Imperial College London
Department of Computing

Reasoning about Concurrent Indexes

by

Pedro da Rocha Pinto

Submitted in partial fulfilment of the requirements for the
MSc Degree in Advanced Computing of Imperial College London

September 2010

Abstract

Index data structures such as B+ trees and hash tables are used widely as the core of databases and file systems. In a world where concurrent systems are common as servers and largely increasing as personal computers, we need a way to make sure concurrent implementations do not contain bugs. We use the fact that multiple implementations share a common high-level specification and use concurrent abstract predicates to prove that an implementation satisfies such a specification. This approach allows us to reuse the abstract specification and enables high-level reasoning independent from the low-level implementation.

Acknowledgements

First, I thank my supervisor Philippa Gardner, for her continuous support during my project, for believing in me and creating a good environment with very intelligent people to work with. Without her encouragement and constant guidance, I could not have finished this project.

I also thank Mark Wheelhouse for the constant encouragement and guidance and serving as my second supervisor. For always being there to meet and explore the ideas and help me solve what seemed unsolvable. For explaining to me concurrent abstract predicates among many other things. For reviewing my work and proofreading it on very short notice and diligently. For asking me hard questions and showing interest in what I was doing.

I am also greatly indebted to Mike Dodds, from Cambridge University, who demonstrated an avid interest in my project, specially about the abstract specifications. For the quick feedback, insightful questions and interesting discussions on how to improve what was being developed. Mike helped me understand some technical details about concurrent abstract predicates which were very important.

I would like to thank Thomas Dinsdale-Young for being ready to help me and discuss ideas and suggest reading material I was not aware of. To Gian Dzik for all the discussions about concurrency and separation logic, about the relationship of indexes and file systems.

Finally, I would like to thank my family and my friends for the unconditional support and encouragement to pursue my interests and believing in me. I also would like to thank to everyone who was I forgot to mention that contributed with suggestions to this project.

Contents

Chapter 1

Introduction

Concurrency has always been notoriously difficult for programming and formal methods. Traditionally, coarse-grained algorithms were used: a sequential data structure is made concurrent by using a single mutual exclusion lock which ensures that only one operation is accessing the data structure at any time. This approach allows simple reasoning, however, it negates some of the benefits of concurrent systems.

In recent years, fine-grained algorithms started to be widely used with the increase of multicore systems. These algorithms allow threads to update different parts of the data structure in parallel. This requires synchronisation techniques such as locking schemes, or non-blocking designs involving atomic instructions such as compare-and-swap to maintain a coherent shared state. Fine-grained algorithms are highly error-prone and hard to debug, yet the advantages in efficiency have made them widely used. We want to provide a formal system to reason about these fine-grained algorithms.

Originally, we were interested in reasoning about a specific set of fine grained concurrent algorithms for accessing a B^{Link} tree [27]. The B^{Link} tree is a widely used data structure, in particular for databases and file systems. However, for the scope of this project we want to provide a more abstract view of the data structure. We realised that we could be more general since a B^{Link} tree is just an instance of a concurrent index. Hence, we wanted to specify the index abstractly at the high-level without considering specific implementations. We show that some specific implementations (node list, B^{Link} tree and a hash table) correctly implement this specification (i.e. the behaviour is consistent with the high-level view).

We chose to work with concurrent abstract predicates [9] because RGSep [29] is too restrictive and carries the details of the internal implementation in the high-level specifications. However, we discovered that concurrent abstract predicates were not powerful enough to provide a specification that was capable of reasoning about all the programs we wanted to verify. In particular there were many intuitive properties that we wanted to specify which concurrent abstract predicates could not express. We extended the high-level specification with permissions and later we created interference environments in order to achieve our desired expressivity. We ended with a high-level specification that allowed us to reason about all programs in an intuitive way.

1.1 Contributions and project outline

The main contributions of this project are the introduction of abstract specifications for concurrent indexes and the introduction of a new action model for reasoning about index implementations.

The abstract specifications for concurrent indexes presented enable simple, modular, high-level proofs without dealing with the details of the implementation. They allow us to prove any reasonable use of a concurrent index and to describe its contents where possible and also prove safety properties.

Here is the outline of the project:

- In chapter 2 we provide a brief overview of a B^{Link} tree structure and the logics which are used throughout this project (separation logic, rely/guarantee, RGSep, concurrent abstract

predicates). We give a brief description of our previous work [25], which reasons about a B^{Link} tree implementation using RGSep. We also complete the insert operation proof omitted from [25], in appendix B.

- In chapter 3 we provide an abstract specification of a concurrent index. We start with specifications using concurrent abstract predicates similar in style to [9]. However, we found that we could not express many intuitive properties of our concurrent index mode. We discuss these limitations and provide the solutions that allow us to express our intuitions formally. In particular we made use of a permissions model and extended the predicates of [9].

- In chapter 4 we reason about several low-level implementations of concurrent indexes. We then show that these implementations are correct with respect to the high-level specifications provided in chapter 3. We first provide a simplified node list implementation which maintains the complexity of the internal details of the action model we wish to explain. We then show how this can be extended to provide a B^{Link} tree implementation. Finally, we show how we can provide a hash table implementation of the concurrent index which internally makes use of our node list implementation.

- In chapter 5 we evaluate and discuss our results and the advantages and disadvantages of our approach.

Chapter 2

Technical background

We start by looking at a concurrent index implementation, the B^{Link} tree, which we will use throughout this work. We provide an informal introduction to separation logic and rely/guarantee reasoning. Then we present RGSep, which combines separation logic with rely/guarantee reasoning. We then show the main logic used in this work, concurrent abstract predicates. Finally, we look at other available related logics.

2.1 B^{Link} tree

The B^{Link} tree [19, 27] is a concurrent B+ tree [1, 8]. It is an ordered, n-ary branching search tree which supports three basic concurrent operations, search, insertion and deletion. Like a B+ tree, the pointers to data are only stored at the fringe of the tree in the leaf nodes. All nodes in the tree at the same depth are connected, forming a null-terminated single-linked list as shown by Figure 2.1. When there are only readers (threads that only search for data), we do not need

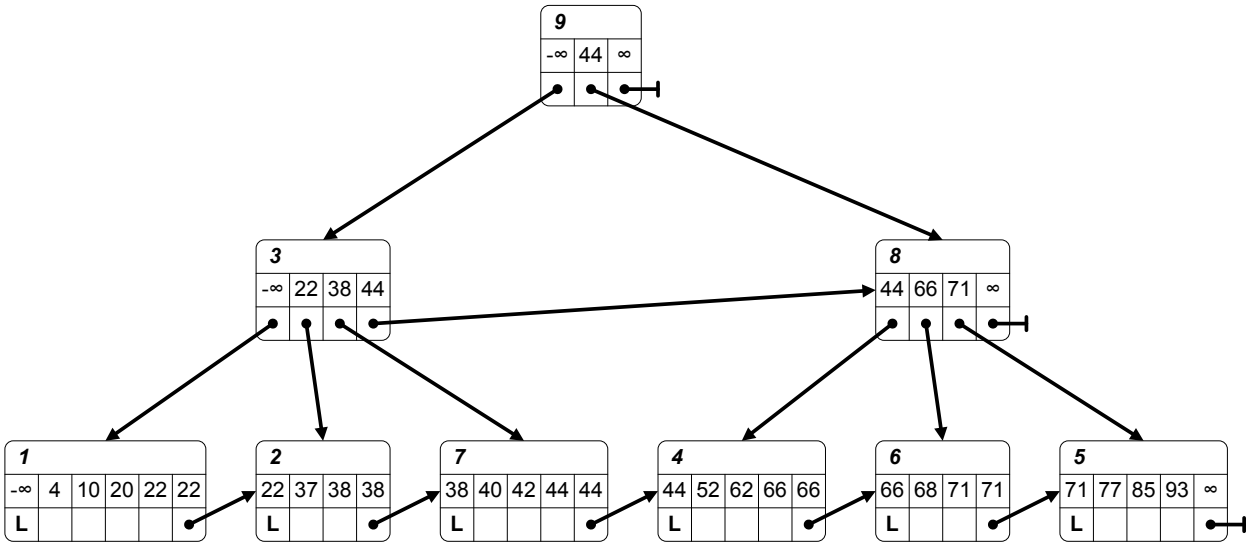

Figure 2.1: A B^{Link} tree.

to consider concurrent techniques to reason about trees. However, when there are also threads performing insertions and deletions (which change the tree), this is no longer trivial.

There are many variants of the B^{Link} tree. Some propose top-down algorithms while others bottom-up. In top-down solutions [18, 28, 11] an algorithm which will update the tree (insertion or deletion) locks the subtree which it will affect. Some of these solutions [2] allow the presence of threads which do not change the tree. In bottom-up solutions [11, 19, 20, 27] an algorithm which updates the tree is similar to a search algorithm on the way down (i.e. the algorithm moves from the root until it reaches a leaf node). On the way up (i.e. when the algorithm starts at a leaf node and moves to higher level nodes) it performs changes to keep the tree balanced and locks

one [19, 27] up to four nodes [11]. The locks ensure that the changes are made atomically and deal with race conditions when two threads move up on the same path. The bottom-up solutions have a higher degree of concurrency when two threads updating the tree have overlapping affecting subtrees. The bottom-up solutions also usually require locking fewer nodes simultaneously.

When choosing the variant of the B^{Link} tree about which to reason the decision was made to use the algorithms presented by Sagiv[1] [27], which provide a bottom-up solution. This variant allows a high degree of concurrency and only locks at most one node with the exception of a corner case in insertion, which requires two locks.

The deletion differs from the traditional B+ tree in the sense that it does not trigger further updates at higher levels of the tree, it simply deletes the value at the leaf node. This means that a leaf node can be less than half full, unlike in a normal B+ tree. This problem is addressed by Sagiv's compression algorithm, but involves changing the initial algorithms, so we will ignore this for now. We start describing the tree by looking at the structure of the nodes located at the fringe,

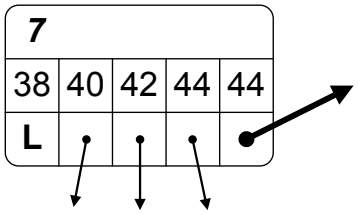

Figure 2.2: A leaf node.

known as leaf nodes, one of which is shown in Figure 2.2. A leaf node is located at a memory position in the heap (in this case the address is 7). Each list has an ordered set of key-value pairs, a minimum and maximum value (being 38 and 44 in the example) such that the ordered set of key-value pairs does not contain a key less than or equal to the minimum value and larger than the maximum value. Finally the leaf has a pointer to its right sibling, called a link. If the leaf is the rightmost node then it will be null. The leaf node contains the actual values stored in the tree. We consider all nodes which are not leaf nodes as intermediate nodes (one is shown in Figure 2.3).

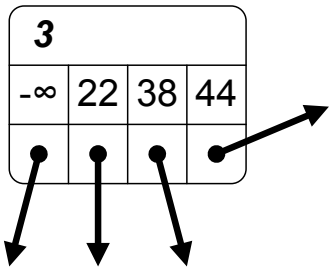

Figure 2.3: An intermediate node.

Like the leaf node they contain a set of key-value pairs, they have a minimum and maximum value, a link pointer to the right sibling at the same level and an address. The main difference is that it also has a pointer associated with the minimum value which with the rest of the set of key-value pairs, all point to nodes at the lower depth in the tree. These are called the children of the node. We assume that node accesses are atomic and that to read a node no lock is required.

The search operation looks through the tree trying to find the data entry associated with a key value. As a consequence of the structure of the tree, if the data entry exists in the tree, it will be located in a leaf node where the the key value associated with the data is greater than the minimum value of that leaf node and less than or equal to its maximum value. Since the search operation only performs readings it acquires no locks.

[1] B^{Link} trees are said to be a variant of B* trees in [27], but this is not the case. They are in fact a variant of a B+ tree. B* trees are B+ trees except each node is required to always be at least 2/3 full, instead of 1/2 full like B+ trees.

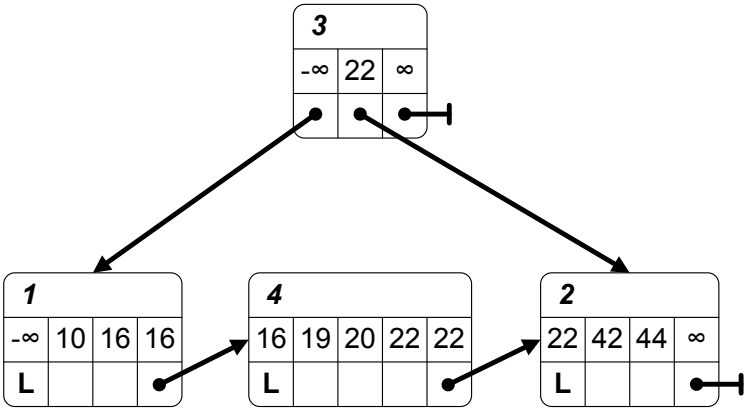

Figure 2.4: A broken B^{Link} tree.

The insert operation, like the search operation, looks through the tree trying to find the node which should contain the data entry associated with a key value. If the data does not exist in the node then it will insert it. Since the node might be full (i.e. the ordered set has the maximum allowed number of elements in a node), then it must be split into two. When this happens, the tree will be broken until a new reference to the new node is added to the higher level (figure 2.4 shows an example of a possible tree after an insertion), it then moves to a higher level of the tree in order to correct the tree as a consequence of the new node. It repeats this until the tree is no longer broken (search operation still works). Finally, it returns true if it was able to insert or false if the value already existed at the leaf node.

The delete operation is very similar to search. Conceptually it too searches for the leaf node which could contain the data entry to delete. When reaching that node, if the key associated with the data entry does not exist, it returns false, since it cannot delete something which does not exist at that time in the tree. Otherwise it removes the value-pointer pair from the leaf node and returns true.

Until now we have seen an incomplete data structure. The complete version includes a block called a prime block, which contains references to every leftmost node in the tree. This block has a constant address unlike the root. Therefore it allows every thread to know where to start their algorithms by reading the prime block and reading the first address. The tree has several invariants:

- A node minimum value is always constant.
- A thread at a leaf node can always reach a leaf node with a higher minimum value if one exists.

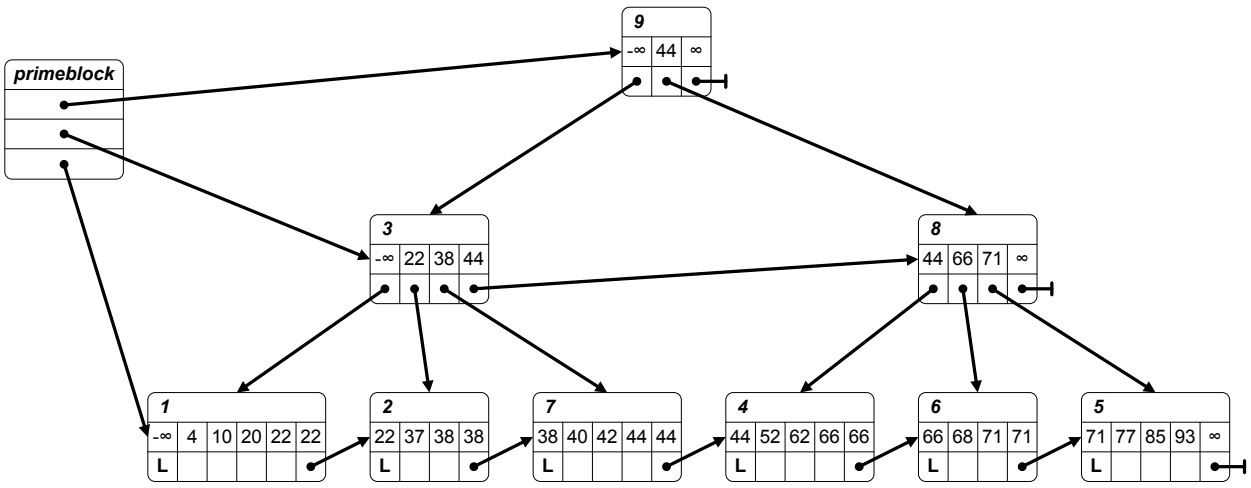

Figure 2.5: A full B^{Link} tree.

- A leftmost node has a minimum value of $-\infty$.
- A rightmost node has a maximum value of $+\infty$.
- If a node has a right sibling, then the right sibling minimum value is the same as the maximum value of the node.
- There always exists a root node.

The first invariant is the most important and implies that the nodes always have the correct minimum values for their children, i.e. the set of nodes which can be accessed directly at the lower level of the tree from the node. The next most important invariant is the second and implies that a thread can always find the intended value-pointer pair as long as the desired value is larger than the minimum value of the current node.

2.2 Separation logic

Separation logic [26, 15, 3] deals with the problem of how to reason about mutable data structures. It is based on the logic of bunched implications (BI) [22]. We use the RAM model here, which is often considered to be the basic model, to provide a basic introduction to separation logic.

Separation logic is expressed by Hoare triples in the form $\{P\}\ C\ \{Q\}$. A triple has a fault avoiding partial correctness interpretation: if a state satisfies the assertion P then when we run command C on that state, then either the command diverges (does not terminate) or it results in a state that satisfies Q. The command does not fault.

Separation logic differs from Hoare logic [14], in the sense that the assertions describe changes to just part of the memory (local reasoning). Its specifications describe only their footprint, i.e. the changes made by a command. Everything that is not described by the specification will be unchanged.

This new approach makes use of a new logical connective called the separating conjunction $*$. An assertion $P * Q$ is interpreted as the state being split into two parts, one described by P and other by Q. Separating conjunction formally captures the essence of local reasoning with the following rule:

$$\frac{\vdash_{SL} \{P\}\ C\ \{Q\} \quad \mathrm{mod}(C) \cap \mathrm{free}(R) = \emptyset}{\vdash_{SL} \{P * R\}\ C\ \{Q * R\}} \quad \text{(SL-Frame)}$$

This rule says that if the assertion P is separate from assertion R and the command C only affects P, then if the command C finishes Q will remain separate from assertion R. The side condition states that nothing mentioned in R is affected by C.

Assertions in separation logic that describe the heap are given by the following grammar:

$$P, Q ::= false \mid emp \mid e = e' \mid e \mapsto e' \mid \exists x\ .\ P \mid P \Rightarrow Q \mid P * Q \mid P \mathrel{-\!\!\circledast} Q$$

The connectives $\neg$, $\wedge$, $\vee$, the quantifier $\forall$, and $true$ behave in the classical way. emp stands for the empty heap. The assertion $e \mapsto e'$ states that the heap cell with address e has contents e'. The separation conjunction, $P * Q$, an important operator for separation logic, states that if a heap satisfies it, then it can be split in two parts, one satisfying P and the other satisfying Q. The other new connective is septraction, $P \mathrel{-\!\!\circledast} Q$, which represents removing P from Q. The heap can be extended with a state satisfying P and the extended heap satisfies Q. We write an underscore $\_$ in the place of an expression whose value we do not care about. Formally, we have $x \mapsto \_ \stackrel{\text{def}}{=} \exists y\ .\ x \mapsto y$. The separating conjunction has an iterative version defined as follows:

$$\circledast_{i=1}^{n} P_i \stackrel{\text{def}}{=} P_1 * \ldots * P_n$$

We now present the axioms for the stack and heap model of separation logic. Known as small axioms, they describe the change of a command in the smallest heap. We use the SL-Frame rule

when there is a larger heap, since the rest of the heap remains unchanged. We have commands to allocate, lookup, mutate and deallocate memory in the heap.

$$\begin{array}{lll} \{emp\} & \mathbf{x} := \mathbf{cons}(e) & \{\mathbf{x} \mapsto e\} \\ \{e \mapsto e'\} & \mathbf{x} := [e] & \{e \mapsto e' \wedge \mathbf{x} = e'\} \\ \{e = \_\} & [e] := e' & \{e \mapsto e'\} \\ \{e \mapsto \_\} & \mathbf{dispose}(e) & \{emp\} \end{array}$$

There are several semantically defined classes of assertions that possess useful properties. We will define them now since they will be used later on.

Definition 1 (Pure assertions)**.** *An assertion is pure if and only if it does not specify the heap, only the interpretation of logical variables.*

Definition 2 (Intuitionistic assertions)**.** *An assertion is intuitionistic if and only if it specifies a lower bound on the heap, i.e. if a heap satisfies an assertion P, then any larger heap also satisfies P.*

Definition 3 (Exact assertions)**.** *An assertion is exact if and only if it specifies exactly one resource.*

Definition 4 (Precise assertions)**.** *An assertion is precise if and only if there exists for all heaps at most one subheap that satisfies the assertion.*

Finally, here is a rule for disjoint concurrency:

$$\frac{\vdash_{SL} \{P_1\}\ C_1\ \{Q_1\} \quad \vdash_{SL} \{P_2\}\ C_2\ \{Q_2\} \quad \begin{array}{l} \mathrm{mod}(C1) \cap \mathrm{free}(P2, Q2) = \emptyset \\ \mathrm{mod}(C2) \cap \mathrm{free}(P1, Q1) = \emptyset \end{array}}{\vdash_{SL} \{P_1 * P_2\}\ C_1 \| C_2\ \{Q_1 * Q_2\}} \quad \text{(SL-Par)}$$

This rule states that if two threads need disjoint resources to execute, they can be executed safely in parallel. If they terminate, each will own resources disjoint from each other. Hence, the postcondition is the composition of the two thread postconditions. The side conditions states that nothing mentioned in either threads specification is affected by the other thread.

A simple example of this is a parallel composition of two heap alternations on different cells:

$$\begin{array}{c} \{\ x \mapsto 3 * y \mapsto 3\ \} \\ \{\ x \mapsto 3\ \} \quad \| \quad \{\ y \mapsto 3\ \} \\ x := 4 \quad \| \quad y := 5 \\ \{\ x \mapsto 4\ \} \quad \| \quad \{\ y \mapsto 5\ \} \\ \{\ x \mapsto 4 * y \mapsto 5\ \} \end{array}$$

The $*$ in the precondition guarantees that x and y are not aliases.

We have only presented a brief outline of separation logic. There are many extensions to it based on other models, such as heap based on pointer arithmetic [23], heaps with permissions [4], variable as resource [5, 24] and an abstract separation logic [7] which allows reasoning about abstract local functions over abstract resource models. There also exists an extension to deal with resource sharing called concurrent separation logic [21] which we will explain briefly.

2.2.1 Concurrent separation logic

Separation logic does not permit sharing of resources among threads. Hence, we cannot reason about concurrent programs involving inter-thread communication.

Concurrent separation logic [21] overcomes this limitation by introducing resource invariants. The proof rules have the form $J \vdash_{SL} \{P\}\ C\ \{Q\}$, where J is a precise separation logic assertion representing an invariant that is true about the program separately from the precondition and postcondition. J holds at all times during the execution of the program except when a thread is inside an atomic block.

We have the following rule for atomic commands, which grants threads temporary access to the invariant, J, within an atomic command:

$$\frac{emp \vdash_{SL} \{P * J\}\ C\ \{Q * J\} \quad J \text{ is precise}}{J \vdash_{SL} \{P\}\ \langle C \rangle\ \{Q\}} \quad \text{(SL-Atomic)}$$

The following rule allows us to take some local state, R, and treat it as shared state for the duration of the command C:

$$\frac{J * R \vdash_{SL} \{P\}\ C\ \{Q\} \quad R \text{ is precise}}{J \vdash_{SL} \{P * R\}\ \langle C \rangle\ \{Q * R\}} \quad \text{(SL-ResFrame)}$$

2.3 Rely/guarantee

Rely/guarantee is a method introduced by Jones [16, 17] which enables post-hoc verification of concurrent algorithms. It allows us to describe interference by having two relations, the rely R and the guarantee G. The former describes the effects on the shared state which the program can tolerate, and the latter describes the effects on the shared state by the program itself.

The specifications in rely/guarantee are given by four components (P, R, G, Q). P stands for the precondition and Q for the postcondition. Both describe the whole behaviour of one thread. The precondition P asserts the conditions for which it makes sense to run a program, while the postcondition Q relates the initial state (before starting the program) to the final state (at the conclusion of the program). The postcondition asserts the overall changes to the state made by the program. The rely R and the guarantee G describe the atomic actions made by the environment and the program respectively. They relate the shared state changes before and after each atomic action. One can see R as the interference that the program tolerates from the environment and G as the interference the program induces.

The precondition P and postcondition Q must be stable under the rely R. This means that P and Q should not be invalidated by any action done by the environment. This allows us to compose threads in parallel as long we make sure that each thread's guarantee is implied by the rely condition of the other threads.

The specification of the interference is global, i.e. it must be checked against every state update. This is a limitation towards finding a satisfactory compositional approach for reasoning about concurrent problems using rely/guarantee.

2.4 RGSep

RGSep [29] subsumes separation logic and rely/guarantee, combining their advantages without some of their weaknesses. It assumes that the state is composed of two disjoint parts: the local state accessible by a single thread and the shared state accessible by all threads. It uses separation logic to deal with the local state and rely/guarantee to handle the shared state.

2.4.1 Local and shared state assertions

We start by formally defining the syntax of assertions in RGSep with the following grammar:

$p, q ::=$	P	Local assertion
	$\boxed{P}$	Shared assertion
	$p * q$	Separating conjunction
	$p \wedge q$	Normal conjunction
	$p \vee q$	Disjunction
	$\forall x\ .\ p$	Universal quantification
	$\exists x\ .\ p$	Existential quantification

Where P is an assertion in separation logic that describes the local state. $\boxed{P}$ describes a shared assertion also using separation logic, this is called a boxed assertion. The separating conjunction

is multiplicative over the local state and additive over the shared state. This means that the separation conjunction splits the local state, while at the shared state we have $\boxed{P} * \boxed{Q}$ if and only if $\boxed{P \wedge Q}$.

2.4.2 Describing interference

RGSep describes interference in terms of actions in the form $P \rightsquigarrow Q$. Actions describe changes on the shared state. We interpret an action $P \rightsquigarrow Q$ as "the part of shared state that is satisfied by P before the action will be replaced by a part satisfied by Q". The remaining part of the shared state remains unchanged. Consider the following action:

$$x \mapsto M \rightsquigarrow x \mapsto N \wedge N \geq M$$

The action describes that the heap cell with address x may be changed but never decremented. The precondition and the postcondition have shared logical variables, in this example M and N, and are assumed to be existentially bound. Like in separation logic, we describe the smallest specification possible and use the frame rule when there is a larger state.

RGSep rely and guarantee conditions are represented as a set of actions. The relational semantics of a set of actions are the reflexive and transitive closure of the union of the semantics of each action in the set. This means that each action is allowed to run an arbitrary number of times respecting the other actions.

$$[\![P_1 \rightsquigarrow Q_1, \ldots, P_n \rightsquigarrow Q_n]\!] = \left(\bigcup_{i=1}^{n} [\![P_i \rightsquigarrow Q_i]\!] \right)^*$$

As a sanity condition, we require that the assertions in the action are precise (see Definition 4). A guarantee allows a specification $P \rightsquigarrow Q$ if its effect is contained in the guarantee itself.

Definition 5. *$(P \rightsquigarrow Q) \subseteq G$ if and only if $[\![P \rightsquigarrow Q]\!] \subseteq [\![G]\!]$*

2.4.3 Stability of assertions

When constructing a proof in rely/guarantee we require that every precondition and postcondition is stable under environment interference. An assertion S is stable under interference of a relation R if and only if when the assertion S holds at the beginning and after making an update satisfying R, the state still satisfies S.

Definition 6 (Stability)**.** *$S; R \Rightarrow S$ if and only if for all states s_1 and s_2 such that assertion S is satisfied at state s_1 and $(s_1, s_2) \in R$, the assertion S is satisfied at state s_2.*

Since RGSep represents the interference R as a set of actions, stability is reduced to a simple syntactic check. For a single action $[\![P \rightsquigarrow Q]\!]$, the following separation logic implication is necessary and sufficient:

Lemma 7 (Checking stability)**.**

$$S; [\![P \rightsquigarrow Q]\!] \Rightarrow S \text{ if and only if } \models_{SL} (P \mathrel{-\!\!\circledast} S) * Q \Rightarrow S.$$

$$S; (R_1 \cup R_2)^* \Rightarrow S \text{ if and only if } S; R_1 \Rightarrow S \text{ and } S; R_2 \Rightarrow S.$$

Informally, the first property states that from a state satisfying S, if we remove the part of the state satisfying P and replace it with a state satisfying Q, the result should satisfy S. If there is no sub-state of S satisfying P, the action cannot run making $P \mathrel{-\!\!\circledast} S$ false and the implication will hold. The second property states that an assertion S is stable under interference of a set of actions R when it is stable under interference of every action in R.

RGSep allows interference on the shared state, but not on the local state. Therefore, the interference only affects the parts that describe the shared state in an RGSep assertion. We say that an assertion in RGSep is (syntactically) stable under R if all of its boxed assertions are stable under R.

2.4.4 Proof rules

Specifications of a command C are quadruples (p, R, G, q), where p is a precondition which sets the conditions where C can be executed for the local and shared state, R is a rely relation, given by a set of actions which describe the interference caused by the environment, G is a guarantee which describes the interference caused by the program on the shared state and q is a postcondition which asserts the local and shared state if the command C terminates.

The judgement $\vdash C$ **sat** (p, R, G, q) means that during any execution of C at an initial state satisfying p and under environment interference R, the program does not fault, causes interference at most G and if it terminates, satisfies q in its final state.

RGSep inherits the frame rule from separation logic: if a program runs safely with initial state p it can also run with additional state r. Since the program runs safely without r, it cannot access the additional state, hence r is still true at the end. Since the frame, r, may also specify the shared state, the frame rule checks that r is stable under interference from both the program and its environment. Otherwise, they may invalidate r during their execution. In the case when r does not mention the shared state, the stability check is trivially satisfied.

$$\frac{\vdash C \textbf{ sat } (p, R, G, q) \quad r \text{ stable under } (R \cup G)}{\vdash C \textbf{ sat } (p * r, R, G, q * r)} \quad (\textsc{Frame})$$

It is valid to prove a stronger specification than the required one, we can weaken the specification by either strengthening the precondition or weakening the postcondition. We use the following rule:

$$\frac{\vdash C \textbf{ sat } (p', R', G', q') \quad R \subseteq R' \quad G' \subseteq G \quad p \Rightarrow p' \quad q' \Rightarrow q}{\vdash C \textbf{ sat } (p, R, G, q)} \quad (\textsc{Weaken})$$

If command c does not interfere with the shared state then it can be expressed with the same rules as separation logic, that is:

$$\frac{\vdash_{\text{SL}} \{P\}\ c\ \{Q\}}{\vdash c \textbf{ sat } (P, \emptyset, \emptyset, Q)} \quad (\textsc{Prim})$$

It is worth noting that RGSep is a sound logic. However, since it is not in the scope of this work to present the full details of the logic, we do not present a proof here. Vafeiadis presents a soundness proof and an extensive list of proof rules to RGSep [29, 30].

2.5 Concurrent Abstract Predicates

We now describe informally concurrent abstract predicates [9] as we have introduced the majority of the concepts in the previous sections. The core idea is to define an abstract specification for a concurrent module and to be able to prove that a concrete implementation satisfies this specification. We introduce the concept of concurrent abstract predicates by giving an example specification and implementation of a lock module. This simple example (from [9]) demonstrates how we reason about shared resources.

2.5.1 Lock specification

A lock module has functions **lock**(x) and **unlock**(x), for acquiring and releasing a lock respectively. It also contains a function to create a new lock, we call it **makelock**(n). This allocates a lock followed by a contiguous block of memory of size n. We specify these functions as follows:

$$\begin{array}{lll} \{isLock(x)\} & \textbf{lock}(\text{x}) & \{isLock(x) * Locked(x)\} \\ \{Locked(x)\} & \textbf{unlock}(\text{x}) & \{emp\} \\ \{emp\} & \textbf{makelock}(\text{n}) & \left\{ \begin{array}{l} \exists .ret = x \wedge isLock(x) * Locked(x) \\ * (x+1) \mapsto \_ * \ldots * (x+n) \mapsto \_ \end{array} \right\} \end{array}$$

This abstract specification, which is presented by the module to the client, is independent of the underlying implementation. The assertions $isLock(x)$ and $Locked(x)$ are abstract predicates. The

predicate $isLock(x)$ asserts that x is a lock which can be acquired by the thread which has this assertion. The predicate $Locked(x)$ asserts that the current thread holds the lock. The connective $*$ is the separating conjunction from separation logic.

The abstract predicates satisfy the following axioms, which are also presented to the client:

$$\begin{array}{rcl} isLock(x) & \Leftrightarrow & isLock(x) * isLock(x) \\ Locked(x) * Locked(x) & \Leftrightarrow & false \end{array}$$

The first axioms allows a client to freely share the knowledge that x is a lock. The second axiom implies that the lock x can only be locked once.

2.5.2 Implementation

We now consider the following compare-and-swap lock implementation:

```
lock(x) {
  local b;
  do
    ⟨b := !CAS(&x, 0, 1)⟩
  while(b)
}

unlock(x) {
  ⟨[x] := 0⟩
}

makelock(n) {
  local x := alloc(n + 1);
  [x] := 1;
  return x;
}
```

Here the lock function is implemented by calling a compare-and-swap (CAS) action on the locks value. The compare-and-swap action atomically compares the contents of a memory location to a given value and, only if they are the same, modifies the contents of that memory location to a given new value.

