_t}'s status after executing line 45 is *Removed*, by claim 66, R_{i_t}'s status at C_j is also *Removed*. Therefore, $t < n$. Since $R_{i_{t+1}}$ is R_{i_t}'s i-successor at C_{j-1}, it is $R_{i_{t-1}}$'s i-successor at C_j. In addition, since s_j only affects $R_{i_{t-1}}$'s $next[i]$ field, by definition R_{i_n} is i-reachable at C_j - a contradiction.

We get a contradiction either when s_j is the execution of line 25 or the execution of line 47. Since these are the only possible options, R_1 is i-reachable at C_j.

5) Suppose $R_1 <_i R_2$. Assume by contradiction that R_1 i-follows R_2 at C_j. By definition, there exist records $R_{i_0}, R_{i_1}, \ldots, R_{i_n}$ for which $R_{i_0} = R_2$, $R_{i_n} = R_1$ and for every $0 < t \leq n$, R_{i_t} is the i-successor of $R_{i_{t-1}}$ at C_{j-1}. By the induction assumption (invariant 2), for every $0 < t \leq n$, R_{i_t} is not an i-infant at C_{j-1}. Therefore, by claim 67, for every $0 < t \leq n$, either $R_{i_{t-1}} <_i R_{i_t}$ or $R_{i_t} <_i R_{i_{t-1}}$.

Since $R_{i_n} <_i R_{i_0}$, there must exist a minimal t among $1, \ldots, n$ for which $R_{i_t} <_i R_{i_{t-1}}$. By the induction assumption, R_{i_t} does not i-follow $R_{i_{t-1}}$ at C_{j-1} and thus, s_j updates $R_{i_{t-1}}$'s i-successor. Therefore, s_j is either the execution of line 24, 25 or 47.

If s_j is the execution of line 24 then $newRecord \rightsquigarrow R_{i_{t-1}}$ and $curr_i \rightsquigarrow R_{i_t}$ at C_j. By claim 68, $R_{i_{t-1}}.data[i] \leq R_{i_t}.data[i]$. Since $R_{i_t} <_i R_{i_{t-1}}$, by definition $R_{i_{t-1}}.data[i] = R_{i_t}.data[i]$ and there exists j' for which R_{i_t} is an i-infant at $C_{j'}$ and $R_{i_{t-1}}$ is not an i-infant at $C_{j'}$. By claim 65, $R_{i_{t-1}}$ is an i-infant at C_j (since line 25 has not been executed yet). Since R_{i_t} is not an i-infant at C_j, there does not exist such j' - a contradiction.

If s_j is the execution of line 25 then $pred_i \rightsquigarrow R_{i_{t-1}}$ and $newRecord \rightsquigarrow R_{i_t}$ at C_j. By claim 68, $R_{i_{t-1}}.data[i] < R_{i_t}.data[i]$ and thus, $R_{i_{t-1}} <_i R_{i_t}$ - a contradiction.

If s_j is the execution of line 47 then $pred_i \rightsquigarrow R_{i_{t-1}}$. Let R be the record for which $victim \rightsquigarrow R$ at C_j. Then R_{i_t} is R's i-successor at C_{j-1}. By the induction assumption, $R_{i_{t-1}} <_i R$ and $R <_i R_{i_t}$. Since the $<_i$ relation is a transitive relation by definition, $R_{i_{t-1}} <_i R_{i_t}$ - a contradiction.

Since every possible case derives a contradiction, if $R_1 <_i R_2$ then R_1 does not i-follow R_2 at C_j.

6) Since R_2 i-follows R_1 at C_{j-1}, by the induction assumption (invariant 2), R_2 is not an i-infant at C_{j-1}. Since R_2 i-follows R_1 at C_{j-1}, by definition there exist records $R_{i_0}, R_{i_1}, \ldots, R_{i_n}$ for which $R_{i_0} = R_1$, $R_{i_n} = R_2$ and for every $0 < t \leq n$, R_{i_t} is the i-successor of $R_{i_{t-1}}$ at C_{j-1}. In addition, by the induction assumption (invariant 2), for every $0 < t \leq n$, R_{i_t} is not an i-infant at C_{j-1}.

Assume by contradiction that R_2 does not i-follow R_1 at C_j. Let t be the minimal value among $0, \ldots, n$ for which R_2 i-follows R_{i_t} at C_j. Notice that since R_2 does not i-follow R_1 and does i-follow itself (by definition) at C_j, $0 < t \leq n$. Since R_2 does not i-follow $R_{i_{t-1}}$ at C_j, R_{i_t} is not the i-successor of $R_{i_{t-1}}$ at C_j (and it does not i-follow it as well). Therefore, s_j is either the execution of line 24, 25 or 47.

Suppose s_j is the execution of line 24 for which $newRecord \rightsquigarrow R_{i_{t-1}}$. That means that s_j is executed during the *add* execution for which $R_{i_{t-1}}$ is created in line 10. Therefore, by claim 65, $R_{i_{t-1}}$ is an i-infant at C_j. By the induction assumption (invariant 3), there does not exist $j'' < j$ such that $s_{j''}$ updates $R_{i_{t-1}}$'s i-successor in line 25 or 47. In addition, since line 24 can be executed at most once during every *add* execution, $R_{i_{t-1}}$'s $next[i]$ field is null at C_{j-1}. Therefore, R_{i_t} is not the i-successor of $R_{i_{t-1}}$ at C_{j-1} - a contradiction.

