A Scalable Linearizable Multi-Index Table

Gal Sheffi
A Scalable Linearizable Multi-Index Table

Research Thesis

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Gal Sheffi

Submitted to the Senate of the Technion — Israel Institute of Technology
Nisan 5777 Haifa April 2017
This research was carried out under the supervision of Prof. Erez Petrank, in the Faculty of Computer Science.

The Technion’s funding of this research is hereby acknowledged.
Contents

List of Figures

Abstract .................................................. 1

1 Introduction .......................................... 3

2 Preliminaries .......................................... 5
  2.1 Table Terminology .................................... 5
  2.2 Table Operations .................................... 5

3 The Algorithm .......................................... 7
  3.1 Internal Data Structure .............................. 8
  3.2 The Lock-Free Table ................................. 10
    3.2.1 The Lock-Free find Method ....................... 10
    3.2.2 The Lock-Free add Operation ..................... 11
    3.2.3 The Lock-Free remove Operation .................. 13
    3.2.4 The Lock-Free retrieve Operation ............... 14
  3.3 The Lock-Based Table ............................... 17
    3.3.1 The Lock-Based find Method ...................... 18
    3.3.2 The Lock-Based add Operation .................... 18
    3.3.3 The Lock-Based remove Operation ................. 20
    3.3.4 The Lock-Based retrieve Operation ............... 21
  3.4 Adding a Fast Skip-List Indexing ................... 24

4 Correctness ............................................ 25
  4.1 The Lock-Free Table ............................... 26
    4.1.1 Table Invariants ................................. 26
    4.1.2 Linearizability proof ............................ 44
    4.1.3 Progress .......................................... 53
  4.2 The Lock-Based Table ............................... 64
    4.2.1 Table Invariants ................................. 64
    4.2.2 Linearizability proof ............................ 73
    4.2.3 Deadlock Freedom ................................ 80
List of Figures

3.1 A Table Example ........................................ 7
3.2 The Lock-Free find Method ................................ 10
3.3 The Lock-Free add Operation ............................. 12
3.4 The Lock-Free remove Operation ...................... 14
3.5 The Lock-Free retrieve Operation ..................... 15
3.6 The Lock-Based find Method ............................. 18
3.7 The Lock-Based add Operation .......................... 19
3.8 The Lock-Based remove Operation .................... 20
3.9 The Lock-Based retrieve Operation ................... 22

5.1 Throughput Graphs ....................................... 83
Abstract

Concurrent data structures typically index data using a single primary key and provide fast access to data associated with a given key value. However, it is often required to access information via multiple primary and secondary keys, and even through additional properties that do not represent keys for the given data. One way to implement a data structure that allows access via multiple indexing is to use an existing data structure that allows access via a single index, and traverse the items exhaustively to find items by a secondary key. However, using this method, it is difficult to achieve either correctness (linearizability) or good performance. In this thesis, we propose to obtain multi-indexing access by using an efficient concurrent data structure for each of the indexed fields. We then coordinate these structures to obtain correctness and good performance. We propose lock-free and lock-based designs of a table with multiple indexing, supporting linearizable inserts, deletes, and retrieve operations. We have implemented the proposed designs and ran measurements on a 64-core AMD platform. The obtained performance demonstrates efficiency and high scalability. In particular, measurements show that lock-free solutions perform better than lock-based ones when contention is high, as well as in most practical scenarios.
Chapter 1

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 [RG03, TWF16]. 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.

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 construction that uses a transactional type of concurrency (see [CBM08, DSS06, SK16]).

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 [HW90], 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 a full-fledged database typically supports many more types of queries and a transactional type of concurrency. They also often employ large structures and involved algorithms to allow spilling data to hard disks. 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. [JCSL]). However, this solution creates severe problems for efficiency and linearizability. From performance standpoint, finding an item given a secondary key requires traversing the table, which is costly. But more importantly, it is difficult to design a linearized solution without holding a coarse-grained lock on the table during the entire operation. For example, the add operation must be able to check whether any of the unique keys exist in the table, the retrieve operation must be able to obtain a linearizable view of all elements in order to determine whether an element is currently in the list, etc. One possibility for obtaining linearizability is to employ a snapshot mechanism for all the operations (using, e.g., [GBH15, PT13]). But that would still require a costly traversal of the table.

We begin in Chapter 2 with some basic notations and definitions and continue in Chapter 3 with a detailed description of the lock-free and lock-based table variants (an optimization for both variants appears in Section 3.4). The correctness proof of both table variants appears in Chapter 4, and the linearization points of the algorithm are specified in Sections 4.1.2 and 4.2.2. Chapter 5 presents the performance measurements, Chapter 6 presents some related work and we conclude in Chapter 7.
Chapter 2

Preliminaries

Our model for concurrent multi-threaded computation follows the linearizability model of [HW90]. 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. In addition, we assume an automatic memory management.

2.1 Table Terminology

A table is a set of $n$-tuples $(e_1, \ldots, e_n)$ 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 $(e_1, \ldots, e_n)$ and $(e'_1, \ldots, e'_n)$ in $T$ satisfy $e_i \neq e'_i$. Otherwise (i.e., $f_i$ is non-unique), $T$ may contain two different tuples $(e_1, \ldots, e_n)$ and $(e'_1, \ldots, e'_n)$ such that $e_i = e'_i$.

An interesting special case of the table data structure is the map data structure. A map is essentially a data structure that represents a function from keys to values. A map can be defined as a table $M$ with two fields: a unique field $f_1$ that holds the key and a non-unique field $f_2$ that holds the data.

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 $\infty$. The elements $-\infty$ and $\infty$ represent the minimal and the maximal possible element that can be stored in a tuple.

2.2 Table Operations

The proposed table supports the add, remove and retrieve operations. The add operation receives a tuple $(e_1, \ldots, e_n)$ and adds it to the table if and only if for every unique field number $i$ the table does not contain a tuple $(e'_1, \ldots, e'_n)$ 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 if 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.
Chapter 3

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.

Figure 3.1: A table example, 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)
In this chapter we describe the algorithm. We start by presenting the data structure we use (in Section 3.1). We then continue with a detailed description of the lock-free table algorithm (Section 3.2) and of the lock-based variant (Section 3.3). Finally, we explain the additional fast skip-list indexing (Section 3.4).

3.1 Internal Data Structure

Our table organizes the stored tuples as a collection of sorted linked-lists. For each field $f_i$, the table maintains a linked-list $L_i$ such that if tuple $\langle \ldots, e_i, \ldots \rangle$ precedes tuple $\langle \ldots, e'_i, \ldots \rangle$ in $L_i$ then $e_i \leq e'_i$. In other words, 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 updated simultaneously and atomically, using a single cas execution (as in Line 8 of Figure 3.2). For example, this can be realized in Java using an atomic object that represents a pointer and a boolean flag.

In order to implement a simultaneous insertion (or deletion) of a record into all the linked-lists at once, we use the status field. 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 \((-\infty, \ldots, -\infty)\) and \((\infty, \ldots, \infty)\), 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 3.1 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.
3.2 The Lock-Free Table

In this section we describe the lock-free table design. Lock-freedom means that when the program threads (executing the table operations) are run for a sufficiently long time, at least one of the threads makes progress (the operation it executes eventually terminates).

We say that a record is logically in the table if it is physically in all internal linked-lists and its status is InTable. 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.

Although parts of the add and remove operations may include collaborations of several threads (as explained later), those collaborations do not affect the identity of the thread executing a successful add or remove of a record. The thread executing a successful insertion of a record is the thread that created it (even if other threads have physically inserted it into some of the lists, or changed its status from Pending to InTable), and the thread executing a successful removal of a record is the one that changed its status (from InTable to Removed).

1: find(val, i)
2: retry:
3: pred ← head
4: curr ← pred.next[i]
5: while true do
6:   succ, marked ← curr.next[i]
7:   while marked do
8:     snip ← pred.next[i].cas(curr, succ, false, false)
9:   if !snip
10:     goto retry
11:   curr ← pred.next[i]
12:   succ, marked ← curr.next[i]
13: if curr.data[i] ≥ val
14:   return pred, curr
15: pred ← curr
16: curr ← succ

Figure 3.2: The Lock-Free find Method

3.2.1 The Lock-Free 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 3.2). 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] ≥ val and (3) curr is pred’s successor in L_i.

When adding a new record R such that R.data[i] = val, pred serves as the prede-
cessor, and \textit{curr} serves as the successor in \(L_i\). Notice that since threads can operate concurrently, there is no guarantee that the returned records will continue to satisfy property (3). Therefore, the calling operation always checks the relevance of the returned values (as will later be explained) and if needed, calls the \textit{find} method again.

The method starts its traversal from \textit{head} (which is the first record in all internal linked-lists, and in particular in \(L_i\)). Assuming \(val\) is always bigger than \(-\infty\) (no operation requires searching for such value), the initial record referenced by \textit{curr} is \textit{head}'s successor in \(L_i\) (line 4). The \textit{pred} and \textit{curr} references are advanced throughout \(L_i\), until the relevant two records are found and returned (line 14). Eventually, \textit{curr} points to the first record \(R\) in \(L_i\) whose \(next[i]\) pointer is not marked and satisfies \(R.data[i] \geq val\), and \textit{pred} is its predecessor in \(L_i\). Assuming \(val\) is always smaller than \(\infty\), a potential \textit{curr} can always be found.

In addition to finding the described records, the \textit{find} method also helps physically removing records that have been logically removed (see [H01]). Records are removed from the table during unsuccessful \textit{add} and successful \textit{remove} executions.

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

The \textit{cas} execution may fail for two reasons. The first one is that \textit{curr} is no longer \textit{pred}'s successor in \(L_i\) (either \textit{curr} has been physically removed or a new record has been added between them) and the second possible reason for a \textit{cas} failure is that \textit{pred}'s \(i\)-th \(next\) pointer has meanwhile become marked. In these cases, when a \textit{cas} execution in line 8 fails, the method can’t make sure that the traversal can continue from the same point, and the traversal starts all over again (line 10).

### 3.2.2 The Lock-Free \textit{add} Operation

The \textit{add} operation (shown in Figure 3.3) receives a tuple \(tup\), creates a new corresponding record \(R\) with the data of \(tup\) and a \textit{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 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 \textit{helpAdding} method. When returning from the \textit{helpAdding} method, if the insertion was successful, \(R\)'s status is either \textit{InTable} or \textit{Removed} (the latter 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 \textit{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.