We relate the lock implementation to our lock specification by providing a concrete interpretation of the abstract predicates. The abstract predicates are interpreted as assertions about the internal interference of the module as well as the internal state of the module.

2.5.3 Describing interference

To describe this internal interference we extend separation logic with two assertions.

The shared region assertion $\boxed{P}^r_{I(\overrightarrow{x})}$ specifies that there is a shared region of memory, identified by label r, and that region satisfies P. The shared state is indivisible, therefore all threads maintain a consistent view of it. The possible actions on the shared state are declared by the environment $I(\overrightarrow{x})$.

The other new assertion is the permission assertion$[A]^r_\pi$. It specifies that the thread has permission π to perform the action A in the region r, provided the action is in that region's environment. The permission π is a fractional permission. When $0 < \pi < 1$ both the thread and the environment can do the action. When $\pi = 1$ only the current thread can do the action.

Now, we can give concrete interpretations of our lock predicates:

$$\begin{array}{rcl} isLock(x) & \equiv & \exists r, \pi \,.\, [LOCK]^r_\pi * \boxed{(x \mapsto 0 * [UNLOCK]^r_1) \vee x \mapsto 1}^r_{I(r,x)} \\ Locked(x) & \equiv & \exists r \,.\, [UNLOCK]^r_1 * \boxed{x \mapsto 1}^r_{I(r,x)} \end{array}$$

The abstract predicate $isLock(x)$ is interpreted by the concrete, implementation-specific assertion on the right-hand side. This specifies that the thread has permission to acquire the lock as the permission $[LOCK]^r_\pi$ is in its local state. The assertion also specifies that the shared region satisfies the module's invariant: either the lock is unlocked and the region holds the full permission $[UNLOCK]^r_1$ to unlock the lock, or the lock is locked and the unlocking permission is gone (it is in some other thread's local state).

The abstract predicate $Locked(x)$ specifies that the permission assertion $[UNLOCK]^r_1$ is in the current thread's local state. This means that the current thread has full permission to unlock the lock in region r. It also states that the lock is locked in the shared region assertion.

The actions allowed on the lock's shared region are declared in $I(r, x)$. Actions describe the interference of the current thread and the environment on the shared state. They have the form $A : P \rightsquigarrow Q$ and are similar to actions in RGSep. The actions for the lock module are:

$$I(r,x) \stackrel{\text{def}}{=} \left(\begin{array}{rl} LOCK: & x \mapsto 0 * [UNLOCK]_1^r \rightsquigarrow x \mapsto 1, \\ UNLOCK: & x \mapsto 1 \rightsquigarrow x \mapsto 0 * [UNLOCK]_1^r \end{array} \right)$$

The $LOCK$ action requires that the full permission to unlock the lock $[UNLOCK]_1^r$ is in the shared region and also that lock is not already unlocked. By performing the action it locks the lock and moves the full unlock permission into the thread's local state. Meanwhile, the $UNLOCK$ action requires that the lock is currently locked and that the current thread holds the full permission to unlock the lock $[UNLOCK]_1^r$. The result of the action is to unlock the lock and move the $[UNLOCK]_1^r$ permission from the thread's local state back into the shared state. Note that the actions only describe the effects at the shared state as in RGSep.

2.5.4 Stability of assertions

Soundness of the concurrent abstract predicates system requires that the abstract predicates are self-stable with respect to the actions. This means that for every action permitted by the module, each predicate must remain true. Self-stability ensures that a client can use these predicates without having to consider the module's internal interference.

We say that an assertion is stable if and only if it cannot be falsified by the interference from other threads that it permits. Similarly, we say that a predicate environment is stable if and only if all the predicates it defines are stable.

2.5.5 Proof system

Judgements in concurrent abstract predicates have the form $\Delta; \Gamma \vdash \{P\}\ C\ \{Q\}$, where Δ contains the predicate definitions and axioms, Γ contains the abstract specifications, and the local Hoare triple $\{P\}\ C\ \{Q\}$ has the standard fault-avoiding partial correctness interpretation as in separation logic.

We will omit the proofs rules as they are mainly, simple modifications of the RGSep proof rules.

Finally, we will not show here that the lock implementation satisfies the abstract specification (proof available in [9]).

2.6 Other proof methods

Lately there have been two proposed logics, Separated Assume-Guarantee Logic (SAGL) [13] and Local Rely-Guarantee Reasoning (LRG) [12], also based on separation logic and rely/guarantee. RGSep and SAGL partition the memory into shared and private parts, while LRG does not.

The main difference between RGSep and SAGL is their purpose. RGSep subsumes both separation logic and rely/guarantee while SAGL is a logic to reason about assembly code, it was not meant to deal with different abstraction levels. It does not subsume separation logic and it does not have the standard version of the frame rule.

Comparing RGSep with LRG, they have many things in common, since the latter was based on the former. However LRG differs deeply in the way it models the program state. It does not have a physical partition of private and shared resources at the program states, which makes the assertion language more complex. On the other hand, it introduces a hiding rule in the logic, which allows one to hide locally shared resources from global specifications and to support sharing of dynamically created resources. Also, it does not require specifying private and shared resources separately like in RGSep (with boxed assertions for shared resources), or SAGL which uses two separation logic assertions. LRG uses just separation logic assertions, but requires an asserter to interpret the boundary, which neither RGSep or SAGL do.

There have been other approaches, one of them is deny guarantee [10]. It is a reformulation of rely/guarantee to reason about dynamically scoped concurrency. It also uses separation logic, to

allow interference to be dynamically split and recombined. We have deny and guarantee permissions instead of rely and guarantee. The deny permission defines what the environment cannot do, while the guarantee specifies what a thread can do. It can encode all of the original rely-guarantee proofs.

2.7 Related work

In related work [25], we have reasoned about B^{Link} trees using RGSep. We have formalised the data structure using separation logic. Rather than thinking of it as a tree structure, we used invariants and knowledge from linked lists to decrease its complexity. We have modelled the changes of the shared state by a set of actions using rely/guarantee. We have used the acquired knowledge to provide formal proofs of correctness to the algorithms using RGSep. However, we have only provided a partial proof to the insertion algorithm. We have now completed the proof and we present it in appendix B.

In that work we have worked with low-level specifications for each operation. Each specification contained the rely and the guarantee which specified the interference on the shared state of the thread performing the operations and the other threads which could be running concurrently. Despite our success in doing such correctness proofs, we wished to create a more general specification which did not depend heavily on the internal details of the implementation, in this case, the B^{Link} tree. Meanwhile, concurrent abstract predicates [9] were being developed and partially solved the problem described before, though, with a major limitation. It required an abstract disjointness of resources which was not very realistic when applied in real problems. In this work we solve both problems and generalise the specifications to concurrent indexes rather than just a single implementation of the B^{Link} tree.

Chapter 3

Specifications for concurrent indexes

In this chapter, we try to give an answer to the question of what makes a good specification from the user point of view. We provide an initial solution using concurrent abstract predicates and then extend it using permissions. We generalise the specification to support different types of interference. We evaluate the specification and try to solve the problems in a generalised model.

3.1 Overview

Before we have reasoned about B^{Link} tree operations in respect to low-level specifications [25]. These specifications were defined in RGSep style, and so contained preconditions and postconditions associated with each operation, a rely and a guarantee set. The rely and the guarantee of each operation defined the interference in the data structure which could occur while the operation occurred. These specifications have two major limitations. The first is that the rely and guarantee sets are given at the specification level, yet they deal with specific details of the implementation and should not be shown to a user of the data structure. The other major limitation is the assumption that no concurrent changes were being made to the same key value, in order to avoid race conditions while reasoning. In practise, the data structure was designed to allow this behaviour and the user should be aware that two threads could be concurrently inserting (or deleting) a pointer in the tree to the same key value. This concurrent insertion/deletion notion only exists at the high-level, since at the implementation level there are critical regions which deal with the race conditions. We will address both problems by providing abstract specifications (i.e. high-level specifications), which can then be used to reason at a high-level about operations in the B^{Link} tree.

A B^{Link} tree is manipulated by a specific set of operations, which provide a way to locate, insert and remove pointers associated with a key value. This set of operations is not exclusive to a B^{Link} tree or a B+ tree, in fact, both data structures belong to a class of structures which we will call an index. Other possible (but not exclusive) implementations of an index are balanced trees or hash tables.

Using an abstract specification, we should be able to reason about the index abstract data type regardless of its implementation. This means that we should not care about the internal interference that occurs at the low-level. This interference not only changes according to each implementation, but should also be hidden from a user perspective. To achieve this kind of abstraction we will use concurrent abstract predicates [9].

3.1.1 Index

There are no standard conventions for defining an abstract data type. We will use the same abstract concepts as the ones used for the B^{Link} tree. We assume an index h is a mapping from key values to pointers (or nothing).

$$h : Values \rightarrow Pointers \cup \{Nothing\}$$

The index h is manipulated by three operations:

$r :=$ SEARCH(h, v)	if there is a mapping from value v to pointer p in index h, then it assigns to r the value p or NULL otherwise.
$r :=$ INSERT(h, v, p)	if there is a mapping from value v to some pointer q in index h, then it assigns to r the value FALSE. Otherwise it creates a mapping from value v to pointer p, and assigns to r the value TRUE.
$r :=$ DELETE(h, v)	if there is a mapping from value v to some pointer p in index h, then it removes it and assigns to r the value TRUE. Otherwise it assigns to r the value FALSE.

Each command depends on a finite subset of the domain, therefore we can talk about partial indexes. This allows us to make specifications which do not require knowledge of the whole index, making the preconditions and postconditions smaller. A partial index denotes the information for a finite set of keys. We can combine partial indexes exactly when they have a different set of value keys.

Since we are working with concurrent indexes, the descriptions given above might not appear to be true all the time. Assume a user is inserting a pointer in the index, while another is deleting it. The first user might believe that there is a mapping in the index for the key value he just inserted, but that might not be the case since the second user may have immediately deleted it. We will see how to provide a formal specification which can handle every possible computation and take into account concurrent manipulations at the high level.

3.1.2 Objectives

When designing a specification we will aim to achieve the following properties:

- Locality - A specification should be local and as small as possible. It should describe the minimum knowledge to run an operation. This captures the programmers intuition when using an operation.

- Abstract - A specification should be independent from a particular implementation. It should not contain details such as the internal interference of each operation and it should be general enough in order to be applied to the majority of the index implementations.

- Formal - A specification should not be described in an ambiguous way, otherwise different implementations assume different high-level behaviour of the same operation. This would confuse users as there would be different interpretations of the specification leading to unexpected results. To avoid this problem a specification must be described in a formal fashion.

- Reusability - It is not desirable to create a new specification every time we want to reason about a concurrent program. Indexes are widely used and they share common features which we can abstract in a common specification. Since there is a huge variety of concurrent indexes, we will try to cover the most common implementations and one should be able to use our specification as a guideline for creating a specification for an uncommon case.

- Completeness - A specification should allow us to reason about safety of an operation and, where possible, to reason about the contents and return values of the operations that manipulate the index concurrently. There are times, due to race conditions, where we cannot determine the contents of the index. However, we should be able to say that the computation does not fault. In the case where there are no race conditions, we should be able to prove precisely what are the return values of each operation and their behaviour. If there are race conditions that affect the computation partially (meaning that we might lose the knowledge about the contents of the index at some point in time), we might be able to determine precisely the end result. If that is the case, our specification should allow us to reason with the same results.

- Soundness - A specification must not allow us to prove incorrect results. It can allow weaker results depending on how we use it, but never wrong ones.

3.2 Concurrent Abstract Predicates

We will use the ideas of concurrent abstract predicates [9] as an initial approach to provide formal specifications of the three operations which manipulate the index. Our specifications are based on the idea that the abstract specification should be independent of the underlying implementation. We also note that an index, like a set, can be described as a set of disjoint elements despite being contained in a single shared data structure.

3.2.1 Predicates

Considering the descriptions of each operation given before, and adapting the set specification presented in [9], we have the following predicates:

$in(h, v, p)$	there is a mapping in the index h from v to p.
$out(h, v)$	there is no mapping in the index h from v.

We define $own(h, v)$ as the disjunction of these two predicates, that is:

$$own(h, v) \stackrel{\text{def}}{=} in(h, v, \_) \vee out(h, v)$$

These assertions do not only capture knowledge about an index, but also exclusive permission to alter that index by changing its contents.

Finally we use $\_$ to describe an unknown pointer value. Note that it is not syntactic sugar to $in(h, v, \_) \equiv \exists p \,.\, in(h, v, p)$. The reason we make $\_$ part of our syntax is to ensure that each predicate is stable by itself, a condition which is required for the soundness of the proof system.

3.2.2 Axioms

The exclusivity of permissions is captured by the module's axiom:

$$own(h, v) * own(h, v) \Longrightarrow false$$

3.2.3 Specifications

We specify the module's commands using the following local Hoare triples:

$$\begin{array}{lll}
\{\ in(h,v,p)\ \} & r := \text{SEARCH}(h, v) & \{\ in(h,v,p) \wedge r = p\ \} \\
\{\ out(h,v)\ \} & r := \text{SEARCH}(h, v) & \{\ out(h,v) \wedge r = null\ \} \\
\{\ in(h,v,q)\ \} & r := \text{INSERT}(h, v, p) & \{\ in(h,v,q) \wedge r = false\ \} \\
\{\ out(h,v)\ \} & r := \text{INSERT}(h, v, p) & \{\ in(h,v,p) \wedge r = true\ \} \\
\{\ in(h,v,p)\ \} & r := \text{DELETE}(h, v) & \{\ out(h,v) \wedge r = true\ \} \\
\{\ out(h,v)\ \} & r := \text{DELETE}(h, v) & \{\ out(h,v) \wedge r = false\ \}
\end{array}$$

3.2.4 Reasoning

Since the abstract predicates denote information for a finite set of keys, we can combine them when they state different sets of keys. We can reason disjointly about index predicates, even though they may be implemented by a single shared structured.

Consider the simple program which performs a sequential search for key value v_2 and stores the associated pointer. Then, concurrently, it inserts the pointer for the key value v_1 and deletes the

key v_2. We now have no pointer associated with the key value v_2 and there is a pointer associated with the key value v_1 at the index. Formally, we have the following program and specifications:

$$\begin{array}{c} \{\ out(h, v_1) * in(h, v_2, p)\ \} \\ r := \text{SEARCH}(h, v_2); \\ r_1 := \text{INSERT}(h, v_1, r) \quad \| \quad r_2 := \text{DELETE}(h, v_2) \\ \{\ in(h, v_1, p) * out(h, v_2)\ \} \end{array}$$

Using the specifications we can prove the correctness of the program as following:

$$\begin{array}{c} \{\ out(h, v_1) * in(h, v_2, p)\ \} \\ r := \text{SEARCH}(h, v_2); \\ \{\ out(h, v_1) * in(h, v_2, p) \wedge r = p\ \} \\ \begin{array}{cc} \{\ out(h, v_1) \wedge r = p\ \} & \{\ in(h, v_2, p)\ \} \\ r_1 := \text{INSERT}(h, v_1, r) & r_2 := \text{DELETE}(h, v_2) \\ \{\ in(h, v_2, p) \wedge r_1 = true\ \} & \{\ out(h, v_2) \wedge r_2 = true\ \} \end{array} \\ \{\ in(h, v_1, p) \wedge r_1 = true * out(h, v_2) \wedge r_2 = true\ \} \\ \{\ in(h, v_1, p) * out(h, v_2)\ \} \end{array}$$

Note that we will not cover the details of creating an actual index and just assume that when it happens, all predicates assume no values in the index.

3.3 Adding permissions

The specification given before does not allow reasoning about concurrent search operations on the same key because they assume a disjoint model of concurrency. In practise, in order to reason about concurrent search operations, we want to allow two threads to search for the same value as this was already possible when doing a low-level proof with RGSep. This means that we have to allow two threads to share a predicate between them in order to search. We can achieve that by using permissions as presented by Boyland [6].

3.3.1 Predicates

We extend our predicates with a permission i where $0 < i \leq 1$. We have the following predicates which extend the previous ones:

$in(h, v, p, i)$	there is a mapping in the index h from v to p with permission i.
$out(h, v, i)$	there is no mapping in the index h from v with permission i.

3.3.2 Axioms

The i component is used to record the splitting of the predicate. We can split both the predicates $in(h, v, p, i)$ and $out(h, v, i)$ by splitting the i permission. This is captured by the new module axioms, which are as follows:

$$\begin{array}{rcl} in(h, v, p, i) * in(h, v, p, j) & \iff & in(h, v, p, i + j) \\ out(h, v, i) * out(h, v, j) & \iff & out(h, v, i + j) \\ own(h, v, i) * own(h, v, j) \wedge i + j > 1 & \implies & false \end{array}$$

3.3.3 Specifications

We want to allow concurrent searches but only one thread inserting or deleting a specific mapping for a key value at the index. Therefore a thread is required to hold a predicate $in(h, v, p, i)$ or $out(h, v, i)$ to search, but it must hold $in(h, v, p, 1)$ or $out(h, v, 1)$ (i.e. total permission) to perform an insertion or deletion. This ensures that search operations on the same key value can be in parallel while insertions and deletions on the same key value must be sequential.

The index operations have the following specifications:

$$
\begin{array}{ccc}
\{\ in(h,v,p,i)\ \} & r := \text{SEARCH}(h,v) & \{\ in(h,v,p,i) \wedge r = p\ \} \\
\{\ out(h,v,i)\ \} & r := \text{SEARCH}(h,v) & \{\ out(h,v,i) \wedge r = null\ \} \\
\{\ in(h,v,q,1)\ \} & r := \text{INSERT}(h,v,p) & \{\ in(h,v,q,1) \wedge r = false\ \} \\
\{\ out(h,v,1)\ \} & r := \text{INSERT}(h,v,p) & \{\ in(h,v,p,1) \wedge r = true\ \} \\
\{\ in(h,v,p,1)\ \} & r := \text{DELETE}(h,v) & \{\ out(h,v,1) \wedge r = true\ \} \\
\{\ out(h,v,1)\ \} & r := \text{DELETE}(h,v) & \{\ out(h,v,1) \wedge r = false\ \}
\end{array}
$$

3.3.4 Reasoning

We can now reason about concurrent searches using fractional permissions. Consider the following program:

$$
\begin{array}{c}
\{\ in(h,v,p,1)\ \} \\
r_1 := \text{DELETE}(h,v) \\
r_2 := \text{SEARCH}(h,v) \quad \| \quad r_3 := \text{SEARCH}(h,v) \\
r_4 := \text{INSERT}(h,v,q) \\
\{\ in(h,v,q,1)\ \}
\end{array}
$$

We now have concurrent search operations which should not affect the contents of the index. Using our specifications with fractional permissions we can prove the correctness of the program as follows:

$$
\begin{array}{c}
\{\ in(h,v,p,1)\ \} \\
r_1 := \text{DELETE}(h,v) \\
\{\ out(h,v,1) \wedge r_1 = true\ \} \\
\{\ out(h,v,1/2) \wedge out(h,v,1/2) \wedge r_1 = true\ \} \\
\left\{\begin{array}{l} out(h,v,1/2) \\ \wedge\, r_1 = true \end{array}\right\} \qquad \| \qquad \{\ out(h,v,1/2)\ \} \\
r_2 := \text{SEARCH}(h,v) \qquad\qquad r_3 := \text{SEARCH}(h,v) \\
\left\{\begin{array}{l} out(h,v,1/2) \\ \wedge\, r_1 = true \wedge r_2 = null \end{array}\right\} \qquad \{\ out(h,v,1/2) \wedge r_3 = null\ \} \\
\left\{\begin{array}{l} out(h,v,1/2) \wedge r_1 = true \wedge r_2 = null \\ *\, out(h,v,1/2) \wedge r_3 = null \end{array}\right\} \\
\{\ out(h,v,1) \wedge r_1 = true \wedge r_2 = null \wedge r_3 = null\ \} \\
r_4 := \text{INSERT}(h,v,q) \\
\{\ in(h,v,q,1) \wedge r_1 = true \wedge r_2 = null \wedge r_3 = null \wedge r_4 = true\ \} \\
\{\ in(h,v,q,1)\ \}
\end{array}
$$

We split the $out(h,v,1)$ predicate into $out(h,v,1/2) * out(h,v,1/2)$ allowing us to prove the concurrent search operations. We then merge the predicates into a single predicate to regain full permission and continue the proof.

These specifications model the assumption that we can have multiple readers but only one writer. In practise the underlying implementations of index support multiple writers and multiple readers and the specifications should reflect this.

3.4 Generalising the specification

The main problem with our previous specifications is that they restrict the interaction between threads to avoid any race conditions. In practise, from a high-level perspective, we want to allow concurrent changes in the index, even if we cannot infer the resulting state of the index. We want to be able to capture this idea and reason about it using abstract specifications. To do that we need to understand the limitations of the previous specifications and how can we overcome them.

Our previous specifications assumed the common notion that we can have multiple readers, but only one writer for each key value in the index. This guarantees that we always know the outcome of the computation, since there are no race conditions. We want to be able to reason about

manipulations in the index as precisely as possible, meaning that when no race conditions occur we wish to maintain the full knowledge about the data structure contents. When race conditions do occur, we have to analyse the possibilities.

When one thread is modifying a specific mapping of a value in the index, while an arbitrary number of threads are just searching for it, we cannot know what will be the outcome of the search operations, as the pointer correspondent to the value may or may not be there. The outcome will depend on the interaction with the thread which is modifying the index concurrently. However, we can know the final contents of the index, as they depend exclusively on the modifying thread. We can actually know the returning value of the same operation too.

We have another case, which occurs when several threads are inserting (but not deleting) and others are searching for the same key value. In this case, we cannot know if each operation will succeed individually, but we know that the combination of all the threads will have created a mapping to some pointer in the index. We can consider an analogous situation but with threads deleting.

There is one last case, where any number of threads can be searching, inserting or deleting the same key value. Here we lose all information about that specific key in the index. However we should be able to reason that the execution of all operations concurrently does indeed work (no faults occur) and is a valid computation.

3.4.1 Predicates

In order to capture all the previous cases and still maintain local specifications we create predicates which instead of expressing only if there is a mapping to a key value or not also express implicitly the amount of interference in the shared state correspondent to a specific key value. We need the following set of abstract predicates, where h, v and p have the same meaning as before in 3.3.1:

$in_{def}(h, v, p, i)$	the value is definitely assigned.
$out_{def}(h, v, i)$	the value is definitely not assigned.
$in_{one}(h, v, p, i)$	the value might be assigned.
$out_{one}(h, v, i)$	the value might not be assigned.
$in_{ins}(h, v, p, i)$	the value is assigned and there are no deletions in the interference environment.
$out_{ins}(h, v, i)$	the value is not assigned and there are no deletions in the interference environment.
$in_{del}(h, v, p, i)$	the value is assigned and there are no insertions in the interference environment.
$out_{del}(h, v, i)$	the value is not assigned and there are no insertions in the interference environment.
$unk(h, v, i)$	nothing is known about the value in the index.

We use i and j as permission fractions with $0 < i, j \leq 1$.

We have five classes of predicates and each class describes some information about the environment. The predicates $in_{def}(h, v, p, i)$ and $out_{def}(h, v, p, i)$ cover the cases where there are no race conditions. Either there is one thread changing or multiple threads reading. The predicates $in_{one}(h, v, p, i)$ and $out_{one}(h, v, i)$ describe that there is at most one thread changing while multiple threads are reading. The predicates $in_{ins}(h, v, p, i)$ and $out_{ins}(h, v, i)$ describe that there are multiple threads inserting and multiple threads reading. Analogously, $in_{del}(h, v, p, i)$ and $out_{del}(h, v, i)$ describe the same but threads can only delete instead of inserting. Finally, $unk(h, v, i)$ describes that there can be multiple threads changing and reading the state. This predicate is useful if we want to prove safety of a program or to describe unknown outcomes of the computation.

We use k to identify the permission with fraction $1/2 < k \leq 1$ which is used with the predicates $in_{one}(h, v, p, i)$ and $out_{one}(h, v, i)$. The permission k is a relaxed version of the full permission, since it assures there is only one thread altering the shared state, but at the same time, it allows other threads to perform reads.

3.4.2 Axioms

We now present the axioms of our system. Some have the usual meaning, such as splitting permissions. Some axioms describe how to switch between each kind of predicate. To do that we require the total permission. This makes sure that no threads are changing a particular key value in the index. With total permission we can also make any predicate weaker by stating that we do not know the state of the index for a key value.

We have the following axioms for our predicates:

$$
\begin{aligned}
in_{def}(h,v,p,1) &\iff in_{one}(h,v,p,1)\\
out_{def}(h,v,1) &\iff out_{one}(h,v,1)\\
in_{def}(h,v,p,1) &\iff in_{ins}(h,v,p,1)\\
out_{def}(h,v,1) &\iff out_{ins}(h,v,1)\\
in_{def}(h,v,p,1) &\iff in_{del}(h,v,p,1)\\
out_{def}(h,v,1) &\iff out_{del}(h,v,1)\\
in_{def}(h,v,p,i) * in_{def}(h,v,p,j) &\iff in_{def}(h,v,p,i+j)\\
out_{def}(h,v,i) * out_{def}(h,v,j) &\iff out_{def}(h,v,i+j)\\
in_{def}(h,v,p,1) &\implies unk(h,v,1)\\
out_{def}(h,v,1) &\implies unk(h,v,1)\\
in_{one}(h,v,p,i) * in_{one}(h,v,p,j) &\iff in_{one}(h,v,p,i+j)\\
out_{one}(h,v,i) * out_{one}(h,v,j) &\iff out_{one}(h,v,i+j)\\
in_{one}(x,v,p,k) * in_{one}(h,v,q,i) &\iff in_{one}(h,v,\_,k+i)\\
in_{one}(x,v,p,k) * out_{one}(h,v,i) &\iff in_{one}(h,v,p,k+i)\\
out_{one}(x,v,k) * in_{one}(h,v,p,i) &\iff out_{one}(h,v,k+i)\\
in_{one}(h,v,p,1) &\implies unk(h,v,1)\\
out_{one}(h,v,1) &\implies unk(h,v,1)\\
in_{ins}(h,v,p,i) * in_{ins}(h,v,p,j) &\iff in_{ins}(h,v,p,i+j)\\
out_{ins}(h,v,i) * out_{ins}(h,v,j) &\iff out_{ins}(h,v,i+j)\\
in_{ins}(x,v,p,i) * in_{ins}(h,v,q,j) &\implies in_{ins}(h,v,\_,i+j)\\
in_{ins}(x,v,p,i) * out_{ins}(h,v,j) &\implies in_{ins}(h,v,p,i+j)\\
in_{ins}(h,v,p,1) &\implies unk(h,v,1)\\
out_{ins}(h,v,1) &\implies unk(h,v,1)\\
in_{del}(h,v,p,i) * in_{del}(h,v,p,j) &\iff in_{del}(h,v,p,i+j)\\
out_{del}(h,v,i) * out_{del}(h,v,j) &\iff out_{del}(h,v,i+j)\\
out_{del}(x,v,i) * in_{del}(h,v,p,j) &\implies out_{del}(h,v,i+j)\\
in_{del}(h,v,p,1) &\implies unk(h,v,1)\\
out_{del}(h,v,1) &\implies unk(h,v,1)\\
unk(h,v,i) * unk(h,v,j) &\iff unk(h,v,i+j)
\end{aligned}
$$

Most of these cases are simple. We will focus on the interesting cases.

The axiom $in_{one}(x,v,p,k) * out_{one}(h,v,i) \iff in_{one}(h,v,p,k+i)$ describes the splitting of a predicate into one that allows a thread to modify the index, requiring permission k and other which only allows searching, which due to the environment behaviour, has an unknown outcome.

The axiom $in_{ins}(x,v,p,i) * in_{ins}(h,v,q,j) \implies in_{ins}(h,v,\_,i+j)$ describes that two threads have performed an insert operation and that the outcome has an unknown value, but we are sure that there is a mapping to the key value in the index.

The axiom $in_{ins}(x,v,p,i) * out_{ins}(h,v,j) \implies in_{ins}(h,v,p,i+j)$ describes that one thread has inserted a value in the index while others possibly searched for it, the value must be in the index and therefore we can combine both predicates. There are analogous axioms for the environment where only deletions and searches can occur.

Finally, note that once we move to an $unk(h,v,i)$ predicate we are stuck there.

3.4.3 Specifications

We have the following specifications for our commands:

$$
\begin{array}{lll}
\{\ in_{def}(h,v,p,i)\ \} & r := \text{SEARCH}(h,v) & \{\ in_{def}(h,v,p,i) \wedge r = p\ \} \\
\{\ out_{def}(h,v,i)\ \} & r := \text{SEARCH}(h,v) & \{\ out_{def}(h,v,i) \wedge r = null\ \} \\
\{\ in_{def}(h,v,q,i)\ \} & r := \text{INSERT}(h,v,p) & \{\ in_{def}(h,v,q,i) \wedge r = false\ \} \\
\{\ out_{def}(h,v,1)\ \} & r := \text{INSERT}(h,v,p) & \{\ in_{def}(h,v,p,1) \wedge r = true\ \} \\
\{\ in_{def}(h,v,p,1)\ \} & r := \text{DELETE}(h,v) & \{\ out_{def}(h,v,1) \wedge r = true\ \} \\
\{\ out_{def}(h,v,i)\ \} & r := \text{DELETE}(h,v) & \{\ out_{def}(h,v,i) \wedge r = false\ \} \\
\{\ in_{one}(h,v,p,k)\ \} & r := \text{SEARCH}(h,v) & \{\ in_{one}(h,v,p,k) \wedge r = p\ \} \\
\{\ out_{one}(h,v,k)\ \} & r := \text{SEARCH}(h,v) & \{\ out_{one}(h,v,k) \wedge r = null\ \} \\
\{\ in_{one}(h,v,p,i)\ \} & r := \text{SEARCH}(h,v) & \left\{ \begin{array}{l} (in_{one}(h,v,p,i) \wedge r = \_) \\ \vee \\ (out_{one}(h,v,i) \wedge r = null) \end{array} \right\} \\
\{\ out_{one}(h,v,i)\ \} & r := \text{SEARCH}(h,v) & \left\{ \begin{array}{l} (out_{one}(h,v,i) \wedge r = null) \\ \vee \\ (in_{one}(h,v,\_,i) \wedge r = \_) \end{array} \right\} \\
\{\ in_{one}(h,v,q,k)\ \} & r := \text{INSERT}(h,v,p) & \{\ in_{one}(h,v,q,k) \wedge r = false\ \} \\
\{\ out_{one}(h,v,k)\ \} & r := \text{INSERT}(h,v,p) & \{\ in_{one}(h,v,p,k) \wedge r = true\ \} \\
\{\ in_{one}(h,v,p,k)\ \} & r := \text{DELETE}(h,v) & \{\ out_{one}(h,v,k) \wedge r = true\ \} \\
\{\ out_{one}(h,v,k)\ \} & r := \text{DELETE}(h,v) & \{\ out_{one}(h,v,k) \wedge r = false\ \} \\
\{\ in_{ins}(h,v,p,i)\ \} & r := \text{SEARCH}(h,v) & \{\ in_{ins}(h,v,p,i) \wedge r = p\ \} \\
\{\ out_{ins}(h,v,i)\ \} & r := \text{SEARCH}(h,v) & \left\{ \begin{array}{l} (out_{ins}(h,v,i) \wedge r = null) \\ \vee \\ (in_{ins}(h,v,\_,i) \wedge r = \_) \end{array} \right\} \\
\{\ in_{ins}(h,v,q,i)\ \} & r := \text{INSERT}(h,v,p) & \{\ in_{ins}(h,v,q,i) \wedge r = false\ \} \\
\{\ out_{ins}(h,v,i)\ \} & r := \text{INSERT}(h,v,p) & \{\ in_{ins}(h,v,\_,i) \wedge (r = true \vee r = false)\ \} \\
\{\ in_{del}(h,v,p,i)\ \} & r := \text{SEARCH}(h,v) & \left\{ \begin{array}{l} (in_{del}(h,v,p,i) \wedge r = p) \\ \vee \\ (out_{del}(h,v,i) \wedge r = null) \end{array} \right\} \\
\{\ out_{del}(h,v,i)\ \} & r := \text{SEARCH}(h,v) & \{\ out_{del}(h,v,i) \wedge r = null\ \} \\
\{\ in_{del}(h,v,p,i)\ \} & r := \text{DELETE}(h,v) & \{\ out_{del}(h,v,i) \wedge (r = true \vee r = false)\ \} \\
\{\ out_{del}(h,v,i)\ \} & r := \text{DELETE}(h,v) & \{\ out_{del}(h,v,i) \wedge r = false\ \} \\
\{\ unk(h,v,i)\ \} & r := \text{SEARCH}(h,v) & \{\ unk(h,v,i) \wedge (r = p \vee r = null)\ \} \\
\{\ unk(h,v,i)\ \} & r := \text{INSERT}(h,v,p) & \{\ unk(h,v,i) \wedge (r = true \vee r = false)\ \} \\
\{\ unk(h,v,i)\ \} & r := \text{DELETE}(h,v) & \{\ unk(h,v,i) \wedge (r = true \vee r = false)\ \}
\end{array}
$$

3.4.4 Reasoning

The examples shown before are still valid now, but use $in_{def}(h,v,p,i)$ and $out_{def}(h,v,i)$ instead. We will now demonstrate how to reason about programs which have race conditions.