Suppose s_j is the execution of line 25, for which $pred_i \rightsquigarrow R_{i_{t-1}}$ at C_j. Notice that $R_{i_{t-1}}$'s i-th lock is held by the thread executing this *add* operation, since before executing line 16. By claim 64, the i-successor of $R_{i_{t-1}}$ when executing line 16 is still its i-successor at C_{j-1}. Therefore, $curr_i \rightsquigarrow R_{i_t}$ at C_j. Now let R be the record for which $newRecord \rightsquigarrow R$ at C_j. Since R's i-successor after executing line 24 is R_{i_t}, and since the executing thread holds R's i-th lock, by claim 64, R_{i_t} is R's i-successor at C_j. Since R is $R_{i_{t-1}}$'s i-successor at C_j, by definition, R_{i_t} i-follows $R_{i_{t-1}}$ at C_j - a contradiction.

Suppose s_j is the execution of line 47 for which $pred_i \rightsquigarrow R_{i_{t-1}}$. When checking the condition in line 40, $R_{i_{t-1}}$'s i-th lock is already obtained by the executing thread. Therefore, by claim 64, $R_{i_{t-1}}$'s i-successor at this point is equal to $R_{i_{t-1}}$'s i-successor at C_{j-1}. Therefore, $victim \rightsquigarrow R_{i_t}$ at C_j. In addition, $R_{i_{t+1}}$ is $R_{i_{t-1}}$'s i-successor at C_j. Notice that indeed $t < n$ in this case, since R_{i_t}'s status when executing line 45 is *Removed* (R_2's status at C_j is *InTable*, and by claim 66, it cannot be *Removed* at an earlier configuration). Now, since $R_{i_{t+1}}$ is R_{i_t}'s i-successor at C_j, R_2 i-follows $R_{i_{t+1}}$. Since $R_{i_{t+1}}$ is $R_{i_{t-1}}$'s i-successor at C_j, R_2 also i-follows $R_{i_{t-1}}$ at C_j - a contradiction.

Since every possible case derives a contradiction, R_2 i-follows R_1 at C_j.

7) If R_1 is an i-infant at C_{j-1} then s_j is the execution of line 25, for which $newRecord \rightsquigarrow R_1$. Let R be the record for which $curr_i \rightsquigarrow R$ at C_j. Let s_{j_1} be the execution of line 24 for which R becomes R_1's i-successor. Since R_1's i-th lock is held by the thread executing this *add* operation, by claim 64, R is also R_1's i-successor at C_j. By the induction assumption (invariant 3), R is not an i-infant at C_{j-1} and therefore, by the induction assumption, *tail* i-follows R at C_{j-1}. Since s_j only affects R_1's $next[i]$ field (and by invariant 5, R_1 is not in the path from R to *tail* at C_{j-1}), *tail* i-follows R at C_j as well. By definition, *tail* i-follows R_1 at C_j in this case.
Now, suppose R_1 is not an i-infant at C_{j-1}. By the induction assumption, *tail* i-follows R_1 at C_{j-1}. By claim 62, *tail*'s status at C_j is *InTable* and thus, by invariant 6, *tail* i-follows R_1 at C_j in this case as well.

Now, given a legal execution $\alpha = C_0 \cdot s_1 \cdot C_1 \cdot s_2 \cdot C_2 \cdot \ldots \cdot s_k \cdot C_k$, we are able to show that all table invariants hold for α, using lemmas 70, 71, 72 and 74. Notice that being logically in the table is defined as having an *InTable* status and being i-reachable for every $1 \leq i \leq N_T$.

Lemma 70: For every $1 \leq i \leq N_T$ and $0 \leq j \leq k$, *head* and *tail* are i-reachable at C_j.

Proof 49: Since *head* is i-reachable at C_j by definition, we only need to show that *tail* is i-reachable at C_j. By claim 62, *tail*'s status at C_j is *InTable*. In addition, by definition, *tail* is not an i-infant at C_j. By claim 69 (invariant 4), *tail* is i-reachable at C_j.

Lemma 71: Let R be a record and let $0 \leq j \leq k$. R is logically in the table if and only if its status is *InTable*.

Proof 50: By definition, if R is logically in the table at C_j then its status is *InTable* at C_j. Therefore, we only need to show the opposite direction. Assume that R's status at C_j is *InTable*. In order to show that R is logically in the table, we need to show that for every $1 \leq i \leq N_T$, R is i-reachable at C_j.

Let $1 \leq i \leq N_T$. Assume by contradiction that R is an i-infant at C_j. Since records are created with a *Pending* status, R's status has already been changed in line 26. Before executing line 26, R ceased being an i-infant at C_j (when line 25 was executed during the i-th loop iteration) - a contradiction. Therefore, R is not an i-infant at C_j. Since its status is not *Removed* at C_j, by claim 69 (invariant 4), R is i-reachable at C_j.

Lemma 72: Let R_1 and R_2 be two different records, let $0 \leq j \leq k$ and let $1 \leq i \leq N_T$. If R_2 i-follows R_1 then $R_1 <_i R_2$.