17: \textit{add}(tup)
18: newRecord \leftarrow \text{makerecord}(tup) \quad \triangleright \text{Creating the new record}
19: \text{helpAdding}(\text{newRecord}, 1) \quad \triangleright 1 \text{ stands for the first list}
20: \textbf{if} newRecord.status == Failed \quad \triangleright \text{Insertion was unsuccessful}
21: \quad \textbf{for} i = 1 \textbf{ to } N_T \textbf{ do }
22: \quad \quad \text{succ, marked} \leftarrow \text{newRecord.next}[i]
23: \quad \quad \textbf{while} !\text{marked} \textbf{ do }
24: \quad \quad \quad \text{newRecord.next}[i].cas(succ, succ, false, true)
25: \quad \quad \quad \text{succ, marked} \leftarrow \text{newRecord.next}[i]
26: \text{return} false \quad \triangleright \text{Insertion was successful}
27: \textbf{else} \text{ return true}
28: \text{helpAdding}(\text{newRecord}, m)
29: \quad \textbf{for} i = m \textbf{ to } N_T \textbf{ do }
30: \quad \text{helpAddingField}(\text{newRecord}, i)
31: \quad \text{newRecord.status.cas(Pending, InTable)} \quad \triangleright \text{Logical insertion}
32: \text{helpAddingField}(\text{newRecord}, i)
33: \quad \textbf{while} newRecord.status == Pending \textbf{ do }
34: \quad \quad \text{succ} \leftarrow newRecord.next[i] \quad \triangleright \text{For a later cas}
35: \quad \quad \text{pred, curr} \leftarrow \text{find(newRecord.data}[i], i)
36: \quad \quad \text{tmp} \leftarrow \text{curr}
37: \quad \quad \textbf{while} \text{tmp.data}[i] == newRecord.data[i] \textbf{ do }
38: \quad \quad \quad \text{if} \text{tmp} == \text{newRecord} \quad \triangleright \text{The record is already in the list}
39: \quad \quad \quad \text{return}
40: \quad \quad \text{tmp} \leftarrow \text{tmp.next}[i]
41: \quad \textbf{if} Unique(i)
42: \quad \quad \textbf{if} curr.data[i] == newRecord.data[i]
43: \quad \quad \quad \textbf{if} curr.status == Pending
44: \quad \quad \quad \quad \text{helpAdding(curr, i + 1)} \quad \triangleright \text{Help adding curr}
45: \quad \quad \quad \text{if} curr.status == InTable \quad \triangleright \text{Insertion failed}
46: \quad \quad \quad \quad \text{newRecord.status.cas(Pending, Failed)}
47: \quad \quad \quad \text{break}
48: \quad \textbf{if} newRecord.next[i].cas(succ, curr, false, false)
49: \quad \quad \textbf{if} pred.next[i].cas(curr, newRecord, false, false)
50: \quad \quad \quad \text{return} \quad \triangleright \text{The record was added to the list}

Figure 3.3: The Lock-Free \textit{add} Operation

Notice that at this point, R’s status cannot be Pending anymore. It either has been changed to Failed during an unsuccessful insert (due to a violation of the uniqueness property of a certain list), or to InTable before the helpAdding method returned.

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). Notice that the executing thread does not check whether the cas operation is unnecessary. If the insertion was unsuccessful, then R’s status is already Failed. If
the insertion was successful, then the cas operation either updates R’s status, or fails because another thread has already updated it.

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. As we prove in Chapter 4, curr is the only record that may be logically in the table and contain the same i-th property. 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},...,L_{NiT} (line 44), since it is guaranteed that it has already been inserted into L_1,...,L_i (as we also prove in Chapter 4). 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.

3.2.3 The Lock-Free remove Operation

The remove operation (shown in Figure 3.4) receives a value val and a unique field number i, and tries to logically remove a record R such that R.data[i] = val. We assume
51: \textit{remove}(val, i)  
52: \quad \textit{pred, victim} ← \textit{find}(val, i) \quad \triangleright \text{We do not need \textit{pred}}  
53: \quad \textbf{if} \ \textit{victim}.\text{data}[i] \neq val  
54: \quad \quad \text{return} \ false \quad \triangleright \text{There is no such item}  
55: \quad \textbf{else if} \ \textit{!victim}.\text{status}.\text{cas}(\text{InTable, Removed})  
56: \quad \quad \text{return} \ false \quad \triangleright \text{The record has already been removed}  
57: \quad \textbf{for} \ j = 1 \ \textbf{to} \ N_T \ \textbf{do}  
58: \quad \quad \textit{succ, marked} ← \textit{victim}.\text{next}[j]  
59: \quad \quad \textbf{while} \ !\text{marked} \ \textbf{do}  
60: \quad \quad \quad \textit{victim}.\text{next}[j].\text{cas}(\textit{succ, succ, false, true})  
61: \quad \quad \quad \textit{succ, marked} ← \textit{victim}.\text{next}[j]  
62: \quad \text{return} \ true \quad \triangleright \text{The record was logically removed}

Figure 3.4: The Lock-Free \textit{remove} Operation

that $i$ is a unique field number, because otherwise, there can be several records in the table whose $i$-th property is $val$. In line 52, the \textit{remove} operation calls the \textit{find} method in order to determine whether there exists a record $R$ which is logically in the table and for which $R$.\text{data}[i] = val$. The potential victim for removal is assigned into the \textit{victim} local variable. If \textit{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 \textit{victim} from the table (by setting \textit{victim}’s status to \textit{Removed}) fails, the operation is considered to be a failure, and it returns in line 56.

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

3.2.4 The Lock-Free \textit{retrieve} Operation

The \textit{retrieve} operation (shown in Figure 3.5) 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 $R$.\text{data} for which $R$ is logically in the table and $R$.\text{data}[i] = val$).

Interestingly, for the lock-free algorithm, the \textit{retrieve} (or even a \textit{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 \textit{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 \textit{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.

63: \texttt{retrieve(val, i)}

64: \textbf{while} true \textbf{do}
65: \hspace{1em} \texttt{tmpSet} $\leftarrow \emptyset$
66: \hspace{1em} curr $\leftarrow$ head
67: \textbf{while} curr.data[i] $<$ val \textbf{do}
68: \hspace{1em} curr $\leftarrow$ curr.next[i]
69: \hspace{1em} \textbf{if} curr.data[i] $>$ val
70: \hspace{1em} \textbf{return} tmpSet
71: \hspace{1em} \textbf{first} $\leftarrow$ curr
72: \textbf{while} curr.data[i] $==$ val \textbf{do}
73: \hspace{1em} status $\leftarrow$ curr.status
74: \hspace{1em} \textbf{if} (status $==$ InTable) || (status $==$ Pending)
75: \hspace{1em} tmpSet $\leftarrow$ tmpSet $\cup$ \{curr, status\}
76: \hspace{1em} curr $\leftarrow$ curr.next[i]
77: \hspace{1em} curr $\leftarrow$ head
78: \textbf{while} curr.data[i] $<$ val \textbf{do}
79: \hspace{1em} \textbf{if} first $==$ curr
80: \hspace{1em} \texttt{tuples} $\leftarrow \emptyset$
81: \hspace{1em} \texttt{valid} $\leftarrow$ true
82: \hspace{1em} \textbf{for} \langle R, status \rangle \textbf{in} tmpSet \textbf{do}
83: \hspace{1em} \textbf{if} R.status $==$ status
84: \hspace{1em} \hspace{1em} \textbf{if} status $==$ InTable
85: \hspace{1em} \hspace{1em} \hspace{1em} \texttt{tuples} $\leftarrow$ \texttt{tuples} $\cup$ \{R.data\}
86: \hspace{1em} \hspace{1em} \textbf{else}
87: \hspace{1em} \hspace{1em} \hspace{1em} \texttt{valid} $\leftarrow$ false
88: \hspace{1em} \hspace{1em} \texttt{break}
89: \hspace{1em} \textbf{if} valid
90: \hspace{1em} \textbf{return} \texttt{tuples}

\textbf{Figure 3.5: The Lock-Free retrieve Operation}

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 and there exist in the literature various ways to deal with it in a lock-free or even wait-free manner (e.g., [PT13]). We chose a simple lock-free solution for this problem. First, notice that a new record with a given property is always added before all existing records with that property. Now, 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 $curr$ that satisfies $curr.data[i] \geq val$. 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 \texttt{tmpSet} set. Relevant records are records $R$ such that $R.data[i] = val$ and whose status is either \texttt{Pending} or \texttt{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 \textit{first} local variable. If it is not, then the whole procedure starts again (from line 64). Otherwise, it is guaranteed (see Chapter 4) that the snapshot saved in \texttt{tmpSet} is a correct snapshot of all records in the table that have \textit{val} in their \textit{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 \textit{status} (lines 83-89). If all pairs satisfy this condition, all the relevant tuples (represented by saved records with an \texttt{InTable} status) are added to the \textit{tuples} set (line 86), and returned as output (line 91). In Chapter 4 we prove that the set of tuples returned by the operation is the set of exactly all relevant tuples at the operation’s linearization point.
3.3 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. First, 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.

We now give a detailed description of the lock based variant of the table. 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 $R.locks[i]$. 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: \textit{find}(val, i)
2: \hspace{1em} \texttt{pred} ← \texttt{head}
3: \hspace{1em} \texttt{curr} ← \texttt{pred.next}[i]
4: \hspace{1em} \textbf{while true} do  \quad \triangledown \text{While not having the desired records}
5: \hspace{2em} \textbf{if} \texttt{curr.data}[i] \geq \textit{val} \quad \triangledown \text{Return the current pair of records}
6: \hspace{2em} \textbf{return} \texttt{pred}, \texttt{curr}
7: \hspace{2em} \texttt{pred} ← \texttt{curr}
8: \hspace{2em} \texttt{curr} ← \texttt{pred.next}[i]

Figure 3.6: The Lock-Based \textit{find} Method

3.3.1 The Lock-Based \textit{find} Method

Like in the lock-free variant, here the \textit{find} auxiliary method (shown in Figure 3.6) locates a record in $L_i$, by finding its potential predecessor and successor in the list. However, since physical removals are done only during \textit{remove} executions, the lock-based \textit{find} method does not physically remove records from the lists. This method receives a value $val$ and a field number $i$, and returns the \texttt{pred} and \texttt{curr} references. As in the lock-free version, it holds that (1) $\texttt{pred.data}[i] < \textit{val}$, (2) $\texttt{curr.data}[i] \geq \textit{val}$ and (3) \texttt{curr} is \texttt{pred}'s successor in $L_i$. 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 variant, the traversal starts by setting \texttt{pred} to reference \texttt{head} (which is the first record in $L_i$) and setting \texttt{curr} to reference \texttt{head}'s successor in $L_i$ (lines 2-3). After this initialization, the \texttt{pred} and \texttt{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 $\infty$, 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 \textit{find} output, \texttt{pred} and \texttt{curr} might not satisfy the above conditions anymore (either \texttt{pred} or \texttt{curr} have been meanwhile removed from the table, or a new record has been added between them). In such cases, the \textit{find} method is called again in order to find a new couple of records.