Consider a program which performs a concurrent search and insert operation on the same key value as following:

$$
\begin{array}{c}
\{\ out_{def}(h,v,1)\ \} \\
r_1 := \text{SEARCH}(h,v) \quad \| \quad r_2 := \text{INSERT}(h,v,q) \\
\{\ in_{def}(h,v,1)\ \}
\end{array}
$$

The correctness proof for the program is as follows:

$$
\begin{array}{c}
\{\ out_{def}(h,v,1)\ \} \\
\{\ out_{one}(h,v,1)\ \} \\
\{\ out_{one}(h,v,1/4) * out_{one}(h,v,3/4)\ \} \\
\begin{array}{c|c|c}
\{\ out_{one}(h,v,1/4)\ \} & & \{\ out_{one}(h,v,3/4)\ \} \\
r_1 := \text{SEARCH}(h,v) & \| & r_2 := \text{INSERT}(h,v,q) \\
\left\{ \begin{array}{l} (in_{one}(h,v,\_,1/4) \wedge r_1 = \_) \\ \vee\, (out_{one}(h,v,1/4) \wedge r_1 = null) \end{array} \right\} & & \{\ in_{one}(h,v,q,3/4) \wedge r_2 = true\ \}
\end{array} \\
\left\{ \begin{array}{l} (in_{one}(h,v,\_,1/4) \wedge r_1 = \_) \\ \vee\, (out_{one}(h,v,1/4) \wedge r_1 = null) \\ * in_{one}(h,v,q,3/4) \wedge r_2 = true \end{array} \right\} \\
\{\ in_{one}(h,v,q,1) \wedge (r_1 = \_ \vee r_1 = null) \wedge r_2 = true\ \} \\
\{\ in_{def}(h,v,q,1) \wedge (r_1 = \_ \vee r_1 = null) \wedge r_2 = true\ \} \\
\{\ in_{def}(h,v,q,1)\ \}
\end{array}
$$

We have a race condition between both threads on key value v. Since one of them is writing and the other is reading, we weaken the predicate $out_{def}(h,v,1)$ to $out_{one}(h,v,1)$ which allows a single thread writing and any reader. We then split the predicate into $out_{one}(h,v,1/4) * out_{one}(h,v,3/4)$ because we require to keep one of the predicates with permission k (i.e. greater than $1/2$) in order to prove the insert operation. At the end of the concurrent execution, we merge the predicates into $in_{one}(h,v,q,1)$ since the predicate with permission k dominates the others, as expressed by our axioms. Finally we can make the predicate $in_{one}(h,v,q,1)$ into $in_{def}(h,v,q,1)$ as we have full permission.

Consider now a program which performs concurrent deletes on the same key value as following:

$$
\begin{array}{c}
\{\ in_{def}(h,v,p,1)\ \} \\
r_1 := \text{DELETE}(h,v) \quad \| \quad r_2 := \text{DELETE}(h,v) \\
\{\ out_{def}(h,v,1)\ \}
\end{array}
$$

The correctness proof for the program is as follows:

$$
\begin{array}{c}
\{\ in_{def}(h,v,p,1)\ \} \\
\{\ in_{del}(h,v,p,1)\ \} \\
\{\ in_{del}(h,v,p,1/2) * in_{del}(h,v,p,1/2)\ \} \\
\begin{array}{c|c|c}
\{\ in_{del}(h,v,p,1/2)\ \} & & \{\ in_{del}(h,v,p,1/2)\ \} \\
r_1 := \text{DELETE}(h,v) & \| & r_2 := \text{DELETE}(h,v) \\
\left\{ \begin{array}{l} out_{del}(h,v,1/2) \\ \wedge\, (r_1 = true \vee r_1 = false) \end{array} \right\} & & \left\{ \begin{array}{l} out_{del}(h,v,1/2) \\ \wedge\, (r_2 = true \vee r_2 = false) \end{array} \right\}
\end{array} \\
\{\ out_{del}(h,v,1/2) \wedge (r_1 = true \vee r_1 = false) * out_{del}(h,v,1/2) \wedge (r_2 = true \vee r_2 = false)\ \} \\
\{\ out_{del}(h,v,1) \wedge (r_1 = true \vee r_1 = false) \wedge (r_2 = true \vee r_2 = false)\ \} \\
\{\ out_{def}(h,v,1) \wedge (r_1 = true \vee r_1 = false) \wedge (r_2 = true \vee r_2 = false)\ \} \\
\{\ out_{def}(h,v,1)\ \}
\end{array}
$$

Again, we have a race condition between both threads on key value v. However in this case both threads are performing deletes on the same key value. We weaken the predicate $in_{def}(h,v,p,1)$ to $in_{del}(h,v,p,1)$ which allows a multiple threads deleting but none inserting. We then split the predicate into $in_{del}(h,v,p,1/2) * in_{del}(h,v,p,1/2)$ in order to prove the concurrent operations. At the end of the concurrent execution, we merge the predicates into $out_{del}(h,v,1)$ and make it into $out_{def}(h,v,1)$ as we have full permission. Note that we do not know which delete succeeds.

Consider the following program:

$$
\begin{array}{c}
\{\ in_{def}(h,v,p,1)\ \} \\
r_1 := \text{DELETE}(h,v) \quad \| \quad r_2 := \text{INSERT}(h,v,q) \\
\{\ unk(h,v,1)\ \}
\end{array}
$$

We cannot know the order of the delete and insert operations, so we must use the $unk(h, v, 1)$ predicate which forces an unknown postcondition. However we can still prove the safety of the program:

$$
\begin{array}{c}
\{\ in_{def}(h,v,p,1)\ \} \\
\{\ unk(h,v,1)\ \} \\
\{\ unk(h,v,1/2) * unk(h,v,1/2)\ \} \\
\begin{array}{c|c|c}
\{\ unk(h,v,1/2)\ \} & & \{\ unk(h,v,1/2)\ \} \\
r_1 := \text{DELETE}(h,v) & \| & r_2 := \text{INSERT}(h,v,q) \\
\left\{\begin{array}{l} unk(h,v,1/2) \\ \wedge (r_1 = true \vee r_1 = false) \end{array}\right\} & & \left\{\begin{array}{l} unk(h,v,1/2) \\ \wedge (r_2 = true \vee r_2 = false) \end{array}\right\}
\end{array} \\
\{\ unk(h,v,1/2) \wedge (r_1 = true \vee r_1 = false) * unk(h,v,1/2) \wedge (r_2 = true \vee r_2 = false)\ \} \\
\{\ unk(h,v,1) \wedge (r_1 = true \vee r_1 = false) \wedge (r_2 = true \vee r_2 = false)\ \} \\
\{\ unk(h,v,1)\ \}
\end{array}
$$

3.5 Limitations

Our specification covers the possible race conditions but it has some major limitations which we will discuss and understand how we can overcome them.

3.5.1 Interference environments

We have considered all possible race conditions, but our interference environments are too restrictive. Consider the following example:

$$
r_1 := \text{SEARCH}(h,v,r) \quad \Big\| \quad \begin{array}{c} r_2 := \text{INSERT}(h,v,p) \\ r_3 := \text{SEARCH}(h,v) \quad \| \quad r_4 := \text{SEARCH}(h,v) \\ r_5 := \text{DELETE}(h,v) \end{array}
$$

With our current specification and assuming we have as precondition $out_{def}(h, v, 1)$ we have a race condition between both threads on key v. Since one of them is writing and the other is reading, we can use the predicate $out_{one}(h, v, 1)$ and use that predicate to continue the proof. That requires splitting the permission between both threads, the left thread only requires permission i while the right one requires permission k to perform the insert and delete operations. The proof sketch so far looks like this:

$$
\begin{array}{c}
\{\ out_{def}(h,v,1)\ \} \\
\{\ out_{one}(h,v,1)\ \} \\
\{\ out_{one}(h,v,1/4) * out_{one}(h,v,3/4)\ \} \\
\begin{array}{c|c|c}
\{\ out_{one}(h,v,1/4)\ \} & & \{\ out_{one}(h,v,3/4)\ \} \\
r_1 := \text{SEARCH}(h,v,r) & & r_2 := \text{INSERT}(h,v,p) \\
\left\{\begin{array}{l} out_{one}(h,v,1/4) \\ * \\ (r_1 = \_ \vee r_1 = null) \end{array}\right\} & & \{\ in_{one}(h,v,p,3/4) \wedge r_2 = p\ \} \\
 & & \{\ in_{one}(h,v,p,1/8) * in_{one}(h,v,p,5/8) \wedge r_2 = p\ \} \\
 & \{\ in_{one}(h,v,p,5/8)\ \} & \begin{array}{c|c} \left\{\begin{array}{l} in_{one}(h,v,p,1/8) \\ \wedge r_2 = p \end{array}\right\} & \\ r_4 := \text{SEARCH}(h,v) & \left\{\begin{array}{l} (in_{one}(h,v,p,1/8) \\ \wedge r_4 = \_) \\ \vee \\ (out_{one}(h,v,p,1/8) \\ \wedge r_4 = null) \\ \wedge r_2 = p \end{array}\right\} \end{array} \\
 & \left\{\begin{array}{l} in_{one}(h,v,p,5/8) \\ * r_3 = p \end{array}\right\} & \{\ \ldots\ \} \\
 & & r_5 := \text{DELETE}(h,v) \\
 & & \{\ \ldots\ \}
\end{array}
\end{array}
$$

The problem comes when we try to prove the concurrent search operations after the insert operation in the right hand thread. Since we cannot give both permission k we will end up with an unknown return value when we should not. From the point of view of the right thread, all operations should occur without any race conditions, since in fact, there is no interference from other threads visible to it. We should be able to prove the right thread as a definite environment and assume an unknown environment in the left one.

Our specification does not consider that, from the point of view of an interference environment, we can have an arbitrary number of threads reading at any point in time without causing any interference (reads do not lock or change any values). We have to note that the other threads can only perform search operations for a key value and not insert or delete operations, as that would cause interference. To solve this problem we introduce the notion of a principal environment parametric to a key value. A principal environment for a specific key value is defined as the only environment that can both read and write that key value at some point in time. All threads outside the scope of the principal environment for a key value can only perform search operations for that key value. Concretely, we introduce a new predicate called $read(h, v)$ which describes that we are outside the principal environment for key value v and that we can only perform reads. We can get this predicate from a principal environment predicate or by duplicating the predicate itself. The predicate says that if a thread has the permission to read a key value it can give that permission to any other thread since it will not affect the principal environment. Note that we cannot know the state of the index for a key value from this predicate. We assume there can be anything and therefore all return values will be unknown when performing a search operation with this predicate as a precondition.

3.5.2 Recovery

Our current specification assumes that when we reach an unknown predicate we are stuck with it. This is a very strong assumption. It says that if there is a race condition where we cannot determine the contents of the index for a key value we can only prove that the operations that manipulate this key value are safe from this point onward. Consider the following example:

$$\begin{array}{c} r_1 := \text{DELETE}(h, v) \quad \| \quad r_2 := \text{INSERT}(h, v, q) \\ r_3 := \text{DELETE}(h, v) \end{array}$$

With our current specification if we have as precondition the predicate $in_{def}(h, v, p, 1)$, which guarantees that at the beginning of the execution we have full permission to manipulate the key value, we can only prove safety and will have as postcondition $unk(h, v, i)$. One can see that despite a race condition existing initially, the last operation is sequential and assures that there will be no pointer associated with the key value v. We wish to get as postcondition the predicate $out_{def}(h, v, 1)$ instead. Moreover, we should always be able to recover from an unknown environment when performing a sequential write operation.

To recover from an unknown environment we can consider two new specifications, for insertion and deletion, which when given full permission and the unknown predicate, result in a definite state. However, we must note that at the insert case we do not know the pointer corresponding to the key value. We only know that it exists in the index.

Now assume we have variation of the previous program as following:

$$\begin{array}{c} r_1 := \text{DELETE}(h, v) \end{array} \quad \left\| \quad \begin{array}{c} r_2 := \text{INSERT}(h, v, p) \\ r_3 := \text{DELETE}(h, v) \end{array} \right.$$

We have two threads which contain concurrent inserts and deletes, creating a race condition and therefore we cannot know which one will occur first. However, we have a very important detail, both threads do a delete operation on the same key value as their final operation. With our current specification, assuming a precondition of $out_{def}(h, v, 1)$, we can only reach $unk(h, v, 1)$ as postcondition. We should be able to conclude that the postcondition is in fact $out_{def}(h, v, 1)$ as, despite the existing race conditions, we can still determine the contents of the index for that key

value. In fact, we can think that after each delete operation there could be an arbitrary number of search operations and that would not affect the postcondition of the whole execution. This is because the search operations do not affect the contents of the index. We should be able to capture all of this knowledge and recover to a known environment from an unknown one.

To recover from either concurrent inserts or deletes as the last writing operation of all threads we need to know which were the last operations which occurred at the end of each thread after all of their executions. Since we want to keep all of our specifications local, we can introduce that information by expanding the unknown predicate to contain the last writing operation if it existed at all. We can use the predicates $unk_{ins}(h, v, i)$ and $unk_{del}(h, v, i)$ to indicate that an insert and delete occurred respectively. We can then merge then them accordingly in each case. If there are only insert and search operations we can say there is a mapping in the index. If there are only delete and search operations we say there is no mapping in the index. We should note that when a search operation occurs, it should maintain the information of the last writer if it exists at all. Finally, if insert and delete operations occur we return to our original $unk(h, v, i)$ predicate.

3.6 Extending the specification

We have presented solutions to the problems we mentioned in the previous section and we will now show how to incorporate them in our generalised specification.

Our first solution considers a principal environment modulo an arbitrary number of threads which only read the shared system. We can consider that the recover solution is only used in this principal environment since all computation outside it does not influence the shared state. Since both solutions do not interfere with each other we can apply both at the same time when extending the generalised specification.

3.6.1 Predicates

We extend the predicates given in subsection 3.4.1 with new ones to allow a finer notion of environment and to allow recovery from predicate $unk(h, v, i)$ to a known environment. We need the following extra set of abstract predicates, where h, v and i have the same meaning as in 3.3.1:

$read(h, v)$	nothing is known about the value v in the index h and the thread can only perform searches.
$unk_{ins}(h, v, i)$	nothing is known about the value v in the index h and the last operation of the thread was an insert.
$unk_{del}(h, v, i)$	nothing is known about the value v in the index h and the last operation of the thread was a delete.

The predicate $read(h, v)$ is used to allow any concurrent thread to read the index without any knowledge of its contents. This allows us to compose new threads which only read with an existing program without modifying its environment.

The predicates $unk_{ins}(h, v, i)$ and $unk_{del}(h, v, i)$ have the same meaning as $unk(h, v, i)$ except that they contain information about the last operation made by a thread. This allows a less restrictive way to recover from an unknown environment to a known one.

3.6.2 Axioms

We extend the axioms given in subsection 3.4.2 with the following ones:

$$
\begin{array}{rcl}
in_{def}(h,v,p,i) * read(h,v) & \iff & in_{def}(h,v,p,i) \\
out_{def}(h,v,i) * read(h,v) & \iff & out_{def}(h,v,i) \\
in_{one}(h,v,p,i) * read(h,v) & \iff & in_{one}(h,v,p,i) \\
out_{one}(h,v,i) * read(h,v) & \iff & out_{one}(h,v,i) \\
in_{ins}(h,v,p,i) * read(h,v) & \iff & in_{ins}(h,v,p,i) \\
out_{ins}(h,v,i) * read(h,v) & \iff & out_{ins}(h,v,i) \\
in_{del}(h,v,p,i) * read(h,v) & \iff & in_{del}(h,v,p,i) \\
out_{del}(h,v,i) * read(h,v) & \iff & out_{del}(h,v,i) \\
unk(h,v,i) * read(h,v) & \iff & unk(h,v,i) \\
read(h,v) * read(h,v) & \iff & read(h,v) \\
unk_{ins}(h,v,i) * read(h,v) & \iff & unk_{ins}(h,v,i) \\
unk_{ins}(h,v,i) * unk_{ins}(h,v,j) & \iff & unk_{ins}(h,v,i+j) \\
unk_{ins}(h,v,1) & \implies & in_{def}(h,v,\_,1) \\
unk_{del}(h,v,i) * read(h,v) & \iff & unk_{del}(h,v,i) \\
unk_{del}(h,v,i) * unk_{del}(h,v,j) & \iff & unk_{del}(h,v,i+j) \\
unk_{del}(h,v,1) & \implies & out_{def}(h,v,1) \\
unk_{del}(h,v,i) * unk_{ins}(h,v,j) & \iff & unk(h,v,i+j)
\end{array}
$$

Again, most of these cases are simple. We will focus on the interesting ones.

The axiom $in_{def}(h,v,p,i) * read(h,v) \iff in_{one}(h,v,p,i)$ describes the splitting of a predicate into itself and one that allows a thread to read the index without any knowledge about its contents for a particular key value. There are analogous axioms for each of our predicates in order to allow this at any point of the proof.

The axiom $unk_{ins}(h,v,1) \implies in_{def}(h,v,\_,1)$ describes that all the previous threads, despite being in an unknown environment, did an insert operation as the last writing operation (i.e. there could have been multiple searches after that insert operation) and therefore there must be a mapping in the index for that key value with unknown contents. There are analogous axioms for deletion.

3.6.3 Specifications

We add to the specifications given at the subsection 3.4.3 the following ones:

$$
\begin{array}{ccc}
\{\ unk(h,v,i)\ \} & r := \text{INSERT}(h,v,p) & \{\ unk_{ins}(h,v,i) \wedge (r = true \vee r = false)\ \} \\
\{\ unk(h,v,i)\ \} & r := \text{DELETE}(h,v) & \{\ unk_{del}(h,v,i) \wedge (r = true \vee r = false)\ \} \\
\{\ read(h,v)\ \} & r := \text{SEARCH}(h,v) & \{\ read(h,v) \wedge (r = \_ \vee r = null)\ \} \\
\{\ unk_{ins}(h,v,i)\ \} & r := \text{SEARCH}(h,v) & \{\ unk_{ins}(h,v,i) \wedge (r = \_ \vee r = null)\ \} \\
\{\ unk_{ins}(h,v,i)\ \} & r := \text{INSERT}(h,v,p) & \{\ unk_{ins}(h,v,i) \wedge (r = true \vee r = false)\ \} \\
\{\ unk_{ins}(h,v,i)\ \} & r := \text{DELETE}(h,v) & \{\ unk_{del}(h,v,i) \wedge (r = true \vee r = false)\ \} \\
\{\ unk_{del}(h,v,i)\ \} & r := \text{SEARCH}(h,v) & \{\ unk_{del}(h,v,i) \wedge (r = \_ \vee r = null)\ \} \\
\{\ unk_{del}(h,v,i)\ \} & r := \text{INSERT}(h,v,p) & \{\ unk_{ins}(h,v,i) \wedge (r = true \vee r = false)\ \} \\
\{\ unk_{del}(h,v,i)\ \} & r := \text{DELETE}(h,v) & \{\ unk_{del}(h,v,i) \wedge (r = true \vee r = false)\ \}
\end{array}
$$

Finally, we note that the predicates $in_{one}(h,v,p,i)$ and $out_{one}(h,v,i)$ are now redundant and can be removed from the specifications.

3.6.4 Reasoning

We can now reason about any combination of programs which manipulate a concurrent index and still maintaining information if possible about its contents. We will show how to reason about the examples which were considered as limitations before.

The first example shows the principal environment solution. Consider the following program:

$$
\begin{array}{c}
\{\ out_{def}(h,v,1)\ \} \\
r_1 := \text{SEARCH}(h,v,r) \quad \| \quad r_2 := \text{INSERT}(h,v,p) \\
r_3 := \text{SEARCH}(h,v) \quad \| \quad r_4 := \text{SEARCH}(h,v) \\
r_5 := \text{DELETE}(h,v) \\
\{\ out_{def}(h,v,1)\ \}
\end{array}
$$

The correctness proof for the program is as follows:

$$
\begin{array}{c}
\{\ out_{def}(h,v,1)\ \} \\
\{\ read(h,v) * out_{def}(h,v,1)\ \} \\
\{\ out_{def}(h,v,1)\ \} \\
r_2 := \text{INSERT}(h,v,p) \\
\{\ in_{def}(h,v,p,1) \wedge r_2 = true\ \} \\
\{\ read(h,v)\ \} \quad \| \quad \{\ in_{def}(h,v,p,1/2)\ \} \quad \| \quad \left\{ \begin{array}{l} in_{def}(h,v,p,1/2) \\ \wedge\ r_2 = true \end{array} \right\} \\
r_1 := \text{SEARCH}(h,v,r) \quad \| \quad r_3 := \text{SEARCH}(h,v) \quad \| \quad r_4 := \text{SEARCH}(h,v) \\
\left\{ \begin{array}{l} read(h,v) \\ \wedge\ (r_1 = true \\ \vee\ r_1 = false) \end{array} \right\} \quad \| \quad \{\ in_{def}(h,v,p,1/2) \wedge r_3 = p\ \} \quad \| \quad \left\{ \begin{array}{l} in_{def}(h,v,p,1/2) \\ \wedge\ r_2 = true \wedge r_4 = p \end{array} \right\} \\
\left\{ \begin{array}{l} in_{def}(h,v,p,1/2) \wedge r_3 = p \\ *\ in_{def}(h,v,p,1/2) \wedge r_2 = true \wedge r_4 = p \end{array} \right\} \\
\{\ in_{def}(h,v,p,1) \wedge r_2 = true \wedge r_3 = p \wedge r_4 = p\ \} \\
r_5 := \text{DELETE}(h,v) \\
\{\ out_{def}(h,v,1) \wedge r_2 = true \wedge r_3 = p \wedge r_4 = p \wedge r_5 = true\ \} \\
\left\{ \begin{array}{l} read(h,v) \wedge (r_1 = true \vee r_1 = false) \\ *\ out_{def}(h,v,1) \wedge r_2 = true \wedge r_3 = p \wedge r_4 = p \wedge r_5 = true \end{array} \right\} \\
\{\ out_{def}(h,v,1) \wedge (r_1 = true \vee r_1 = false) \wedge r_2 = true \wedge r_3 = p \wedge r_4 = p \wedge r_5 = true\ \} \\
\{\ out_{def}(h,v,1)\ \}
\end{array}
$$

We start with the predicate $out_{def}(h,v,1)$ and turn it into $read(h,v) * out_{def}(h,v,1)$. This allow us to use the $read(h,v,)$ predicate at the left hand thread which performs a simple search operation and maintain the principal environment at the right hand thread. With the predicate $out_{def}(h,v,1)$ the proof at the right hand thread is similar to the examples shown before. It is important to notice that at the end of the execution of both threads we end up with the $out_{def}(h,v,1)$ predicate, as we merged the predicate $read(h,v)$ with it.

We now show how to deal with recovery using a sequential operation. Consider the following program:

$$
\begin{array}{c}
\{\ in_{def}(h,v,p,1)\ \} \\
r_1 := \text{DELETE}(h,v) \quad \| \quad r_2 := \text{INSERT}(h,v,q) \\
r_3 := \text{DELETE}(h,v) \\
\{\ out_{def}(h,v,1)\ \}
\end{array}
$$

The correctness proof for the program is as follows:

$$
\begin{array}{c}
\{\ in_{def}(h,v,p,1)\ \} \\
\{\ unk(h,v,1)\ \} \\
\{\ unk(h,v,1/2) * unk(h,v,1/2)\ \} \\
\begin{array}{c|c}
\{\ unk(h,v,1/2)\ \} & \{\ unk(h,v,1/2)\ \} \\
r_1 := \text{DELETE}(h,v) & r_2 := \text{INSERT}(h,v,q) \\
\left\{\begin{array}{l} unk(h,v,1/2) \\ \wedge (r_1 = true \vee r_1 = false) \end{array}\right\} & \left\{\begin{array}{l} unk(h,v,1/2) \\ \wedge (r_2 = true \vee r_2 = false) \end{array}\right\}
\end{array} \\
\left\{\begin{array}{l} unk(h,v,1/2) \wedge (r_1 = true \vee r_1 = false) \\ * unk(h,v,1/2) \wedge (r_2 = true \vee r_2 = false) \end{array}\right\} \\
\{\ unk(h,v,1) \wedge (r_1 = true \vee r_1 = false) \wedge (r_2 = true \vee r_2 = false)\ \} \\
r_3 := \text{DELETE}(h,v) \\
\left\{\begin{array}{l} unk_{def}(h,v,1) \wedge (r_1 = true \vee r_1 = false) \\ \wedge (r_2 = true \vee r_2 = false) \wedge (r_3 = true \vee r_3 = false) \end{array}\right\} \\
\left\{\begin{array}{l} out_{def}(h,v,1) \wedge (r_1 = true \vee r_1 = false) \\ \wedge (r_2 = true \vee r_2 = false) \wedge (r_3 = true \vee r_3 = false) \end{array}\right\} \\
\{\ out_{def}(h,v,1)\ \}
\end{array}
$$

We have a race condition between both threads, we cannot know which one inserts or deletes first. Therefore, we are forced to make the predicate $in_{def}(h,v,p,1)$ into $unk(h,v,1)$ to continue the proof. However, the last operation of the program is a sequential delete operation that leads to the predicate $unk_{def}(h,v,1)$. Since we have full permission we can use the axiom to make it a $out_{def}(h,v,1)$ predicate which assures that there is no pointer associated with the key value v in the index h.

We will now show how to recover from an unknown environment using the information of the writing threads. Consider the following program:

$$
\begin{array}{c}
\{\ out_{def}(h,v,1)\ \} \\
\begin{array}{c|c}
r_1 := \text{DELETE}(h,v) & \begin{array}{c} r_2 := \text{INSERT}(h,v,q) \\ r_3 := \text{DELETE}(h,v) \end{array}
\end{array} \\
\left\{\begin{array}{l} out_{def}(h,v,1) \wedge (r_1 = true \vee r_1 = false) \\ \wedge (r_2 = true \vee r_2 = false) \wedge (r_3 = true \vee r_3 = false) \end{array}\right\}
\end{array}
$$

The correctness proof for the program is as follows:

$$
\begin{array}{c}
\{\ out_{def}(h,v,1)\ \} \\
\{\ unk(h,v,1)\ \} \\
\{\ unk(h,v,1/2) * unk(h,v,1/2)\ \} \\
\begin{array}{c|c}
\begin{array}{c} \{\ unk(h,v,1/2)\ \} \\ r_1 := \text{DELETE}(h,v) \\ \left\{\begin{array}{l} unk_{del}(h,v,1/2) \\ \wedge (r_1 = true \vee r_1 = false) \end{array}\right\} \end{array} & \begin{array}{c} \{\ unk(h,v,1/2)\ \} \\ r_2 := \text{INSERT}(h,v,q) \\ \left\{\begin{array}{l} unk_{ins}(h,v,1/2) \\ \wedge (r_2 = true \vee r_2 = false) \end{array}\right\} \\ r_3 := \text{DELETE}(h,v) \\ \left\{\begin{array}{l} unk_{del}(h,v,1/2) \\ \wedge (r_2 = true \vee r_2 = false) \\ \wedge (r_3 = true \vee r_3 = false) \end{array}\right\} \end{array}
\end{array} \\
\left\{\begin{array}{l} unk_{del}(h,v,1/2) \wedge (r_1 = true \vee r_1 = false) \\ * unk_{del}(h,v,1/2) \wedge (r_2 = true \vee r_2 = false) \wedge (r_3 = true \vee r_3 = false) \end{array}\right\} \\
\left\{\begin{array}{l} unk_{del}(h,v,1) \wedge (r_1 = true \vee r_1 = false) \\ \wedge (r_2 = true \vee r_2 = false) \wedge (r_3 = true \vee r_3 = false) \end{array}\right\} \\
\left\{\begin{array}{l} out_{def}(h,v,1) \wedge (r_1 = true \vee r_1 = false) \\ \wedge (r_2 = true \vee r_2 = false) \wedge (r_3 = true \vee r_3 = false) \end{array}\right\}
\end{array}
$$

Again, we have a race condition between both threads. We must weaken the predicate $out_{def}(h,v,1)$ into $unk(h,v,1)$ and split it to prove both threads. At the right hand side, we have a single delete

Environment	Permission	Interference (current thread)	Interference (other threads)
def	1	All actions	No actions
def	i	No actions	No actions
ins	i	Only inserts	Only inserts
del	i	Only deletes	Only deletes
unk	i	All actions	All actions
$read$		No actions	All actions

Table 3.1: Environments and described interference

operation which, using our specification, results in the $unk_{del}(h, v, 1/2)$ which by itself does not allow us to get a stronger predicate. However, at the left hand side, there is an insert operation followed by another delete operation, which results also in a $unk_{del}(h, v, 1/2)$ predicate. Combining both predicates we get predicate $unk_{del}(h, v, 1)$ which, since we have full permission, can be turned into the $out_{def}(h, v, 1)$ predicate.

3.7 Evaluation

We have defined abstract specifications which capture the knowledge about a concurrent index and also allow us to prove safety even in the presence of race conditions. Our specifications are all local and only require the knowledge about the key value which the operation will manipulate. We also capture that a concurrent index data structure should support parallel threads reading and writing at the same time. One advantage of our specification is that it does not require the assumption that the operations are linearisable, since in practise that might not be true.

Our predicates describe not only information about the index for a specific key value, they also capture some information about the environment or even history of the computation of the current thread. This allows great flexibility and power and still maintaining simple compositional specifications.

Another advantage of our specification is that it allow us to treat each mapping in the index as a separate predicate at the high-level. This means that not only that we can make our assertions as small as possible, it also allows dealing with race conditions for a specific key value, without propagating to other key values.

Our specification aims to give the most meaningful information about the contents of the index and in general it achieves this. We can see by table 3.1, that our predicates address all possibilities of interference which can occur on the shared state by the current thread and the other threads: There is one limitation though when it comes to concurrent insert operations without any deletions. We do not know which pointer was inserted, but we should be able to tell that it belongs to a set of pointers when we have full permission. Without full permission one can only say that it might be in that set.

There is still much work to be done when it comes to the specifications. Our specification introduces several new novel concepts which can be applied in many other programs, specially the ones which have readers/writers problems.

Finally, we have not considered factors such as time efficiency or space usage, among others, as it is out of the scope of this work. We believe one can extend our specification to tackle this issues according to their particular aims.

Chapter 4

Implementations of concurrent indexes

In chapter 3 we presented abstract specifications for a concurrent index. In this chapter, we show how to prove that a concrete implementation satisfies our abstract specification. The predicates are not just interpreted as assertions about the internal state of the module, but also as assertions about the internal interference of the module.

4.1 Overview

We will show how to, given a concrete implementation of a concurrent index, prove the correctness of its operations according to the abstract specification we have presented before. In order to make the explanation simpler, we will consider a node list which is a linked list where each element is a leaf node from a B^{Link} tree. This way we have simpler algorithms, yet, we maintain the complexity of the internal details of the action model which we wish to explain. The node list is manipulated by three operations, search, insert and delete, which have the same definition as the specifications we provided in the previous chapter. We will give a concrete implementation of each operation which will manipulate the node list and we also give concrete definitions for each abstract predicate. The abstract predicates will have to capture the interference of the environment and must allow us to prove the partial correctness of each specification.

Finally, using the same abstract specification, we will explain how to prove the correctness of other index implementations (a B^{Link} tree and a hash table) against the specifications extending the ideas used for the node list.

4.2 Node list

We will start by describing a node list implementation which satisfies our high-level specifications for concurrent indexes.

4.2.1 Model

We have reasoned about B^{Link} trees in the past using RGSep, we will use similar ideas to create our model and formalise a list node using separation logic. Our program state consists of a heap and a variable store.

Formalising the data structure

In order to reason about algorithms on a node list we have to formalise precisely what this is. We start by defining a value-pointer list and a new binary operation for them.

Definition 8 (Value-pointer list)**.** *A value-pointer list is an empty list or a pair (v, p) followed by another list. The list is ordered in an increasing way by the values. It is defined as:*

$$s = [] \mid (v, p) : s$$

The key value v is an integer and p is a pointer to a data entry. We treat the value-pointer list as an ordered set. Therefore we assume operations such as union, intersection and complement with the classical meaning, yet maintaining its order.

We introduce a new binary operation for value-pointer lists, $s \leq s'$, with the interpretation that if a value-pointer pair is in the list s, then it will also be in the list s', i.e. all elements in s exist in s'. If the operation $\leq$ is applied to numbers it has the classical meaning. Formally we define the new operation as:

$$s_1 \leq s_2 \equiv \forall (v, p) \,.\, (v, p) \in s_1 \Rightarrow (v, p) \in s_2$$

We assume that thread identifiers are positive integers and let T represent the set containing all thread identifiers and zero.

Definition 9 (Node predicate)**.** *A node is represented by a predicate as follows:*

$$node(t_{id}, v_0, s, v_{i+1}, p_{i+1})$$

The value $t_{id} \in T$ tells us if the node has been locked by a thread. When zero, the node is unlocked, otherwise it has the thread identifier of the locking thread. The values v_0 and v_{i+1} are the lower bound and upper bound, respectively, on the entries in the node. p_{i+1} is the pointer to the next node of the list. This pointer is also called a link. It may be null if the node is the last node of the list. Finally s is a list of value-pointer pairs.