Proof 51: By claim 69 (invariant 2), R_2 is not an i-infant at C_j and thus, by claim 67, either $R_1 <_i R_2$ or $R_2 <_i R_1$. By claim 69 (invariant 5), $R_1 <_i R_2$.

Claim 73: Let R_1 and R_2 be two records, let $0 \leq j \leq k$ and let $1 \leq i \leq N_T$ such that i is a unique field number. If R_2 is the i-successor of R_1 at C_j then $R_1.data[i] < R_2.data[i]$.

Proof 52: We are going to prove claim 73 by induction. The lemma holds for C_0 since $head.data[i] < tail.data[i]$ for every $1 \leq i \leq N_T$.

Now, assume that the lemma holds for every $j' < j$ and $1 \leq i \leq N_T$ such that i is a unique field number. Let $1 \leq i \leq N_T$ such that i is a unique field number. We are going to show that the lemma holds for i and j. Let R_1 and R_2 be two records such that R_2 is the i-successor of R_1 at C_j. In particular, R_2 i-follows R_1 at C_j. Therefore, by lemma 72, $R_1.data[i] \leq R_2.data[i]$. Assume by contradiction that $R_1.data[i] = R_2.data[i]$. By the induction assumption, R_2 is not the i-successor of R_1 at C_{j-1}. Therefore, s_j is either the execution of line 24, 25 or 47.

If s_j is the execution of line 24 then $newRecord \rightsquigarrow R_1$ and $curr_i \rightsquigarrow R_2$ at C_j. Since this line is executed and i is a unique field number, the condition in line 19 must not hold for R_1 R_2. Therefore, $R_1.data[i] \neq R_2.data[i]$ - a contradiction. If s_j is the execution of line 25 then $pred_i \rightsquigarrow R_1$ and $newRecord \rightsquigarrow R_2$ at C_j. By claim 68, $R_1.data[i] < R_2.data[i]$ - a contradiction. If s_j is the execution of line 47 then $pred_i \rightsquigarrow R_1$ and $victim \rightsquigarrow R_2$ at C_j. By claim 68, $R_1.data[i] < R_2.data[i]$ - a contradiction.

Since every case derives a contradiction, $R_1.data[i] \neq R_2.data[i]$. Therefore, if R_2 is the i-successor of R_1 at C_j, then $R_1.data[i] < R_2.data[i]$.

Lemma 74: Let R_1 and R_2 be two records, let $0 \leq j \leq k$ and let $1 \leq i \leq N_T$ such that i is a unique field number. If both R_1 and R_2 are i-reachable at C_j then $R_1.data[i] \neq R_2.data[i]$.

Proof 53: Assume by contradiction that there exist two records R_1 and R_2 such that both R_1 and R_2 are i-reachable at C_j and $R_1.data[i] = R_2.data[i]$. By claim 59, either R_1 i-follows R_2 or R_2 i-follows R_1 at C_j. Without loss of generality, we assume that R_2 i-follows R_1 at C_j. By definition, there exist records $R_{i_0}, \ldots, R_{i_n}$ such that $R_{i_0} = R_1$, $R_{i_n} = R_2$ and for every $0 < t \leq n$, R_{i_t} is the i-successor of $R_{i_{t-1}}$ at C_j. Since $R_1.data[i] = R_2.data[i]$, by lemma 72, $R_{i_0}.data[i] = R_{i_1}.data[i]$ - a contradiction to claim 73. Therefore, if both R_1 and R_2 are i-reachable at C_j then $R_1.data[i] \neq R_2.data[i]$.

B. Linearizability proof

We are now going to show that our lock-based table is linearizable. Let $\alpha = C_0 \cdot s_1 \cdot C_1 \cdot s_2 \cdot C_2 \cdot \ldots \cdot s_k \cdot C_k$ be a legal execution. As in the lock-free version, we are going to define linearization points for each of the *add*, *remove* and *retrieve* operations, terminated before C_k. Notice that here, we also define a linearization point for the *find* executions, since the linearization points of some of the table operations rely on them.

a) find: Assume a *find(val, i)* execution was invoked at s_{j_1} and returned at s_{j_2} ($0 < j_1 < j_2 \leq k$), and let R_1 and R_2 be the two records returned as output from this call (respectively). We are going to show that there exists a configuration C_j, which is between the invocation and termination of this call, for which R_1 is i-reachable and R_2

is R_1's i-successor at C_j. In order to do so, we are going to use claims 75, 76 and 77.

Claim 75: Let R be a record, $0 \leq j \leq k$ and $1 \leq i \leq N_T$. If R's status at C_j is *Removed* then R's i-successor at C_j is equal to R's i-successor at C_{j-1}.

Proof 54: The claim is vacuously true at C_0. Now assume that the claim holds for $0, \ldots, j-1$. We are going to show that the claim still holds for j. Assume by contradiction that R's status at C_j is *Removed* and that R's i-successor at C_j is different from R's i-successor at C_{j-1}. Then s_j is either the execution of line 24, 25 or 47.