3.3.2 The Lock-Based \textit{add} Operation

The lock-based \textit{add} operation (shown in Figure 3.7) receives a tuple \texttt{tup}, creates a new corresponding record $R$ with a \texttt{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 $R'.data[i] = R.data[i]$). 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
9: add(tup)  
10: newRecord ← makerecord(tup)  \(\triangleright\) Creating the new record  
11: retry:  \(\triangleright\) Starting a new adding attempt  
12: for i = 1 to N_T do  
13: pred_i, curr_i ← find(newRecord.data[i], i)  
14: pred_i.locks[i].lock()  
15: newRecord.locks[i].lock()  
16: if (pred_i.status \neq \text{InTable}) || (pred_i.next[i] \neq \text{curr}_i)  
17: release all locks and goto retry  
18: else if Uniqu(i)  \(\triangleright\) If the i-th field is unique  
19: if curr_i.data[i] == R.data[i]  
20: if curr_i.status == \text{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] ← curr_i  
25: pred_i.next[i] ← newRecord  \(\triangleright\) Physical insertion  
26: newRecord.status ← \text{InTable}  \(\triangleright\) Logical insertion  
27: release all locks and return true

Figure 3.7: The Lock-Based add Operation

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 \(i\)-th predecessor in the \(\text{pred}_i\) local variable and the \(i\)-th successor in \(\text{curr}_i\). After finding each pair of records, we obtain \(\text{pred}_i\)'s \(i\)-th lock (line 14), obtain \(R\).locks[i], and check whether \(\text{pred}_i\) still satisfies the find’s output conditions. That includes checking whether \(\text{pred}_i\) is indeed in the table (otherwise, we cannot assume that it is still physically in \(L_i\) after obtaining the lock) and whether \(\text{pred}_i\)'s successor in \(L_i\) is indeed \(\text{curr}_i\) (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 \(\text{pred}_i\)'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 \(\text{pred}_i\) and \(\text{curr}_i\) satisfy the conditions, if \(i\) is a unique field number, we check whether \(\text{curr}_i.data[i] = R.data[i]\). If they are equal and \(\text{curr}_i\) 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 \(\text{curr}_i\) is not in the table (its status is not \text{InTable}), it means that its status is \text{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 \text{Removed}. If \(\text{curr}_i\)’s
status is \textit{Removed} then it has already been physically removed (otherwise, we would not have succeeded in obtaining its lock). Since \textit{pred} is in the table and still is \textit{curr}'s predecessor in \(L_i\), it is impossible that \textit{curr} has been physically removed. Therefore, if \textit{curr}'s status is not \textit{InTable}, it has to be \textit{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 \textit{next}[\(i\)] references and update them to reference the \textit{curr}'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 \textit{pred}.\textit{next}[\(i\)] to reference \(R\) (line 25). After the physical insertion is done, we complete the operation by setting \(R\)'s status to \textit{InTable} (line 26), releasing all locks and returning true.

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

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure3.8.png}
\caption{The Lock-Based \texttt{remove} Operation}
\end{figure}

### 3.3.3 The Lock-Based \texttt{remove} Operation

The \texttt{remove} operation (shown in Figure 3.8) receives a value \textit{val} and a field number \(i\) and tries to remove a record \(R\) such that \(R.data[i] = val\). As in the lock-free variant, 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 \texttt{find} method, in line 29 (notice that the \textit{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 \texttt{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 follows \(R\) in \(L_i\), either \(R'.data[i] > val\) or \(R'.status \neq InTable\). Therefore, if
victim.data[i] > val or victim.status ≠ 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 procedure 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 Lm, trying to obtain victim’s predecessor’s locks[m] and victim’s locks[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 predecessor in Lm, and that pred_m’s status is InTable. If one of these conditions does not hold, then we need to search for victim’s predecessor in Lm again, and we start a new removal trial (line 41). As explained before, pred_m’s successor in Lm and pred_m’s status cannot change after we obtain pred_m.locks[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 successor in Lm to be victim’s successor in Lm (lines 46-47). After the logical and physical removals are done, we release all locks and return true.

### 3.3.4 The Lock-Based retrieve Operation

Like in the lock-free variant, the retrieve operation (shown in Figure 3.9) 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
retrieve(val, i)

tuples ← ∅

if curr.data[i] > val
    return tuples

pred.locks[i].lock()

while curr.data[i] == val do
    curr.locks[i].lock()

if curr.status ≠ InTable
    release all locks and goto retry

tuples ← tuples ∪ {curr.data}

curr ← curr.next[i]

release all locks and return tuples

Figure 3.9: The Lock-Based retrieve Operation

each record $R'$ satisfying $R'.data[i] = val$, it obtains $R'.locks[i]$. 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.
3.4 Adding a Fast Skip-List Indexing

We improve access to the table by using a skip-list index [SK16]. 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 variant, 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 3.1, instead of starting from the head record, our index can be used to start traversing from the record which represents ⟨80, John, 1003⟩ (marked with *).

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 the `insert(R)` and `remove(R)` operations, 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 the `getPrev(v)` operation that returns the latest record pointer \(R\) in \(S_i\) that satisfies \(R.data[i] < v\). As noted in [SK16], such \(S_i\) can be realized by a standard linearizable skip list; in our work it is implemented as a simple wrapper of Java’s `ConcurrentSkipListSet`.

Using the index During `find(v, i)` and `retrieve(v, i)` executions, 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 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 record \(R\) 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 a 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.
Chapter 4

Correctness

In this chapter we elaborate the proof for correctness of the algorithms described in Sections 3.2 and 3.3. For both variants, we prove some basic table invariants and give a linearizability proof. In addition, we prove that our lock-free variant is indeed lock-free, and that our lock-based variant is deadlock-free.
4.1 The Lock-Free Table

This section contains the proofs of our lock-free table’s invariants (in Section 4.1.1), a linearizability proof (in Section 4.1.2) and a proof that our table implementation is lock-free (in Section 4.1.3). 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.

4.1.1 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.

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 \( \infty \) 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. 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 $0 \leq t < m$, $R_t$’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 $R_{i_0} = R_{j_0} = \text{head}$, $t \geq 1$ 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.

### 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 the thread’s 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 $R.status \in \{\text{Removed, Failed}\}$ at $C_{j'}$ then $R$’s status at $C_j$ is equal to $R$’s status at $C_{j'}$.  

27
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. 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 $R.status = Pending$ at $C_{j'}$ and $R.status = Removed$ 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, $R.status \in \{Removed, Failed\}$ at $C_{j-1}$. In this case, from (2), it also holds that $R.status \in \{InTable, Removed\}$ at $C_j$. 

28
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. $\text{head.status} = \text{tail.status} = \text{InTable at } C_j$.

2. For every $i \in \{1, \ldots, N_T\}$, head and tail are not i-marked at $C_j$.

Proof. At initialization, head and tail are created with an InTable status. We assume that for every remove(val, i) invocation, $\text{val} \notin \{-\infty, \infty\}$. 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 tails' statuses never change during the execution. Therefore, (1) always holds, and (2) holds consequentially, from claim 8. \qed

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.

\[ \{\text{Removed, Failed}\} \text{ at } C_j. \] 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 Failed $\in \{\text{Removed, Failed}\}$. 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 Removed $\in \{\text{Removed, Failed}\}$.
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 $R \not\in \{\text{head}, \text{tail}\}$ be a record. If there does not exist $j' \leq j$ for which $s_{j'}$ is a successful cas execution in line 49, satisfying $\text{newRecord} \leadsto R$, 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. 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 $<_i R <_i \text{tail}$ for every record $R$. \hfill $\square$

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 $C_j$. 

30
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 $\text{pred} \rightsquigarrow R_1$, $\text{curr} \rightsquigarrow R_2$ and $\text{succ} \rightsquigarrow R_3$. 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 $\langle R_3, \text{false} \rangle$ 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 $\text{newRecord} \rightsquigarrow R_1$ and $\text{curr} \rightsquigarrow R_2$, or of line 49, having $\text{pred} \rightsquigarrow R_1$ and $\text{newRecord} \rightsquigarrow R_2$. 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 $\langle R_2, \text{false} \rangle$ 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 $\text{pred} \rightsquigarrow R_1$ and $\text{newRecord} \rightsquigarrow R_2$. 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 $\langle R_2, \text{false} \rangle$ 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 $\infty$ 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. 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 $\text{pred} \rightsquigarrow R_1$ and $\text{curr} \rightsquigarrow R_2$, when $\langle \text{pred, curr} \rangle$ is the output of a find(val, i) execution, then $R_1.data[i] < val \leq R_2.data[i]$.

Proof. 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
As mentioned, claim 20 assumes that the \textit{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 (\textit{pred} and \textit{curr}) point to actual records (and not nulls) throughout the method execution (and in particular, when returning).

\textit{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 $\text{pred} \Rightarrow X$ or $\text{curr} \Rightarrow X$ at $C_j$, during a \texttt{find(val, i)} execution, then $X$ is a record. In particular, if $\text{pred} \Rightarrow X$ then $X.data[i] < \text{val}$.

2. If $R_2$ is the \text{i}-successor of $R_1$ at $C_j$ then $R_2$ is not an \text{i}-infant at $C_j$.

3. If $R_1$ is not an \text{i}-infant at $C_j$ then there exists $j' \leq j$ for which $R_1$ is \text{i}-reachable at $C_{j'}$.

4. If $\text{pred} \Rightarrow R_1$ and $\text{curr} \Rightarrow R_2$ at $C_j$, during a \texttt{find(val, i)} execution, then $R_1$ and $R_2$ are not \text{i}-infants at $C_j$.

5. If $\text{pred} \Rightarrow R_1$ and $\text{curr} \Rightarrow R_2$ at $C_j$, during a \texttt{find(val, i)} execution, then there exists $j_1 \leq j$ for which $R_1$ is \text{i}-reachable and $R_2$ is the \text{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 \text{i}-infants at $C_{j-1}$ and $R_2$ is not \text{i}-marked at $C_j$, then $R_2$ belongs to $R_1$’s \text{i}-followers set at $C_{j-1}$.

7. If $s_j$ is a successful \texttt{cas} execution in line 48, for which \texttt{newRecord} $\Rightarrow R_1$ and $\text{curr} \Rightarrow R_2$, then $R_1 <_i R_2$.

8. If $R_2$ is the \text{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 \text{i}-infant at $C_{j-1}$, (2) $R_2$’s \text{i}-inclusive followers set at $C_{j-1}$ is a subset of $R_1$’s \text{i}-inclusive followers set at $C_{j-1}$.

Then $R_2$’s \text{i}-inclusive followers set at $C_{j-1}$ is a subset of $R_1$’s \text{i}-followers set at $C_{j-1}$.

10. If \texttt{R1} is not \text{i}-marked and is an \text{i}-predecessor of \texttt{R2} at $C_{j-1}$, and either is \text{i}-marked or is not an \text{i}-predecessor of \texttt{R2} at $C_j$, then $s_j$ is either an \text{i}-marking, an \text{i}-snipping, an \text{i}-redirection or an \text{i}-insertion.

11. If $R_1 <_i R_2$ then $R_1$ does not \text{i}-follow $R_2$ at $C_j$. 

32
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. 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 the condition checked in line 13 did not hold, $X.data[i] < i$. 

33
Suppose \( s_j \) is the execution of line 16, and that \( curr \sim X \) at \( C_j \). Let \( R \) be the record for which \( pred \sim R \) 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 \( pred \sim R \) 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 \( pred \sim R_1 \) and \( succ \sim R_2 \), a successful \( cas \) execution in line 48 for which \( newRecord \sim R_1 \) and \( curr \sim R_2 \), or a successful \( cas \) execution in line 49 for which \( pred \sim R_1 \) and \( newRecord \sim R_2 \).

If \( s_j \) is a successful \( cas \) execution in line 8 for which \( pred \sim R_1 \) and \( succ \sim R_2 \), 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 \( newRecord \sim R_1 \) and \( curr \sim R_2 \), 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 \( pred \sim R_1 \) and \( newRecord \sim R_2 \), 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 \( j' \leq j - 1 < j \) for which \( R_1 \) is \( i \)-reachable at \( C_{j'} \). Otherwise, \( s_j \) is a successful \( cas \) execution in line 49 for which \( newRecord \sim R_1 \). Let \( R_3 \) be the record for which \( pred \sim R_3 \) 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 \( j \leq j \), the invariant holds.