Assume k to be a known constant integer value, greater than zero. We let s, the list of value-pointer pairs in a node have at most $2k$ pairs. This means that a node can have at most $2k$ pairs, and the minimum number of pairs is zero.

A node list is made of disjoint nodes. Therefore we can use separation logic to model the node list and define it precisely.

Definition 10 (Node list)**.** *Let $data = [(v_1, p_1), \ldots, (v_n, p_n)]$ be a value-pointer list, then:*

$$\begin{array}{lcl} nodeList(list, data) & \stackrel{def}{=} & \exists min_1, \ldots, min_m, a_1, \ldots, a_m, s_1, \ldots, s_m \,. \\ & \circledast_{i=1}^{m-1} & a_i \mapsto node(\_, min_i, s_i, min_{i+1}, a_{i+1}) \\ & * & a_m \mapsto node(\_, min_m, s_m, +\infty, null) \\ & \wedge & min_1 < \ldots < min_m \\ & \wedge & min_1 = -\infty \\ & \wedge & data = \bigcup_{i=1}^{m} s_i \\ & \wedge & m > 0 \\ & \wedge & list = a_1 \end{array}$$

where $data = \bigcup_{i=1}^{m} s_i$ is the disjoint union (i.e. no (v, p) pairs occurs more than once).

The *nodeList* predicate takes two parameters, *list* and *data*. The first parameter contains the address of the first node of the node list, this address is constant. The second parameter contains the concatenation of all value-pointer lists of each node in the linked list. We can deduce the correct node location of each value-pair in that list because the value must be greater than the node minimum value and less than or equal to the node maximum value. Implicitly the predicate implies that each node does not overflow its capacity.

Program state

The program state consists of a working heap h and a variable store σ.

Definition 11 (Heap)**.** *A heap h is a finite partial function that maps heap addresses to node data:*

$$h : \quad Addr \rightharpoonup_{fin} Node$$

The heap is divided up such that each thread has a private section of the heap for internal use and there exists a single area that is shared by all threads which contains the node list.

If a node is reachable from the first node of the list, it is part of the shared state. One should notice that a node can be in the heap, even containing pointers to the list structure, without belonging to the shared state, if it is not reachable from the first node.

The local state of a thread is composed by its private section. It can be described as every node accessible, by the thread, in the heap excluding the shared state. This will include all nodes allocated in the heap before a thread modifies the list to point to them.

Definition 12 (Store)**.** *A store σ is a set of finite partial functions that map integer variables $Var_{\mathbb{Z}} = \{i, j, \dots\}$ to integers, address variables $Var_{Addrs} = \{p, q, \dots\}$ to heap addresses, boolean variables $Var_{\mathbb{B}} = \{b, \dots\}$ to boolean values and node data variables $Var_{Node} = \{A, B, \dots\}$ to node data:*

$$\sigma : \quad (Var_{\mathbb{Z}} \rightharpoonup_{fin} \mathbb{Z}) \times (Var_{Addr} \rightharpoonup_{fin} Addr \cup \{null\}) \times (Var_{\mathbb{B}} \rightharpoonup_{fin} \mathbb{B}) \times (Var_{Node} \rightharpoonup_{fin} Node)$$

4.2.2 Programming language

To reason about the programs within our model, we require a language for those programs to be written in. Our language is as simple and general as possible, consisting only of imperative sequencing, conditionals, while loops, parallel composition and basic expressions. We also provide commands to manipulate the heap and store, which allow easy manipulation of nodes.

Heap commands

The heap commands manipulate the heap. Since the heap contains nodes, these commands will directly modify nodes. Since the shared state is included in the heap, we require that all the commands that manipulate the heap are atomic, i.e. are indivisible operations. We have the following heap commands:

$C_H ::=$	$q := \text{NEW}()$	Allocate new node
	$A := \text{GET}(x)$	Read node operation
	$\text{PUT}(A, x)$	Write node operation
	$\text{LOCK}(x)$	Lock node operation
	$\text{UNLOCK}(x)$	Unlock node operation

The allocation of a new node will return the address of the newly allocated node. That node will have size $2k$ (k defined as in section 4.2.1). A newly allocated node belongs to the local state of a thread, since there is no reference from the shared list to that node.

The read node operation returns a copy of the contents of the node with address x in the heap.

Writing to a node at the heap consists of atomically replacing the node with address x in the heap, by the contents of the node A located in the store.

The lock and unlock operations acquire and release the lock on a node in the heap with address x. A thread can only unlock a node if it has previously acquired the lock. If it is the case that a thread wants to lock a node currently locked by another thread, then that thread remains blocked until the other thread releases the node. All of these operations are atomic, therefore, between the lock and unlock operations all changes on the node are executed with mutual execution with threads that also acquire the lock and release it. This means that read operations can be done over a node even if it is locked by another thread.

The command NEW() only changes the local state. All the other heap commands can change both the local and shared state.

Store commands

The store commands, as with the heap commands, consist of commands that work with nodes. These commands do not manipulate the shared state and, as such, they are not required to be atomic.

The commands which operate on nodes in a store are the following:

$C_N ::=$	$v := \text{LOWVALUE}(A)$	Reads the minimum value of node A
	$v := \text{HIGHVALUE}(A)$	Reads the maximum value of node A
	$b := \text{ISSAFE}(A)$	Tests if it is safe to insert into node A
	$p := \text{NEXT}(A, v)$	Reads the pointer in A that contains the right path for key v
	$b := \text{ISIN}(A, v)$	Tests if a pair with key v exists in node A
	$p := \text{LOOKUP}(A, v)$	Reads the pointer with key v in node A
	$\text{ADDPAIR}(A, v, p)$	Inserts the pair (v, p) into node A
	$\text{REMOVEPAIR}(A, v)$	Deletes the pair (v, p) from node A
	$B := \text{REARRANGE}(A, v, p, q)$	Splits the node A into $A$and B

The command $\text{LOWVALUE}(A)$ and $\text{HIGHVALUE}(A)$ return the minimum and maximum values of a node, respectively, in the store.

The command $\text{ISSAFE}(A)$ tests if it is safe to insert a value pointer pair into the node. It returns true if the node has fewer than $2k$ pairs in its value-pointer list and false otherwise.

The command $\text{NEXT}(A,v)$ returns the address in node A of the next node to follow in order to find the data corresponding to the value v. The command should only be used if the value v is larger than the maximum value of the node A, and it will return the link pointer (i.e. address of the right sibling).

The command $\text{ISIN}(A, v)$ returns true if there exists a pair in node A with the value v. Otherwise it returns false.

The command $\text{ADDPAIR}(A,v,p)$ inserts the pair (v, p) into the value-pointer list of the node A (faults if A is already full). Analogously, $\text{REMOVEPAIR}(A,v)$ removes the pair in the node with a value v (faults if there was not one).

The last command, $\text{REARRANGE}(A, v, p, q)$, is the most complex of all. It splits the contents of the value-pointer list of node A and the pair (v, p) into two lists. Then updates both nodes A and B with the new value-pointer lists with the minimum value of the node B equal to the maximum value of node A. The link pointer of A will point to B. The maximum value and link pointer of node B are copied from the ones originally in node A. Finally it gets the maximum value and link pointer of node A to be the minimum value of B and the address of B on the heap, which is given by the argument q.

Language commands

We define a simple concurrent imperative programming language with the following commands:

$C ::=$	**skip**	Empty command
	$C_1; C_2$	Sequential composition
	if (B) $\{C_1\}$ **else** $\{C_2\}$	Conditional command
	while (B) $\{C\}$	Loop command
	$C_1 \| C_2$	Parallel composition
	C_H	Heap command
	C_N	Store node command

We assume the existence of Boolean expressions B which do not modify the heap. The conditional command evaluates the Boolean expression B, if it is equal to true, then it runs the command C_1, otherwise it runs C_2. The loop command runs the command C while the Boolean expression B is equal to true. Parallel composition can be used to express the concurrent execution of two threads, where one runs command C_1 and the other runs C_2. We do not require this when writing our tree operations, but it is useful when dealing with operations on the tree at a higher level of abstraction. Finally, we assume the existence of integer expressions which do not dereference memory.

Command axioms

We present axioms for each command using separation logic small axioms. These describe the effects of a command on the smallest heap for which it can succeed. We can use the frame rule to scale these commands up to larger heaps. We will use t_{id} as the identifier of the thread performing the lock operation.

The axioms for the heap and store commands are the following, where $t = 0$ or $t = t_{id}$ for some thread identifier t_{id}:

$$\{\ emp\ \}\quad q := \textsc{new}()\quad \left\{\begin{array}{l} q = x \wedge x \mapsto N \\ \wedge\, N = node(0, 0, \emptyset, 0, null) \end{array}\right\}$$

$$\left\{\begin{array}{l} x \mapsto N \\ \wedge\, N = node(t, v_0, s, v_{i+1}, p_{i+1}) \end{array}\right\}\quad A := \textsc{get}(x)\quad \left\{\begin{array}{l} x \mapsto N \\ \wedge\, N = node(t, v_0, s, v_{i+1}, p_{i+1}) \\ \wedge\, A = N \end{array}\right\}$$

$$\left\{\begin{array}{l} x \mapsto N \\ \wedge\, N = node(t, v_0, s, v_{i+1}, p_{i+1}) \\ \wedge\, A = M \\ \wedge\, M = node(t', v'_0, s', v'_{i+1}, p'_{i+1}) \end{array}\right\}\quad \textsc{put}(A, x)\quad \left\{\begin{array}{l} x \mapsto M \wedge A = M \\ \wedge\, M = node(t', v'_0, s', v'_{i+1}, p'_{i+1}) \end{array}\right\}$$

$$\left\{\begin{array}{l} x \mapsto N \\ \wedge\, N = node(0, v_0, s, v_{i+1}, p_{i+1}) \end{array}\right\}\quad \textsc{lock}(x)\quad \left\{\begin{array}{l} x \mapsto N \\ \wedge\, N = node(t_{id}, v_0, s, v_{i+1}, p_{i+1}) \end{array}\right\}$$

$$\left\{\begin{array}{l} x \mapsto N \\ \wedge\, N = node(t_{id}, v_0, s, v_{i+1}, p_{i+1}) \end{array}\right\}\quad \textsc{unlock}(x)\quad \left\{\begin{array}{l} x \mapsto N \\ \wedge\, N = node(0, v_0, s, v_{i+1}, p_{i+1}) \end{array}\right\}$$

$$\left\{\begin{array}{l} emp \\ \wedge\, A = node(t, v_0, s, v_{i+1}, p_{i+1}) \end{array}\right\}\quad v := \textsc{lowValue}(A)\quad \left\{\begin{array}{l} emp \\ \wedge\, A = node(t, v_0, s, v_{i+1}, p_{i+1}) \\ \wedge\, v = v_0 \end{array}\right\}$$

$$\left\{\begin{array}{l} emp \\ \wedge\, A = node(t, v_0, s, v_{i+1}, p_{i+1}) \end{array}\right\}\quad v := \textsc{highValue}(A)\quad \left\{\begin{array}{l} emp \\ \wedge\, A = node(t, v_0, s, v_{i+1}, p_{i+1}) \\ \wedge\, v = v_{i+1} \end{array}\right\}$$

$$\left\{\begin{array}{l} emp \\ \wedge\, A = node(t, v_0, s, v_{i+1}, p_{i+1}) \\ \wedge\, s = [(v_1, p_1), \ldots, (v_i, p_i)] \\ \wedge\, i < 2k \end{array}\right\}\quad b := \textsc{isSafe}(A)\quad \left\{\begin{array}{l} emp \\ \wedge\, A = node(t, v_0, s, v_{i+1}, p_{i+1}) \\ \wedge\, s = [(v_1, p_1), \ldots, (v_i, p_i)] \\ \wedge\, i < 2k \\ \wedge\, b = true \end{array}\right\}$$

$$\left\{\begin{array}{l} emp \\ \wedge\, A = node(t, v_0, s, v_{i+1}, p_{i+1}) \\ \wedge\, s = [(v_1, p_1), \ldots, (v_i, p_i)] \\ \wedge\, i = 2k \end{array}\right\}\quad b := \textsc{isSafe}(A)\quad \left\{\begin{array}{l} emp \\ \wedge\, A = node(t, v_0, s, v_{i+1}, p_{i+1}) \\ \wedge\, s = [(v_1, p_1), \ldots, (v_i, p_i)] \\ \wedge\, i = 2k \\ \wedge\, b = false \end{array}\right\}$$

$$\left\{\begin{array}{l} emp \\ \wedge\, A = node(t, v_0, s, v_{i+1}, p_{i+1}) \\ \wedge\, v > v_{i+1} \end{array}\right\}\quad p := \textsc{next}(A, v)\quad \left\{\begin{array}{l} emp \\ \wedge\, A = node(t, v_0, s, v_{i+1}, p_{i+1}) \\ \wedge\, v > v_{i+1} \wedge p = p_{i+1} \end{array}\right\}$$

$$\left\{\begin{array}{l} emp \\ \wedge\, A = node(t, v_0, s, v_{i+1}, p_{i+1}) \\ \wedge\, (v, p) \in s \end{array}\right\}\quad b := \textsc{isIn}(A, v)\quad \left\{\begin{array}{l} emp \\ \wedge\, A = node(t, v_0, s, v_{i+1}, p_{i+1}) \\ \wedge\, (v, p) \in s \wedge b = true \end{array}\right\}$$

$$\left\{\begin{array}{l} emp \\ \wedge\, A = node(t, v_0, s, v_{i+1}, p_{i+1}) \\ \wedge\, (v, p) \notin s \end{array}\right\}\quad b := \textsc{isIn}(A, v)\quad \left\{\begin{array}{l} emp \\ \wedge\, A = node(t, v_0, s, v_{i+1}, p_{i+1}) \\ \wedge\, (v, p) \notin s \wedge b = false \end{array}\right\}$$

$$\left\{\begin{array}{l} emp \\ \wedge\, A = node(t, v_0, s, v_{i+1}, p_{i+1}) \\ \wedge\, (v, q) \in s \end{array}\right\}\quad p := \textsc{lookup}(A, v)\quad \left\{\begin{array}{l} emp \\ \wedge\, A = node(t, v_0, s, v_{i+1}, p_{i+1}) \\ \wedge\, (v, q) \in s \wedge p = q \end{array}\right\}$$

$$\left\{\begin{array}{l} emp \\ \wedge\, A = node(t, v_0, s, v_{i+1}, p_{i+1}) \\ \wedge\, s = [(v_1, p_1), \ldots, (v_i, p_i)] \\ \wedge\, i < 2k \wedge (v, p) \notin s \end{array}\right\}\quad \textsc{addPair}(A, v, p)\quad \left\{\begin{array}{l} emp \\ \wedge\, A = node(t, v_0, z, v_{i+1}, p_{i+1}) \\ \wedge\, s = [(v_1, p_1), \ldots, (v_i, p_i)] \\ \wedge\, i < 2k \wedge (v, p) \notin s \\ \wedge\, z = [(w_1, q_1), \ldots, (w_{i+1}, q_{i+1})] \\ \wedge\, z = s \cup (v, p) \wedge (v, p) \in z \end{array}\right\}$$

$$\left\{\begin{array}{l} emp \\ \wedge\, A = node(t, v_0, s, v_{i+1}, p_{i+1}) \\ \wedge\, s = [(v_1, p_1), \ldots, (v_i, p_i)] \\ \wedge\, (v, p) \in s \end{array}\right\} \quad \text{removePair}(A, v) \quad \left\{\begin{array}{l} emp \\ \wedge\, A = node(t, v_0, z, v_{i+1}, p_{i+1}) \\ \wedge\, s = [(v_1, p_1), \ldots, (v_i, p_i)] \\ \wedge\, (v, p) \in s \\ \wedge\, z = [(w_1, q_1), \ldots, (w_{i-1}, q_{i-1})] \\ \wedge\, z = s \backslash (v, p) \wedge (v, p) \notin z \end{array}\right\}$$

$$\left\{\begin{array}{l} emp \\ \wedge\, A = node(t, v_0, s, v_{i+1}, p_{i+1}) \\ \wedge\, s = [(v_1, p_1), \ldots, (v_i, p_i)] \\ \wedge\, v_0 < v \leq v_{i+1} \wedge i = 2k \end{array}\right\} \quad B := \text{rearrange}(A, v, p, q) \quad \left\{\begin{array}{l} emp \\ \wedge\, A = node(t, v_0, s_1, w_k, q) \\ \wedge\, B = node(0, w_k, s_2, v_{i+1}, p_{i+1}) \\ \wedge\, s = [(v_1, p_1), \ldots, (v_i, p_i)] \\ \wedge\, v_0 < v \leq v_{i+1} \\ \wedge\, z = [(w_1, q_1), \ldots, (w_{2k+1}, q_{2k+1}) \\ \wedge\, z = s \cup (v, p) \\ \wedge\, s_1 = [(w_1, q_1), \ldots, (w_k, q_k)] \\ \wedge\, s_2 = [(w_{k+1}, q_{k+1}), \ldots, (w_{2k+1}, q_{2k+1})] \end{array}\right\}$$

We use the usual axioms for the language commands.

4.2.3 Algorithms

In the previous section we have shown a programming language with a set of heap and basic commands that manipulate the heap and store respectively. In this section we write the algorithms that manipulate the node list (i.e. search, insert and delete) in that language.

Search

```
r := search(h, v) {
  current := h;
  A := get(current);
  moveRight;
  if (isIn(A, v)) {
    r := lookup(A, v);
  } else {
    r := null;
  }
}

moveRight ≜ {
  while (v > highValue(A)) {
    current := next(A, v);
    A := get(current);
  }
}
```

Figure 4.1: Node list search algorithm.

The search algorithm (see figure 4.1) starts reading the first node. If the maximum value is less than the key value v then it means that the current node is not the one desired. However, the desired node is accessible from the first node, by moving to the right in the node list. It repeats this same process on the node to the right until it reaches the correct node. It then checks if the key value exists in that node and, if so, it returns the corresponding pointer to the data entry, otherwise it returns null.

Insert

The insert algorithm (see figure 4.2) starts by locking the first node. It then tests if it is the correct node to insert the pair (v, p) into. A node is the correct one to insert a value-pointer pair (v, p) into if the value v is greater than the minimum value of the node and less than or equal to the maximum value. If it is not the correct node, it unlocks the first node and, it uses the procedure moveRight to reach the correct node. Again it locks the node and checks it to see if it really is. It repeats this process until the correct node is found. The reason why we require this locking is because other threads can insert or delete new values which can induce the creation of new nodes. Therefore, unless the algorithm locks the node, every test to see if it is the correct node could be invalidated before actually inserting the new value-pointer pair.

Assuming the algorithm has the lock of the correct node, it must check that the node is safe to insert into (has fewer than $2k$ pairs). If it is, then the algorithm inserts the new value-pointer pair,

```
r := INSERT(h, v, p) {
  current := h;
  completed := FALSE;
  found := FALSE;
  while (found = FALSE) {
    found := TRUE;
    LOCK(current);
    A := GET(current);
    if (ISIN(A, v)) {
      UNLOCK(current);
      r := FALSE;
      completed := TRUE;
    } else if (v > HIGHVALUE(A)) {
      UNLOCK(current);
      found := FALSE;
      MOVERIGHT;
    }
  }
  if (completed = FALSE) {
    if (ISSAFE(A)) {
      INSERTINTOSAFE;
    } else {
      INSERTINTOUNSAFE;
    }
    r := TRUE;
  }
}
```

```
INSERTINTOSAFE ≙ {
  ADDPAIR(A, v, p);
  PUT(A, current);
  UNLOCK(current);
}

INSERTINTOUNSAFE ≙ {
  q := NEW();
  B := REARRANGE(A, v, p, q);
  PUT(B, q);
  PUT(A, current);
  UNLOCK(current);
}
```

```
r := DELETE(h, v) {
  current := h;
  found := FALSE;
  while (found = FALSE) {
    found := TRUE;
    LOCK(current);
    A := GET(current);
    if (ISIN(A, v)) {
      REMOVEPAIR(A, v);
      PUT(A, current);
      UNLOCK(current);
      r := TRUE;
    } else {
      UNLOCK(current);
      if (v > HIGHVALUE(A)) {
        found := FALSE;
        MOVERIGHT;
      } else {
        r := FALSE;
      }
    }
  }
}
```

Figure 4.2: Node list insert and delete algorithms.

stops and returns success. Otherwise, the node is full and it must be split into two to accommodate the new value-pointer pair. The existing pairs must be split to balance between the current and newly created nodes.

When the algorithm splits the node it splits the contents of the node and the pair to be inserted into the current node and the newly created node. The newly created node will have maximum value equal to the current node, while the maximum value of the current node and new minimum value of the created node will be set to a suitable value and equal. The current node link will be set to the new node. All of these changes are made in the store, therefore it must update the actual tree. To do this, it first writes the newly created node into the heap. Since there are no references from the tree to it, this node is still part of the local state. The algorithm then updates the current node with the new contents, making the new node part of the shared state and accessible to other threads.

Delete

The delete algorithm (see figure 4.2) is similar to the insert algorithm until it reaches the correct node. When it has found the correct node, it deletes the pair (v, p) associated with the value v and returns true, otherwise it does nothing and returns false. In the case where the node is not the correct one, it keeps moving to the right until it reaches the correct node. If so, it locks and repeats as described before. A node is said to be the correct one if the minimum value is less than the value key v and the maximum value is greater than or equal to it.

4.2.4 Actions

We have shown the commands and algorithms in previous sections. We now formalise the effects that the heap commands have on the shared state. We present actions for each behaviour that a thread may exhibit on the shared state. All of the actions respect the node list predicate, that is, if

the predicate was valid, it will remain valid after any of the actions described in this section. This means that the node list predicates are preserved by the actions.

Interpretation of predicates

Our abstract specification contains predicates which describe the interference in the environment. The abstract predicates are parametric to a specific key value in the concurrent index and only describe the interference about that key value. The interference is described by the commands which other threads can do concurrently which manipulate the same key value. Therefore, the possible interference of the shared state will be the set of possible changes carried out by the commands which are allowed for a particular environment at the high-level.

Considering our implementation, when we are searching for a key value we are not concerned if a node is locked or not, we are only concerned about the key value and if there is a corresponding pointer in our index at the time we perform the reading of the node in the shared state.

The predicates $in_{def}(h, v, p, 1)$ and $out_{def}(h, v, 1)$ describe that a thread can read or change the key value v and the other threads can only perform reading operations. If we consider the same predicates but with permission i, then they describe that a thread can perform only reading operations and the other threads can only perform reading operations too.

The predicates $in_{ins}(h, v, p, i)$ and $out_{ins}(h, v, i)$ describe that a thread can read and insert the key value v and other threads can do the same, no thread can perform delete operations. The predicates $in_{del}(h, v, p, i)$ and $out_{del}(h, v, i)$ describe analogous situations but threads can delete instead of inserting.

The predicates $unk(h, v, i)$ describe that all threads can read and change the key value v.

The predicate $read(h, v)$ describe that a a thread can only read the key value v and the other threads can read and change the same key value.

We have described the interference that each predicate allows and we can see that all possible command combinations are covered for our index module.

Additionally, our shared data structure will have locking and unlocking actions, but they are not visible at the high-level and therefore not captured by the abstract specification. Depending on the implementation there will be more or less actions which modify the shared state, but only the ones which modify the shared state are reflected at the high-level.

Action model

In order to represent the actions which manipulate the abstract resources and at the same time allow modifications dependent of the implementation itself, we we will develop an action model which contains two different regions of tokens. One region of actions which allow manipulation of the key value and another region of actions, which in our particular implementation, will contain the tokens to lock and unlock. When a thread performs an insert or delete operation, it is required to lock the node which it will change. This means that to actually perform a change we need to lock the node that contains the key value.

We first describe some predicates that will help us to describe the action model:

$$\begin{aligned} isLock(x) &\equiv \exists r, \pi\,.\, \boxed{\begin{array}{r}(x \mapsto node(0, v_0, s, v_{i+1}, p_{i+1}) * [UNLOCK(x)]^r_1) \\ \vee\, x \mapsto node(t_{id}, v_0, s, v_{i+1}, p_{i+1})\end{array}}^{r}_{L(r',h)} * [LOCK(x)]^r_\pi \\ Locked(x) &\equiv \exists r\,.\, \boxed{x \mapsto node(t_{id}, v_0, s, v_{i+1}, p_{i+1})}^{r}_{L(r',h)} * [UNLOCK(x)]^r_1 \end{aligned}$$

The abstract predicate $isLock(x)$ describes a node, x, that can be locked. It specifies that the local state contains permission $[LOCK(x)]^r_\pi$ meaning that a thread can acquire the lock for node x. It also asserts that the shared region satisfies that either the node is unlocked and the region holds the full permission to unlock the node, or the lock is locked and the unlocking permission is no longer in the shared state.

The predicate $Locked(x)$ describes that the node x is locked. It is interpreted as the permission assertion $[UNLOCK(x)]^r_1$ in the local state, giving the current thread full permission to unlock the

lock in region r and the shared region assertion stating that the node is locked. Finally, we note that $Locked(x) * Locked(x) \Rightarrow false$.

We now present all the actions permitted on the shared state for the node list module:

$$LOCK(x) : x \mapsto node(0, v_0, s, v_{i+1}, p_{i+1}) * [UNLOCK(x)]_1^r \rightsquigarrow x \mapsto node(t_{id}, v_0, s, v_{i+1}, p_{i+1})$$

$$UNLOCK(x) : x \mapsto node(t_{id}, v_0, s, v_{i+1}, p_{i+1}) \rightsquigarrow x \mapsto node(0, v_0, s, v_{i+1}, p_{i+1}) * [UNLOCK(x)]_1^r$$

$$CHANGE(v) : \forall x \,.\, [MOD(x, v)]_1^{r'} \rightsquigarrow Locked(x) * [CHANGE(v)]_1^{r'}$$

$$MOD(x, v) : \begin{cases} \left(\begin{array}{c} |s| < 2k \wedge Locked(x) * \\ x \mapsto node(t_{id}, v_0, s, v_{i+1}, p_{i+1} \\ * [CHANGE(v)]_1^{r'} \end{array} \right) \rightsquigarrow \left(\begin{array}{c} x \mapsto node(t_{id}, v_0, s \cup \{(v,p)\}, v_{i+1}, p_{i+1}) \\ * [MOD(x, v)]_1^{r'} \end{array} \right) \\ \left(\begin{array}{c} |s| = 2k \wedge Locked(x) * \\ x \mapsto node(t_{id}, v_0, s, v_{i+1}, p_{i+1} * \\ [CHANGE(v)]_1^{r'} \end{array} \right) \rightsquigarrow \left(\begin{array}{c} x \mapsto node(t_{id}, v_0, s', v_0', y) * \\ [MOD(x, v)]_1^{r'} * \\ y \mapsto node(0, v_0', s'', v_{i+1}, p_{i+1}) \\ * [UNLOCK(y)]_1^r * isLock(y) \wedge \\ s' \cup s'' = s \cup \{(v,p)\} \end{array} \right) \\ \left(\begin{array}{c} Locked(x) * \\ x \mapsto node(t_{id}, v_0, s, v_{i+1}, p_{i+1} \\ * [CHANGE(v)]_1^{r'} \end{array} \right) \rightsquigarrow \left(\begin{array}{c} x \mapsto node(t_{id}, v_0, s \backslash \{(v, \_)\}, v_{i+1}, p_{i+1}) \\ * [MOD(x, v)]_1^{r'} \end{array} \right) \\ \left(\begin{array}{c} Locked(x) * \\ [CHANGE(v)]_1^{r'} \end{array} \right) \rightsquigarrow [MOD(x, v)]_1^{r'} \end{cases}$$

$$INSERT(v) : \forall x \,.\, [INS(x, v)]_i^{r'} \rightsquigarrow Locked(x) * [INSERT(v)]_i^{r'}$$

$$INS(x, v) : \begin{cases} \left(\begin{array}{c} |s| < 2k \wedge Locked(x) * \\ x \mapsto node(t_{id}, v_0, s, v_{i+1}, p_{i+1} \\ * [INSERT(v)]_i^{r'} \end{array} \right) \rightsquigarrow \left(\begin{array}{c} x \mapsto node(t_{id}, v_0, s \cup \{(v,p)\}, v_{i+1}, p_{i+1}) \\ * [INS(x, v)]_i^{r'} \end{array} \right) \\ \left(\begin{array}{c} |s| = 2k \wedge Locked(x) * \\ x \mapsto node(t_{id}, v_0, s, v_{i+1}, p_{i+1} \\ * [INSERT(v)]_i^{r'} \end{array} \right) \rightsquigarrow \left(\begin{array}{c} x \mapsto node(t_{id}, v_0, s', v_0', y) * \\ [INS(x, v)]_i^{r'} * \\ y \mapsto node(0, v_0', s'', v_{i+1}, p_{i+1}) * \\ [UNLOCK(y)]_1^r * isLock(y) \wedge \\ s' \cup s'' = s \cup \{(v,p)\} \end{array} \right) \\ \left(\begin{array}{c} Locked(x) * \\ [INSERT(v)]_i^{r'} \end{array} \right) \rightsquigarrow [INS(x, v)]_i^{r'} \end{cases}$$

$$DELETE(v) : \forall x \,.\, [DEL(x, v)]_i^{r'} \rightsquigarrow Locked(x) * [DELETE(v)]_i^{r'}$$

$$DEL(x, v) : \begin{cases} \left(\begin{array}{c} Locked(x) * \\ x \mapsto node(t_{id}, v_0, s, v_{i+1}, p_{i+1}) \\ * [DELETE(v)]_i^{r'} \end{array} \right) \rightsquigarrow \left(\begin{array}{c} x \mapsto node(t_{id}, v_0, s \backslash \{(v, \_)\}, v_{i+1}, p_{i+1}) \\ * [DEL(x, v)]_i^{r'} \end{array} \right) \\ \left(\begin{array}{c} Locked(x) * \\ [DELETE(v)]_i^{r'} \end{array} \right) \rightsquigarrow [DEL(x, v)]_i^{r'} \end{cases}$$

The $LOCK(x)$ action requires that the shared state region contains the unlocked lock for node x and full permission on the token $[UNLOCK(x)]_1^r$ to unlock the lock of the same node. The result of the action is to lock the node and to move the full unlock permission to the thread's local state. By moving $[UNLOCK(x)]_1^r$ permission into the local state it allows the locking thread to release the lock afterwards (no other thread can lock or unlock the node). The local state is not explicitly represented in the action since only the shared state has interference.

The $UNLOCK(x)$ action requires the shared region r to contain the node x locked. By performing the action the node is unlocked and the $[UNLOCK(x)]_1^r$ permission is moved into the

shared state. The thread must start with $[UNLOCK(x)]_1^r$ in its local state in order to move it to the shared state as a result of the action.

Intuitively, the $CHANGE(v)$ actions allow a node to be altered while a thread owns the lock of a node, as long as all changes preserve the list's contents with the exception of the key value-pointer correspondent to v. By giving up the *Locked* permission on a node and the $CHANGE(v)$ token a thread can acquire the $MOD(x, v)$ token which gives it the right to modify node n up to the containment of (v, p). The first $MOD(x, v)$ action simply adds a new key value-pointer to the node x. The second deals with the case when the node x is full and a thread adds new key value-pointer to the node x splitting the node. The third consists of removing a key value-pointer from node x. The final action corresponds to doing nothing. This deals with the case when we the thread is at the wrong node.

The $INSERT(v)$ and $INS(x, v)$ have the same behaviour has $CHANGE(v)$ and $MOD(x, v)$, with the exception that they not require full permission to change the shared state (they require only fractional permission i) and they do not allow the deletion action of the correspondent key value-pointer to the node.

The $DELETE(v)$ and $DEL(x, v)$ serve as the analogous of $INSERT(v)$ and $INS(x, v)$ but no insertions are allowed instead.