If s_j is the execution of line 24, then $newRecord \rightsquigarrow R$. Since records are created with a *Pending* status, by claim 66, there exists $j' < j$ for which R's status at $C_{j'}$ is *InTable*. Notice that a record's status can only become *InTable* after the execution of line 26 during the *add* execution for which it was created in line 10. Since R was created in line 26 of the current *add* execution, this means that line 24 cannot be executed at s_j - a contradiction.

If s_j is the execution of line 25 then $pred_i \rightsquigarrow R$ at C_j. When executing line 16 during the i-th loop iteration, R's status is *InTable*. R is partially locked since before the execution of line 16, and until C_j. Therefore, by claim 60, R's status at C_j is also *InTable* - a contradiction.

If s_j is the execution of line 47 then then $pred_i \rightsquigarrow R$ at C_j. When executing line 40, during the i-th loop iteration, R's status is *InTable*. R is partially locked since before the execution of line 40, and until C_j. Therefore, by claim 60, R's status at C_j is also *InTable* - a contradiction.

Since each possible case derives a contradiction, if R's status at C_j is *Removed* then R's i-successor at C_j is equal to R's i-successor at C_{j-1}.

Claim 76: There exists $j_1 < j < j_2$ for which R_1 is i-reachable at C_j.

Proof 55: If $R_1 =$ *head* then by definition, it is i-reachable for every $j_1 < j < j_2$ and we are done. Otherwise, We are going to prove that each record which is referenced by the $pred$ local variable during the execution is i-reachable at some point during the execution (and in particular, R_1). Since the execution terminates, there is a finite series of records $R_{i_0}, R_{i_1}, \ldots, R_{i_n}$ which are assigned into the $pred$ local variable during the execution. Let $s_{t_0}, s_{t_1}, \ldots, s_{t_n}$ be the steps assigning them into the $pred$ variable, respectively ($t_0 < t_1 < \ldots < t_n$). Notice that $R_{i_0} =$ *head* and that s_{t_0} is the execution of line 2. In addition, R_{i_n} is R_1 and s_{t_n} is the last execution of line 7 during the method execution.
We are going to prove by induction on m that there exists $j_1 < j \leq t_m$ for which R_{i_m} is i-reachable at C_j.
Base Case (m = 0): By definition, R_{i_0} is i-reachable at C_{t_0}.
Induction Step: Assume that there exists $j_1 < j' \leq t_{m-1}$ for which $R_{i_{m-1}}$ is i-reachable at $C_{j'}$. If R_{i_m} is $R_{i_{m-1}}$'s i-successor at $C_{j'}$, then we are done. Otherwise, let $j' < j \leq t_m$ be the minimal value for which R_{i_m} is $R_{i_{m-1}}$'s i-successor at C_j. Notice that since R_{i_m} is $R_{i_{m-1}}$'s i-successor at C_{t_m}, j must exist. By claim 69 (invariant 3), $R_{i_{m-1}}$ is not an i-infant at $C_{j'}$ and thus, not an i-infant at C_j. In addition, by the minimality of j, $R_{i_{m-1}}$'s i-successor at C_j is different from its i-successor at C_{j-1}. Therefore, by claim 75, $R_{i_{m-1}}$'s status at C_j is not *Removed*. By claim 69 (invariant 4), $R_{i_{m-1}}$ is i-reachable at C_j and thus, R_{i_m} is also i-reachable at C_j.

Claim 77: There exists $j_1 < j < j_2$ for which R_1 is i-reachable and R_2 is R_1's i-successor at C_j.

Proof 56: Let $s_{j'}$ be the assignment of R_2 into the method's $curr$ local variable. Then $s_{j'}$ must be either the execution of line 3 or 8. In both cases, R_2 is R_1's i-successor at $C_{J'}$. If R_1 is i-reachable at $C_{j'}$ then we are done (by setting $j = j'$). Otherwise, by claim 76, there exists $j_1 < j'' \leq j'$ for which R_1 is i-reachable at $C_{j''}$.

If R_2 is R_1's i-successor at $C_{j''}$, then we are done. Otherwise, let $j'' < j \leq j'$ be the minimal value for which R_2 is R_1's i-successor at C_j. Notice that since R_2 is R_1's i-successor at $C_{j'}$, j must exist. By claim 69 (invariant 3), R_1 is not an i-infant at $C_{j''}$ and thus, not an i-infant at C_j. In addition, by the minimality of j, R_1's i-successor at C_j is different from its i-successor at C_{j-1}. Therefore, by claim 75, R_1's status at C_j is not *Removed*. By claim 69 (invariant 4), R_1 is i-reachable at C_j. In addition, R_2 is R_1's i-successor at C_j.
The configuration C_j, guaranteed by claim 77, is the linearization point of every *find* execution that has terminated.

b) add: We are going to define the linearization points of each *add(tup)* execution that has terminated during α. The definitions of successful and unsuccessful executions remain the same as in Section A.

Claim 78: Given an *add(tup)* operation which was invoked at s_{j_1} and terminated at s_{j_2} ($0 < j_1 < j_2 \leq k$), let R be the record created in line 10 (meaning, $newRecord \rightsquigarrow R$ during the execution). If the operation returned $true$ then there exists $j_1 < j < j_2$ for which R is logically in the table at C_j.