4. If \( pred \sim R_1 \) and \( curr \sim R_2 \) 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 $\mapsto R_2$ at $C_j$, curr $\mapsto R_2$ 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 $\mapsto R_1$ 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 $\mapsto R_1$ 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 $C_j$.

5. If pred $\mapsto R_1$ and curr $\mapsto R_2$ 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 $\mapsto R_1$. 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 \texttt{cas} execution in line 49, having \texttt{newRecord} \( \sim R_1 \).

Let \( R_3 \) be the record for which \texttt{pred} \( \sim R_3 \) at \( C_{j-1} \). By invariant 4, \( R_3 \) is not an \( i \)-infant at \( C_{j-2} \). In addition, since the \texttt{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 \texttt{find} call in line 35, from claim 20 it holds that \( R_1.data[i] = R_2.data[i] \).

We denote the \texttt{tmp} \( \leftarrow \texttt{curr} \) 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 \texttt{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 \texttt{cas} execution in line 48 is successful, \( R_1 \) is not \( i \)-marked at \( C_{j_3} \). 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 \texttt{tmp} \( \sim R_3 \) 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 $tmp \sim R_3$.

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 $tmp \sim R_4$. 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 $pred \sim R_1$ and $succ \sim R_2$. Let $R_3$ be the record for which $curr \sim R_3$ 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 $newRecord \sim R_1$ and $curr \sim R_2$. From invariant 7, $R_1 <_i R_2$.

If $s_j$ is a successful cas execution in line 49 then $pred \sim R_1$ and $newRecord \sim R_2$. $R_2.data[i]$ 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 $0 < t \leq m$, $R_{i_t}$ 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 $0 < t \leq m$, $R_{i_t}$ 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 $R_2$ and $R_3$ 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 marked 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 $newRecord \rightsquigarrow R_1$ 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 $R_1.data[i] \leq R.data[i]$. 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 $pred \sim R_1$ and $curr \sim R_2$. Let $R_3$ be the record for which $newRecord \sim R_3$. $s_j$ 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 $0 < t \leq m$, $R_{i_t-1} \neq R_{i_t}$ and $R_{i_t}$ is $R_{i_{t-1}}$’s $i$-successor at $C_j$. By the
induction assumption (invariant 2), for every $0 < t \leq m$, $R_{it}$ is also not an $i$-infant at $C_j$ and therefore, from claim 14, either $R_{it} < i R_{it-1}$ or $R_{it-1} < i R_{it}$. Now, let $0 < t \leq m$ be the minimal value for which $R_{it} < i R_{it-1}$ (since $R_1 < i R_2$, it must exist). By the induction assumption, $R_{it}$ does not $i$-follow $R_{it-1}$. Therefore, by invariant 10, $s_j$ is either an $i$-snipping, an $i$-redirection or an $i$-insertion, changing $R_{it-1}$’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 $R_{it-1}$’s $i$-successor and an $i$-predecessor of $R_{ti}$ at $C_{j-1}$. By definition, $R_{it}$ $i$-follows $R_{it-1}$ at $C_{j-1}$ - a contradiction. Suppose $s_j$ is an $i$-redirection. Then by definition 17, $R_{it-1} < i R_{it}$ - a contradiction. Suppose $s_j$ is an $i$-insertion. Then by definition 18, $R_{it-1} < i R_{it}$ - a contradiction. Therefore, $R_1$ does not $i$-follow $R_2$ at $C_j$.

12. Suppose $R_j$ is $i$-reachable and is not $i$-marked at $C_{j-1}$. By definition, there exist records $R_{jo}, \ldots, R_{jm}$ for which $head = R_{jo}$, $R_1 = R_{jm}$ and for every $0 < t \leq m$, $R_{it}$ is $R_{it-1}$’s $i$-successor at $C_{j-1}$.

Assume by contradiction that $R_1$ is not $i$-reachable at $C_j$. If $R_{jm-1}$ 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 $R_{jm-1}$’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 $R_{jm-1}$’s $i$-followers set at $C_{j-1}$. By definition 17, $R_1$ belongs to $R_{jm-1}$’s $i$-followers set at $C_j$ and thus, it is $i$-reachable at $C_j$ - a contradiction.

Therefore, $R_{jm-1}$ is not $i$-reachable at $C_j$. Let $0 \leq t < m - 1$ be the maximal value for which $R_{it}$ 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 $R_{it}$’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 $R_{it}$’s $i$-successor. Suppose $s_j$ is an $i$-snipping. Then $R_{it+2}$ is $i$-reachable at $C_j$ - a contradiction. Suppose $s_j$ is an $i$-redirection or an $i$-insertion. Then $R_1$ $i$-follows $R_{it}$ 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 case execution in line 49, having $newRecord \leadsto R_1$. 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 $pred \leadsto R_3$ 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 \( R_{t_0}, \ldots, R_{t_m} \) for which \( R_1 = R_{t_0}, R_2 = R_{t_m} \) and for every \( 0 < t \leq m \), \( R_t \) is \( R_{t_{t-1}} \)'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 \( R_{t_{m-1}} \) 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 \( R_{t_{m-1}} \)'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 \( R_{t_{m-1}} \)'s \( i \)-followers set at \( C_{j-1} \). By definition 17, \( R_2 \) belongs to \( R_{t_{m-1}} \)'s \( i \)-followers set at \( C_j \) and thus, it \( i \)-follows \( R_1 \) at \( C_j \) - a contradiction.

Therefore, \( R_{t_{m-1}} \) does not \( i \)-follow \( R_1 \) at \( C_j \). Let \( 0 \leq t < m - 1 \) be the maximal value for which \( R_{t_i} \) \( i \)-follows \( R_1 \) at \( C_j \) (\( t \) must exist, since by definition, \( R_1 \) \( i \)-follows \( R_1 \) at \( C_j \)). By definition, \( R_{t_{i+1}} \) is not \( R_i \)'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 \( R_{t_i} \)'s \( i \)-successor. Suppose \( s_j \) is an \( i \)-snipping. Then \( R_{t_{i+2}} \) \( i \)-follows \( R_1 \) at \( C_j \) - a contradiction. Suppose \( s_j \) is an \( i \)-redirection or an \( i \)-insertion. Then \( R_2 \) \( i \)-follows \( R_{t_i} \) 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, \( \text{head} \) and \( \text{tail} \) 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 \( 0 \leq j \leq k \) and let \( 1 \leq i \leq N_T \). Then \( \text{head} \) and \( \text{tail} \) are \( i \)-reachable at \( C_j \).

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

**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 \( 0 \leq j \leq k \), \( 1 \leq i \leq N_T \) and let \( R_1 \) and \( R_2 \) be two different records. If \( R_2 \) \( i \)-follows \( R_1 \) at \( C_j \) then \( R_1.data[i] \leq R_2.data[i] \).
Proof. 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 $R_1.data[i] \leq R_2.data[i]$.

Lemma 24. Let $\alpha = C_0 \cdot s_1 \cdot C_1 \cdot s_2 \cdot C_2 \ldots \cdot s_k \cdot C_k$ be a legal execution, let $0 \leq j \leq k$ 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. 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 $\alpha' = C_0 \cdot s_1 \cdot C_1 \ldots C_{j-1}$ 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 $\rightarrow R$. 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 $\alpha = C_0 \cdot s_1 \cdot C_1 \cdot s_2 \cdot C_2 \ldots \cdot s_k \cdot C_k$ 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. Since head.data[i] < tail.data[i], the claim is vacuously true at $C_0$. Assume that the lemma holds throughout $\alpha' = C_0 \cdot s_1 \cdot C_1 \ldots C_{j-1}$ 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
Assume without loss of generality that we get a contradiction here as well. Therefore, when executing line 45, $R_3$ (the case 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 contradiction. Now, assume by contradiction that $R_2$'s status at $C_j$ is InTable. If $R_2$'s status 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.** 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 $0 < t \leq m$, $R_t$ 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 $R_1.data[i] \neq R_2.data[i]$.

### 4.1.2 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 \textit{add}, \textit{remove} and \textit{retrieve} operations, terminated before \( C_k \). In addition, it will be convenient to define linearization points for calls to the \textit{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 \textit{add} call inserting them into the table, and there does not exist an already linearized \textit{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.

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

\textit{Claim 27.} If a \textit{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 \((\text{pred} \rightarrow R_1, \text{curr} \rightarrow R_2)\), 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 \( R_3.data[i] < \text{val} \), \( R_3 \) \( i \)-precedes \( R_1 \) at \( C_j \).
5. For every record \( R_3 \) which is \( i \)-reachable at \( C_j \) and satisfying \( R_3.data[i] \geq \text{val} \), \( R_3 \) \( i \)-follows \( R_2 \) at \( C_j \).

\textit{Proof.} 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 \( R_3.data[i] < \text{val} \). 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, \( R_2.data[i] \geq \text{val} \). Since \( R_3.data[i] < \text{val} \leq R_2.data[i] \), 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 \( R_3.data[i] < \text{val} \), \( R_3 \) \( i \)-precedes \( R_1 \) at \( C_j \). Now suppose \( R_3 \) is \( i \)-reachable at \( C_j \) and \( R_3.data[i] \geq \text{val} \). 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 \textit{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 \textit{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 \textit{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 \textit{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 \textit{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. \(\square\)

We define the linearization point of a $\text{find}(val, i)$ 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.

**add** After defining the linearization points of each $\text{find}(val, i)$ execution which terminated, we can define the linearization points of each $\text{add}(tup)$ 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 $\text{add}(tup)$ 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 \textit{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, \(\text{newRecord} \bowtie R\) during the execution). If the operation returned \textit{true} then there exists \(j_1 < j < j_2\) for which \(R\) is logically in the table at \(C_j\).

\textbf{Proof.} Using lemma 24, it suffices to show that there exists \(j_1 < j < j_2\) for which \(R\)'s status at \(C_j\) is \textit{InTable}.

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

We define the linearization point of a successful \textit{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 \textit{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 \(tup = (e_1, \ldots, e_{N_T})\) 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.

\textbf{Claim 29.} Given an \textit{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, \(\text{newRecord} \bowtie R\) during the execution). If the operation returned \textit{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 \textit{InTable} at \(C_j\) and for which \(R'.data[i] = R.data[i]\).

\textbf{Proof.} If the operation returned \textit{false}, then \(R\)'s status when executing line 20 was \textit{Failed}. Since new records are created with a \textit{Pending} status, \(R\)'s status was changed in a successful \textit{cas} execution in line 46, during a \textit{helpAddingField(newRecord,i)} for which \(\text{newRecord} \bowtie R\). Let \(R'\) be the record for which \(curr \bowtie R'\) when the \textit{cas} is
We define the linearization point of an unsuccessful \texttt{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.

**remove** We define a successful \texttt{remove} execution as an execution which returned \textit{true}, and an unsuccessful \texttt{remove} execution as an execution which returned \textit{false}. Assume a successful \texttt{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, \textit{victim} \( \rightarrow R \)). 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 \texttt{cas} execution in line 55 changing \( R \)'s status from \textit{InTable} to \textit{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 \texttt{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 \texttt{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, \textit{victim} \( \rightarrow R \) during the execution). If the operation returned \textit{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.** Let \( C_{j_3} \) be the linearization point of the \texttt{find(val, i)} call in line 52, and let \( R_2 \) be the \texttt{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 \( C_{j_1} \). 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 \( R_2 < R_1 \), 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 \( R < R_1 \) 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 \texttt{remove} call), by claim 25, \( R_1 \)'s status at \( C_{j_3} \) is not \textit{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 \textit{false} in line 54, we choose \( j = j_3 \). Obviously, \( j_1 < j < j_2 \).