We now require interference environments, which will use the actions presented before. We have the following assertions:

$$Def(r', h) \stackrel{\text{def}}{=} \begin{pmatrix} LOCK(x) \\ UNLOCK(x) \\ MOD(x, v) \\ CHANGE(v) \end{pmatrix}$$

$$Ins(r', h) \stackrel{\text{def}}{=} \begin{pmatrix} LOCK(x) \\ UNLOCK(x) \\ INS(x, v) \\ INSERT(v) \end{pmatrix}$$

$$Del(r', h) \stackrel{\text{def}}{=} \begin{pmatrix} LOCK(x) \\ UNLOCK(x) \\ DEL(x, v) \\ DELETE(v) \end{pmatrix}$$

$$Unk(r', h) \stackrel{\text{def}}{=} \begin{pmatrix} LOCK(x) \\ UNLOCK(x) \\ INS(x, v) \\ INSERT(v) \\ DEL(x, v) \\ DELETE(v) \end{pmatrix}$$

$$Read(r', h) \stackrel{\text{def}}{=} \begin{pmatrix} LOCK(x) \\ UNLOCK(x) \\ MOD(x, v) \\ CHANGE(v) \\ INS(x, v) \\ INSERT(v) \\ DEL(x, v) \\ DELETE(v) \end{pmatrix}$$

The concrete interpretation of the predicates for this implementation of the index module are defined as follows:

$$
\begin{aligned}
in_{def}(h,v,p,i) &\equiv \exists S\,.\,\boxed{nodeList(h,S)\wedge(v,p)\in S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_{i} \\
out_{def}(h,v,i) &\equiv \exists S\,.\,\boxed{nodeList(h,S)\wedge(v,\_)\notin S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_{i} \\
in_{ins}(h,v,p,i) &\equiv \exists S,q\,.\,(i<1\wedge(\boxed{nodeList(h,S)\wedge(v,q)\in S}^{r'}_{Ins(r',h)} * [INSERT(v)]^{r'}_{i}) \\
&\vee\ (i=1\wedge\boxed{nodeList(h,S)\wedge(v,p)\in S}^{r'}_{Def(r',h)} * [INSERT(v)]^{r'}_{i} \\
out_{ins}(h,v,i) &\equiv \exists S\,.\,(i<1\wedge\boxed{nodeList(h,S)}^{r'}_{Ins(r',h)} * [INSERT(v)]^{r'}_{i}) \\
&\vee\ (i=1\wedge\boxed{nodeList(h,S)\wedge(v,\_)\notin S}^{r'}_{Def(r',h)} * [INSERT(v)]^{r'}_{i}) \\
in_{del}(h,v,p,i) &\equiv \exists S\,.\,(i<1\wedge\boxed{nodeList(h,S)}^{r'}_{Del(r',h)} * [DELETE(v)]^{r'}_{i}) \\
&\vee\ (i=1\wedge\boxed{nodeList(h,S)\wedge(v,p)\in S}^{r'}_{Def(r',h)} * [DELETE(v)]^{r'}_{i}) \\
out_{del}(h,v,i) &\equiv \exists S\,.\,\boxed{nodeList(h,S)\wedge(v,\_)\notin S}^{r'}_{Del(r',h)} * [DELETE(v)]^{r'}_{i} \\
unk(h,v,i) &\equiv \exists S\,.\,\boxed{nodeList(h,S)}^{r'}_{Unk(r',h)} * [INSERT(v)]^{r'}_{i} * [DELETE(v)]^{r'}_{i} \\
unk_{ins}(h,v,i) &\equiv \exists S\,.\,(i<1\wedge\boxed{nodeList(h,S)}^{r'}_{Unk(r',h)} * [INSERT(v)]^{r'}_{i} * [DELETE(v)]^{r'}_{i} \\
&\vee\ (i=1\wedge\boxed{nodeList(h,S)\wedge(v,\_)\in S}^{r'}_{Def(r',h)} * [INSERT(v)]^{r'}_{i} \\
&\quad * [DELETE(v)]^{r'}_{i}) \\
unk_{del}(h,v,i) &\equiv \exists S\,.\,(i<1\wedge\boxed{nodeList(h,S)}^{r'}_{Unk(r',h)} * [INSERT(v)]^{r'}_{i} * [DELETE(v)]^{r'}_{i}) \\
&\vee\ (i=1\wedge\boxed{nodeList(h,S)\wedge(v,\_)\notin S}^{r'}_{Def(r',h)} * [INSERT(v)]^{r'}_{i} \\
&\quad * [DELETE(v)]^{r'}_{i}) \\
read(h,v) &\equiv \exists S\,.\,\boxed{nodeList(h,S)}^{r'}_{Read(r',h)}
\end{aligned}
$$

The predicates $in_{def}(h,v,p,i)$ and $out_{def}(h,v,i)$ are self-stable. For the only action available to another thread is $CHANGE(v)$ and that requires full permission. The shared assertions are invariant under the action $CHANGE(v)$ without full permission. With full permission the assertion holds as a consequence of the fact that no other thread can perform an action besides locking and unlocking.

The predicates $in_{ins}(h,v,p,i)$ and $out_{ins}(h,v,i)$ are self-stable. For the only action available to another thread is $INSERT(v)$. The shared assertions are invariant under all changes $INSERT(v)$. This is true because the assertion of the shared state is either too weak to specify if there is a value key-pointer pair (v,p) in the index or it asserts that there is. Since there can be no $DELETE(v)$ actions, the assertions are always satisfied. When we have full permission we can specify that there is no value key-pointer pair $(v,\_)$ in the index as there are no actions available to another threads. The predicates $in_{del}(h,v,p,i)$ and $out_{del}(h,v,i)$ are self-stable and the explanation is analogous.

We have that the predicates $unk(h,v,i)$, $unk_{ins}(h,v,i)$ and $unk_{del}(h,v,i)$ are self-stable. For the shared assertions, which only say that there exists a node list, are invariant under all actions. When we have full permission, we can make the shared assertions stronger as a consequence of the fact that no other thread can perform an action.

We have the predicate $read(h,v)$ is self-stable as its shared assertion is always invariant under all actions.

Finally, we explain the axioms which allows us to switch between each environment work. When a thread switches from one environment to another it must have full permission. This means that no other thread can perform an action which changes the shared state. Remember that the $read(h,v)$ predicate can always exist but it does not influence the shared state. Therefore, to make our axioms work, we are forced to give up the tokens of the current environment to get the ones from the new environment. For example, we have that $in_{def}(h,v,p,1) \Leftrightarrow in_{del}(h,v,p,1)$. Since we are at the def environment, we give up on the $[CHANGE(v)]^{r'}_{1}$ token and receive the $[DELETE(v)]^{r'}_{1}$. The other axioms are analogous.

The way the abstract specification is built, it enforces environments and therefore it restricts the interference that the shared state is under. This will make it easier to prove each operation.

4.2.5 Verification

To make sure our implementation is correct we need to check that our algorithms satisfy all cases of our high-level specification. However, since that would be very extensive to show it here, we will omit the majority of the proofs as all they are quite similar.

We start by giving a proof for the SEARCH$(h,\ v)$ operation in an environment where no race conditions occur. The main loop searches through the list checking if our key value is between the minimum value and maximum value of the current node. The thread starts at the left most node, moving to the right node until it finds the correct one. When it does, it simply checks if the value exists and returns it. Since it does not modify any node, it does not interfere with the rest of the state. The proof is as follows:

$\{\ in_{def}(h,v,p,i)\ \}$
$r :=$ SEARCH(h,v) $\{$
$\left\{\ \exists S\ .\ \boxed{nodeList(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_i\ \right\}$
$current := h;$
$\left\{\ \exists S\ .\ \boxed{nodeList(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_i \wedge current = h\ \right\}$
$A :=$ GET$(current);$
$\left\{\begin{array}{l} \exists S, v', v'', s\ .\ \boxed{nodeList(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_i \wedge current = h \\ \wedge\ A = node(\_, v', s, v'', \_) \end{array}\right\}$
MOVERIGHT;
$\left\{\begin{array}{l} \exists S, v', v'', s, h'\ .\ \boxed{nodeList(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_i \wedge current = h' \\ \wedge\ A = node(\_, v', s, v'', \_) \wedge v \leq v'' \end{array}\right\}$
if (ISIN(A,v)) $\{$
$\left\{\begin{array}{l} \exists S, v', v'', s, h'\ .\ \boxed{nodeList(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_i \wedge current = h' \\ \wedge\ A = node(\_, v', s, v'', \_) \wedge v \leq v'' \end{array}\right\}$
$r :=$ LOOKUP$(A,v);$
$\left\{\begin{array}{l} \exists S, v', v'', s, h'\ .\ \boxed{nodeList(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_i \wedge current = h' \\ \wedge\ A = node(\_, v', s, v'', \_) \wedge v \leq v'' \wedge r = p \end{array}\right\}$
$\}$ **else** $\{$
$\{\ false\ \}$
$r :=$ NULL;
$\{\ false\ \}$
$\}$
$\left\{\begin{array}{l} \exists S, v', v'', s, h'\ .\ \boxed{nodeList(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_i \wedge current = h' \\ \wedge\ A = node(\_, v', s, v'', \_) \wedge v \leq v'' \wedge r = p \end{array}\right\}$
$\}$
$\{\ in_{def}(h,v,p,i) \wedge r = p\ \}$

and the proof for the help function MOVERIGHT:

$$\left\{ \begin{array}{l} \exists S, v', v'', s \,.\, \boxed{nodeList(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \wedge current = h \\ \wedge\, A = node(\_, v', s, v'', \_) \end{array} \right\}$$

MOVERIGHT $\triangleq$ {

$$\left\{ \begin{array}{l} \exists S, v', v'', s \,.\, \boxed{nodeList(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \wedge current = h \\ \wedge\, A = node(\_, v', s, v'', \_) \end{array} \right\}$$

while ($v >$ HIGHVALUE(A)) {

$$\left\{ \begin{array}{l} \exists S, v', v'', s, h' \,.\, \boxed{nodeList(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \\ \wedge\, A = node(\_, v', s, v'', \_) \wedge v > v'' \end{array} \right\}$$

$current$:= NEXT(A, v);

$$\left\{ \begin{array}{l} \exists S, v', v'', s, h' \,.\, \boxed{nodeList(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \\ \wedge\, A = node(\_, v', s, v'', \_) \wedge v > v'' \end{array} \right\}$$

A := GET($current$);

$$\left\{ \begin{array}{l} \exists S, v', v'', s, h' \,.\, \boxed{nodeList(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \\ \wedge\, A = node(\_, v', s, v'', \_) \end{array} \right\}$$

}

$$\left\{ \begin{array}{l} \exists S, v', v'', s, h' \,.\, \boxed{nodeList(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \\ \wedge\, A = node(\_, v', s, v'', \_) \wedge v \leq v'' \end{array} \right\}$$

}

$$\left\{ \begin{array}{l} \exists S, v', v'', s, h' \,.\, \boxed{nodeList(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \\ \wedge\, A = node(\_, v', s, v'', \_) \wedge v \leq v'' \end{array} \right\}$$

We now give a proof for the INSERT($h,\ v,\ p$) operation in a similar environment as the search previous proof.

$\{\ out_{def}(h, v, 1)\ \}$

r := INSERT(h, v, p) {

$$\left\{ \exists S, q \,.\, \boxed{nodeList(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \right\}$$

$current := h$;

$$\left\{ \exists S, q \,.\, \boxed{nodeList(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \wedge current = h \right\}$$

$completed$:= FALSE;

$$\left\{ \begin{array}{l} \exists S, q \,.\, \boxed{nodeList(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \wedge current = h \\ \wedge\, completed = false \end{array} \right\}$$

$found$:= FALSE;

$$\left\{ \begin{array}{l} \exists S, q \,.\, \boxed{nodeList(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \wedge current = h \\ \wedge\, completed = false \wedge found = false \end{array} \right\}$$

while ($found$ = FALSE) {

$$\left\{ \begin{array}{l} \exists S, q \,.\, \boxed{nodeList(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \\ \wedge\, completed = false \wedge found = false \end{array} \right\}$$

$found$:= TRUE;

$$\left\{ \begin{array}{l} \exists S, q \,.\, \boxed{nodeList(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \\ \wedge\, completed = false \wedge found = true \end{array} \right\}$$

LOCK($current$);

$$\left\{ \begin{array}{l} \exists S, q \,.\, \boxed{nodeList(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \\ \wedge\, completed = false \wedge found = true * Locked(current) \end{array} \right\}$$

A := GET($current$);

$$\left\{ \begin{array}{l} \exists S, q, h', v', v'', s \,.\, \boxed{nodeList(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \\ \wedge\, completed = false \wedge found = true * Locked(current) \wedge A = node(t_{id}, v', s, v'', \_) \end{array} \right\}$$

if (ISIN(A, v)) {

$\{\ false\ \}$

UNLOCK($current$);

$\{\ false\ \}$

r := FALSE;

$\{\ false\ \}$

$completed$:= TRUE;

$\{\ false\ \}$

$\{\ false\ \}$

} **else if** ($v >$ HIGHVALUE(A)) {

$\left\{\ \exists S, q, h', v', v'', s\ .\ \boxed{nodeList(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \right.$
$\left.\ \wedge\ completed = false \wedge found = true * Locked(current) \wedge A = node(t_{id}, v', s, v'', \_) \wedge v > v''\ \right\}$

UNLOCK($current$);

$\left\{\ \exists S, q, h', v', v'', s\ .\ \boxed{nodeList(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \right.$
$\left.\ \wedge\ completed = false \wedge found = true \wedge A = node(t_{id}, v', s, v'', \_) \wedge v > v''\ \right\}$

$found$:= FALSE;

$\left\{\ \exists S, q, h', v', v'', s\ .\ \boxed{nodeList(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \right.$
$\left.\ \wedge\ completed = false \wedge found = false \wedge A = node(t_{id}, v', s, v'', \_) \wedge v > v''\ \right\}$

MOVERIGHT;

$\left\{\ \exists S, q, h', v', v'', s\ .\ \boxed{nodeList(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \right.$
$\left.\ \wedge\ completed = false \wedge found = false \wedge A = node(t_{id}, v', s, v'', \_) \wedge v \leq v''\ \right\}$

}

$\left\{\ (\exists S, q, h', v', v'', s\ .\ \boxed{nodeList(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \right.$
$\left.\ \wedge\ completed = false \wedge found = false \wedge A = node(t_{id}, v', s, v'', \_) \wedge v \leq v'')\ \right\}$

$\vee$

$\left\{\ (\exists S, q, h', v', v'', s\ .\ \boxed{nodeList(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \right.$
$\left.\ \wedge\ completed = false \wedge found = true * Locked(current) \wedge A = node(t_{id}, v', s, v'', \_) \wedge v \leq v'')\ \right\}$

}

$\left\{\ \exists S, q, h', v', v'', s\ .\ \boxed{nodeList(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \right.$
$\left.\ \wedge\ completed = false \wedge found = true * Locked(current) \wedge A = node(t_{id}, v', s, v'', \_) \wedge v \leq v''\ \right\}$

if ($completed$ = FALSE) {

$\left\{\ \exists S, q, h', v', v'', s\ .\ \boxed{nodeList(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \right.$
$\left.\ \wedge\ completed = false \wedge found = true * Locked(current) \wedge A = node(t_{id}, v', s, v'', \_) \wedge v \leq v''\ \right\}$

if (ISSAFE(A)) {

$\left\{\ \exists S, q, h', v', v'', s\ .\ \boxed{nodeList(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \right.$
$\left.\ \wedge\ completed = false \wedge found = true * Locked(current) \wedge A = node(t_{id}, v', s, v'', \_) \wedge v \leq v'' \right.$
$\left.\ \wedge\ |s| < 2k\ \right\}$

INSERTINTOSAFE;

$\left\{\ \exists S, q\ .\ \boxed{nodeList(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1\ \right\}$

} **else** {

$\left\{\ \exists S, q, h', v', v'', s\ .\ \boxed{nodeList(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \right.$
$\left.\ \wedge\ completed = false \wedge found = true * Locked(current) \wedge A = node(t_{id}, v', s, v'', \_) \wedge v \leq v'' \right.$
$\left.\ \wedge\ |s| = 2k\ \right\}$

INSERTINTOUNSAFE;

$\left\{\ \exists S, q\ .\ \boxed{nodeList(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1\ \right\}$

}

$\left\{\ \exists S, q\ .\ \boxed{nodeList(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1\ \right\}$

r := TRUE;

$\left\{\ \exists S, q\ .\ \boxed{nodeList(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \wedge r = true\ \right\}$

}

$\left\{\ \exists S, q\ .\ \boxed{nodeList(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \wedge r = true\ \right\}$

}

$\{\ in_{def}(h, v, p, 1) \wedge r = true\ \}$

and the proof for the helper function INSERTINTOSAFE:

$$\left\{ \begin{array}{l} \exists S, q, h', v', v'', s \,.\, \boxed{nodeList(h, S) \wedge (v, q) \notin S}^{r'}_{Def(r', h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \\ \wedge\, completed = false \wedge found = true * Locked(current) \wedge A = node(t_{id}, v', s, v'', \_) \wedge v \leq v'' \\ \wedge\, |s| < 2k \end{array} \right\}$$

INSERTINTOSAFE $\triangleq$ {

$$\left\{ \begin{array}{l} \exists S, q, h', v', v'', s \,.\, \boxed{nodeList(h, S) \wedge (v, q) \notin S}^{r'}_{Def(r', h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \\ \wedge\, completed = false \wedge found = true * Locked(current) \wedge A = node(t_{id}, v', s, v'', \_) \wedge v \leq v'' \\ \wedge\, |s| < 2k \end{array} \right\}$$

ADDPAIR(A, v, p);

$$\left\{ \begin{array}{l} \exists S, q, v', h', v'', s \,.\, \boxed{nodeList(h, S) \wedge (v, q) \notin S}^{r'}_{Def(r', h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \\ \wedge\, completed = false \wedge found = true * Locked(current) \wedge A = node(t_{id}, v', s \cup (v, p), v'', \_) \wedge v \leq v'' \end{array} \right\}$$

PUT$(A, current)$;

$$\left\{ \begin{array}{l} \exists S, h', v', v'', s \,.\, \boxed{nodeList(h, S) \wedge (v, p) \in S}^{r'}_{Def(r', h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \\ \wedge\, completed = false \wedge found = true * Locked(current) \wedge A = node(t_{id}, v', s \cup (v, p), v'', \_) \wedge v \leq v'' \end{array} \right\}$$

UNLOCK$(current)$;

$$\left\{ \exists S \,.\, \boxed{nodeList(h, S) \wedge (v, p) \in S}^{r'}_{Def(r', h)} * [CHANGE(v)]^{r'}_1 \right\}$$

}

$$\left\{ \exists S \,.\, \boxed{nodeList(h, S) \wedge (v, p) \in S}^{r'}_{Def(r', h)} * [CHANGE(v)]^{r'}_1 \right\}$$

and the proof for the helper function INSERTINTOUNSAFE:

$$\left\{ \begin{array}{l} \exists S, q, h', v', v'', s \,.\, \boxed{nodeList(h, S) \wedge (v, q) \notin S}^{r'}_{Def(r', h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \\ \wedge\, completed = false \wedge found = true * Locked(current) \wedge A = node(t_{id}, v', s, v'', \_) \wedge v \leq v'' \\ \wedge\, |s| = 2k \end{array} \right\}$$

INSERTINTOUNSAFE $\triangleq$ {

$$\left\{ \begin{array}{l} \exists S, q, h', v', v'', s \,.\, \boxed{nodeList(h, S) \wedge (v, q) \notin S}^{r'}_{Def(r', h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \\ \wedge\, completed = false \wedge found = true * Locked(current) \wedge A = node(t_{id}, v', s, v'', \_) \wedge v \leq v'' \\ \wedge\, |s| = 2k \end{array} \right\}$$

$q :=$ NEW();

$$\left\{ \begin{array}{l} \exists S, q, h', v', v'', s \,.\, \boxed{nodeList(h, S) \wedge (v, q) \notin S}^{r'}_{Def(r', h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \\ \wedge\, completed = false \wedge found = true * Locked(current) \wedge A = node(t_{id}, v', s, v'', \_) \wedge v \leq v'' \\ \wedge\, |s| = 2k \wedge q \mapsto node(0, \_, \_, \_, \_) \end{array} \right\}$$

$B :=$ REARRANGE(A, v, p, q);

$$\left\{ \begin{array}{l} \exists S, q, h', v', v'', v''', s, s', s'' \,.\, \boxed{nodeList(h, S) \wedge (v, q) \notin S}^{r'}_{Def(r', h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \\ \wedge\, completed = false \wedge found = true * Locked(current) \wedge A = node(t_{id}, v', s', v''', q) \wedge v \leq v'' \\ \wedge\, q \mapsto node(0, \_, \_, \_, \_) \wedge B = node(0, v''', s'', v'', \_) \wedge s \cup \{(v, p)\} = s' \cup s'' \end{array} \right\}$$

PUT(B, q);

$$\left\{ \begin{array}{l} \exists S, q, h', v', v'', v''', s, s', s'' \,.\, \boxed{nodeList(h, S) \wedge (v, q) \notin S}^{r'}_{Def(r', h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \\ \wedge\, completed = false \wedge found = true * Locked(current) \wedge A = node(t_{id}, v', s', v''', q) \wedge v \leq v'' \\ \wedge\, q \mapsto B \wedge B = node(0, v''', s'', v'', \_) \wedge s \cup \{(v, p)\} = s' \cup s'' \end{array} \right\}$$

PUT$(A, current)$;

$$\left\{ \begin{array}{l} \exists S, q, h', v', v'', v''', s, s', s'' \,.\, \boxed{nodeList(h, S) \wedge (v, q) \in S}^{r'}_{Def(r', h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \\ \wedge\, completed = false \wedge found = true * Locked(current) \end{array} \right\}$$

UNLOCK$(current)$;

$$\left\{ \begin{array}{l} \exists S, q, h', v', v'', v''', s, s', s'' \,.\, \boxed{nodeList(h, S) \wedge (v, q) \in S}^{r'}_{Def(r', h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \\ \wedge\, completed = false \wedge found = true \end{array} \right\}$$

}

$$\left\{ \exists S, q \,.\, \boxed{nodeList(h, S) \wedge (v, p) \in S}^{r'}_{Def(r', h)} * [CHANGE(v)]^{r'}_1 \right\}$$

Finally, we give a proof for the DELETE$(h,\ v)$ operation also in an environment where no race conditions occur.

$\{\ in_{def}(h, v, p, 1)\ \}$
$r :=$ DELETE(h, v) $\{$
$\left\{\ \exists S\ .\ \boxed{nodeList(h, S) \wedge (v, p) \in S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1\ \right\}$
$current := h;$
$\left\{\ \exists S\ .\ \boxed{nodeList(h, S) \wedge (v, p) \in S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \wedge current = h\ \right\}$
$found :=$ FALSE;
$\left\{\ \exists S\ .\ \boxed{nodeList(h, S) \wedge (v, p) \in S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \wedge current = h \wedge found = false\ \right\}$
while $(found =$ FALSE$)$ $\{$
$\left\{\ \exists S, h'\ .\ \boxed{nodeList(h, S) \wedge (v, p) \in S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \wedge found = false\ \right\}$
$found :=$ TRUE;
$\left\{\ \exists S, h'\ .\ \boxed{nodeList(h, S) \wedge (v, p) \in S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \wedge found = true\ \right\}$
LOCK$(current)$;
$\left\{\ \exists S, h'\ .\ \boxed{nodeList(h, S) \wedge (v, p) \in S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \wedge found = true * Locked(current)\ \right\}$
$A :=$ GET$(current)$;
$\left\{\ \exists S, v', v'', s, h'\ .\ \boxed{nodeList(h, S) \wedge (v, p) \in S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \wedge found = true * Locked(current) \wedge A = node(t_{id}, v', s, v'',\ )\ \right\}$
if (ISIN(A, v)) $\{$
$\left\{\ \exists S, v', v'', s, h'\ .\ \boxed{nodeList(h, S) \wedge (v, p) \in S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \wedge found = true * Locked(current) \wedge A = node(t_{id}, v', s, v'',\ ) \wedge (v, \_) \in s\ \right\}$
REMOVEPAIR(A, v);
$\left\{\ \exists S, v', v'', s, h'\ .\ \boxed{nodeList(h, S) \wedge (v, p) \in S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \wedge found = true * Locked(current) \wedge A = node(t_{id}, v', s \backslash (v, \_), v'',\ ) \wedge (v, \_) \notin s\ \right\}$
PUT$(A, current)$;
$\left\{\ \exists S, v', v'', s, h'\ .\ \boxed{nodeList(h, S) \wedge (v, \_) \notin S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \wedge found = true * Locked(current)\ \right\}$
UNLOCK$(current)$;
$\left\{\ \exists S, v', v'', s, h'\ .\ \boxed{nodeList(h, S) \wedge (v, \_) \notin S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \wedge found = true\ \right\}$
$r :=$ TRUE;
$\left\{\ \exists S, v', v'', s, h'\ .\ \boxed{nodeList(h, S) \wedge (v, \_) \notin S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \wedge found = true \wedge r = true\ \right\}$
$\}$ **else** $\{$
$\left\{\ \exists S, v', v'', s, h'\ .\ \boxed{nodeList(h, S) \wedge (v, p) \in S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \wedge found = true * Locked(current) \wedge A = node(t_{id}, v', s, v'',\ ) \wedge (v, \_) \notin s\ \right\}$
UNLOCK$(current)$;
$\left\{\ \exists S, v', v'', s, h'\ .\ \boxed{nodeList(h, S) \wedge (v, p) \in S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \wedge found = true \wedge A = node(t_{id}, v', s, v'',\ ) \wedge (v, \_) \notin s\ \right\}$
if $(v >$ HIGHVALUE$(A))$ $\{$
$\left\{\ \exists S, v', v'', s, h'\ .\ \boxed{nodeList(h, S) \wedge (v, p) \in S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_1 \wedge current = h' \wedge found = true \wedge A = node(t_{id}, v', s, v'',\ ) \wedge (v, \_) \notin s \wedge v > v''\ \right\}$

$$\left\{ \begin{array}{l} \exists S, v', v'', s, h' \,.\, \boxed{nodeList(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_{1} \wedge current = h' \\ \wedge\, found = true \wedge A = node(t_{id}, v', s, v'',) \wedge (v, \_) \notin s \wedge v > v'' \end{array} \right\}$$

found := FALSE;

$$\left\{ \begin{array}{l} \exists S, v', v'', s, h' \,.\, \boxed{nodeList(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_{1} \wedge current = h' \\ \wedge\, found = false \wedge A = node(t_{id}, v', s, v'',) \wedge (v, \_) \notin s \wedge v > v'' \end{array} \right\}$$

moveRight;

$$\left\{ \begin{array}{l} \exists S, v', v'', s, h' \,.\, \boxed{nodeList(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_{1} \wedge current = h' \\ \wedge\, found = false \wedge A = node(t_{id}, v', s, v'',) \wedge v \leq v'' \end{array} \right\}$$

} **else** {

$$\{\ false\ \}$$

r := FALSE;

$$\{\ false\ \}$$

}

$$\left\{ \begin{array}{l} \exists S, v', v'', s, h' \,.\, \boxed{nodeList(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_{1} \wedge current = h' \\ \wedge\, found = false * \wedge A = node(t_{id}, v', s, v'',) \wedge v \leq v'' \end{array} \right\}$$

}

$$\left\{ \begin{array}{l} (\exists S, v', v'', s, h' \,.\, \boxed{nodeList(h,S) \wedge (v,\_) \notin S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_{1} \wedge current = h' \\ \wedge\, found = true \wedge r = true) \\ \vee\, (\exists S, v', v'', s, h' \,.\, \boxed{nodeList(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_{1} \wedge current = h' \\ \wedge\, found = false \wedge A = node(t_{id}, v', s, v'',) \wedge v \leq v'') \end{array} \right\}$$

}

$$\left\{ \exists S \,.\, \boxed{nodeList(h,S) \wedge (v,\_) \notin S}^{r'}_{Def(r',h)} * [CHANGE(v)]^{r'}_{1} \wedge r = true \right\}$$

}

$$\{\ out_{def}(h, v, 1) \wedge r = true\ \}$$

4.3 B^{Link} tree

We have seen how to give a correct implementation of our high-level specification. The same process can be applied to other implementations. We will describe how to extend the ideas applied of the node list to a B^{Link} tree implementation which is described in section 2.1.

4.3.1 Model

In this section we extend the previous model. We formalise the B^{Link} tree using separation logic. Our program state also consists of a heap and a variable store.

Formalising the data structure

In the node list there existed a single type of node, in the B^{Link} tree we consider two types:

- Leaf nodes exist at the fringe of the tree (also known as level one or lowest level) and contain pointers to data. The nodes in the node list can be seen as leaf nodes.
- Intermediate nodes are all the nodes in the tree which are not at the fringe (there are none if the height of the tree is one). They contain pointers to other nodes.

We simply call them nodes when we do not know or care which kind they are.

Definition 13 (Node predicate)**.** *We use the predicates for a leaf node, an intermediate node and a generic node, respectively, as:*

$$leaf(t_{id}, v_0, s, v_{i+1}, p_{i+1})$$
$$inner(t_{id}, v_0, p_0, s, v_{i+1}, p_{i+1})$$
$$node(t_{id}, v_0, s, v_{i+1}, p_{i+1})$$

In a leaf node, all of the pointers in this list are to data entries, whilst in an inner node all of these pointers are to other tree nodes.

The value $t_{id} \in T$ tells us if the node has been locked by a thread. When zero, the node is unlocked, otherwise it has the thread identifier of the locking thread. The values v_0 and v_{i+1} are the lower bound and upper bound, respectively, on the entries in the node. p_0 is the pointer to a subtree with values less than the node, which in the leaf case, is a null pointer. p_{i+1} is the pointer to the next node at the current level of the tree. This pointer is also called a link. It may be null if the node is the rightmost node of its level. Finally s is a list of value-pointer pairs.

Assume k to be a known constant integer value, greater than zero. We let s, the list of value-pointer pairs in a node have at most $2k$ pairs. This means that a leaf node can have at most $2k$ pairs, where each pointer in the pair points to data. An inner node can have up to $2k + 1$ pairs, where each pointer in the pair points to a node in the lower level. Respectively the minimum number of pairs is zero and one.

A B^{Link} tree is a superimposed structure. It contains a tree and at the same time a linked list at each level. At leaf level, the linked list will be of leaf nodes, while at the other levels, they will be intermediate nodes. We should take into account that these linked lists always have at least one element and that the first element will have minimum value $-\infty$ (and the last node always has maximum value $+\infty$). Each node that belongs to these linked lists will be disjoint in the heap. Therefore we can use separation logic to model these structures and define them precisely.

Definition 14 (Leaf list)**.** *Let* $top = [(min_1, a_1), \ldots, (min_m, a_m)]$ *and* $data = [(k_1, v_1), \ldots, (k_n, v_n)]$ *be two value-pointer lists, then:*

$$\begin{array}{rcl} leafList(top, data) & \stackrel{def}{=} & \exists s_1, \ldots, s_m \,.\; \circledast_{i=1}^{m-1} a_i \mapsto leaf(\_, min_i, s_i, min_{i+1}, a_{i+1}) \\ & * & a_m \mapsto leaf(\_, min_m, s_m, +\infty, null) \\ & \wedge & min_1 = -\infty \\ & \wedge & data = \bigcup_{i=1}^{m} s_i \\ & \wedge & m > 0 \end{array}$$

The *leafList* predicate takes two parameters, *top* and *data*. The first parameter contains an arbitrary number of value-pointer pairs such that for each pair the value corresponds to the minimum value v_0 of a leaf node and the correspondent pointer gives a reference to that node. We take into account that the maximum value of a leaf node is always equal to the minimum value of the next node of the linked list. If there is no next node, i.e. the node is the last element of the list, then the maximum value is $+\infty$. The second parameter contains the concatenation of all value-pointer lists of each node in the linked list. We can deduce the correct node location of each value-pair in that list because the value must be greater than the node minimum value and less than or equal to the node maximum value. Implicitly the predicate implies that each node does not overflow its capacity.

Definition 15 (Inner list)**.** *Let* $top = [(min_1, a_1), \ldots, (min_m, a_m)]$ *and* $data = [(k_1, q_1), \ldots, (k_n, q_n)]$ *be two value-pointer lists, then:*

$$\begin{array}{rcl} innerList(top, data) & \stackrel{def}{=} & \exists s_1, \ldots, s_m \,.\; \circledast_{i=1}^{m-1} a_i \mapsto inner(\_, min_i, p_i, s_i, min_{i+1}, a_{i+1}) \\ & * & a_m \mapsto inner(\_, min_m, p_m, s_m, +\infty, null) \\ & \wedge & min_1 = -\infty \\ & \wedge & data = \bigcup_{i=1}^{m} ((min_i, p_i) \cup s_i) \\ & \wedge & m > 0 \end{array}$$

The inner list is very similar to the leaf list. However each inner node can accommodate at most one more value-pointer pair than a leaf node. That extra pair value corresponds to the minimum value and the pointer references a node at a lower level, which itself has the same minimum value.

We wish to define a predicate to represent the whole tree. Instead of considering the typical view of a tree as a node that itself points to several subtrees, we consider the tree as a series of linked lists where some nodes have pointers to other linked lists at a lower level of the tree. This view of the tree was considered to allow larger flexibility. Even when considering a single thread modifying the tree, there will be certain points in time where the tree cannot be defined simply

as an n-ary tree, as there will be nodes which are not referenced by a node in a higher level, but only from a node on the left at the same level. This happens, for example, when a node has to split. When considering several threads performing these kinds of changes in a tree, depending on the scheduler, one could consider the tree as a root referencing only a few elements of a very large linked list. Therefore, the logic predicate should be able to express this.

Definition 16 (Tree)**.** *We define a tree by induction over n as follows:*

$$\begin{array}{rcl} tree_1(pb, x : top, data) & \stackrel{def}{=} & \exists p \,.\, leafList(x : top, data) \\ & \wedge & x = (-\infty, p) \\ & \wedge & pb = [p] \end{array}$$

$$\begin{array}{rcl} tree_{n+1}(p : ps, x : top, data) & \stackrel{def}{=} & \exists l, l' \,.\, innerList(x : top, l) \\ & * & tree_n(ps, l', data) \\ & \wedge & x = (-\infty, p) \\ & \wedge & l \leq l' \end{array}$$

Where *pb* is the prime block, it is a list of pointers to each leftmost node in the tree. $x : top$ corresponds to a value-pointer list with at least one element. That first element's pointer is the address of the leftmost node at level n, and the prime block contains a pointer to it. A node is considered to be the root if the first element in the prime block references it. This is one possible interpretation, we discuss this in detail in section 4.3.4. The variable *data* is a value-pointer list which is the concatenation of value-pointer pairs at the fringe of the tree. We can see this forms a set from an abstract point of view. If a value-pointer pair is in that set, then the tree which implements that set, will contain some equivalent pair.