Proof 57: The operation returned $true$ in line 27. Let s_j be the execution of line 26. Obviously, $j_1 < j < j_2$ and R's status at C_j is *InTable*.
We define the linearization point of a successful *add(tup)* execution that has terminated to be C_j, described in claim 78. From claim 78, it is guaranteed that the new record created during the operation, and which represents the tuple tup, has been logically added into the table during the execution.

Given an unsuccessful *add(tup)* operation, we want to set a linearization point which is between the invocation and the termination of the operation, for which a record representing tup cannot be logically inserted into the table. Formally, let $tup = \langle e_1, \ldots, e_{N_T} \rangle$ and let $i \in \{1, \ldots, N_T\}$ be a unique field number. We want to show that at the linearization point there exists a record R' which is not the record created in line 10 during the operation, satisfying $R'.data[i] = e_i$, and is logically in the table during that linearization point.

Claim 79: Given an *add(tup)* operation which was invoked at s_{j_1} and terminated at s_{j_2} ($0 < j_1 < j_2 \leq k$), let R be the record created in line 10 (meaning, $newRecord \rightsquigarrow R$ during the execution). If the operation returned $false$ then there exist *(1)* $j_1 < j < j_2$, *(2)* $i \in \{1, \ldots, N_T\}$ for which i is a unique field number, and *(3)* a record $R' \neq R$ whose status is *InTable* at C_j and for which $R'.data[i] = R.data[i]$.

Proof 58: The operation returned $false$ in line 21. Let R' be the record for which the conditions checked in line 19 and 20 held before returning. In addition, let s_j be the execution of line 20. Obviously, $j_1 < j < j_2$, and the status of R' at C_j is *InTable*. In addition, there exists $i \in \{1, \ldots, N_T\}$ for which i is a unique field number and $R'.data[i] = R.data[i]$. Therefore, j satisfies all of the needed conditions.
We define the linearization point of an unsuccessful *add(tup)* execution that has terminated to be C_j, described in claim 79. From claim 79, it is guaranteed that inserting the new record into the table at C_j would break the uniqueness property of our table.

c) remove: The definitions of successful and unsuccessful *remove* executions remain the same as in Section A as well.
Assume a successful *remove(val, i)* was invoked at s_{j_1} and returned at s_{j_2}. Let R be the record found in line 29 (meaning, $victim \rightsquigarrow R$ throughout the execution). We want to set a linearization point s_j for which $j_1 < j < j_2$, and s_j is an operation step that logically removes R from the table.

Notice that, since the operation returns in line 48, we can choose s_j to be the execution of line 45, setting the record's status to *Removed*. The only thing we still need to show that the record's status is indeed *InTable* before this step (meaning, by lemma 71, this step logically removes the record from the table). We are going to show it in claim 80.

Claim 80: Let s_j be the execution of line 45 and let R be the record for which $victim \rightsquigarrow R$ at C_j. Then R's status at C_{j-1} is *InTable*.

Proof 59: When executing line 43, R's status is *InTable*. Since all of R's locked are obtained by the executing thread before executing line 43, by claim 61, R's status at C_{j-1} is also *InTable*.
Now, assume that an unsuccessful *remove(val, i)* operation was invoked at s_{j_1} and returned at s_{j_2} ($0 < j_1 < j_2 \leq k$). We want to set a linearization point C_j between the invocation and termination of the operation ($j_1 < j < j_2$), for which there does not exist a record that can be removed by the operation (meaning, the operation failure is justified). An unsuccessful *remove* returns either in line 31, 38 or 44. We are going to set different linearization points, based on the line in which the operation returns.
If the operation returns in line 31, let R_1 and R_2 be the records for which $pred \rightsquigarrow R_1$ and $victim \rightsquigarrow R_2$ at C_{j_2-1}. In addition, let C_j be the linearization point of the *find* call, invoked in line 29. Obviously, $j_1 < j < j_2$.

Claim 81: There does not exist a record R which is logically in the table at C_j and for which $R.data[i] = val$.

Proof 60: By claim 77, R_1 is i-reachable and R_2 is R_1's i-successor at C_j (meaning, R_2 is i-reachable at C_j as well). By claim 68, $R_1.data[i] < val \leq R_2.data[i]$. Now, assume by contradiction that there exists a record R which is logically in the table at C_j and for which $R.data[i] = val$. In particular, R is i-reachable at C_j.

If the operation returns because $R_2.data[i] \neq val$, then $val < R_2.data[i]$. By claim 59, either R i-follows R_1 or R_1 i-follows R at C_j. In addition, either R i-follows R_2 or R_2 i-follows R at C_j. By lemma 72, R i-follows R_1 and R_2 i-follows R at C_j. Since R_2 is R_1's i-successor at C_j, we get a contradiction.

Otherwise, if $R_2.data[i] = val$ then the operation returns because R_2's status at C_j is not *InTable*. By lemma 74, if $R \neq R_2$ then R is not i-reachable at C_j. Therefore, $R \neq R_2$, meaning R's status at C_j is not *InTable* - a contradiction.

Since every possible case derives a contradiction, there does not exist a record R which is logically in the table at C_j and for which $R.data[i] = val$.
By claim 81, if the operation returns in line 31 then there does not exist a record that can be logically removed from the table at C_j. Therefore, C_j is the operation's linearization point in this case.