From claim 20, \( R_2.\text{data}[i] < \text{val} \leq R.\text{data}[i] \). Since the operation returned in line 54, \( R.\text{data}[i] \neq \text{val} \) and therefore, \( \text{val} < R.\text{data}[i] \). Since \( R.\text{data}[i] \neq \text{val} \) and there does not exist any other record \( R_1 \) for which \( R_1.\text{data}[i] = \text{val} \), 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 \textit{false} in line 56 and \( R \)'s status at \( C_{j_3} \) is not \textit{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 \( R_1.\text{data}[i] = \text{val} \), and that is logically in the table at \( C_j \). Therefore, \( j \) satisfies all conditions for that case as well.

If the operation returned \textit{false} in line 56, but \( R \)'s status at \( C_{j_3} \) is \textit{InTable}, let \( s_{j_4} \) be the unsuccessful \textit{cas} execution in line 55. Since the \textit{cas} execution in line 55 is unsuccessful, \( R \)'s status at \( C_{j_{4-1}} \) is not \textit{InTable}. Let \( C_j \) be the first configuration between \( C_{j_3} \) and \( C_{j_{4-1}} \) for which \( R \)'s status is not \textit{InTable}. Obviously, \( j_1 < j < j_2 \).

Since \( R \) is logically in the table at \( C_{j-1} \) and \( R.\text{data}[i] = \text{val} \) (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 \( R_1.\text{data}[i] = \text{val} \). 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 \( R_1.\text{data}[i] = \text{val} \). Since \( R \) is also not logically in the list at \( C_j \) (its status is not \textit{InTable} at \( C_j \)), there does not exist a record \( R_1 \) which is logically in the table at \( C_j \) and for which \( R_1.\text{data}[i] = \text{val} \). Therefore, \( j \) satisfies all conditions for that case as well. \( \square \)

We define the linearization point of an unsuccessful \textit{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.

\textbf{retrieve} For each \textit{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 \( t = \langle e_1, \ldots, e_{N_T} \rangle \) which are currently in the table and satisfy \( t.e_i = \text{val} \). Formally, assume a \textit{retrieve(val, i)} execution, invoked at \( s_j \) 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 \( R.\text{data}[i] = \text{val} \), it holds that \( R.\text{data} \in S \) if and only if \( R \) is logically in the table at \( C_j \).

\textbf{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 \( C_j \).

\textbf{Proof}. We are going to prove the claim by induction on \( m \).

\textbf{Base Case} (\( m = 2 \)): the claim obviously holds for \( j_1 \).

\textbf{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 \( 1 \leq t < m \), \( R_t \) 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. 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, \( R_{m-1} < i < R_m \).

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-follow \( R_{m-1} \) 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 \textit{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 \( j'_{t} \) be the step assigning \( R'_{t+1} \) into \( curr \). Obviously, \( j'_0 < j'_1 < \ldots < j'_{n-1} \) and for every \( 1 \leq t < n \), \( R'_t \) is an i-predecessor of \( R'_{t+1} \) at \( C_{j'_t} \). From claim 31, there exists \( j'_1 \leq j'' < j'_{n-1} \) 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. 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 been 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, \( R < R'_n \). In addition, it must hold that \( R_{n-1}.data[i] < val \) and thus, \( R_{n-1} < R \). 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 \( s_j \) be 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 \( curr \sim R \), \( R \) 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), \( R_{m+k-1} < R_{m+k} \). Therefore, since \( R_{m+k}.data[i] = val \) (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 \( R_{m-1} < R \).
We are now going to prove that our table implementation is lock-free. Meaning, if all which are in the table at

Proof. Let \( R \) be a record whose status is Pending during \( \alpha' \). 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 ≤ t ≤ 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. 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. 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 · 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. 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 ≤ j, let 1 ≤ i ≤ 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}.  

54
Proof. By definition, tail belongs to head and tails’ i-inclusive followers set at C₀ for every 1 ≤ i ≤ Nₜ. Now, let 1 ≤ i ≤ Nₜ and assume that for every record R₁ and for every j’ ≤ j – 1, if R₁ is not an i-infant at Cⱼ, then tail belongs to R₁’s i-inclusive followers set at Cⱼ.’

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

If R is an i-infant at Cⱼ₋₁, then by claim 21 (invariant 10) sⱼ is an i-insertion for which newRecord ↼ R. Let R₁ be the record for which pred ↼ R₁ at Cⱼ. By claim 21 (invariant 4), R₁ is not an i-infant at Cⱼ₋₁ and therefore, as explained above, tail belongs to its i-inclusive followers set at Cⱼ. Since R is R₁’s i-successor at Cⱼ, by definition tail also belongs to R’s i-inclusive followers set at Cⱼ.

Claim 41. Let R be the record for which newRecord ↼ R during a helpAddingField(newRecord, i) execution. Let sⱼ₁ be the execution of line 34 and let sⱼ₂ be the last execution of line 37 (during the same while loop iteration). If R is not an i-infant at Cⱼ₁ and is not i-marked at Cⱼ₂ then tmp ↼ R at Cⱼ₂.

Proof. From claim 8, R is not i-marked at Cⱼ for every j₁ ≤ j ≤ j₂. In addition, by definition, R is not an i-infant at Cⱼ for every j₁ ≤ j ≤ j₂. From claim 13, R is i-reachable at Cⱼ for every j₁ ≤ j ≤ j₂.

If R was returned as the curr output variable from the find call in line 35, then the claim obviously holds. Otherwise, let Cⱼ₀ be that find’s linearization point, guaranteed by claim 27. Let R₁₀, R₁₁, . . . , R₁ₙ be the records that the tmp local variable references during the execution of lines 36-40 and for every 1 ≤ t ≤ n, let sⱼₙ be the step reading R₁ₙ into tmp. R₁ₙ is the record returned as the curr output parameter for the find call and therefore (by claim 27), it is i-reachable at Cⱼ₀. Since j₁ < j₀ < j₂, R is also i-reachable at Cⱼ₀ and from claim 6, either R i-follows R₁₀ or R₁ₙ i-follows R at Cⱼ₀.

From claim 27, R₁₀’s i-predecessor at Cⱼ₀ (returned as the first output parameter from the find call) is also i-reachable at Cⱼ₀ and thus, either i-follows or i-precedes R at Cⱼ₀ (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 Cⱼ₀. Meaning, R cannot i-precede R₁₀ at Cⱼ₀.

Therefore, R i-follows R₁₀ at Cⱼ₀.

Assume by contradiction that R₁ₙ is not R. Let t be the minimal value among 0, . . . , n for which R does not i-follow R₁ₚ at Cⱼₚ. As shown above, t > 0. In addition, since R₁ₚ is not R, it must hold that R₁ₚ.data[i] ≠ R.data[i]. Since R₁ₚ₋₁.data[i] = R.data[i] and R₁ₙ₋₁ is an i-predecessor of R₁ₚ at Cⱼₚ, by lemma 23, R₁ₚ.data[i] > R.data[i]. Therefore, by claim 21 (invariant 11), R cannot i-follow R₁ₚ at Cⱼₚ and thus, t ≤ n.

Since R i-follows R₁ₚ₋₁ at Cⱼₚ₋₁, by definition, R belongs to R₁ₚ₋₁’s i-inclusive followers set at Cⱼₚ₋₁. From claim 21 (invariant 14), and since R is not i-marked before Cⱼ₂, R belongs to R₁ₚ₋₁’s i-inclusive followers set at Cⱼₚ. Since R₁ₚ is R₁ₚ₋₁’s i-successor
at $C_{j_i}$ and $R$ does not $i$-follow $R_{i_t}$ at $C_{j_{i_t}}$, we get a contradiction. Conclusion: $R_{i_t}$ is $R$. Therefore, $\text{tmp} \rightsquigarrow R$ at $C_{j_2}$.

Claim 42. There is a finite number of observable $i$-redirections during $\alpha'$.

Proof. By definition, $i$-redirections are done only during $\text{helpAddingField}(\text{newRecord}, i)$ in a $\text{while}$ 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 $\alpha'$. From claim 35, there is a finite number of records with a $\text{Pending}$ status during $\alpha'$. 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 $\text{newRecord} \rightsquigarrow R$.

Since the $\text{cas}$ in line 49 is executed only if the $\text{cas}$ in line 48 is successful, it must hold that the $\text{cas}$ 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 $\alpha'$ for which $\text{newRecord} \rightsquigarrow R$. 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 $\text{newRecord} \rightsquigarrow R$, there is an observable $i$-redirection that is executed during a $\text{while}$ 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 $\alpha'$.

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 $\alpha'$ 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 $\alpha'$ at all.

Claim 43. There is a finite number of $i$-snippings during $\alpha'$.

Proof. Let $1 \leq i \leq N_T$. Since the only existing records can be $\text{head}$, $\text{tail}$ 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 $\alpha'$. Then there exists a record $R_2$ for which $\text{curr} \rightsquigarrow R_2$ when executing the $i$-snipping, infinitely many times during $\alpha'$. 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 $C_j$:

56
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 \(R_{i_0} = \text{head}, R_{i_n} = R_2\) 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 \(\text{head}\) 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 \(\alpha'\). Therefore, it is an \(i\)- redirection. Since \(R_{i_t}\) is either \(\text{head}\) or has an \(i\)- predecessor at \(C_{j-1}\), by claim 21 (variant 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 \(\alpha'\), 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 \(\text{curr} \sim R_2\) 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 \(\alpha'\).

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 \(\alpha'\) at all.

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

Proof. Since the only existing records can be \(\text{head}, \text{tail}\) and records created in line 18, there are at most \(2 + k + m\) records. Assume by contradiction that the condition checked
Suppose \( R \) is the record for which \( \text{pred} \sim R \) at \( C_{j_2} \). If \( R_i \) is head, then by claim 9, \( R_i \) is not \( i \)-marked at \( C_{j_2} \). Otherwise, it has been assigned into the \( \text{pred} \) variable in line 15, during the current traversal. Let \( s_{j_3} \) be the last time for which line 7 was executed when \( curr \sim R_1 \) during the current traversal. Since the traversal is still ongoing at \( C_{j_2} \), and that \( curr \sim R_1 \) when getting to line 15, the condition did not hold. Therefore, \( R_i \) was not \( i \)-marked at \( C_{j_3} \). Since there are no \( i \)-markings after \( C_k \), and \( k < j_1 < j_3 \), \( R_i \) 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 \( R_i \)'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 \( R_i \)'s \( i \)-successor at \( C_{j_2} \) as well (by claim 21, invariant 4, \( R_i \) is not an \( i \)-infant at \( C_{j_4} \)). If \( s_{j_4} \) is the execution of line 16 then there exists \( k < j_1 < j_3 \) for which \( R \) was assigned into the \( \text{succ} \) variable, either in line 6 or 12 at \( s_{j_5} \). Obviously, \( R \) was \( R_i \)'s \( i \)-successor at \( C_{j_5} \), and for the above reasons, it is also still \( R_i \)'s \( i \)-successor at \( C_{j_2} \).