As a consequence of $l \leq l'$, we implicitly make each pointer p_0 of a intermediate node in an inner list to point to a node which has the same minimum value as itself. If that wasn't the case, then the condition $l \leq l'$ would be false. With these predicates we can now define a complete B^{Link} tree as follows:

Definition 17 (B^{Link} tree)**.** *Let h be the constant or global address of the prime block. The B^{Link} tree predicate is defined as follows:*

$$\begin{array}{rcl} BTree(h, S) & \stackrel{def}{=} & \exists pb, n, l \,.\, tree_n(pb, l, S) \\ & * & h \mapsto pb \end{array}$$

Finally we can state that the current number of levels in the tree will be equal to the number of pointers in the prime block. Also the root will be the first element in that list. The prime block address is constant and known by every thread. Its contents are allowed to change, but by analysis we can trivially deduce that if an element is in the prime block it will remain there forever. The prime block can only increase the number of elements. They are added at the beginning of the list, which means that the root can change, although all the leftmost nodes' addresses remain the same.

Program state

The program state consists of a working heap h and a variable store σ, which are extensions from the ones used to the node list.

Definition 18 (Heap)**.** *A heap h is a finite partial function that maps heap addresses to node data:*

$$h : \quad (Addr \rightharpoonup_{fin} Node) \times (Addr \rightharpoonup_{fin} PrimeBlock)$$

The heap is divided up in a way that each thread has a private section of the heap for internal use and there exists a single area that is shared by all threads which contain the B^{Link} tree.

The shared state is located in the heap and consists of the prime block and the tree which it points to. If a node is reachable from the prime block, it is part of the shared state. One should

notice that a node can be in the heap, even containing pointers to the tree structure, without belonging to the shared state, if it is not reachable from the prime block.

The local state of a thread is made of its private section. It can be described as every accessible node, by the thread, in the heap excluding the shared state. This will be all nodes allocated in the heap before a thread modifies the tree to point to them.

Definition 19 (Store)**.** *A store σ is a set of finite partial functions that map integer variables $Var_{\mathbb{Z}} = \{i, j, \dots\}$ to integers, address variables $Var_{Addrs} = \{p, q, \dots\}$ to heap addresses, boolean variables $Var_{\mathbb{B}} = \{b, \dots\}$ to boolean values, node data variables $Var_{Node} = \{A, B, \dots\}$ to node data, prime block variables $Var_{PrimeBlock} = \{PB, \dots\}$ to prime block data and stack variables $Var_{Stack} = \{stack, \dots\}$ to stack data:*

$$\begin{aligned}\sigma : \ & (Var_{\mathbb{Z}} \rightharpoonup_{fin} \mathbb{Z}) \times (Var_{Addr} \rightharpoonup_{fin} Addr \cup \{null\}) \times (Var_{\mathbb{B}} \rightharpoonup_{fin} \mathbb{B}) \\ & \times (Var_{Node} \rightharpoonup_{fin} Node) \times (Var_{Addr} \rightharpoonup PrimeBlock) \times (Var_{Stack} \rightharpoonup Stack)\end{aligned}$$

4.3.2 Programming language

As before we will use a simple and general language which will be an extension of the language used at the node list.

Heap commands

We extend the heap commands , given for the node list (see section 4.2.2), to allow the manipulation of the prime block as following:

$C_H ::=$	$\dots$	
	$PB :=$ getPrimeBlock(h)	Read prime block operation
	putPrimeBlock(h, PB)	Write prime block operation

The new commands are analogous to the read node and write node operations, but instead of dealing with tree nodes, they deal with the prime block. The main difference, which is a technical detail, is that a thread should have the lock on the current root when rewriting the contents of the prime block, but this condition has to be assured by the thread which uses the command.

Store commands

We extend the store commands, given for the node list (see section 4.2.2), with additional node commands and manipulation of the prime block and a stack of pointers.

The commands which read and change a prime block on a store are defined as follows:

$C_{PB} ::=$	$r :=$ root(PB)	Return the first element in the prime block PB
	addRoot(PB, r)	Add address r to the prime block PB
	$q :=$ getNodeLevel$(PB, level)$	Read leftmost node address at level *level*
	$b :=$ isRoot(PB, v)	Test if node with address v is the root of the prime block PB

A prime block in a store can be changed only by adding address elements to its head. All the other operations consist only of reading the prime block or testing its contents. The command getNodeLevel(PB, *level*) returns the element at position *level* if we started counting from the last element of the list (recall the leaf level is level one).

We now give the list of new commands that operate on nodes in the store:

$C_N ::=$	$\dots$	
	$A :=$ newNode(w, p, v, q, u)	Creates a node
	$b :=$ isLeaf(A)	Tests if the node A is a leaf

The command newNode(w, p, v, q, u) returns a new node in the store, with size $2k$. Its minimum and maximum value are equal to w and u respectively. The pointer to the node at the lower level,

with the same minimum value, will be equal to p (this command is not used to create leaf nodes). The pointer to the node on the right will be null and the value-pointer list in the node contains a pointer with the pair $(v,\ q)$.

The command $\textsc{isLeaf}(A)$ simply needs to check if the pointer to the node at the lower level, with the same minimum value, is null. If so, then it is a leaf node, otherwise it is an intermediate node.

We provide a set of operations to manage a stack of pointers in the store, which are:

$C_S ::=$	$stack := \textsc{newStack}()$	Create new stack
	$\textsc{push}(stack, v)$	Add element v to the top of the stack
	$v := \textsc{pop}(stack)$	Remove top element of the stack
	$b := \textsc{isEmpty}(stack)$	Test if the stack is empty

A stack is very similar to the prime block, i.e. both are lists of pointers. Conceptually we will assume the stack to be a list of pointers, but unlike the prime block, it allows the removal of elements.

4.3.3 Language commands

Finally we extend the concurrent imperative programming language with the following commands:

$C ::=$	$\ldots$	
	C_{PB}	Store prime block command
	C_S	Store stack command

Command axioms

We will present new formal specification for each command in order to avoid confusion between the *node* and *leaf* predicate. The axioms for the heap and store commands are the following:

$$\{\ emp\ \}\quad q := \textsc{new}()\quad \left\{\begin{array}{l} q = x \wedge x \mapsto N \\ \wedge\ N = node(0, 0, \emptyset, 0, null)\end{array}\right\}$$

$$\left\{\begin{array}{l} x \mapsto N \\ \wedge\ N = node(t, v_0, s, v_{i+1}, p_{i+1})\end{array}\right\}\quad A := \textsc{get}(x)\quad \left\{\begin{array}{l} x \mapsto N \\ \wedge\ N = node(t, v_0, s, v_{i+1}, p_{i+1}) \\ \wedge\ A = N\end{array}\right\}$$

$$\left\{\begin{array}{l} x \mapsto N \\ \wedge\ N = node(t, v_0, s, v_{i+1}, p_{i+1}) \\ \wedge\ A = M \\ \wedge\ M = node(t', v'_0, s', v'_{i+1}, p'_{i+1})\end{array}\right\}\quad \textsc{put}(A, x)\quad \left\{\begin{array}{l} x \mapsto M \wedge A = M \\ \wedge\ M = node(t', v'_0, s', v'_{i+1}, p'_{i+1})\end{array}\right\}$$

$$\left\{\begin{array}{l} x \mapsto N \\ \wedge\ N = node(0, v_0, s, v_{i+1}, p_{i+1})\end{array}\right\}\quad \textsc{lock}(x)\quad \left\{\begin{array}{l} x \mapsto N \\ \wedge\ N = node(t_{id}, v_0, s, v_{i+1}, p_{i+1})\end{array}\right\}$$

$$\left\{\begin{array}{l} x \mapsto N \\ \wedge\ N = node(t_{id}, v_0, s, v_{i+1}, p_{i+1})\end{array}\right\}\quad \textsc{unlock}(x)\quad \left\{\begin{array}{l} x \mapsto N \\ \wedge\ N = node(0, v_0, s, v_{i+1}, p_{i+1})\end{array}\right\}$$

$$\{\ h \mapsto N \wedge N = p : ps\ \}\quad PB := \textsc{getPrimeBlock}(h)\quad \left\{\begin{array}{l} h \mapsto N \wedge N = p : ps \\ \wedge\ PB = N\end{array}\right\}$$

$$\left\{\begin{array}{l} h \mapsto N \wedge N = p : ps \\ \wedge\ PB = M \wedge M = q : qs\end{array}\right\}\quad \textsc{putPrimeBlock}(h, PB)\quad \left\{\begin{array}{l} h \mapsto M \wedge PB = M \\ \wedge\ M = q : qs\end{array}\right\}$$

$$\{\ emp \wedge PB = p : ps\ \}\quad r := \textsc{root}(PB)\quad \{\ emp \wedge PB = p : ps \wedge r = p\ \}$$

$$\{\ emp \wedge PB = ps \wedge r = p\ \}\quad \textsc{addRoot}(PB, r)\quad \{\ emp \wedge PB = p : ps \wedge r = p\ \}$$

$$\left\{\begin{array}{l} emp \wedge PB = [p_m, \ldots, p_1] \\ \wedge\ level = n \wedge 1 \leq n \leq m\end{array}\right\}\quad q := \textsc{getNodeLevel}(PB, level)\quad \left\{\begin{array}{l} emp \wedge PB = [p_m, \ldots, p_1] \\ \wedge\ level = n \wedge 1 \leq n \leq m \wedge q = p_n\end{array}\right\}$$

$$\{\ emp \wedge PB = p : ps \wedge v = p\ \}\quad b := \textsc{isRoot}(PB, v)\quad \{\ emp \wedge PB = p : ps \wedge v = p \wedge b = true\ \}$$

$$\left\{\begin{array}{l} emp \wedge PB = p : ps \wedge v = q \\ \wedge\ p \neq q \end{array}\right\}\quad b := \textsc{isRoot}(PB, v)\quad \left\{\begin{array}{l} emp \wedge PB = p : ps \wedge v = q \\ \wedge\ p \neq q \wedge b = false \end{array}\right\}$$

$$\left\{\begin{array}{l} emp \wedge w = v_0 \wedge p = p_0 \wedge v = v_1 \\ \wedge\ q = p_1 \wedge u = v_{i+1} \end{array}\right\}\quad A := \textsc{newNode}(w, p, v, q, u)\quad \left\{\begin{array}{l} emp \wedge w = v_0 \wedge p = p_0 \wedge v = v_1 \\ \wedge\ q = p_1 \wedge u = v_{i+1} \\ \wedge\ A = inner(0, v_0, p_0, s, v_{i+1}, null) \\ \wedge\ s = (v_1, p_1) \end{array}\right\}$$

$$\left\{\begin{array}{l} emp \\ \wedge\ A = node(t, v_0, s, v_{i+1}, p_{i+1}) \end{array}\right\}\quad v := \textsc{lowValue}(A)\quad \left\{\begin{array}{l} emp \\ \wedge\ A = node(t, v_0, s, v_{i+1}, p_{i+1}) \\ \wedge\ v = v_0 \end{array}\right\}$$

$$\left\{\begin{array}{l} emp \\ \wedge\ A = node(t, v_0, s, v_{i+1}, p_{i+1}) \end{array}\right\}\quad v := \textsc{highValue}(A)\quad \left\{\begin{array}{l} emp \\ \wedge\ A = node(t, v_0, s, v_{i+1}, p_{i+1}) \\ \wedge\ v = v_{i+1} \end{array}\right\}$$

$$\left\{\begin{array}{l} emp \\ \wedge\ A = node(t, v_0, s, v_{i+1}, p_{i+1}) \\ \wedge\ s = [(v_1, p_1), \ldots, (v_i, p_i)] \\ \wedge\ i < 2k \end{array}\right\}\quad b := \textsc{isSafe}(A)\quad \left\{\begin{array}{l} emp \\ \wedge\ A = node(t, v_0, s, v_{i+1}, p_{i+1}) \\ \wedge\ s = [(v_1, p_1), \ldots, (v_i, p_i)] \\ \wedge\ i < 2k \\ \wedge\ b = true \end{array}\right\}$$

$$\left\{\begin{array}{l} emp \\ \wedge\ A = node(t, v_0, s, v_{i+1}, p_{i+1}) \\ \wedge\ s = [(v_1, p_1), \ldots, (v_i, p_i)] \\ \wedge\ i = 2k \end{array}\right\}\quad b := \textsc{isSafe}(A)\quad \left\{\begin{array}{l} emp \\ \wedge\ A = node(t, v_0, s, v_{i+1}, p_{i+1}) \\ \wedge\ s = [(v_1, p_1), \ldots, (v_i, p_i)] \\ \wedge\ i = 2k \\ \wedge\ b = false \end{array}\right\}$$

$$\left\{\begin{array}{l} emp \\ \wedge\ A = leaf(t, v_0, s, v_{i+1}, p_{i+1}) \end{array}\right\}\quad b := \textsc{isLeaf}(A)\quad \left\{\begin{array}{l} emp \\ \wedge\ A = leaf(t, v_0, s, v_{i+1}, p_{i+1}) \\ \wedge\ b = true \end{array}\right\}$$

$$\left\{\begin{array}{l} emp \\ \wedge\ A = inner(t, v_0, p_0, s, v_{i+1}, p_{i+1}) \end{array}\right\}\quad b := \textsc{isLeaf}(A)\quad \left\{\begin{array}{l} emp \\ \wedge\ A = inner(t, v_0, p_0, s, v_{i+1}, p_{i+1}) \\ \wedge\ b = false \end{array}\right\}$$

$$\left\{\begin{array}{l} emp \\ \wedge\ A = inner(t, v_0, p_0, s, v_{i+1}, p_{i+1}) \\ \wedge\ s = [(v_1, p_1), \ldots, (v_i, p_i)] \\ \wedge\ v_j < v \leq v_{j+1} \leq v_{i+1} \end{array}\right\}\quad p := \textsc{next}(A, v)\quad \left\{\begin{array}{l} emp \\ \wedge\ A = inner(t, v_0, p_0, s, v_{i+1}, p_{i+1}) \\ \wedge\ s = [(v_1, p_1), \ldots, (v_i, p_i)] \\ \wedge\ v_j < v \leq v_{j+1} \leq v_{i+1} \\ \wedge\ p = p_j \end{array}\right\}$$

$$\left\{\begin{array}{l} emp \\ \wedge\ A = node(t, v_0, s, v_{i+1}, p_{i+1}) \\ \wedge\ v > v_{i+1} \end{array}\right\}\quad p := \textsc{next}(A, v)\quad \left\{\begin{array}{l} emp \\ \wedge\ A = node(t, v_0, s, v_{i+1}, p_{i+1}) \\ \wedge\ v > v_{i+1} \wedge p = p_{i+1} \end{array}\right\}$$

$$\left\{\begin{array}{l} emp \\ \wedge\ A = node(t, v_0, s, v_{i+1}, p_{i+1}) \\ \wedge\ (v, p) \in s \end{array}\right\}\quad b := \textsc{isIn}(A, v)\quad \left\{\begin{array}{l} emp \\ \wedge\ A = node(t, v_0, s, v_{i+1}, p_{i+1}) \\ \wedge\ (v, p) \in s \wedge b = true \end{array}\right\}$$

$$\left\{\begin{array}{l} emp \\ \wedge\ A = node(t, v_0, s, v_{i+1}, p_{i+1}) \\ \wedge\ (v, p) \notin s \end{array}\right\}\quad b := \textsc{isIn}(A, v)\quad \left\{\begin{array}{l} emp \\ \wedge\ A = node(t, v_0, s, v_{i+1}, p_{i+1}) \\ \wedge\ (v, p) \notin s \wedge b = false \end{array}\right\}$$

$$\left\{\begin{array}{l} emp \\ \wedge\ A = node(t, v_0, s, v_{i+1}, p_{i+1}) \\ \wedge\ (v, q) \in s \end{array}\right\}\quad p := \textsc{lookup}(A, v)\quad \left\{\begin{array}{l} emp \\ \wedge\ A = node(t, v_0, s, v_{i+1}, p_{i+1}) \\ \wedge\ (v, q) \in s \wedge p = q \end{array}\right\}$$

$$\left\{\begin{array}{l} emp \\ \wedge\ A = node(t, v_0, s, v_{i+1}, p_{i+1}) \\ \wedge\ s = [(v_1, p_1), \ldots, (v_i, p_i)] \\ \wedge\ i < 2k \wedge (v, p) \notin s \end{array}\right\}\quad \textsc{addPair}(A, v, p)\quad \left\{\begin{array}{l} emp \\ \wedge\ A = node(t, v_0, z, v_{i+1}, p_{i+1}) \\ \wedge\ s = [(v_1, p_1), \ldots, (v_i, p_i)] \\ \wedge\ i < 2k \wedge (v, p) \notin s \\ \wedge\ z = [(w_1, q_1), \ldots, (w_{i+1}, q_{i+1})] \\ \wedge\ z = s \cup (v, p) \wedge (v, p) \in z \end{array}\right\}$$

$$\left\{\begin{array}{l} emp \\ \wedge\ A = node(t, v_0, s, v_{i+1}, p_{i+1}) \\ \wedge\ s = [(v_1, p_1), \ldots, (v_i, p_i)] \\ \wedge\ (v, p) \in s \end{array}\right\}\quad \textsc{removePair}(A, v)\quad \left\{\begin{array}{l} emp \\ \wedge\ A = node(t, v_0, z, v_{i+1}, p_{i+1}) \\ \wedge\ s = [(v_1, p_1), \ldots, (v_i, p_i)] \\ \wedge\ (v, p) \in s \\ \wedge\ z = [(w_1, q_1), \ldots, (w_{i-1}, q_{i-1})] \\ \wedge\ z = s \backslash (v, p) \wedge (v, p) \notin z \end{array}\right\}$$

$$\left\{\begin{array}{l} emp \\ \wedge\ A = leaf(t, v_0, s, v_{i+1}, p_{i+1}) \\ \wedge\ s = [(v_1, p_1), \ldots, (v_i, p_i)] \\ \wedge\ v_0 < v \leq v_{i+1} \wedge i = 2k \end{array}\right\} \quad B := \text{REARRANGE}(A, v, p, q) \quad \left\{\begin{array}{l} emp \\ \wedge\ A = leaf(t, v_0, s_1, w_k, q) \\ \wedge\ B = leaf(0, w_k, s_2, v_{i+1}, p_{i+1}) \\ \wedge\ s = [(v_1, p_1), \ldots, (v_i, p_i)] \\ \wedge\ v_0 < v \leq v_{i+1} \\ \wedge\ z = [(w_1, q_1), \ldots, (w_{2k+1}, q_{2k+1}) \\ \wedge\ z = s \cup (v, p) \\ \wedge\ s_1 = [(w_1, q_1), \ldots, (w_k, q_k)] \\ \wedge\ s_2 = [(w_{k+1}, q_{k+1}), \ldots, (w_{2k+1}, q_{2k+1})] \end{array}\right\}$$

$$\left\{\begin{array}{l} emp \\ \wedge\ A = inner(t, v_0, p_0, s, v_{i+1}, p_{i+1}) \\ \wedge\ s = [(v_1, p_1), \ldots, (v_i, p_i)] \\ \wedge\ v_0 < v < v_{i+1} \wedge i = 2k \end{array}\right\} \quad B := \text{REARRANGE}(A, v, p, q) \quad \left\{\begin{array}{l} emp \\ \wedge\ A = inner(t, v_0, p_0, s_1, w_{k+1}, q) \\ \wedge\ B = inner(0, w_{k+1}, p_{k+1}, s_2, v_{i+1}, p_{i+1}) \\ \wedge\ s = [(v_1, p_1), \ldots, (v_i, p_i)] \\ \wedge\ v_0 < v < v_{i+1} \\ \wedge\ z = [(w_1, q_1), \ldots, (w_{2k+1}, q_{2k+1}) \\ \wedge\ z = s \cup (v, p) \\ \wedge\ s_1 = [(w_1, q_1), \ldots, (w_k, q_k)] \\ \wedge\ s_2 = [(w_{k+1}, q_{k+1}), \ldots, (w_{2k+1}, q_{2k+1})] \end{array}\right\}$$

$$\{\ emp\ \} \quad stack := \text{NEWSTACK}() \quad \{\ emp \wedge stack = []\ \}$$

$$\{\ emp \wedge stack = xs\ \} \quad \text{PUSH}(stack, v) \quad \{\ emp \wedge stack = v : xs\ \}$$

$$\{\ emp \wedge stack = x : xs\ \} \quad v := \text{POP}(stack) \quad \{\ emp \wedge stack = xs \wedge v = x\ \}$$

$$\{\ emp \wedge stack = []\ \} \quad b := \text{ISEMPTY}(stack) \quad \{\ emp \wedge stack = [] \wedge b = true\ \}$$

$$\{\ emp \wedge stack = x : xs\ \} \quad b := \text{ISEMPTY}(stack) \quad \{\ emp \wedge stack = x : xs \wedge b = false\ \}$$

We use the usual axioms for the language commands.

4.3.4 Algorithms

In the previous section we have shown a programming language with a set of heap and basic commands that manipulate the heap and store respectively. In this section we write the algorithms that manipulate the tree (i.e. search, insert and delete) in that language. Our algorithms are based on the ones presented by Sagiv [27], however the delete operation was not explicitly defined and had to be written based on the textual descriptions only. Unlike Sagiv's algorithms, which are defined using both pseudo code and textual comments to address the cases which the pseudo code does not cover, we encode all cases in the code, making the definition of each precise and without margin of interpretation.

Search

```
r := SEARCH(h, v) {
  MOVEDOWN;
  MOVERIGHT;
  if (ISIN(A, v)) {
    r := LOOKUP(A, v);
  } else {
    r := NULL;
  }
}
```

```
MOVEDOWN ≜ {
  PB := GETPRIMEBLOCK(h);
  current := ROOT(PB);
  A := GET(current);
  while (ISLEAF(A) = FALSE) {
    current := NEXT(A, v);
    A := GET(current);
  }
}
```

```
MOVERIGHT ≜ {
  while (v > HIGHVALUE(A)) {
    current := NEXT(A, v);
    A := GET(current);
  }
}
```

Figure 4.3: B^{Link} tree search algorithm.

The search algorithm (see figure 4.3) starts reading the prime block and starts the search at the node pointed to by its first element. One could think of it as the root, but this is not the general case, as between the initial read of the prime block and the reading of its first element, other threads can insert new roots. After reading the first node, it starts moving down the tree until it reaches a leaf node. To decide which path should be taken when moving down, it always chooses the pointer p_j such that its key value is the closest lower value in the node to v. This procedure is called MOVEDOWN.

```
moveDownAndStack ≜ {
  stack := newStack();
  PB := getPrimeBlock(h);
  current := root(PB);
  A := get(current);
  while (isLeaf(A) = false) {
    if (v ≤ highValue(A)) {
      push(stack, current);
    }
    current := next(A, v);
    A := get(current);
  }
}
```

```
insertIntoSafe ≜ {
  addPair(A, v, p);
  put(A, current);
  unlock(current);
  completed := true;
}
```

```
insertIntoUnsafeRoot ≜ {
  q := new();
  B := rearrange(A, v, p, q);
  put(B, q);
  lock(q);
  put(A, current);
  w := minValue(A);
  t := highValue(A);
  u := highValue(B);
  r := new();
  R := newNode(w, current, t, q, u);
  PB := getPrimeBlock(h);
  put(R, r);
  addRoot(PB, r);
  putPrimeBlock(h, PB);
  unlock(current);
  unlock(q);
  completed := true;
}
```

```
insertIntoUnsafe ≜ {
  q := new();
  B := rearrange(A, v, p, q);
  put(B, q);
  put(A, current);
  unlock(current);
  p := q;
  v := highValue(A);
  level := level +1;
  if (isEmpty(stack)) {
    PB := getPrimeBlock(h);
    current := getNodeLevel(PB, level);
  } else {
    current := pop(stack);
  }
}
```

Figure 4.4: B^{Link} tree insert auxiliary algorithms.

When it reaches a leaf node, it reads the node and if the maximum value is less than or equal to the key value v then it means that the current node is not the one desired. However, the desired node is accessible from the current node, by moving to the right of the linked list at the fringe of the tree. Therefore, it repeats the same process for the node on the right until it reaches the correct node. It then checks if the key value exists on that node, and if so, it returns the correspondent pointer to the data entry, otherwise it returns null.

Insert

The insert algorithm (see figures 4.4 and 4.5), like search, reads the prime block and then starts at its first element. It starts moving down as we described before in the search algorithm. The only difference is that all the pointers to lower levels that form the path until it reaches a leaf, (i.e. all pointers returned by the next(A,v)) are pushed onto the stack.

When it reaches a leaf node, it locks it and tests if the node is the correct one to insert into. A node is the correct one to insert a value-pointer pair into if the value is greater than the minimum value of the node and less than or equal to the maximum value. If it is not the correct node, it uses the procedure moveRight to reach the correct node, and again it locks it to see if it really is. It repeats this process until the correct node is found. The reason why we require this locking is because other threads can insert or delete new values which can induce the creation of new nodes. Therefore, unless the algorithm locks the node, every test to see if it is the correct node could be invalidated before actually inserting the new value-pointer pair.

Assuming the algorithm has the lock of the correct node, it must check that the node is safe to insert into (has fewer than $2k$ pairs). If it is, then the algorithm inserts the new value-pointer pair,

stops and returns success. Otherwise, the node is full and it must be split into two to accommodate the new value-pointer pair, and balance the existing pairs between the current and newly created nodes. There are two separate cases to consider here: either the node is the root, or an intermediate (or leaf) node.

When an insert operation checks if a node is a root, it is uses the command isRoot(*PB*, *current*). There are many ways to implement this command, Sagiv proposes two solutions. One is to have a bit at each node to mark if it is the root. The other solution is to read the prime block and check if the first address corresponds to the node being tested. We can take into account that this command is only used by insert, and that it has already locked the node being tested. Therefore, no other thread can perform the same test until the thread releases the lock. This also means that if another thread is creating a new root, which includes rewriting the prime block, then it has this lock, meaning that all the other threads testing the root will have to wait. We decided to consider the second solution, as having an extra bit in each node would further clutter the notation.

Returning to the algorithm itself, if the node is indeed the root, then the algorithm checks if the node is safe to insert into (i.e. has fewer than $2k$ pairs) and if so, it simply adds the pair to the node in such a way that the list of values and pointers in the node remains ordered. However, when the root is not safe, this requires the creation of a new node. The contents of the root and the pair to be inserted will be split between the root and the newly created node. The newly created node will have maximum value $+\infty$ which is taken from the root, while the maximum value of the old root node and new minimum value of the created node will be a suitable value and equal. The root link will be to the new node. All of these changes are made in the store, therefore it must update the actual tree. To do this, it first writes the newly created node into the heap. Since there are no references from the tree to it, this node is still part of the local state. Before rewriting the root, it locks the new node in the heap. The algorithm then updates the root with the new contents, making the new node part of the shared state and accessible to other threads. This also has the side effect of making the top level a list with two elements, both locked by the thread. To create a new root, it first allocates space in the heap and writes in the new node the correct minimum and maximum values. It also inserts the addresses referring to both nodes at the current higher level of the tree. This new root is currently still part of the local state. Finally the algorithm adds the new root address to the prime block and releases both locks on the other nodes.

Before dealing with the case where the node is not a root, we need to understand why the thread does not release the locks immediately after inserting at the root. Consider instead that the newly created node is not locked. Another thread wants to insert into the new node which is now part of the shared state, and this node is already full (i.e. several other threads have inserted into it). It will split it and attempt to insert at the higher level. To prevent this the algorithm maintains the lock on both nodes while creating a new root for the tree.

We now consider the case if the node is not a root. The algorithm also splits the node and it follows a similar procedure to the one described when splitting the old root. The difference is that the thread does not lock the new node and does not create a new root for the tree. Instead, it gets the next address in the stack, which corresponds to the node at the higher level we came from. This node needs to contain a pointer to the new node to preserve the B^{Link} tree structure. If the stack is empty, the algorithm did not insert at the root, another must have created a new root. In this case,we read the pointer to the leftmost node at the higher level from the prime block. We move along this level until we reach the correct node and repeat this loop until we either insert at a safe node or create a new root for the whole tree.

Delete

The delete algorithm (see figure 4.5) is exactly like search until it reaches a leaf node. When it reaches a leaf node, it acquires a lock and checks if the node is still the correct one. When it is the correct node, it deletes the pair associated with the value v, otherwise it returns false. In the case where the leaf node is not the correct one, it keeps moving to the right until it reaches the correct node. If so, it locks and repeats as described before. A node is said to be the correct one if the minimum value is less than the value key and the maximum value is greater than or equal to it.

```
r := INSERT(h, v, p) {
  completed := FALSE;
  MOVEDOWNANDSTACK;
  level := 1;
  while (completed = FALSE) {
    found := FALSE;
    while (found = FALSE) {
      found := TRUE;
      LOCK(current);
      A := GET(current);
      r := TRUE;
      if (ISIN(A, v)) {
        UNLOCK(current);
        r := FALSE;
        completed := TRUE;
      } else if (v > HIGHVALUE(A)) {
        UNLOCK(current);
        found := FALSE;
        MOVERIGHT;
      }
    }
    if (completed = FALSE) {
      if (ISSAFE(A)) {
        INSERTINTOSAFE;
      } else {
        PB := GETPRIMEBLOCK(h);
        if (ISROOT(PB, current)) {
          INSERTINTOUNSAFEROOT;
        } else {
          INSERTINTOUNSAFE;
        }
      }
    }
  }
}
```

```
r := DELETE(h, v) {
  MOVEDOWN;
  found := FALSE;
  while (found = FALSE) {
    found := TRUE;
    LOCK(current);
    A := GET(current);
    if (ISIN(A, v)) {
      REMOVEPAIR(A, v);
      PUT(A, current);
      UNLOCK(current);
      r := TRUE;
    } else {
      UNLOCK(current);
      if (v > HIGHVALUE(A)) {
        found := FALSE;
        MOVERIGHT;
      } else {
        r := FALSE;
      }
    }
  }
}
```

Figure 4.5: B^{Link} tree insert and delete algorithms.

4.3.5 Actions

We have a similar interpretation of predicates as the one used for the node list. However we must extend the action model to deal with intermediate nodes and the prime block.

Action model

In order to represent the actions which manipulate the abstract resources and at the same time allow modifications dependent of the implementation itself, we extend the action model presented for the node list. It contains two different regions of tokens. One region of actions which allow manipulation of the key value and another region of actions, which in our particular implementation, will contain the tokens to lock and unlock. When a thread performs an insert or delete operation, it is required to lock the node which it will change. This means that to actually perform a change we need to lock the node that contains the key value. We will reuse the predicates $isLock(x)$ and $Locked(x)$ as well as the actions $LOCK(X)$ and $UNLOCK(X)$ which were presented for the node list in section 4.2.4.