Now, suppose the operation returns in line 38. Let R_{-1} be the record assigned into the $pred_m$ local variable and let $R_0, R_1, \ldots, R_n$ be the records assigned into the $curr_m$ local variable during the last loop iteration (the one during which the operation returns in line 38). In addition, let C_{t_0} be the linearization point of the *find(victim.data$[m]$, m)* execution, called in line 34 and let $s_{t_1}, s_{t_2}, \ldots, s_{t_n}$ be the steps assigning $R_1, R_2, \ldots, R_n$ into $curr_m$, respectively ($j_1 < t_0 < t_1 < \ldots < t_n < j_2$). Finally, let R be the record for which $victim \rightsquigarrow R$.

Claim 82: There exists $j_1 < j \leq t_n$ for which R's status is *Removed* and R is i-reachable at C_j.

Proof 61: Since the operation returns in line 38, the condition checked in line 30 holds for R. Therefore, there exists $j_1 < j' < t_0$ for which R's status is *InTable* at $C_{j'}$. By lemma 71, R is both i-reachable and m-reachable at $C_{j'}$.

Let $s_{j''}$ be the execution of line 30. Assume by contradiction that R's status is *InTable* at C_j for every $j'' < j \leq t_n$. By lemma 71, R is m-reachable at C_j for every $t_0 \leq j \leq t_n$. In particular, it is m-reachable at C_{t_0}. By claim 77, both R_{-1} and R_0 are m-reachable and at C_{t_0}. In addition, by claim 68, $R_{-1}.data[m] < R.data[m] \leq R_0.data[m]$. Since R is also m-reachable at C_{t_0}, by claim 59 and lemma 72, it must m-follow R_{-1} at C_{t_0}. Since $R \neq R_0$ and R_0 is R_{-1}'s m-successor at C_{t_0}, R m-follows R_0 at C_{t_0} as well. By claim 69 (invariant 5), $R.data[m] = R_0.data[m]$ and thus, $n > 0$.

Let a be the minimal number among $0, 1, \ldots, n$ for which R does not m-follows R_a at C_{t_a}. Notice that since R m-follows R_0 at C_{t_0}, $a > 0$. In addition, since $R_n.data[m] \neq R.data[m]$ and $R_{n-1}.data[m] = R.data[m]$, by lemma 72, $R_n.data[m] > R.data[m]$. Therefore, by claim 69 (invariant 5), R cannot m-follow R_n at C_{t_n} and thus, $a \leq n$.

Since R m-follows R_{a-1} at $C_{t_{a-1}}$ and R's status at C_{t_a} is *InTale*, by claim 21 (invariant 6), R m-follows R_{a-1} at C_{t_a} as well. Since R does not m-follow R_a at C_{t_a} and R_a is R_{a-1}'s m-successor at C_{t_a}, we get a contradiction.

Therefore, there exists $j'' < j \leq t_n$ for which R's status is not *InTable* at C_j. Let j be the minimal number among $j'', j''+1, \ldots, t_n$ for which it holds. Since R's status at $C_{j''}$

is *InTable*, by claim 66, R's status at C_j is *Removed*. By lemma 71, R is i-reachable at C_{j-1}. Since s_j only changes R's status, R is also i-reachable at C_j. Therefore, there exists $j_1 < j \leq t_n$ for which R's status is *Removed* and R is i-reachable at C_j.
By claim 82, if the operation returns in line 38, there exists $j_1 < j \leq t_n$ for which R's status is *Removed* and R is i-reachable at C_j. Since $R.data[i] = val$, by lemma 74, there does not exist a record R' which is logically in the table at C_j and for which $R'.data[i] = val$. Therefore, C_j can serve as the operation's linearization point in this case.
We still need to define the operation's linearization point when it returns in line 44. In this case, let R be the record for which $victim \rightsquigarrow R$ throughout the execution. Then $R.data[i] = val$. In addition, since the operation returns in line 44, R's status when executing line 30 is *InTable* and R's status when executing line 43 is not *InTable*. By claim 66, R's status when executing line 43 is *Removed*. Therefore, there exists $j_1 < j < j_2$ for which s_j changes R's status from *InTable* to *Removed*. By lemma 71, R is i-reachable at C_{j-1}. Since s_j only changes R's status, R is also i-reachable at C_j. Since $R.data[i] = val$, by lemma 74, there does not exist a record R' which is logically in the table at C_j and for which $R'.data[i] = val$. Therefore, C_j can serve as the operation's linearization point in this case.

d) retrieve: Assume a *retrieve*(val, i) execution, invoked at s_{j_1} and terminated at s_{j_2} (returning a set S of tuples). We want to set a linearization point C_j for which *(1)* $j_1 < j < j_2$, *(2)* for every record R satisfying $R.data[i] = val$, it holds that $R.data \in S$ if and only if R is logically in the table at C_j.

Here we also set different linearization points according to the last step executed before the operation terminates. We start with the case when the operation returns in line 54. Let C_j be the linearization point of the *find*(val, i) execution, called in line 52 (obviously, $j_1 < j < j_2$), and let R_0 and R_1 be the two records returned from this call (meaning, $pred \rightsquigarrow R_0$ and $curr \rightsquigarrow R_1$). By claim 68, $R_0.data[i] < val \leq R_1.data[i]$. Since the operation returns in line 54, $val < R_1.data[i]$. By claim 77, R_0 is i-reachable and R_1 is R_0's i-successor at C_j.