Since \( R \) is \( R_i \)'s \( i \)-successor at \( C_{j_2} \), the fact that \( R_i \) 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 \( \alpha' \). We are going to derive a contradiction by showing that some operation from \( S \) terminates during \( \alpha' \).

Claim 45. Every execution of the \text{find} method eventually terminates.

Proof. Suppose \( val \) and \( i \) are the input parameters to a \text{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 \( curr \sim R \) 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

58
is not an observable one. In addition, since there are no $i$-snippings, $i$-insertions and observable $i$-redirections during $\alpha'$, 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 $curr \rightarrow R_n$, 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. Suppose $newRecord$ and $i$ are the input parameters to a $helpAddingField$ invocation, and that $newRecord \rightarrow R$ throughout the execution. If the execution of lines 37-40 terminates before $C_{k_1}$, then we are done. Otherwise, assume by contradiction that the execution never terminates.

Let $s_k$ be the first execution of line 37 after $C_k$, and let $R_0$ be the record for which $tmp \rightarrow R_0$ at $C_{k_1}$. If $curr \rightarrow R_0$ 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. 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 $C_{k_1}$, 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 $curr \rightarrow R_0$ 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 $C_{k_1}$.  

59
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 \( R_n = \text{tail} \) 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.** 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 \( \leadsto R_0 \) at \( C_{k_1} \). If \( R_0 = \text{head} \) 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 \( R_n = \text{tail} \) 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.** 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 \( \leadsto R \) 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 \textit{cas} executions in lines 48-49 fails at every loop iteration, and \textit{R}'s status always remains \textit{Pending}.

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

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

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

Since the \texttt{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 \texttt{pred \rightleftharpoons R_t, curr \rightleftharpoons R_{t+1}}, \(R_t\) is not \textit{i}-marked and \(R_{t+1}\) is \(R_t\)'s \textit{i}-successor at \(C_{k+1}\). In addition, since \(R_t\) is not an \textit{i}-infant at \(C_{k_1}\) (by claim 21, invariant 4), its \textit{i}-successor remains \(R_{t+1}\) and it remains not \textit{i}-marked for every \(k' > k_1\). Therefore, the \textit{cas} executed in line 49 should be successful - a contradiction. Conclusion: the method always terminates when \(i = N_T\).

\underline{Induction Step:} By the induction assumption, the call to \texttt{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. \hfill \Box

\underline{Claim 50.} Every execution of the \texttt{helpAdding} method eventually terminates.

\underline{Proof.} This derives directly by the guaranteed termination of the \texttt{helpAddingField} method (claim 49). \hfill \Box

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

\underline{Claim 51.} Every execution of lines 21-25 eventually terminates.

\underline{Proof.} 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 \textit{R}'s status is \textit{Failed}. From claim 8, it remains \textit{Failed} forever.

Since the only way \textit{R}'s \textit{i}-successor can be changed after \(C_k\) is by an \textit{i}-redirection which is not observable, it can only be changed during a \texttt{helpAddingField(newRecord, i)} call
for which $newRecord \leadsto R$ (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. 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 $newRecord \leadsto R$ (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. 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 3.2, 3.3, 3.4 and 3.5 is a lock-free implementation of a table.
4.2 The Lock-Based Table

This section contains the proofs of our lock-based table’s invariants (in Section 4.2.1), a linearizability proof (in Section 4.2.2) and a proof that our lock-based table implementation is deadlock-free (in Section 4.2.3).

As in the lock-free variant 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 4.1. 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 $\Rightarrow R$, we say that $R$ is an $i$-infant at $C_j$.

4.2.1 Table Invariants

After defining the $i$-infancy term, we can use the $<_i$ partial order, defined in Section 4.1 (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 $R_1.data[i] \leq R_2.data[i]$. 

64
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. See the proof of claim 6 in Section 4.1. □

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. 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 $\text{victim} \rightarrow R$. 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. $R$’s status can only be changed when executing line 26 (and $newRecord \rightsquigarrow R$
or 45 (and $\text{victim} \rightarrow R$). 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 $\text{head}$ and $\text{tails}$’ status at $C_j$ is $\text{InTable}$.

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

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

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

Proof. See the proof of Claim 19 in Section 4.1. Notice that in the lock-based table
version, the $\text{find}$ method is called during $\text{retrieve}$ executions as well. Since we assume
neither $-\infty$ nor $\infty$ are sent as input parameters to the $\text{retrieve}$ 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. \( R \)'s \( i \)-successor can only be changed in line 24 while \( newRecord \leadsto R \), 25 while \( pred_i \leadsto R \) or 47 while \( pred_i \leadsto R \). 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 \( newRecord \leadsto R \), then \( R \) is an \( i \)-infant at \( C_{j-1} \).

Proof. 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 \( newRecord \leadsto R \). 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 \( newRecord \leadsto R \), 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. \( head \) 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, \( newRecord \leadsto R \)). 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 $\leadsto R$. 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.** See the proof of Claim 14 in Section 4.1.

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

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

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 4.1. Notice that here we also rely on the fact that during a $\text{find}$ execution, $\text{pred}$ and $\text{curr}$ 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 $\text{pred} \leadsto X$ or $\text{curr} \leadsto X$ at $C_j$, during a $\text{find(val, i)}$ execution, then $X$ is a record. In particular, if $\text{pred} \leadsto X$ then $X.data[i] < \text{val}.$

67
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 $\text{pred} \leadsto R_1$ and $\text{curr} \leadsto R_2$ at $C_j$, during a $\text{find}(\text{val}, i)$ 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 $\text{Removed}$ 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 $\text{Removed}$ 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. 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 $\mathbf{\alpha}' = C_0 \cdot s_1 \cdot C_1 \cdot \ldots \cdot C_{j-1}$ of the execution. We show that each claim invariant holds throughout $\mathbf{\alpha} = \mathbf{\alpha}' \cdot s_j \cdot C_j$.

1. Suppose $\text{pred} \leadsto X$ at $C_j$. If $\text{pred} \leadsto X$ at $C_{j-1}$ as well then by the induction assumption, $X$ is a record and $X.data[i] < \text{val}$. 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, $X.data[i] = -\infty < \text{val}$. If $s_j$ is the execution of line 7 then $\text{curr} \leadsto X$ at $C_{j-1}$. By the induction assumption, $X$ is a record. In addition, since the condition in line 5 does not hold for $X$, $X.data[i] < \text{val}$.

Now, suppose $\text{curr} \leadsto X$ at $C_j$. If $\text{curr} \leadsto X$ 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 $\text{pred} \leadsto R$ 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 \( \text{newRecord} \leadsto R_1 \) and \( \text{curr}_i \leadsto R_2 \) 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 \( \text{pred}_i \leadsto R_1 \) and \( \text{newRecord} \leadsto R_2 \). By definition 58, \( R_2 \) is not an \( i \)-infant at \( C_j \).

If \( s_j \) is the execution of line 47 then \( \text{pred}_i \leadsto R_1 \) at \( C_{j-1} \). Let \( R_3 \) be the record for which \( \text{victim} \leadsto R_3 \) at \( C_{j-1} \). Then \( R_2 \) is the \( i \)-successor of \( R_3 \) at \( C_{j-1} \). By 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 \( \text{pred} \leadsto R_1 \) 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 \( R_1 = \text{head} \) and by definition, \( R_1 \) is not an \( i \)-infant at \( C_j \). If \( s_j \) is the execution of line 7 then \( \text{curr} \leadsto R_1 \) 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 \( \text{curr} \leadsto R_2 \) 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 \textit{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 \( \text{newRecord} \leadsto R_1 \). Let \( R \) be the record for which \( \text{pred}_i \leadsto R \) 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 \textit{InTable} (the condition checked in line 16 must not hold for \( R \)), by claim 66, \( R \)’s status at \( C_{j-1} \) is \textit{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 \( R_{i_{0}} = \text{head}, R_{i_{n}} = R_1 \) 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 \( R_{it} \) is not \( i \)-reachable at \( C_j \). Since \( R_1 \) is not \( i \)-reachable at \( C_j \) and \( \text{head} \) is always \( i \)-reachable by definition, \( 0 < t \leq n \). Since \( R_{it-1} \) is still \( i \)-reachable at \( C_j \), \( R_{it} \) is no longer \( R_{it-1} \)'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 \( R_{it-1} \) is an \( i \)-infant at \( C_{j-1} \).

If \( s_j \) is the execution of line 25, then \( \text{pred}_i \sim R_{it-1} \) at \( C_j \). Notice that since \( R_{it-1} \)'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 \( R_{it-1} \)'s \( i \)-successor at \( C_{j-1} \) is \( R_{it} \), \( \text{curr}_i \sim R_{it} \) at \( C_{j-1} \) (as well as at \( C_j \)). Let \( R \) be the record for which \( \text{newRecord} \sim R \) at \( C_j \). Since \( R \) is \( R_{it-1} \)'s \( i \)-successor and \( R_{it} \) is \( R \)'s \( i \)-successor at \( C_j \) (since we obtain \( R \)'s \( i \)-th lock, by claim 64, \( R_{it} \) is \( R \)'s \( i \)-successor after the execution of line 24), by definition \( R_{it} \) is \( i \)-reachable at \( C_j \) - a contradiction.

If \( s_j \) is the execution of line 47, then \( \text{pred}_i \sim R_{it-1} \) at \( C_j \). Notice that since \( R_{it-1} \)'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 \( R_{it-1} \)'s \( i \)-successor at \( C_{j-1} \) is \( R_{it} \), \( \text{victim} \sim R_{it} \) at \( C_{j-1} \). Since \( R_{it} \)'s status after executing line 45 is \( \text{Removed} \), by claim 66, \( R_{it} \)'s status at \( C_j \) is also \( \text{Removed} \). Therefore, \( t < n \). Since \( R_{it+1} \) is \( R_{it} \)'s \( i \)-successor at \( C_{j-1} \), it is \( R_{it-1} \)'s \( i \)-successor at \( C_j \). In addition, since \( s_j \) only affects \( R_{it-1} \)'s \( \text{next}[i] \) field, by definition \( R_{it} \) 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 \( 0 < t \leq n \), \( R_{it} \) is the \( i \)-successor of \( R_{it-1} \) at \( C_{j-1} \). By the induction assumption (invariant 2), for every \( 0 < t \leq n \), \( R_{it} \) is not an \( i \)-infant at \( C_{j-1} \). Therefore, by claim 67, for every \( 0 < t \leq n \), either \( R_{it-1} <_i R_{it} \) or \( R_{it} <_i R_{it-1} \).