We extend the other actions according to the B^{Link} link module as following:

$$CHANGE'(v) : \forall x \,.\, [MOD'(x,v)]_1^{r'} \rightsquigarrow Locked(x) * [CHANGE'(v)]_1^{r'}$$

$$MOD'(x,v) : \begin{cases} \left(\begin{array}{c} |s| < 2k \wedge Locked(x) \\ x \mapsto leaf(t_{id}, v_0, s, v_{i+1}, p_{i+1} \\ * [CHANGE'(v)]_1^{r'} \end{array} \right) \rightsquigarrow \left(\begin{array}{c} x \mapsto leaf(t_{id}, v_0, s \cup \{(v,p)\}, v_{i+1}, p_{i+1}) \\ * [MOD'(x,v)]_1^{r'} \end{array} \right) \\ \left(\begin{array}{c} |s| = 2k \wedge Locked(x) * \\ x \mapsto leaf(t_{id}, v_0, s, v_{i+1}, p_{i+1} * \\ [CHANGE'(v)]_1^{r'} \end{array} \right) \rightsquigarrow \left(\begin{array}{c} x \mapsto leaf(t_{id}, v_0, s', v_0', y) * \\ [MOD'(x,v)]_1^{r'} * \\ y \mapsto leaf(0, v_0', s'', v_{i+1}, p_{i+1}) \\ * [UNLOCK'(y)]_1^{r} * isLock(y) \wedge \\ s' \cup s'' = s \cup \{(v,p)\} \end{array} \right) \\ \left(\begin{array}{c} |s| < 2k \wedge Locked(x) * \\ x \mapsto inner(t_{id}, v_0, p_0, s, v_{i+1}, p_{i+1} \\ * [INSERT'(v)]_i^{r'} \end{array} \right) \rightsquigarrow \left(\begin{array}{c} x \mapsto inner(t_{id}, v_0, p_0, s \cup \{(v,p)\}, v_{i+1}, p_{i+1}) \\ * [INS'(x,v)]_i^{r'} \end{array} \right) \\ \left(\begin{array}{c} |s| = 2k \wedge Locked(x) * \\ x \mapsto inner(t_{id}, v_0, p_0, s, v_{i+1}, p_{i+1} \\ * [INSERT'(v)]_i^{r'} \end{array} \right) \rightsquigarrow \left(\begin{array}{c} x \mapsto inner(t_{id}, v_0, p_0, s', v_0', y) * \\ [INS(x,v)]_i^{r'} * \\ y \mapsto inner(0, v_0', p_0', s'', v_{i+1}, p_{i+1}) * \\ [UNLOCK(y)]_1^{r} * isLock(y) \wedge \\ s' \cup \{(v_0', p_0')\} \cup s'' = s \cup \{(v,p)\} \end{array} \right) \\ \left(\begin{array}{c} Locked(x) * \\ x \mapsto leaf(t_{id}, v_0, s, v_{i+1}, p_{i+1} \\ * [CHANGE'(v)]_1^{r'} \end{array} \right) \rightsquigarrow \left(\begin{array}{c} x \mapsto leaf(t_{id}, v_0, s \backslash \{(v, \_)\}, v_{i+1}, p_{i+1}) \\ * [MOD'(x,v)]_1^{r'} \end{array} \right) \\ \left(\begin{array}{c} Locked(x) * \\ [CHANGE'(v)]_1^{r'} \end{array} \right) \rightsquigarrow [MOD'(x,v)]_1^{r'} \end{cases}$$

$$INSERT'(v) : \forall x \,.\, [INS'(x,v)]_i^{r'} \rightsquigarrow Locked(x) * [INSERT'(v)]_i^{r'}$$

$$INS'(x,v) : \begin{cases} \left(\begin{array}{c} |s| < 2k \wedge Locked(x) * \\ x \mapsto leaf(t_{id}, v_0, s, v_{i+1}, p_{i+1} \\ * [INSERT'(v)]_i^{r'}) \end{array}\right) \rightsquigarrow \left(\begin{array}{c} x \mapsto leaf(t_{id}, v_0, s \cup \{(v,p)\}, v_{i+1}, p_{i+1}) \\ * [INS'(x,v)]_i^{r'} \end{array}\right) \\ \left(\begin{array}{c} |s| = 2k \wedge Locked(x) * \\ x \mapsto leaf(t_{id}, v_0, s, v_{i+1}, p_{i+1} \\ * [INSERT'(v)]_i^{r'}) \end{array}\right) \rightsquigarrow \left(\begin{array}{c} x \mapsto leaf(t_{id}, v_0, s', v_0', y) * \\ [INS(x,v)]_i^{r'} * \\ y \mapsto leaf(0, v_0', s'', v_{i+1}, p_{i+1}) \\ * [UNLOCK(y)]_1^r * isLock(y) \wedge \\ s' \cup s'' = s \cup \{(v,p)\} \end{array}\right) \\ \left(\begin{array}{c} |s| < 2k \wedge Locked(x) * \\ x \mapsto inner(t_{id}, v_0, p_0, s, v_{i+1}, p_{i+1} \\ * [INSERT'(v)]_i^{r'}) \end{array}\right) \rightsquigarrow \left(\begin{array}{c} x \mapsto inner(t_{id}, v_0, p_0, s \cup \{(v,p)\}, v_{i+1}, p_{i+1}) \\ * [INS'(x,v)]_i^{r'} \end{array}\right) \\ \left(\begin{array}{c} |s| = 2k \wedge Locked(x) * \\ x \mapsto inner(t_{id}, v_0, p_0, s, v_{i+1}, p_{i+1} \\ * [INSERT'(v)]_i^{r'}) \end{array}\right) \rightsquigarrow \left(\begin{array}{c} x \mapsto inner(t_{id}, v_0, p_0, s', v_0', y) * \\ [INS(x,v)]_i^{r'} * \\ y \mapsto inner(0, v_0', p_0', s'', v_{i+1}, p_{i+1}) \\ * [UNLOCK(y)]_1^r * isLock(y) \wedge \\ s' \cup \{(v_0', p_0')\} \cup s'' = s \cup \{(v,p)\} \end{array}\right) \\ \left(\begin{array}{c} |s| = 2k \wedge Locked(x) * \\ h \mapsto x : xs * \\ x \mapsto leaf(t_{id}, v_0, s, v_{i+1}, p_{i+1} \\ * [INSERT'(v)]_i^{r'}) \end{array}\right) \rightsquigarrow \left(\begin{array}{c} x \mapsto leaf(t_{id}, v_0, s', v_0', y) * \\ [INS(x,v)]_i^{r'} * \\ y \mapsto leaf(t_{id}, v_0', s'', v_{i+1}, p_{i+1}) \\ * isLock(y) \wedge \\ s' \cup s'' = s \cup \{(v,p)\} \end{array}\right) \\ \left(\begin{array}{c} |s| = 2k \wedge Locked(x) * \\ h \mapsto x : xs * \\ x \mapsto inner(t_{id}, v_0, p_0, s, v_{i+1}, p_{i+1} \\ * [INSERT'(v)]_i^{r'}) \end{array}\right) \rightsquigarrow \left(\begin{array}{c} x \mapsto inner(t_{id}, v_0, p_0, s', v_0', y) * \\ [INS(x,v)]_i^{r'} * \\ y \mapsto inner(0, v_0', p_0', s'', v_{i+1}, p_{i+1}) \\ * isLock(y) \wedge \\ s' \cup \{(v_0', p_0')\} \cup s'' = s \cup \{(v,p)\} \end{array}\right) \\ \left(\begin{array}{c} Locked(x) * \\ [INSERT'(v)]_i^{r'} \end{array}\right) \rightsquigarrow [INS'(x,v)]_i^{r'} \end{cases}$$

$$DELETE'(v) : \forall x \,.\, [DEL'(x,v)]_i^{r'} \rightsquigarrow Locked(x) * [DELETE'(v)]_i^{r'}$$

$$DEL'(x,v) : \begin{cases} \left(\begin{array}{c} Locked(x) * \\ x \mapsto leaf(t_{id}, v_0, s, v_{i+1}, p_{i+1}) \\ * [DELETE'(v)]_i^{r'}) \end{array}\right) \rightsquigarrow \left(\begin{array}{c} x \mapsto leaf(t_{id}, v_0, s \backslash \{(v,\_)\}, v_{i+1}, p_{i+1}) \\ * [DEL'(x,v)]_i^{r'} \end{array}\right) \\ \left(\begin{array}{c} Locked(x) * \\ [DELETE'(v)]_i^{r'}) \end{array}\right) \rightsquigarrow [DEL'(x,v)]_i^{r'} \end{cases}$$

The actions presented here are very similar to the ones used for the node list. The difference is that we require actions to insert at intermediate nodes and to create a new root at the $MOD'(x,v)$ and $INS'(x,v)$ actions.

The interference environments and the concrete implementation of the predicates are analogous to the ones used to the node list. The only difference is that we will use the actions we have extended and the *BTree* predicate instead of the *nodeList* predicate.

4.3.6 Verification

As with the node list, to make sure our implementation is correct we need to check that our algorithms satisfy all cases of our high-level specification. However, since that would be very

extensive to show it here, we will omit the majority of the proofs as they are quite similar to the previous ones.

We start by giving a proof for the SEARCH$(h,\ v)$ operation in an environment where no race conditions occur. The proof is as follows:

$\{\ in_{def}(h,v,p,i)\ \}$
$r :=$ SEARCH(h,v) $\{$
$\left\{\ \exists S\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_{i}\ \right\}$
MOVEDOWN;
$\left\{\begin{array}{l} \exists S, v', v'', s, h'\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_{i} \wedge current = h' \\ \wedge\ A = leaf(\_, v', s, v'', \_) \end{array}\right\}$
MOVERIGHT;
$\left\{\begin{array}{l} \exists S, v', v'', s, h'\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_{i} \wedge current = h' \\ \wedge\ A = leaf(\_, v', s, v'', \_) \wedge v \leq v'' \end{array}\right\}$
if (ISIN(A,v)) $\{$
$\left\{\begin{array}{l} \exists S, v', v'', s, h'\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_{i} \wedge current = h' \\ \wedge\ A = leaf(\_, v', s, v'', \_) \wedge v \leq v'' \end{array}\right\}$
$r :=$ LOOKUP(A,v);
$\left\{\begin{array}{l} \exists S, v', v'', s, h'\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_{i} \wedge current = h' \\ \wedge\ A = leaf(\_, v', s, v'', \_) \wedge v \leq v'' \wedge r = p \end{array}\right\}$
$\}$ **else** $\{$
$\{\ false\ \}$
$r :=$ NULL;
$\{\ false\ \}$
$\}$
$\left\{\begin{array}{l} \exists S, v', v'', s, h'\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_{i} \wedge current = h' \\ \wedge\ A = leaf(\_, v', s, v'', \_) \wedge v \leq v'' \wedge r = p \end{array}\right\}$
$\}$
$\{\ in_{def}(h,v,p,i) \wedge r = p\ \}$

and the proof for the help function MOVERIGHT:

$\left\{\begin{array}{l} \exists S, v', v'', s, h'\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_{1} \wedge current = h' \\ \wedge\ A = leaf(\_, v', s, v'', \_) \end{array}\right\}$
MOVERIGHT $\triangleq$ $\{$
$\left\{\begin{array}{l} \exists S, v', v'', s, h'\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_{1} \wedge current = h' \\ \wedge\ A = leaf(\_, v', s, v'', \_) \end{array}\right\}$
while $(v >$ HIGHVALUE$(A))$ $\{$
$\left\{\begin{array}{l} \exists S, v', v'', s, h'\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_{1} \wedge current = h' \\ \wedge\ A = leaf(\_, v', s, v'', \_) \wedge v > v'' \end{array}\right\}$
$current :=$ NEXT(A,v);
$\left\{\begin{array}{l} \exists S, v', v'', s, h'\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_{1} \wedge current = h' \\ \wedge\ A = leaf(\_, v', s, v'', \_) \wedge v > v'' \end{array}\right\}$
$A :=$ GET$(current)$;
$\left\{\begin{array}{l} \exists S, v', v'', s, h'\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_{1} \wedge current = h' \\ \wedge\ A = leaf(\_, v', s, v'', \_) \end{array}\right\}$
$\}$
$\left\{\begin{array}{l} \exists S, v', v'', s, h'\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_{1} \wedge current = h' \\ \wedge\ A = leaf(\_, v', s, v'', \_) \wedge v \leq v'' \end{array}\right\}$
$\}$
$\left\{\begin{array}{l} \exists S, v', v'', s, h'\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_{1} \wedge current = h' \\ \wedge\ A = leaf(\_, v', s, v'', \_) \wedge v \leq v'' \end{array}\right\}$

and the proof for the help function MOVEDOWN:

$\{\ \exists S\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_{i}\ \}$
MOVEDOWN $\triangleq$ {
$\{\ \exists S\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_{i}\ \}$
$PB :=$ GETPRIMEBLOCK(h);
$\{\ \exists S, h'\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_{i} \wedge PB = h' : xs\ \}$
$current :=$ ROOT(PB);
$\{\ \exists S, h'\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_{i} \wedge current = h'\ \}$
$A :=$ GET$(current)$;
$\left\{\ \exists S, v', v'', s, h'\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_{i} \wedge current = h' \right.$
$\left. \wedge\ A = node(\_, v', s, v'', \_)\ \right\}$
while (ISLEAF(A) = FALSE) {
$\left\{\ \exists S, v', v'', s, h'\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_{i} \wedge current = h' \right.$
$\left. \wedge\ A = inner(\_, v', s, v'', \_)\ \right\}$
$current :=$ NEXT(A, v);
$\left\{\ \exists S, v', v'', s, h'\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_{i} \wedge current = h' \right.$
$\left. \wedge\ A = inner(\_, v', s, v'', \_)\ \right\}$
$A :=$ GET$(current)$;
$\left\{\ \exists S, v', v'', s, h'\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_{i} \wedge current = h' \right.$
$\left. \wedge\ A = node(\_, v', s, v'', \_)\ \right\}$
}
$\left\{\ \exists S, v', v'', s, h'\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_{i} \wedge current = h' \right.$
$\left. \wedge\ A = leaf(\_, v', s, v'', \_)\ \right\}$
}
$\left\{\ \exists S, v', v'', s, h'\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_{i} \wedge current = h' \right.$
$\left. \wedge\ A = leaf(\_, v', s, v'', \_)\ \right\}$

The proof for the INSERT$(h,\ v,\ p)$ operation in a similar case is given in appendix A. Finally, we give a proof for the DELETE$(h,\ v)$ operation also in an environment where no race conditions occur.

$\{\ in_{def}(h,v,p,1)\ \}$
$r :=$ DELETE(h, v) {
$\{\ \exists S\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_{1}\ \}$
MOVEDOWN;
$\left\{\ \exists S, v', v'', s, h'\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_{1} \wedge current = h' \right.$
$\left. \wedge\ A = leaf(\_, v', s, v'', \_)\ \right\}$
$found :=$ FALSE;
$\left\{\ \exists S, v', v'', s, h'\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_{1} \wedge current = h' \right.$
$\left. \wedge\ A = leaf(\_, v', s, v'', \_) \wedge found = false\ \right\}$
while ($found$ = FALSE) {
$\left\{\ \exists S, v', v'', s, h'\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_{1} \wedge current = h' \right.$
$\left. \wedge\ A = leaf(\_, v', s, v'', \_) \wedge found = false\ \right\}$
$found :=$ TRUE;
$\left\{\ \exists S, v', v'', s, h'\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_{1} \wedge current = h' \right.$
$\left. \wedge\ A = leaf(\_, v', s, v'', \_) \wedge found = true\ \right\}$
LOCK$(current)$;
$\left\{\ \exists S, v', v'', s, h'\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_{1} \wedge current = h' \right.$
$\left. \wedge\ A = leaf(\_, v', s, v'', \_) \wedge \wedge found = true * Locked(current)\ \right\}$
$A :=$ GET$(current)$;
$\left\{\ \exists S, v', v'', s, h'\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_{1} \wedge current = h' \right.$
$\left. \wedge\ found = true * Locked(current) \wedge A = leaf(t_{id}, v', s, v'',)\ \right\}$

$$\left\{\begin{array}{l}\exists S, v', v'', s, h' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]_1^{r'} \wedge current = h' \\ \wedge\, found = true * Locked(current) \wedge A = leaf(t_{id}, v', s, v'',) \end{array}\right\}$$

if (ISIN(A, v)) {

$$\left\{\begin{array}{l}\exists S, v', v'', s, h' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]_1^{r'} \wedge current = h' \\ \wedge\, found = true * Locked(current) \wedge A = leaf(t_{id}, v', s, v'',) \wedge (v,\_) \in s \end{array}\right\}$$

REMOVEPAIR(A, v);

$$\left\{\begin{array}{l}\exists S, v', v'', s, h' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]_1^{r'} \wedge current = h' \\ \wedge\, found = true * Locked(current) \wedge A = leaf(t_{id}, v', s \backslash (v,\_), v'',) \wedge (v,\_) \notin s \end{array}\right\}$$

PUT$(A, current)$;

$$\left\{\begin{array}{l}\exists S, v', v'', s, h' \,.\, \boxed{BTree(h,S) \wedge (v,\_) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]_1^{r'} \wedge current = h' \\ \wedge\, found = true * Locked(current) \end{array}\right\}$$

UNLOCK$(current)$;

$$\left\{\begin{array}{l}\exists S, v', v'', s, h' \,.\, \boxed{BTree(h,S) \wedge (v,\_) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]_1^{r'} \wedge current = h' \\ \wedge\, found = true \end{array}\right\}$$

$r :=$ TRUE;

$$\left\{\begin{array}{l}\exists S, v', v'', s, h' \,.\, \boxed{BTree(h,S) \wedge (v,\_) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]_1^{r'} \wedge current = h' \\ \wedge\, found = true \wedge r = true \end{array}\right\}$$

} **else** {

$$\left\{\begin{array}{l}\exists S, v', v'', s, h' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]_1^{r'} \wedge current = h' \\ \wedge\, found = true * Locked(current) \wedge A = leaf(t_{id}, v', s, v'',) \wedge (v,\_) \notin s \end{array}\right\}$$

UNLOCK$(current)$;

$$\left\{\begin{array}{l}\exists S, v', v'', s, h' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]_1^{r'} \wedge current = h' \\ \wedge\, found = true \wedge A = leaf(t_{id}, v', s, v'',) \wedge (v,\_) \notin s \end{array}\right\}$$

if $(v >$ HIGHVALUE$(A))$ {

$$\left\{\begin{array}{l}\exists S, v', v'', s, h' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]_1^{r'} \wedge current = h' \\ \wedge\, found = true \wedge A = leaf(t_{id}, v', s, v'',) \wedge (v,\_) \notin s \wedge v > v'' \end{array}\right\}$$

$found :=$ FALSE;

$$\left\{\begin{array}{l}\exists S, v', v'', s, h' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]_1^{r'} \wedge current = h' \\ \wedge\, found = false \wedge A = leaf(t_{id}, v', s, v'',) \wedge (v,\_) \notin s \wedge v > v'' \end{array}\right\}$$

MOVERIGHT;

$$\left\{\begin{array}{l}\exists S, v', v'', s, h' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]_1^{r'} \wedge current = h' \\ \wedge\, found = false \wedge A = leaf(t_{id}, v', s, v'',) \wedge v \leq v'' \end{array}\right\}$$

} **else** {

$\{\ false\ \}$

$r :=$ FALSE;

$\{\ false\ \}$

}

$$\left\{\begin{array}{l}\exists S, v', v'', s, h' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]_1^{r'} \wedge current = h' \\ \wedge\, found = false \wedge A = leaf(t_{id}, v', s, v'',) \wedge v \leq v'' \end{array}\right\}$$

}

$$\left\{\begin{array}{l}(\exists S, v', v'', s, h' \,.\, \boxed{BTree(h,S) \wedge (v,\_) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]_1^{r'} \wedge current = h' \\ \wedge\, found = true \wedge r = true) \\ \vee\, (\exists S, v', v'', s, h' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]_1^{r'} \wedge current = h' \\ \wedge\, found = false \wedge A = leaf(t_{id}, v', s, v'',) \wedge v \leq v'') \end{array}\right\}$$

}

$$\left\{\ \exists S \,.\, \boxed{BTree(h,S) \wedge (v,\_) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]_1^{r'} \wedge r = true\ \right\}$$

}

$\{\ out_{def}(h, v, 1) \wedge r = true\ \}$

4.4 Hash table

We will now see a simple implementation of a hash table to demonstrate how the abstract specifications are compositional.

4.4.1 Model

In this section we formalise the hash table data structure and extend the program state of the node list.

Formalising the data structure

Consider an hash table implemented as an array, of size n, of node lists.

Definition 20 (Hash table)**.** *Let* $data = [(v_1, p_1), \ldots, (v_n, p_n)]$ *be a value-pointer list, then:*

$$\begin{array}{rcl} hashTable(hash, data) & \stackrel{def}{=} & \exists g_1, \ldots, g_n, s_1, \ldots, s_n \,. \\ & \circledast_{i=1}^{n} & (hash + i - 1) \mapsto nodeList(g_i, s_i) \\ & \wedge & data = \bigcup_{i=1}^{m}((min_i, p_i) \cup s_i) \end{array}$$

The *hashTable* predicate takes two parameters, *hash* and *data*. The first parameter contains the address of the first element of the array, this address is constant. The second parameter contains the concatenation of all value-pointer lists of each node in all linked lists.

Program state

The program state consists of a working heap h and a variable store σ.

Definition 21 (Heap)**.** *A heap* h *is a finite partial function that maps heap addresses to node data:*

$$h : \quad Addr \rightharpoonup_{fin} Node \times Addr \rightharpoonup_{fin} Addr$$

The heap is divided up such that each thread has a private section of the heap for internal use and there exists a single area that is shared by all threads which contains the array with n addresses and nodes which make the node lists.

The local state of a thread is composed by its private section. It can be described as every node accessible, by the thread, in the heap excluding the shared state. This will include all nodes allocated in the heap before a thread modifies the list to point to them.

The store is the same as the one used to the node list.

4.4.2 Programming language

We extend the programming language used to the node list with a heap command and a store command as following:

$$\begin{array}{rll} C_H ::= & \ldots & \\ & h := \textsc{getList}(h, v) & \text{Get node list at position } v \text{ in the hash table } h \end{array}$$

The command $\textsc{getList}(h,\ v)$ is used to retrieve the node list address at the array h and position v.

$$\begin{array}{rll} C_N ::= & \ldots & \\ & w := \textsc{hashCode}(v) & \text{Get hash code for value } v \end{array}$$

The command $\textsc{hashCode}(v)$ calculates an hash code, for value v, between 0 and n.

4.4.3 Algorithms

The hash table algorithms (see figure 4.6) are all similar to each other. They start by calculating the hash code of the key value v which will be greater or equal to 0 and less n. With the hash code they get the address of the corresponding node list. Finally, they apply the operation they are performing on the node list using its interface.

```
r := SEARCH(h, v) {
  w := HASHCODE(v);
  g := GETLIST(h, c);
  r := SEARCH(g, v);
}

r := INSERT(h, v, p) {
  w := HASHCODE(v);
  g := GETLIST(h, c);
  r := INSERT(g, v, p);
}

r := DELETE(h, v) {
  w := HASHCODE(v);
  g := GETLIST(h, c);
  r := DELETE(g, v);
}
```

Figure 4.6: Hash table algorithms.

4.4.4 Actions

Since the hash code calculation is local and does not change the shared state, all the actions will be the ones from the node list. This happens because our algorithms make use the of the node lists internally and each node list is disjoint from each other in the heap.

4.4.5 Verification

The verification of the algorithms can be done by proving that the hash code function always reaches the correct node list, this is out of the scope of this work. The rest of the proof is done by using the high-level specification of the node list which is the same as the hash table itself. We present a proof for the search operation as an example. Notice that we use the predicates $in_{def}(h, v, p, i)$ for the hash table implementation and $in_{def}(g, v, p, i)$ for the node list implementation. Finally, we use the $hashCode(v)$ to represent the hash code command.

$\{\ in_{def}(h, v, p, i)\ \}$
$r :=$ SEARCH(h, v) {
$\left\{\ \exists S\ .\ \boxed{hashTable(h, S) \wedge (v, p) \in S}^{r'}_{Def(r', h)} * [CHANGE(v)]^{r'}_{i}\ \right\}$
$w :=$ HASHCODE(v);
$\left\{\ \exists S\ .\ \boxed{hashTable(h, S) \wedge (v, p) \in S}^{r'}_{Def(r', h)} * [CHANGE(v)]^{r'}_{i} \wedge w = hashCode(v)\ \right\}$
$g :=$ GETLIST(h, c);
$\left\{\begin{array}{l} \exists S, S'\ .\ \boxed{hashTable(h, S) \wedge (v, p) \in S}^{r'}_{Def(r', h)} * [CHANGE(v)]^{r'}_{i} \wedge w = hashCode(v) \\ \wedge\ in_{def}(g, v, p, i) \end{array}\right\}$
$r :=$ SEARCH(g, v);
$\left\{\begin{array}{l} \exists S, S'\ .\ \boxed{hashTable(h, S) \wedge (v, p) \in S}^{r'}_{Def(r', h)} * [CHANGE(v)]^{r'}_{i} \wedge w = hashCode(v) \\ \wedge\ in_{def}(g, v, p, i) \wedge r = p \end{array}\right\}$
}
$\{\ in_{def}(h, v, p, i) \wedge r = p\ \}$

4.5 Evaluation

We have successfully defined a low-level implementation of the abstract specification for three particular implementations. We used a node list as a motivation to investigate how to discover the low-level details. This turned out to be a good choice as it has the main ideas of other complex data structures in terms of the action model but is easier to model in terms of complexity of the data structure itself.

Our approach to the B^{Link} tree was made easier by reusing and extending the concepts applied to the node list. We can see the node list as the leaf list in the B^{Link} tree. Therefore, it makes sense that we only needed to extend the number of actions, as the high-level specification is only related to the leaf list contents, everything else in the tree is just a way to access the leaf list in a more efficient way. We also believe that a similar approach can be used for other concurrent implementations that use a tree structure.

In our third implementation, a hash table, we were able to use the abstract specifications of another concurrent index (the node list) to prove the correctness of the implementation of the hash table itself. This was possible because the hash table was implemented using the node list. We can

see that the same approach can be applied to software that uses data structures, as long as the data structures have a formal abstract specification that hides the details of the implementation.

We can see that threads can lock and unlock nodes without affecting the high-level view and as such, we have a different permission region for these actions. By combining them with the actions which will affect the abstract view, we allow flexibility at the low-level and still maintain a correct abstract specification at the high-level. We can see that the actions which do affect the high-level can only occur when the abstract level allows them, as the abstract level implicitly defines the amount of interference which can occur at any point in time. Taking this into account, we can see that the low-level will have to restrict its computation in order to avoid affecting the abstract view. This means that there is a relationship between the high-level and the low-level in terms of the interference and the actions themselves which can occur. We need to investigate this further to understand if allowing stronger abstract specifications will influence the low-level.

We can also see that at the low-level things do not occur at the same time, assuming traditional interleaving semantics, however, at the high-level we still have a fiction of concurrency.

Finally, we want to investigate if it is better to first build the high-level specifications and then give a concrete implementation or to start at the low-level and infer the high-level specification as is normally done in separation logic.

Chapter 5

Evaluation

We take a critical view of the work presented, and see what remains to be done.

We have reasoned about B^{Link} trees before using RGSep. We can see that the concurrent abstract predicates have a high-level specification and a low-level specification while the RGSep has only the last. If we consider the low-level implementation specification (that is, the concrete implementation of the high-level specification), we can see that there are many similarities in the models, command axioms and even the action models themselves. It should be interesting to pursue further investigation in this area, as it might be possible to extend RGSep to enable abstraction at the high-level.

We have provided abstract specifications that capture high-level behaviour and hide the details of an implementation. However, we have not covered correctness conditions, such as linearisability. It is important to explore the effects that linearisability has in our high-level specification. More importantly, what are the implications at the low-level. Since we are giving an abstract specification which can be implemented in many different ways, when we impose a correctness condition like linearisability, we are restricting the possible implementations which satisfy our specification. What are the implications in the action model itself and, is it possible to simplify the concurrent abstract predicates internal model by imposing a restrictive model of concurrent computation?

While implementing the node list and the B^{Link} tree, we had to prove that the high-level axioms were satisfied at the low-level. We did by imposing a system where a thread could exchange possible action tokens if it had full permission. We believe that these kind of systems require further work and might be a possible extension to the concurrent abstract predicates in general.

Finally, there seem to be many specifications which are very similar both at the high-level and at the low-level. Since we are required to implement every high-level specification, axiom or predicate, this leads to a considerable amount of work which, in many cases, is repetitive. The development of automated tools would be very useful if we wish to apply these concepts on a larger scale.

Chapter 6

Conclusions and future work

We have provided abstract specifications which allow reasoning about concurrent operations which manipulate an index. We have found out that we can get past the normal restriction of sequential writers and concurrent readings and still get some information from the contents of the index. We were able to capture that knowledge in formal specifications and were able to prove several implementations and still maintain the abstraction at the high-level. The action model developed should also be easily extendable to other data structures besides concurrent indexes. We believe that the continuation of this work for other data structures will enhance our capability to prove concurrent programs and improve the reusability of the proofs.

6.1 Future work

There are many possible ways that we could extend the current work. We will give a brief summary of some work that remains to be done and possible new directions.

6.1.1 Abstract specifications

We have provided abstract specifications that seem to capture the high-level behaviour and still allow different implementations at the low-level. Even though these specifications seem to be well adapted for real programs, there is still work to be done in this area. In particular deallocation of abstract resources at the high-level or dealing with the creation of the data structure. It seems important to explore what restrictions the high-level specification imposes at the low-level and how this relates to correctness conditions, such as linearisability. The reverse also seems very important, imposing stronger correctness conditions at the implementation level should enable stronger abstract specifications. It is not clear yet what the relationship between both levels is. Another interesting development can be using the same ideas to tackle other readers/writers problems which require formal verification.

6.1.2 Concurrent abstract predicates

We have seen that concurrent abstract predicates allow us to use abstraction in our logic in a similar way to how programmers use abstraction when developing programs. This allow us to contain the complexity of proof at the higher-levels and at the same time define in a formal way what a module operation does. These ideas could be built-in to programming languages, even used as abstract web interfaces, such as web services, in order to increase the formalism of their specifications and at the same time allow automated tools to prove their correctness against a specification.

6.1.3 Verification tool

An obvious extension to this project would be the creation of an automated verification tool that would support concurrent abstract predicates and allow abstractions such as the specifications developed. If we could specify an abstract specification for some module operations, the program

should be able to infer, with some help of the user, if a concrete implementation is correct. This process could be repeated in order to be applied at several layers in a bigger project. We hope that the continuous development in this new logic will result in such a verification tool in the future.

6.1.4 Fault recovery

Another possible extension to this work is to allow specifications which allow "faults" at the low-level, such that we have to restart or rollback some thread that was performing an operation. It should be interesting to embed logics such as concurrent abstract predicates with mechanisms that enable such reasoning at the low-level and still be able to prove the abstract specification, as in practise there exists algorithms which use these ideas when implementing an algorithm in order to improve their efficiency.

Bibliography

[1] R. Bayer and E. McCreight. Organization and Maintenance of Large Ordered Indices. *Acta Informatica*, 1:173–189, 1972.

[2] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta informatica*, 9(1):1–21, 1977.

[3] R. Bornat, C. Calcagno, and P. O'Hearn. Local reasoning, separation and aliasing. 2004.

[4] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, page 270. ACM, 2005.

[5] R. Bornat, C. Calcagno, and H. Yang. Variables as resource in separation logic. *Electronic Notes in Theoretical Computer Science*, 155:247–276, 2006.

[6] J. Boyland. Checking interference with fractional permissions. *Static Analysis*, pages 1075–1075, 2003.

[7] C. Calcagno, P. O'Hearn, and H. Yang. Local action and abstract separation logic. In *Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science*, pages 366–378. IEEE Computer Society, 2007.

[8] D. Comer. The ubiquitous B-tree. 1979.

[9] T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Slovenia*, 2010.

[10] M. Dodds, X. Feng, M. Parkinson, and V. Vafeiadis. Deny-guarantee reasoning. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, pages 363–377. Springer-Verlag Berlin, Heidelberg, 2009.

[11] C. Ellis. Concurrent search and insertion in 2–3 trees. *Acta Informatica*, 14(1):63–86, 1980.

[12] X. Feng. Local rely-guarantee reasoning. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 315–327. ACM, 2009.

[13] X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *Programming languages and systems: 16th European Symposium on Programming, ESOP 2007, held as part of the Joint European Conferences on Theory and Practics [ie Practice] of Software, ETAPS 2007, Braga, Portugal, March 24-April 1, 2007: proceedings*, page 173. Springer-Verlag New York Inc, 2007.

[14] C. Hoare. An axiomatic basis for computer programming. 1969.

[15] S. Ishtiaq and P. O'Hearn. BI as an assertion language for mutable data structures. *ACM SIGPLAN Notices*, 36(3):26, 2001.

[16] C. Jones. *Development methods for computer programs including a notion of interference*. Oxford Univ. Computing Laboratory, Programming Research Group, 1981.

[17] C. B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, pages 321–332. North-Holland, 1983.

[18] Y. Kwong and D. Wood. A new method for concurrency in B-trees. *IEEE Transactions on Software Engineering*, 8(3):211–222, 1982.

[19] P. Lehman and S. Yao. Efficient locking for concurrent operations on b-trees. *ACM Transactions on Database Systems*, 6(4):650–670, 1981.

[20] R. Miller and L. Snyder. Multiple access to B-trees. In *Proceedings of the 1978 Conference on Information Sciences and Systems*, pages 400–408.

[21] P. O'Hearn. Resources, concurrency, and local reasoning. *Theoretical Computer Science*, 375(1-3):271–307, 2007.

[22] P. O'Hearn and D. Pym. The logic of bunched implications. *The Bulletin of Symbolic Logic*, 5(2):215–244, 1999.

[23] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of the 15th International Workshop on Computer Science Logic*, page 19. Springer-Verlag, 2001.

[24] M. Parkinson, R. Bornat, and C. Calcagno. Variables as resource in hoare logics. 2006.

[25] P. Pinto. Reasoning about B^{Link} trees. Technical report, Department of Computing, Imperial College London, April 2010.

[26] J. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.

[27] Y. Sagiv. Concurrent operations on B*-trees with overtaking. *Journal of Computer and System Sciences*, 33(2):275–296, 1986.

[28] J. Ullman. Principles of database systems. 1982.

[29] V. Vafeiadis. Modular fine-grained concurrency verification. 2008.

[30] V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. *Lecture Notes in Computer Science*, 4703:256–271, 2007.