Assume by contradiction that there exists a record R which is logically in the table at C_j and for which $R.data[i] = val$. In particular, R is i-reachable at C_j. By claim 59 and claim 69 (invariant 11), R i-follows R_0 at C_j. Since $R.data[i] < R_1.data[i]$, by claim 69 (invariant 5), R does not i-follow R_1 at C_j - a contradiction.

Therefore, the empty set (returned in line 54) is indeed the set of all tuples $R.data$ such that R is logically in the table at C_j and $R.data[i] = val$. In the case when the operation returns in line 54, C_j (as described above) is its linearization point.
Now, suppose the operation returns in line 64. As in the former case, let R_0 and R_1 be the two records returned from the *find* call in line 52, during the final $retry$ iteration. In addition, let $R_1, \ldots, R_n$ be the records assigned into $curr$ in line 63. Notice that since the operation does not return in line 54, $n \geq 1$. In this case, let s_j be the last execution of line 59. We choose C_j to be the linearization point of the operation in this case. Obviously, $j_1 < j < j_2$.

Claim 83: Let R be a record for which $R.data[i] = val$. Then R is logically in the table at C_j if and only if there exists $1 \leq t \leq n-1$ such that $R = R_t$.

Proof 62: First, we are going to show that for every $1 \leq t \leq n-1$, $R_t.data[i] = val$ and R_t is logically in the table at C_j. Let $1 \leq t \leq n-1$. The condition checked in line 60 does not hold for R_t and therefore. Since we obtain R_t's i-th lock before checking its status, by claim 61, R_t's status at C_j is *InTable*. By lemma 71, R_t is logically in the table at C_j. In addition, since $1 \leq t \leq n-1$, the condition checked in line 58 holds for R_t and thus, $R_t.data[i] = val$.

Now, we are going to show that for every record R such that $R.data[i] = val$, if there does not exist $1 \leq t \leq n$ such that $R = R_t$ then R is not logically in the table at C_j. Assume by contradiction that there exists a record R such that $R.data[i] = val$ and R is logically in the table at C_j. In particular, R is i-reachable at C_j and by claim 59, either R i-follows R_1 or R_1 i-follows R at C_j.

Since the condition checked in line 56 does not hold for R_0 and R_1, and since R_0's i-th lock is obtained before this condition is checked, by claims 61 and 64, R_0's status at C_j is *InTable* and R_1 is R_0's i-successor at C_j. By lemma 71, R_0 is i-reachable at C_j as well. By claim 68, $R_0.data[i] < val$ and thus, by claim 69 (invariant 5), R_0 cannot i-follow R. Since $R \neq R_1$, R does not i-follow R_1 at C_j.

Since either R i-follows R_1 or R_1 i-follows R at C_j, R must i-follow R_1 at C_j. Notice that for every $1 \leq t \leq n-1$, R_t's i-th lock is obtained in line 59 before we read its i-successor in line 63. Therefore, by claim 64, for every $1 \leq t \leq n-1$, R_{t+1} is still R_t's i-successor at C_j. Since there does not exist $1 \leq t \leq n-1$ for which $R = R_t$, it means that R must i-follow R_n at C_j. Since $R_n.data[i] \neq val$ (the condition checked in line 58 does not hold for R_n) and R_n i-follows R_1 at C_j, by lemma 72, $R_n.data[i] > val$. Since $R.data[i] = val$, by lemma 72 R cannot i-follow R_n at C_j.

Therefore, R is not i-reachable at C_j - a contradiction. Conclusion: for every record R satisfying $R.data[i] = val$, R is logically in the table at C_j if and only if there exists $1 \leq t \leq n-1$ such that $R = R_t$.
By claim 83, the set $\{R_1, \ldots R_{n-1}\}$, returned in line 64, is indeed the set of all tuples $R.data$ such that R is logically in the table at C_j and $R.data[i] = val$. In the case when the operation returns in line 64, C_j (as described above) is its linearization point.
After defining linearization points for every *add*, *remove* and *retrieve* execution that has terminated, we end our linearization proof with the following theorem:

Theorem 84: The implementation given in Figure 7, 8, 9 and 10 is a linearizable implementation of a lock-based table.

C. Deadlock Freedom

One of the necessary conditions for a deadlock situation is a circular wait. Meaning, there is a set of waiting threads, $\{t_1, t_2, \ldots t_n\}$, such that t_1 is waiting for a lock held by t_2, t_2

is waiting for a lock held by t_3 and so on until t_n is waiting for a lock held by t_1. We are going to show that a circular wait cannot occur during any legal execution.

In order to do so, we use a partial order $<_l$ of locks and make sure that lock l_1 is never acquired before a lock $l_0 <_l l_1$ (unless l_1 is released before trying to acquire l_0). We define $<_l$ by saying that $R_0.locks[i_0] < R_1.locks[i_1]$ if either $i_0 < i_1$, or $i_0 = i_1$ and $R_0 <_{i_0} R_1$. Notice that since $<_i$ is a constant order (i.e., if $R <_i R'$ then it will never reach a state where $R' <_i R$), $<_l$ is a constant order as well.