Since \( R_{it} <_i R_{i_0} \), there must exist a minimal \( t \) among \( 1, \ldots, n \) for which \( R_{it} <_i R_{it-1} \). By the induction assumption, \( R_{it} \) does not \( i \)-follow \( R_{it-1} \) at \( C_{j-1} \) and thus, \( s_j \) updates \( R_{it-1} \)'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 \( \text{newRecord} \sim R_{it-1} \) and \( \text{curr}_i \sim R_{it} \) at \( C_j \). By claim 68, \( R_{it-1}.\text{data}[i] \leq R_{it}.\text{data}[i] \). Since \( R_{it} <_i R_{it-1} \), by definition \( R_{it-1}.\text{data}[i] = R_{it}.\text{data}[i] \) and there exists \( j' \) for which \( R_{it} \) is an \( i \)-infant at \( C_{j'} \) and \( R_{it-1} \) is not an \( i \)-infant at \( C_{j'} \). By claim 65, \( R_{it-1} \) is an \( i \)-infant at \( C_j \) (since line 25 has not been executed yet). Since \( R_{it} \) 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 \( \text{pred}_i \sim R_{it-1} \) and \( \text{newRecord} \sim R_{it} \) at \( C_j \). By claim 68, \( R_{it-1}.\text{data}[i] < R_{it}.\text{data}[i] \) and thus, \( R_{it-1} <_i R_{it} \) - a contradiction.
If \( s_j \) is the execution of line 47 then \( \text{pred}_i \leadsto R_{t_{i-1}} \). Let \( R \) be the record for which \( \text{victim} \leadsto R \) at \( C_j \). Then \( R_{t_i} \) is \( R \)'s \( i \)-successor at \( C_j \). By the induction assumption, \( R_{t_{i-1}} <_i R \) and \( R <_i R_{t_i} \). Since the \(<_i \) relation is a transitive relation by definition, \( R_{t_{i-1}} <_i R_{t_i} \) - 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 \( R_{t_0}, R_{t_1}, \ldots, R_{t_n} \) for which \( R_{t_0} = R_1, R_{t_n} = R_2 \) and for every \( 0 < t \leq n \), \( R_{t_i} \) is the \( i \)-successor of \( R_{t_{i-1}} \) at \( C_{j-1} \). In addition, by the induction assumption (invariant 2), for every \( 0 < t \leq n \), \( R_{t_i} \) 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 \( R_{t_i} \) 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 \( R_{t_{i-1}} \) at \( C_j \), \( R_{t_i} \) is not the \( i \)-successor of \( R_{t_{i-1}} \) 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 \( \text{newRecord} \leadsto R_{t_{i-1}} \). That means that \( s_j \) is executed during the \( \text{add} \) execution for which \( R_{t_{i-1}} \) is created in line 10. Therefore, by claim 65, \( R_{t_{i-1}} \) is an \( i \)-infant at \( C_j \). By the induction assumption (invariant 3), there does not exist \( j'' < j \) such that \( s_{j''} \) updates \( R_{t_{i-1}} \)'s \( i \)-successor in line 25 or 47. In addition, since line 24 can be executed at most once during every \( \text{add} \) execution, \( R_{t_{i-1}} \)'s \( \text{next}[i] \) field is null at \( C_{j-1} \). Therefore, \( R_{t_i} \) is not the \( i \)-successor of \( R_{t_{i-1}} \) at \( C_{j-1} \) - a contradiction.

Suppose \( s_j \) is the execution of line 25, for which \( \text{pred}_i \leadsto R_{t_{i-1}} \) at \( C_j \). Notice that \( R_{t_{i-1}} \)'s \( i \)-th lock is held by the thread executing this \( \text{add} \) operation, since before executing line 16. By claim 64, the \( i \)-successor of \( R_{t_{i-1}} \) when executing line 16 is still its \( i \)-successor at \( C_{j-1} \). Therefore, \( \text{curr}_{t_i} \leadsto R_{t_i} \) at \( C_j \). Now let \( R \) be the record for which \( \text{newRecord} \leadsto R \) at \( C_j \). Since \( R \)'s \( i \)-successor after executing line 24 is \( R_{t_i} \), and since the executing thread holds \( R \)'s \( i \)-th lock, by claim 64, \( R_{t_i} \) is \( R \)'s \( i \)-successor at \( C_j \). Since \( R \) is \( R_{t_{i-1}} \)'s \( i \)-successor at \( C_j \), by definition, \( R_{t_i} \) \( i \)-follows \( R_{t_{i-1}} \) at \( C_j \) - a contradiction.

Suppose \( s_j \) is the execution of line 47 for which \( \text{pred}_i \leadsto R_{t_{i-1}} \). When checking the condition in line 40, \( R_{t_{i-1}} \)'s \( i \)-th lock is already obtained by the executing thread. Therefore, by claim 64, \( R_{t_{i-1}} \)'s \( i \)-successor at this point is equal to \( R_{t_{i-1}} \)'s \( i \)-successor at \( C_{j-1} \). Therefore, \( \text{victim} \leadsto R_{t_i} \) at \( C_j \). In addition, \( R_{t_{i+1}} \) is \( R_{t_{i-1}} \)'s \( i \)-successor at \( C_j \). Notice that indeed \( t < n \) in this case, since \( R_{t_i} \)'s status when executing line 45 is \( \text{Removed} \) (\( R_2 \)'s status at \( C_j \) is \( \text{InTable} \), and by claim 66, it cannot be \( \text{Removed} \) at an earlier configuration). Now, since \( R_{t_{i+1}} \) is \( R_{t_i} \)'s \( i \)-successor at \( C_j \), \( R_2 \) \( i \)-follows \( R_{t_{i+1}} \). Since \( R_{t_{i+1}} \) is \( R_{t_{i-1}} \)'s \( i \)-successor at \( C_j \), \( R_2 \) \( i \)-follows \( R_{t_{i-1}} \) at \( C_j \).
also \( i \)-follows \( R_{i-1} \) 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 \textit{newRecord} \( \rightarrow R_1 \). Let \( R \) be the record for which \( curr_i \) \( \rightarrow R \) at \( C_j \). Let \( s_i \) 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 \textit{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 \( \textit{next}[i] \) 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 \( \alpha \), 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 \( 1 \leq i \leq N_T \).

\textbf{Lemma 70.} For every \( 1 \leq i \leq N_T \) and \( 0 \leq j \leq k \), \textit{head} and \textit{tail} are \( i \)-reachable at \( C_j \).

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

\textbf{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.

\textit{Proof.} 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 \( C_j \).

\textbf{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 \( R_1 <_i R_2 \).

72
Proof. 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. 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. 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, \( R_{i_0}.data[i] = R_{i_n}.data[i] \) - a contradiction to claim 73. Therefore, if both \( R_1 \) and \( R_2 \) are \( i \)-reachable at \( C_j \) then \( R_1.data[i] \neq R_2.data[i] \).}

4.2.2 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.
**find** Assume a \(\text{find}(\text{val}, i)\) 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.** 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 \(\text{newRecord} \hookrightarrow R\). Since records are created with a \(\text{Pending}\) status, by claim 66, there exists \(j' < j\) for which \(R\)'s status at \(C_{j'}\) is \(\text{InTable}\). Notice that a record’s status can only become \(\text{InTable}\) after the execution of line 26 during the \(\text{add}\) execution for which it was created in line 10. Since \(R\) was created in line 26 of the current \(\text{add}\) execution, this means that line 24 cannot be executed at \(s_j\) – a contradiction.

If \(s_j\) is the execution of line 25 then \(\text{pred}_i \hookrightarrow R\) at \(C_j\). When executing line 16 during the \(i\)-th loop iteration, \(R\)'s status is \(\text{InTable}\). \(R\) 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 \(\text{InTable}\) - a contradiction.

If \(s_j\) is the execution of line 47 then \(\text{pred}_i \hookrightarrow R\) at \(C_j\). When executing line 40, during the \(i\)-th loop iteration, \(R\)'s status is \(\text{InTable}\). \(R\) 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 \(\text{InTable}\) - 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 \(C_{j-1}\). \(\square\)

**Claim 76.** There exists \(j_1 < j < j_2\) for which \(R_1\) is \(i\)-reachable at \(C_j\).

**Proof.** If \(R_1 = \text{head}\) 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 \(\text{pred}\) 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 \(R_{t_0}, R_{t_1}, \ldots, R_{t_n}\) which are assigned into the \(\text{pred}\) local variable during the execution. Let \(s_{t_0}, s_{t_1}, \ldots, s_{t_n}\) be the steps assigning them into the \(\text{pred}\) variable, respectively \((t_0 < t_1 < \ldots < t_n)\). Notice that \(R_{t_0} = \text{head}\) and that \(s_{t_0}\) is the execution of line 2. In addition, \(R_{t_n} = R_1\) 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 \(R_{t_m}\) is \(i\)-reachable at \(C_j\).
Base Case ($m = 0$): By definition, $R_{t_0}$ is $i$-reachable at $C_0$.

Induction Step: Assume that there exists $j_1 < j' \leq t_{m-1}$ for which $R_{t_m-1}$ is $i$-reachable at $C_{j'}$. If $R_{t_m}$ is $R_{t_m-1}$’s $i$-successor at $C_{j'}$, then we are done. Otherwise, let $j' < j \leq t_m$ be the minimal value for which $R_{t_m}$ is $R_{t_m-1}$’s $i$-successor at $C_j$. Notice that since $R_{t_m}$ is $R_{t_m-1}$’s $i$-successor at $C_{t_m}$, $j$ must exist. By claim 69 (invariant 3), $R_{t_m-1}$ is not an $i$-infant at $C_{j'}$ and thus, not an $i$-infant at $C_j$. In addition, by the minimality of $j$, $R_{t_m-1}$’s $i$-successor at $C_j$ is different from its $i$-successor at $C_{j-1}$. Therefore, by claim 75, $R_{t_m-1}$’s status at $C_j$ is not Removed. By claim 69 (invariant 4), $R_{t_m-1}$ is $i$-reachable at $C_j$ and thus, $R_{t_m}$ 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. 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.

$\text{add}$ We are going to define the linearization points of each add(tup) execution that has terminated during $\alpha$. The definitions of successful and unsuccessful executions remain the same as in Section 4.1.

Claim 78. Given an $\text{add(tup)}$ 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, $\text{newRecord} \rightarrow R$ during the execution). If the operation returned $\text{true}$ then there exists $j_1 < j < j_2$ for which $R$ is logically in the table at $C_j$.

Proof. The operation returned $\text{true}$ 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 $\text{add(tup)}$ 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 \(\text{tup}\), has been logically added into the table during the execution.

Given an unsuccessful \(\text{add}(\text{tup})\) operation, we want to set a linearization point which is between the invocation and the termination of the operation, for which a record representing \(\text{tup}\) cannot be logically inserted into the table. Formally, let \(\text{tup} = \langle e_1, \ldots, e_{N_T} \rangle\) 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 \(\text{add}(\text{tup})\) 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, \(\text{newRecord} \sim R\) 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 \(\text{InTable}\) at \(C_j\) and for which \(R'.data[i] = R.data[i]\).

**Proof.** 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 \(\text{InTable}\). 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 \(\text{add}(\text{tup})\) 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.

**remove** The definitions of successful and unsuccessful \(\text{remove}\) executions remain the same as in Section 4.1 as well.

Assume a successful \(\text{remove}(\text{val}, i)\) was invoked at \(s_{j_1}\) and returned at \(s_{j_2}\). Let \(R\) be the record found in line 29 (meaning, \(\text{victim} \sim R\) 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 \(\text{Removed}\). The only thing we still need to show that the record’s status is indeed \(\text{InTable}\) 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 \(\text{victim} \sim R\) at \(C_j\). Then \(R\)’s status at \(C_{j-1}\) is \(\text{InTable}\).

**Proof.** When executing line 43, \(R\)’s status is \(\text{InTable}\). 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 \(\text{InTable}\).
Now, assume that an unsuccessful \textit{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 \textit{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 \(\text{pred} \mapsto R_1\) and \(\text{victim} \mapsto R_2\) at \(C_{j_2-1}\). In addition, let \(C_j\) be the linearization point of the \textit{find} call, invoked in line 29. Obviously, \(j_1 < j < j_2\).

\textbf{Claim 81.} There does not exist a record \(R\) which is logically in the table at \(C_j\) and for which \(R.data[i] = val\).

\textit{Proof.} 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 \textit{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 \textit{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 \(\text{pred}_m\) local variable and let \(R_0, R_1, \ldots, R_n\) be the records assigned into the \(\text{curr}_m\) 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 \textit{find} \((\text{victim.data}[m], m)\) 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 \(\text{curr}_m\), respectively \((j_1 < t_0 < t_1 < \ldots < t_n < j_2)\). Finally, let \(R\) be the record for which \(\text{victim} \mapsto R\).

\textbf{Claim 82.} There exists \(j_1 < j \leq t_n\) for which \(R\)'s status is \textit{Removed} and \(R\) is \(i\)-reachable at \(C_j\).

\textit{Proof.} 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 \textit{InTable} at \(C_{j'}\). By lemma 71, \(R\) is both \(i\)-reachable and \(m\)-reachable at \(C_{j'}\).  

77
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 $R_{t-1}$ and $R_0$ are $m$-reachable and at $C_{t_0}$. In addition, by claim 68, $R_{t-1}.data[m] < R.data[m] \leq R_0.data[m]$. Since $R$ is also $m$-reachable at $C_{t_0}$, by claim 59 and lemma 72, it must $m$-follow $R_{t-1}$ at $C_{t_0}$. Since $R \neq R_0$ and $R_0$ is $R_{t-1}$’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, ..., $n$ 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 $j'', j'' + 1, ..., t_n$ 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 $\leadsto R$ 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.

**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 $\text{find}(val,i)$ 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, $\text{pred} \rightarrow R_0$ and $\text{curr} \rightarrow R_1$). 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 $\text{find}$ call in line 52, during the final retry iteration. In addition, let $R_{l_1}, \ldots, R_{l_n}$ be the records assigned into $\text{curr}$ 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. 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 $\text{InTable}$. 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 $\text{InTable}$ 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 3.6, 3.7, 3.8 and 3.9 is a linearizable implementation of a lock-based table.

### 4.2.3 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 $R_0.locks[i_0] <_l R_1.locks[i_1]$ if either $i_0 < i_1$, or $i_0 = i_1$ and $R_0 <_{i_0} R_1$. Notice that since $<_l$ is a constant order (i.e., if $R <_{i_1} R'$ then it will never reach a state where $R' <_{i_1} R$), $<_l$ 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 < $j_1 < j_2$ ≤ $k$).

**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.** 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, $i_1 = i_2$, $pred_i \leadsto R_1$ and $newRecord \leadsto R_2$. 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, $R_1 <_{i_1} R_2$ 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. 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, $i_1 = i_2$, $pred_i \leadsto R_1$ and $victim \leadsto R_2$. Since $R_1$ is the first output parameter, returned from the $\text{find}(\text{victim.data}[i_1], i_1)$ 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. Notice that locks are only acquired in lines 55 and 59. In addition, it must hold that $i_1 = i_2$. If $pred \leadsto R_1$ 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 $R_2$’s $i_1$-th locks are acquired in line 59. If $R_1$ is the second output parameter, returned from the $\text{find}$ 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 $\text{curr}$ 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 $\text{curr}$ 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 $\text{curr}$ 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 $R_{k_t}$’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 3.6, 3.7, 3.8 and 3.9 is a deadlock-free implementation of a lock-based table.
Chapter 5

Measurements

We compared the performance of our lock-free and lock-based table against each other, and against a global lock-based table (using coarse-grained locking) and an STM table (using software transactional memory as the synchronization tool). All implementations use the skip-list optimization of Section 3.4.

![Throughput graphs](image)

Figure 5.1: Throughput graphs. The x-axis is the number of threads. The y-axis is the throughput in $10^6$ operations per second.

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 two experiments, unique fields were selected uniformly at random from the range $[0,255]$, and in the last two experiments, 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 two experiments, and
from the range \([0, 25 \cdot 10^4 - 1]\) for the last two experiments. The table was initiated
with 128 records in the first two experiments and with \(5 \cdot 10^5\) records in the latter
two experiments. The graphs present the average throughput (operations executed per
millisecond) over all trials. We used two workloads distributions: one with 90% retrieve,
5% add and 5% remove operations, and the other with 50% retrieve, 25% add and 25%
remove operations. Missing points in Figure 5.1 represent inefficient executions that did
not complete within an hour.

We ran our experiments on a machine with 4 AMD Opteron(TM) 6376 2.3GHz
processors, each with 16-core processors (64 threads overall). All implementations
are in Java. We used Java’s\AtomicMarkableReference for the lock-free variant and
Java’s\ConcurrentSkipListSet (e.g. [JCSL]) for the skip-list optimization. We used
TL2 [DSS06] for our STM table. The TL2 synchronization was automatically added
by Deuce [KSF10]. We also ran an experiment with a medium key range of \([0, 10^4 - 1]\).
The results were very similar to the results with the large range (presented in Figure
5.1), and were therefore omitted.

**Results** As shown in Figure 5.1, for a small range of fields (when contention is high),
our lock-free table is far-superior to the other variants and scales well. For large fields
ranges, our lock-free and lock-based variants present a similar performance, and are far
superior to the other two alternatives.
Chapter 6

Related Work

Many sophisticated concurrent data structures (e.g., see [BCP16, BP12, EFR10, KP11, PT14, TBK12]) 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 [S12]. 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 [MHG10, SPEC]).

Many (relational) database systems have built-in support for multiple primary and secondary indexes [RG03, WV01]. Typically, database indexes can be correctly used in the presence of concurrency (e.g., by using transactions [WV01]). 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 [RG03] and snapshot [GBH15], 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 [ACD, ALS10, CD10, K10, KG06, TWF16]. 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 simple in-memory concurrency control, and it is not clear how to convert them into scalable 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 [G15, SK16]. However, in contrast to our specialized table implementations, general transactions provide limited performance due to their high runtime overhead [CBM08, G15], and their excessive thread contention [SK16].
Chapter 7

Conclusions

We proposed designs for a linearizable concurrent table with items that can be accessed quickly via multiple keys. Lock-free and lock-based designs were implemented and measured. On one hand, the proposed design is interesting practically, by offering a scalable and efficient solution to a natural and desirable functionality. On the other hand, this study brings up challenges in designing a concurrent multi-index table, such as the problem of designing a lock-free retrieve operation for the lock-free implementation and the problem of avoiding deadlocks in the fine-grained lock-based implementation. Measurements show that lock-free solutions perform better when contention is high, as well as in most practical scenarios.
Bibliography


Java concurrent skip list. https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentSkipListMap.html.


פעולות. עבור הגריסא המושתמאת במנועיים, מובחן שלא נוצר מצב של קפוא, לולמו, או המ탄ה.

מענחות מלשואניות.

עבורを目指ות סדרתיות של טבלאות, הפתרון הוא פ.viewmodel. נינת התהליךشبهמס교회, אך עבורה
כל אינדקס של הטבלה. בדרכו, נינת לאזור כל אייבר בקובץ עד פאינוודסיה כל אינדקס
הפתרון עבורה טבלת מקבליות אינוי פнтואו. יש להשתתף עם הזדמנד עם התוכנית איבר לכל אינדקסים
לא הפרוגראמה תהיה שליטה על נייגב כים שיניבים שיניבים שליטה תוכנה
יחד עם, על כל התוכנית עם הנקמה של אייבר שלIpAddress להתקל.
מותב הקויש שבחנה התוכנית
הרעיון פאודי, חפיה אייבר על פאינוודסיה שיניבים שליטה התוכנה, שיכן מייצר התוכנה
ל럴ובב, יש לדאוג שהאיזון מושגים הלברנברקר איבר מסדר, על מנטה להשתתף בברגרים קשר לזכר
הטבלה, יש לדאוג שהאיזון מושגים הלברנברקר על אטוארכו קהל מסדר, המבשל, עלייה התוכנה
 contratoados, מוחצף משני.

יש לעש בל סיוווסים במעניין זה הלברנברקר איננום מובחן בכון ביביניו לינארביולית, או פונג, ביבינו
מאפרור ניצולים מוטיבי של כל המוניצד. כל ניצולים פתיה תויווס שיניבים עבורה מבני התוכנית הסייר, נייעו
וא כלל המרכזים שונים עדביoungeול, על מנטה לייצר טבלת מקבליות של פאינוודסיה שיניבים,
אכן מגלת את כל המוניצד הגיע מכוני.
תקציר

כיום, ישנם מכם רוב המחשבים בתוכם מחשבים, אותם מחשבים ממקבילים ואבזורים שיוכנים באופן ממוחשב. מחשבים ממקבילים מת닮ים יותר ממוחשבים בשתי סיבות: הראשונה, כל כר מ אינם מאמנים,聞く נוכחי, במעבדה, והשנייה, כללי נוכחיים במעבדה.?

השימוש באלגוריתמים ממקבילים瓷 שלי אבטחתם. אלגוריתמים ממקבילים מאמנים, על פי המחשבים, הולכים ומגלים, ולא מאמנים. אלגוריתמים ממקבילים מאמנים עליונית את המחשבים, על פיהם נוכחים, ואできないים במעבדה.?

לדוגמה, ואת האיבר הוה נוכחי. את האיבר הוה נוכחי, להציגה, ולא מאמנים, על פיהם נוכחים, ואできないים במעבדה.?

לשם מבנה, מספר עליונית, מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספרعدد מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר bieten מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספרמספר מספר מספר מספר מספר מספרמספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספרמספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספרמספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספר מספרמספר מספר מספר מספר מספר מספר מספר מספר מספר מספרמספר מספר מספר מספר מספרמספר מספר מספר מספר מספר מספר מספרמספר מספר מספרמספר מספרמספר מספרמספר מספרמספר מספרמספר מספרמספר מספרמספר מספרמספר מספרמספר מספרמספר מספרמספר מספרמספר מספרמספר מספרמספר מספרמספר מספרמספר מספרמספר מספרמספר מספרמספר מספרמספר מספרמספר מספרמספר מספרמספר מספרמספר מספרמספר מספרמספר מספרמספר מספרמספרמספרמספר מספרמספרמספרמספר מספרמספרמספרמספר מספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספר מספרמספר מספר מספרמספר מספרמספר מספרמספר מספרמספר מספרמספר מספרמספר מספרמספר מספרמספר מספרמספר מספרמספר מספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספרמספר nhiêu_remaining_numbers
המחקר בוצע בהנחייתו של פרופסור ארז פטרן, בכקלולה למדעי המחשב.

הכותרתinnessררה לטלניוק על ממון מחקר זה.
טבלת מקבילים ומרובות אינדקסים

ть_by
לשם مليי חלקי של הדירישטים לけどלת התוצר
מqueteルלאידע במודיע המורת

גל שפי

רותם למלט ערכנו – מכון טכנולוגי לישראל
נכס התשע”ז חמה אפריל 2017
טבלת מקבילה ומורבות אינדקסים

גל שפי