Notice that all acquired locks are released before returning from a table operation (lines 21, 27, 31, 38, 44, 48 and 64) or starting a new operation trial (lines 17, 22, 41, 57 and 61). Therefore, we only need to show that during an operation trial, locks are acquired according to $<_l$. Let $\alpha = C_0 \cdot s_1 \cdot C_1 \cdot s_2 \cdot C_2 \cdot \ldots \cdot s_k \cdot C_k$ be a legal execution, let R_1 and R_2 be two records, let s_{j_1} be the acquirement of R_1's i_1-th lock and let s_{j_2} be the acquirement of R_2's i_2-th lock, by the same thread ($0 < j_1 < j_2 \leq k$).

Claim 85: If s_{j_1} and s_{j_2} are executed during the same *add* trial then $R_1.locks[i_1] <_l R_2.locks[i_2]$.

Proof 63: Notice that locks are only acquired in lines 14-15. If s_{j_1} and s_{j_2} are executed during different loop iterations, then $i_1 < i_2$ and by definition, $R_1.locks[i_1] <_l R_2.locks[i_2]$. Otherwise, $i_1 = i_2$, $pred_{i_1} \rightsquigarrow R_1$ and $newRecord \rightsquigarrow R_2$. Since R_1 is the first output parameter, returned from the *find($newRecord.data[i_1], i_1$)* call in line 13, by claim 68, $R_1.data[i_1] < R_2.data[i_1]$. By definition, $R_1 <_i R_2$ and thus, $R_1.locks[i_1] <_l R_2.locks[i_2]$.

Claim 86: If s_{j_1} and s_{j_2} are executed during the same *remove* trial then $R_1.locks[i_1] <_l R_2.locks[i_2]$.

Proof 64: Notice that locks are only acquired in lines 39 and 42. If s_{j_1} and s_{j_2} are executed during different loop iterations, then $i_1 < i_2$ and by definition, $R_1.locks[i_1] <_l R_2.locks[i_2]$. Otherwise, $i_1 = i_2$, $pred_{i_1} \rightsquigarrow R_1$ and $victim \rightsquigarrow R_2$. Since R_1 is the first output parameter, returned from the *find($victim.data[i_1], i_1$)* call in line 34, by claim 68, $R_1.data[i_1] < R_2.data[i_1]$. By definition, $R_1 <_i R_2$ and thus, $R_1.locks[i_1] <_l R_2.locks[i_2]$.

Claim 87: If s_{j_1} and s_{j_2} are executed during the same *retrieve* trial then $R_1.locks[i_1] <_l R_2.locks[i_2]$.

Proof 65: Notice that locks are only acquired in lines 55 and 59. In addition, it must hold that $i_1 = i_2$. If $pred \rightsquigarrow R_1$ then by claim 68, $R_1.data[i_1] < val$. Since R_2's i_1's lock is acquired in line 59, $R_2.data[i_1] = val$. By definition, $R_1 <_i R_2$ and thus, $R_1.locks[i_1] <_l R_2.locks[i_2]$.

Otherwise, R_1 and R_2s' i_1-th locks are acquired in line 59. If R_1 is the second output parameter, returned from the *find* execution called in line 52, then by claim 69 (invariant 3), R_1 is not an i_1-infant at C_{j_2}. Otherwise, R_1 has an i_1-predecessor when assigned into the $curr$ local variable in line 63, and by claim 69 (invariant 2), R_1 is not an i_1-infant at C_{j_2} in this case as well. From the same reason, R_2 is also not an i_1-infant at C_{j_2}. Therefore, by claim 67, either $R_1 <_{i_1} R_2$ or $R_2 <_{i_1} R_1$.

Assume by contradiction that $R_1.locks[i_1] <_l R_2.locks[i_2]$ does not hold. By definition, $R_1 <_{i_1} R_2$ does not hold and thus, $R_2 <_{i_1} R_1$. Let $R_{k_0}, \ldots, R_{k_n}$ be the records assigned into the $curr$ local variable during the execution trial, starting from R_1 and ending with R_2 (meaning, $R_{k_0} = R_1$ and $R_{k_n} = R_2$). For every $1 \leq t \leq n$, when R_{k_t} is assigned into the $curr$ local variable, $R_{k_{t-1}}$'s i_1-th lock is already obtained and R_{k_t} is $R_{k_{t-1}}$'s i_1-successor. Since all locks are still held when R_{t_n}'s i_1-th lock is obtained, by claim 64, for every $1 \leq t \leq n$, R_{k_t} is still $R_{k_{t-1}}$'s i_1-successor at this point. Therefore, by definition, R_2 i_1-follows R_1 at this point.

Using claim 69 (invariant 5), we get a contradiction to the fact that $R_2 <_{i_1} R_1$. Therefore, $R_1 <_{i_1} R_2$ and $R_1.locks[i_1] <_l R_2.locks[i_2]$ in this case as well.

By claims 85, 86 and 87, during every table operation trial, locks are acquired according to $<_l$. Since the $<_l$ relation is anti-symmetric by definition, a deadlock situation can never occur. We end our deadlock-freedom proof with the following theorem:

Theorem 88: The implementation given in Figure 7, 8, 9 and 10 is a deadlock-free implementation of a lock-based table.