Appendix A

B^{Link} tree insert proof

$\{\ out_{def}(h, v, 1)\ \}$
$r :=$ INSERT(h, v, p) {
$\left\{\ \exists S, q\ .\ \boxed{BTree(h, S) \wedge (v, q) \notin S}^{r'}_{Def(r', h)} * [CHANGE'(v)]^{r'}_1\ \right\}$
$completed :=$ FALSE;
$\left\{\ \exists S, q\ .\ \boxed{BTree(h, S) \wedge (v, q) \notin S}^{r'}_{Def(r', h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false\ \right\}$
MOVEDOWNANDSTACK;
$\left\{\begin{array}{l} \exists S, q, v', v'', s, h', xs\ .\ \boxed{BTree(h, S) \wedge (v, q) \notin S}^{r'}_{Def(r', h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false \\ \wedge\ stack = xs \wedge current = h' \wedge A = leaf(\_, v', s, v'', \_) \end{array}\right\}$
$level := 1$;
$\left\{\begin{array}{l} \exists S, q, v', v'', s, h', xs\ .\ \boxed{BTree(h, S) \wedge (v, q) \notin S}^{r'}_{Def(r', h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false \\ \wedge\ stack = xs \wedge current = h' \wedge A = leaf(\_, v', s, v'', \_) \wedge level = 1 \end{array}\right\}$
while $(completed =$ FALSE$)$ {
$\left\{\begin{array}{l} (\exists S, q, v', v'', s, h', xs\ .\ \boxed{BTree(h, S) \wedge (v, q) \notin S}^{r'}_{Def(r', h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false \\ \wedge\ stack = xs \wedge current = h' \wedge A = leaf(\_, v', s, v'', \_) \wedge level = 1) \\ \vee \\ (\exists S, v', v'', s, h', xs\ .\ \boxed{BTree(h, S) \wedge (v, p) \in S}^{r'}_{Def(r', h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false \\ \wedge\ stack = xs \wedge current = h' \wedge A = inner(\_, v', \_, s, v'', \_) \wedge level > 1) \end{array}\right\}$
$found :=$ FALSE;
$\left\{\begin{array}{l} (\exists S, q, v', v'', s, h', xs\ .\ \boxed{BTree(h, S) \wedge (v, q) \notin S}^{r'}_{Def(r', h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false \\ \wedge\ stack = xs \wedge current = h' \wedge A = leaf(\_, v', s, v'', \_) \wedge level = 1 \wedge found = false) \\ \vee \\ (\exists S, v', v'', s, h', xs\ .\ \boxed{BTree(h, S) \wedge (v, p) \in S}^{r'}_{Def(r', h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false \\ \wedge\ stack = xs \wedge current = h' \wedge A = inner(\_, v', \_, s, v'', \_) \wedge level > 1 \wedge found = false) \end{array}\right\}$

$\{\ (\exists S, q, v', v'', s, h', xs\ .\ \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false$
$\wedge\ stack = xs \wedge current = h' \wedge A = leaf(\_, v', s, v'', \_) \wedge level = 1 \wedge found = false)$
$\vee$
$(\exists S, v', v'', s, h', xs\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false$
$\wedge\ stack = xs \wedge current = h' \wedge A = inner(\_, v', \_, s, v'', \_) \wedge level > 1 \wedge found = false)\ \}$

while $(found = \text{FALSE})$ {

$\{\ (\exists S, q, v', v'', s, h', xs\ .\ \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false$
$\wedge\ stack = xs \wedge current = h' \wedge A = leaf(\_, v', s, v'', \_) \wedge level = 1 \wedge found = false)$
$\vee$
$(\exists S, v', v'', s, h', xs\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false$
$\wedge\ stack = xs \wedge current = h' \wedge A = inner(\_, v', \_, s, v'', \_) \wedge level > 1 \wedge found = false)\ \}$

$found := \text{TRUE};$

$\{\ (\exists S, q, v', v'', s, h', xs\ .\ \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false$
$\wedge\ stack = xs \wedge current = h' \wedge A = leaf(\_, v', s, v'', \_) \wedge level = 1 \wedge found = true)$
$\vee$
$(\exists S, v', v'', s, h', xs\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false$
$\wedge\ stack = xs \wedge current = h' \wedge A = inner(\_, v', \_, s, v'', \_) \wedge level > 1 \wedge found = true)\ \}$

$\text{LOCK}(current);$

$\{\ (\exists S, q, v', v'', s, h', xs\ .\ \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false$
$\wedge\ stack = xs \wedge current = h' \wedge A = leaf(\_, v', s, v'', \_) \wedge level = 1 \wedge found = true$
$*\ Locked(current))$
$\vee$
$(\exists S, v', v'', s, h', xs\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false$
$\wedge\ stack = xs \wedge current = h' \wedge A = inner(\_, v', \_, s, v'', \_) \wedge level > 1 \wedge found = true$
$*\ Locked(current))\ \}$

$A := \text{GET}(current);$

$\{\ (\exists S, q, v', v'', s, h', xs\ .\ \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false$
$\wedge\ stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s, v'', \_) \wedge level = 1 \wedge found = true$
$*\ Locked(current))$
$\vee$
$(\exists S, v', v'', s, h', xs\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false$
$\wedge\ stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s, v'', \_) \wedge level > 1 \wedge found = true$
$*\ Locked(current))\ \}$

$r := \text{TRUE};$

$\{\ (\exists S, q, v', v'', s, h', xs\ .\ \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false$
$\wedge\ stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s, v'', \_) \wedge level = 1 \wedge found = true$
$*\ Locked(current) \wedge r = true)$
$\vee$
$(\exists S, v', v'', s, h', xs\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false$
$\wedge\ stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s, v'', \_) \wedge level > 1 \wedge found = true$
$*\ Locked(current) \wedge r = true)\ \}$

if $(\text{ISIN}(A, v))$ {

$\{\ false\ \}$

$\text{UNLOCK}(current);$

$\{\ false\ \}$

$r := \text{FALSE};$

$\{\ false\ \}$

$completed := \text{TRUE};$

$\{\ false\ \}$

$\{\ false\ \}$

} **else if** $(v >$ HIGHVALUE$(A))$ {

$\{\ (\exists S, q, v', v'', s, h', xs\ .\ \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false$
$\wedge\ stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s, v'', \_) \wedge level = 1 \wedge found = true$
$*\ Locked(current) \wedge r = true \wedge v > v'')$
$\vee$
$(\exists S, v', v'', s, h', xs\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false$
$\wedge\ stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s, v'', \_) \wedge level > 1 \wedge found = true$
$*\ Locked(current) \wedge r = true \wedge v > v'')\ \}$

UNLOCK$(current)$;

$\{\ (\exists S, q, v', v'', s, h', xs\ .\ \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false$
$\wedge\ stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s, v'', \_) \wedge level = 1 \wedge found = true \wedge v > v''$
$\wedge\ r = true)$
$\vee$
$(\exists S, v', v'', s, h', xs\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false$
$\wedge\ stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s, v'', \_) \wedge level > 1 \wedge found = true \wedge v > v''$
$\wedge\ r = true)\ \}$

$found :=$ FALSE;

$\{\ (\exists S, q, v', v'', s, h', xs\ .\ \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false$
$\wedge\ stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s, v'', \_) \wedge level = 1 \wedge found = false \wedge v > v''$
$\wedge\ r = true)$
$\vee$
$(\exists S, v', v'', s, h', xs\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false$
$\wedge\ stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s, v'', \_) \wedge level > 1 \wedge found = false \wedge v > v''$
$\wedge\ r = true)\ \}$

MOVERIGHT;

$\{\ (\exists S, q, v', v'', s, h', xs\ .\ \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false$
$\wedge\ stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s, v'', \_) \wedge level = 1 \wedge found = false \wedge v \leq v''$
$\wedge\ r = true)$
$\vee$
$(\exists S, v', v'', s, h', xs\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false$
$\wedge\ stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s, v'', \_) \wedge level > 1 \wedge found = false \wedge v \leq v''$
$\wedge\ r = true)\ \}$

}

$\{\ ((\exists S, q, v', v'', s, h', xs\ .\ \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false$
$\wedge\ stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s, v'', \_) \wedge level = 1 \wedge found = false \wedge v \leq v''$
$\wedge\ r = true)$
$\vee$
$(\exists S, v', v'', s, h', xs\ .\ \boxed{BTree(h,S) \wedge (v,q) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false$
$\wedge\ stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s, v'', \_) \wedge level > 1 \wedge found = false \wedge v \leq v''$
$\wedge\ r = true)) \vee$
$((\exists S, q, v', v'', s, h', xs\ .\ \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false$
$\wedge\ stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s, v'', \_) \wedge level = 1 \wedge found = true$
$*\ Locked(current) \wedge v \leq v'') \wedge r = true)$
$\vee$
$(\exists S, v', v'', s, h', xs\ .\ \boxed{BTree(h,S) \wedge (v,p) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false$
$\wedge\ stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s, v'', \_) \wedge level > 1 \wedge found = true$
$*\ Locked(current) \wedge v \leq v'') \wedge r = true))\ \}$

}

$\{\ (\exists S, q, v', v'', s, h', xs\ .\ \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false$
$\wedge\ stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s, v'', \_) \wedge level = 1 \wedge found = true$
$*\ Locked(current) \wedge v \leq v'' \wedge r = true)$
$\vee$
$(\exists S, v', v'', s, h', xs\ .\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false$
$\wedge\ stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s, v'', \_) \wedge level > 1 \wedge found = true$
$*\ Locked(current) \wedge v \leq v'' \wedge r = true)\ \}$

$$\left\{\begin{array}{l}(\exists S, q, v', v'', s, h', xs\,.\ \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\ stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s, v'', \_) \wedge level = 1 \wedge found = true \\ *\ Locked(current) \wedge v \leq v'' \wedge r = true \wedge completed = false) \\ \vee \\ (\exists S, v', v'', s, h', xs\,.\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\ stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s, v'', \_) \wedge level > 1 \wedge found = true \\ *\ Locked(current) \wedge v \leq v'' \wedge r = true \wedge completed = false)\end{array}\right\}$$

if ($completed =$ FALSE) {

$$\left\{\begin{array}{l}(\exists S, q, v', v'', s, h', xs\,.\ \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\ stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s, v'', \_) \wedge level = 1 \wedge found = true \\ *\ Locked(current) \wedge v \leq v'' \wedge r = true \wedge completed = false) \\ \vee \\ (\exists S, v', v'', s, h', xs\,.\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\ stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s, v'', \_) \wedge level > 1 \wedge found = true \\ *\ Locked(current) \wedge v \leq v'' \wedge r = true \wedge completed = false)\end{array}\right\}$$

if (isSafe(A)) {

$$\left\{\begin{array}{l}(\exists S, q, v', v'', s, h', xs\,.\ \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\ stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s, v'', \_) \wedge level = 1 \wedge found = true \\ *\ Locked(current) \wedge v \leq v'' \wedge r = true \wedge |s| < 2k \wedge completed = false) \\ \vee \\ (\exists S, v', v'', s, h', xs\,.\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\ stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s, v'', \_) \wedge level > 1 \wedge found = true \\ *\ Locked(current) \wedge v \leq v'' \wedge r = true \wedge |s| < 2k \wedge completed = false)\end{array}\right\}$$

insertIntoSafe;

$$\left\{\begin{array}{l}\exists S\,.\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = true \\ \wedge\ r = true\end{array}\right\}$$

} **else** {

$$\left\{\begin{array}{l}(\exists S, q, v', v'', s, h', xs\,.\ \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\ stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s, v'', \_) \wedge level = 1 \wedge found = true \\ *\ Locked(current) \wedge v \leq v'' \wedge r = true \wedge |s| = 2k \wedge completed = false) \\ \vee \\ (\exists S, v', v'', s, h', xs\,.\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\ stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s, v'', \_) \wedge level > 1 \wedge found = true \\ *\ Locked(current) \wedge v \leq v'' \wedge r = true \wedge |s| = 2k \wedge completed = false)\end{array}\right\}$$

$PB :=$ getPrimeBlock(h);

$$\left\{\begin{array}{l}(\exists S, q, v', v'', s, h', h', xs, xs'\,.\ \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\ stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s, v'', \_) \wedge level = 1 \wedge found = true \\ *\ Locked(current) \wedge v \leq v'' \wedge r = true \wedge |s| = 2k \wedge PB = h'' : xs' \wedge completed = false) \\ \vee \\ (\exists S, v', v'', s, h', h', xs, xs'\,.\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\ stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s, v'', \_) \wedge level > 1 \wedge found = true \\ *\ Locked(current) \wedge v \leq v'' \wedge r = true \wedge |s| = 2k \wedge PB = h'' : xs' \wedge completed = false)\end{array}\right\}$$

if (isRoot(PB, $current$)) {

$$\left\{\begin{array}{l}(\exists S, q, v', v'', s, h', h'', xs, xs'\,.\ \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\ completed = false * Locked(current) \\ \wedge\ stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s, v'', \_) \wedge level = 1 \wedge found = true \\ \wedge\ v \leq v'' \wedge r = true \wedge |s| = 2k \wedge PB = h'' : xs' \wedge h' = h'') \\ \vee \\ (\exists S, v', v'', s, h', h'', xs, xs'\,.\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\ completed = false * Locked(current) \\ \wedge\ stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s, v'', \_) \wedge level > 1 \wedge found = true \\ \wedge\ v \leq v'' \wedge r = true \wedge |s| = 2k \wedge PB = h'' : xs' \wedge h' = h'')\end{array}\right\}$$

insertIntoUnsafeRoot;

$$\left\{\begin{array}{l}(\exists S\,.\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = true \\ \wedge\ r = true)\end{array}\right\}$$

$$\left\{ \begin{array}{l} (\exists S \,.\, \boxed{BTree(h,S) \land (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \land completed = true \\ \land\, r = true) \end{array} \right\}$$

} **else** {

$$\left\{ \begin{array}{l} (\exists S, q, v', v'', s, h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \land (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \land\, completed = false * Locked(current) \\ \land\, stack = xs \land current = h' \land A = leaf(t_{id}, v', s, v'', \_) \land level = 1 \land found = true \\ \land\, v \le v'' \land r = true \land |s| = 2k \land PB = h'' : xs' \land h' \ne h'') \\ \lor \\ (\exists S, v', v'', s, h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \land (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \land\, completed = false * Locked(current) \\ \land\, stack = xs \land current = h' \land A = inner(t_{id}, v', \_, s, v'', \_) \land level > 1 \land found = true \\ \land\, v \le v'' \land r = true \land |s| = 2k \land PB = h'' : xs' \land h' \ne h'') \end{array} \right\}$$

InsertIntoUnsafe;

$$\left\{ \begin{array}{l} \exists S, v', v'', s, h', xs \,.\, \boxed{BTree(h,S) \land (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \land completed = false \\ \land\, stack = xs \land current = h' \land A = inner(\_, v', \_, s, v'', \_) \land level > 1 \end{array} \right\}$$

}

$$\left\{ \begin{array}{l} (\exists S \,.\, \boxed{BTree(h,S) \land (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \land completed = true \\ \land\, r = true) \\ \lor \\ (\exists S, v', v'', s, h', xs \,.\, \boxed{BTree(h,S) \land (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \land completed = false \\ \land\, stack = xs \land current = h' \land A = inner(\_, v', \_, s, v'', \_) \land level > 1) \end{array} \right\}$$

}

$$\left\{ \begin{array}{l} (\exists S \,.\, \boxed{BTree(h,S) \land (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \land completed = true \\ \land\, r = true) \\ \lor \\ (\exists S, v', v'', s, h', xs \,.\, \boxed{BTree(h,S) \land (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \land completed = false \\ \land\, stack = xs \land current = h' \land A = inner(\_, v', \_, s, v'', \_) \land level > 1) \end{array} \right\}$$

}

$$\left\{ \begin{array}{l} (\exists S \,.\, \boxed{BTree(h,S) \land (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \land completed = true \\ \land\, r = true) \\ \lor \\ (\exists S, v', v'', s, h', xs \,.\, \boxed{BTree(h,S) \land (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \land completed = false \\ \land\, stack = xs \land current = h' \land A = inner(\_, v', \_, s, v'', \_) \land level > 1) \end{array} \right\}$$

}

$$\left\{ \begin{array}{l} \exists S \,.\, \boxed{BTree(h,S) \land (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \land completed = true \\ \land\, r = true \end{array} \right\} \}$$

$$\{\ in_{def}(h, v, p, 1) \land r = true\ \}$$

and the proof for the helper function MOVEDOWNANDSTACK:

$$\left\{ \exists S, q \,.\, \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false \right\}$$

MOVEDOWNANDSTACK $\triangleq$ {

$$\left\{ \exists S, q \,.\, \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false \right\}$$

$stack :=$ NEWSTACK();

$$\left\{ \begin{array}{l} \exists S, q \,.\, \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false \\ \wedge\ stack = [] \end{array} \right\}$$

$PB :=$ GETPRIMEBLOCK(h);

$$\left\{ \begin{array}{l} \exists S, q, h' \,.\, \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false \\ \wedge\ stack = [] \wedge PB = h' : xs \end{array} \right\}$$

$current :=$ ROOT(PB);

$$\left\{ \begin{array}{l} \exists S, q, h' \,.\, \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false \\ \wedge\ stack = [] \wedge current = h' \end{array} \right\}$$

$A :=$ GET($current$);

$$\left\{ \begin{array}{l} \exists S, q, v', v'', s, h' \,.\, \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false \\ \wedge\ stack = [] \wedge current = h' \wedge A = node(\_, v', s, v'', \_) \end{array} \right\}$$

while (ISLEAF(A) = FALSE) {

$$\left\{ \begin{array}{l} \exists S, q, v', v'', s, h', xs \,.\, \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false \\ \wedge\ stack = xs \wedge current = h' \wedge A = inner(\_, v', \_, s, v'', \_) \end{array} \right\}$$

if ($v \leq$ HIGHVALUE(A)) {

$$\left\{ \begin{array}{l} \exists S, q, v', v'', s, h', xs \,.\, \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false \\ \wedge\ stack = xs \wedge current = h' \wedge A = inner(\_, v', \_, s, v'', \_) \wedge v \leq v'' \end{array} \right\}$$

PUSH($stack$, $current$);

$$\left\{ \begin{array}{l} \exists S, q, v', v'', s, h', xs \,.\, \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false \\ \wedge\ stack = h' : xs \wedge current = h' \wedge A = inner(\_, v', \_, s, v'', \_) \wedge v \leq v'' \end{array} \right\}$$

}

$$\left\{ \begin{array}{l} \exists S, q, v', v'', s, h', xs \,.\, \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false \\ \wedge\ stack = xs \wedge current = h' \wedge A = inner(\_, v', \_, s, v'', \_) \end{array} \right\}$$

$current :=$ NEXT(A, v);

$$\left\{ \begin{array}{l} \exists S, q, v', v'', s, h', xs \,.\, \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false \\ \wedge\ stack = xs \wedge current = h' \wedge A = inner(\_, v', \_, s, v'', \_) \end{array} \right\}$$

$A :=$ GET($current$);

$$\left\{ \begin{array}{l} \exists S, q, v', v'', s, h', xs \,.\, \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false \\ \wedge\ stack = xs \wedge current = h' \wedge A = node(\_, v', s, v'', \_) \end{array} \right\}$$

}

$$\left\{ \begin{array}{l} \exists S, q, v', v'', s, h', xs \,.\, \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false \\ \wedge\ stack = xs \wedge current = h' \wedge A = leaf(\_, v', s, v'', \_) \end{array} \right\}$$

}

$$\left\{ \begin{array}{l} \exists S, q, v', v'', s, h', xs \,.\, \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false \\ \wedge\ stack = xs \wedge current = h' \wedge A = leaf(\_, v', s, v'', \_) \end{array} \right\}$$

and the proof for the helper function INSERTINTOSAFE:

$$\left\{\begin{array}{l}(\exists S, q, v', v'', s, h', xs\,.\ \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false\\ \wedge\ stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s, v'', \_) \wedge level = 1 \wedge found = true * Locked(current)\\ \wedge\ v \leq v'' \wedge r = true \wedge |s| < 2k)\\ \vee\\ (\exists S, v', v'', s, h', xs\,.\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false\\ \wedge\ stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s, v'', \_) \wedge level > 1 \wedge found = true * Locked(current)\\ \wedge\ v \leq v'' \wedge r = true \wedge |s| < 2k)\end{array}\right\}$$

INSERTINTOSAFE $\triangleq$ {

$$\left\{\begin{array}{l}(\exists S, q, v', v'', s, h', xs\,.\ \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false\\ \wedge\ stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s, v'', \_) \wedge level = 1 \wedge found = true * Locked(current)\\ \wedge\ v \leq v'' \wedge r = true \wedge |s| < 2k)\\ \vee\\ (\exists S, v', v'', s, h', xs\,.\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false\\ \wedge\ stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s, v'', \_) \wedge level > 1 \wedge found = true * Locked(current)\\ \wedge\ v \leq v'' \wedge r = true \wedge |s| < 2k)\end{array}\right\}$$

ADDPAIR(A, v, p);

$$\left\{\begin{array}{l}(\exists S, q, v', v'', s, h', xs\,.\ \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false\\ \wedge\ stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s \cup (v,p), v'', \_) \wedge level = 1 \wedge found = true\\ * Locked(current) \wedge v \leq v'' \wedge r = true)\\ \vee\\ (\exists S, v', v'', s, h', xs\,.\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false\\ \wedge\ stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', s \cup (v,p), v'', \_) \wedge level > 1 \wedge found = true\\ * Locked(current) \wedge v \leq v'' \wedge r = true)\end{array}\right\}$$

PUT$(A, current)$;

$$\left\{\begin{array}{l}(\exists S, v', v'', s, h', xs\,.\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false\\ \wedge\ stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s \cup (v,p), v'', \_) \wedge level = 1 \wedge found = true\\ * Locked(current) \wedge v \leq v'' \wedge r = true)\\ \vee\\ (\exists S, v', v'', s, h', xs\,.\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false\\ \wedge\ stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', s \cup (v,p), v'', \_) \wedge level > 1 \wedge found = true\\ * Locked(current) \wedge v \leq v'' \wedge r = true)\end{array}\right\}$$

UNLOCK$(current)$;

$$\left\{\begin{array}{l}(\exists S, v', v'', s, h', xs\,.\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false\\ \wedge\ stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s \cup (v,p), v'', \_) \wedge level = 1 \wedge found = true \wedge v \leq v''\\ \wedge\ r = true)\\ \vee\\ (\exists S, v', v'', s, h', xs\,.\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false\\ \wedge\ stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', s \cup (v,p), v'', \_) \wedge level > 1 \wedge found = true \wedge v \leq v''\\ \wedge\ r = true)\end{array}\right\}$$

$completed$:= TRUE;

$$\left\{\begin{array}{l}(\exists S, v', v'', s, h', xs\,.\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = true\\ \wedge\ stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s \cup (v,p), v'', \_) \wedge level = 1 \wedge found = true \wedge v \leq v''\\ \wedge\ r = true)\\ \vee\\ (\exists S, v', v'', s, h', xs\,.\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = true\\ \wedge\ stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', s \cup (v,p), v'', \_) \wedge level > 1 \wedge found = true \wedge v \leq v''\\ \wedge\ r = true)\end{array}\right\}$$

}

$$\left\{\begin{array}{l}\exists S\,.\ \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = true\\ \wedge\ r = true\end{array}\right\}$$

and the proof for the helper function INSERTINTOUNSAFEROOT:

$$\left\{\begin{array}{l}(\exists S, q, v', v'', s, h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s, v'', \_) \wedge level = 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge |s| = 2k \wedge PB = h'' : xs' \wedge h' = h'') \\ \vee \\ (\exists S, v', v'', s, h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s, v'', \_) \wedge level > 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge |s| = 2k \wedge PB = h'' : xs' \wedge h' = h'')\end{array}\right\}$$

INSERTINTOUNSAFEROOT $\triangleq$ {

$$\left\{\begin{array}{l}(\exists S, q, v', v'', s, h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s, v'', \_) \wedge level = 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge |s| = 2k \wedge PB = h'' : xs' \wedge h' = h'') \\ \vee \\ (\exists S, v', v'', s, h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s, v'', \_) \wedge level > 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge |s| = 2k \wedge PB = h'' : xs' \wedge h' = h'')\end{array}\right\}$$

$q :=$ NEW();

$$\left\{\begin{array}{l}(\exists S, q, v', v'', s, h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s, v'', \_) \wedge level = 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge |s| = 2k \wedge PB = h'' : xs' \wedge h' = h'' \wedge q \mapsto leaf(0, \_, \_, \_, \_)) \\ \vee \\ (\exists S, v', v'', s, h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s, v'', \_) \wedge level > 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge |s| = 2k \wedge PB = h'' : xs' \wedge h' = h'' \wedge q \mapsto inner(0, \_, \_, \_, \_, \_))\end{array}\right\}$$

$B :=$ REARRANGE(A, v, p, q);

$$\left\{\begin{array}{l}(\exists S, q, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s', v''', q) \wedge level = 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = h'' : xs' \wedge h' = h'' \wedge q \mapsto leaf(0, \_, \_, \_, \_) \\ \wedge\, B = leaf(0, v''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup s'') \\ \vee \\ (\exists S, p''', v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s', v''', q) \wedge level > 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = h'' : xs' \wedge h' = h'' \wedge q \mapsto inner(0, \_, \_, \_, \_, \_) \\ \wedge\, B = inner(0, v''', p''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup \{(v''', p''')\} \cup s'')\end{array}\right\}$$

PUT(B, q);

$$\left\{\begin{array}{l}(\exists S, q, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s', v''', q) \wedge level = 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = h'' : xs' \wedge h' = h'' \wedge q \mapsto B \\ \wedge\, B = leaf(0, v''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup s'') \\ \vee \\ (\exists S, p''', v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s', v''', q) \wedge level > 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = h'' : xs' \wedge h' = h'' \wedge q \mapsto B \\ \wedge\, B = inner(0, v''', p''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup \{(v''', p''')\} \cup s'')\end{array}\right\}$$

$$\left\{\begin{array}{l}(\exists S, q, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s', v''', q) \wedge level = 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = h'' : xs' \wedge h' = h'' \wedge q \mapsto B \\ \wedge\, B = leaf(0, v''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup s'') \\ \vee \\ (\exists S, p''', v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s', v''', q) \wedge level > 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = h'' : xs' \wedge h' = h'' \wedge q \mapsto B \\ \wedge\, B = inner(0, v''', p''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup \{(v''', p''')\} \cup s'')\end{array}\right\}$$

LOCK(q);

$$\left\{\begin{array}{l}(\exists S, q, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s', v''', q) \wedge level = 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = h'' : xs' \wedge h' = h'' \wedge q \mapsto B \\ \wedge\, B = leaf(0, v''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup s'' * Locked(q)) \\ \vee \\ (\exists S, p''', v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s', v''', q) \wedge level > 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = h'' : xs' \wedge h' = h'' \wedge q \mapsto B \\ \wedge\, B = inner(0, v''', p''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup \{(v''', p''')\} \cup s'' * Locked(q))\end{array}\right\}$$

PUT(A, $current$);

$$\left\{\begin{array}{l}(\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s', v''', q) \wedge level = 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = h'' : xs' \wedge h' = h'' \wedge q \mapsto B \\ \wedge\, B = leaf(0, v''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup s'' * Locked(q) \\ \wedge\, current \mapsto A) \\ \vee \\ (\exists S, p''', v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s', v''', q) \wedge level > 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = h'' : xs' \wedge h' = h'' \wedge q \mapsto B \\ \wedge\, B = inner(0, v''', p''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup \{(v''', p''')\} \cup s'' * Locked(q) \\ \wedge\, current \mapsto A)\end{array}\right\}$$

w := MINVALUE(A);

$$\left\{\begin{array}{l}(\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s', v''', q) \wedge level = 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = h'' : xs' \wedge h' = h'' \wedge q \mapsto B \\ \wedge\, B = leaf(0, v''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup s'' * Locked(q) \\ \wedge\, current \mapsto A \wedge w = v') \\ \vee \\ (\exists S, p''', v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s', v''', q) \wedge level > 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = h'' : xs' \wedge h' = h'' \wedge q \mapsto B \\ \wedge\, B = inner(0, v''', p''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup \{(v''', p''')\} \cup s'' * Locked(q) \\ \wedge\, current \mapsto A \wedge w = v')\end{array}\right\}$$

$$\left\{\begin{array}{l}(\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s', v''', q) \wedge level = 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = h'' : xs' \wedge h' = h'' \wedge q \mapsto B \\ \wedge\, B = leaf(0, v''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup s'' * Locked(q) \\ \wedge\, current \mapsto A \wedge w = v') \\ \vee \\ (\exists S, p''', v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s', v''', q) \wedge level > 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = h'' : xs' \wedge h' = h'' \wedge q \mapsto B \\ \wedge\, B = inner(0, v''', p''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup \{(v''', p''')\} \cup s'' * Locked(q) \\ \wedge\, current \mapsto A \wedge w = v') \end{array}\right.$$

$t :=$ highValue(A);

$$\left\{\begin{array}{l}(\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s', v''', q) \wedge level = 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = h'' : xs' \wedge h' = h'' \wedge q \mapsto B \\ \wedge\, B = leaf(0, v''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup s'' * Locked(q) \\ \wedge\, current \mapsto A \wedge w = v' \wedge t = v'') \\ \vee \\ (\exists S, p''', v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s', v''', q) \wedge level > 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = h'' : xs' \wedge h' = h'' \wedge q \mapsto B \\ \wedge\, B = inner(0, v''', p''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup \{(v''', p''')\} \cup s'' * Locked(q) \\ \wedge\, current \mapsto A \wedge w = v' \wedge t = v'') \end{array}\right.$$

$u :=$ highValue(B);

$$\left\{\begin{array}{l}(\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \not\in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s', v''', q) \wedge level = 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = h'' : xs' \wedge h' = h'' \wedge q \mapsto B \\ \wedge\, B = leaf(0, v''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup s'' * Locked(q) \\ \wedge\, current \mapsto A \wedge w = v' \wedge t = v'' \wedge u = v'') \\ \vee \\ (\exists S, p''', v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s', v''', q) \wedge level > 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = h'' : xs' \wedge h' = h'' \wedge q \mapsto B \\ \wedge\, B = inner(0, v''', p''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup \{(v''', p''')\} \cup s'' * Locked(q) \\ \wedge\, current \mapsto A \wedge w = v' \wedge t = v'' \wedge u = v'') \end{array}\right.$$

$$\left\{\begin{array}{l}(\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s', v''', q) \wedge level = 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = h'' : xs' \wedge h' = h'' \wedge q \mapsto B \\ \wedge\, B = leaf(0, v''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup s'' * Locked(q) \\ \wedge\, current \mapsto A \wedge w = v' \wedge t = v'' \wedge u = v'') \\ \vee \\ (\exists S, p''', v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s', v''', q) \wedge level > 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = h'' : xs' \wedge h' = h'' \wedge q \mapsto B \\ \wedge\, B = inner(0, v''', p''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup \{(v''', p''')\} \cup s'' * Locked(q) \\ \wedge\, current \mapsto A \wedge w = v' \wedge t = v'' \wedge u = v'')\end{array}\right.$$

$r :=$ NEW();

$$\left\{\begin{array}{l}(\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s', v''', q) \wedge level = 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = h'' : xs' \wedge h' = h'' \wedge q \mapsto B \\ \wedge\, B = leaf(0, v''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup s'' * Locked(q) \\ \wedge\, current \mapsto A \wedge w = v' \wedge t = v'' \wedge u = v'' \wedge r \mapsto inner(0, \_, \_, \_, \_)) \\ \vee \\ (\exists S, p''', v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s', v''', q) \wedge level > 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = h'' : xs' \wedge h' = h'' \wedge q \mapsto B \\ \wedge\, B = inner(0, v''', p''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup \{(v''', p''')\} \cup s'' * Locked(q) \\ \wedge\, current \mapsto A \wedge w = v' \wedge t = v'' \wedge u = v'' \wedge r \mapsto inner(0, \_, \_, \_, \_))\end{array}\right.$$

$R :=$ NEWNODE(w, $current$, t, q, u);

$$\left\{\begin{array}{l}(\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s', v''', q) \wedge level = 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = h'' : xs' \wedge h' = h'' \wedge q \mapsto B \\ \wedge\, B = leaf(0, v''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup s'' * Locked(q) \\ \wedge\, current \mapsto A \wedge w = v' \wedge t = v'' \wedge u = v'' \wedge r \mapsto inner(0, \_, \_, \_, \_) \\ \wedge\, R = inner(0, w, current, t, q, u)) \\ \vee \\ (\exists S, p''', v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s', v''', q) \wedge level > 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = h'' : xs' \wedge h' = h'' \wedge q \mapsto B \\ \wedge\, B = inner(0, v''', p''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup \{(v''', p''')\} \cup s'' * Locked(q) \\ \wedge\, current \mapsto A \wedge w = v' \wedge t = v'' \wedge u = v'' \wedge r \mapsto inner(0, \_, \_, \_, \_) \\ \wedge\, R = inner(0, w, current, t, q, u))\end{array}\right.$$

$PB :=$ GETPRIMEBLOCK(h);

$$\left\{\begin{array}{l}(\exists S, p, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,q) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s', v''', q) \wedge level = 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = current : xs' \wedge h' = h'' \wedge q \mapsto B \\ \wedge\, B = leaf(0, v''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup s'' * Locked(q) \\ \wedge\, current \mapsto A \wedge w = v' \wedge t = v'' \wedge u = v'' \wedge r \mapsto inner(0, \_, \_, \_, \_) \\ \wedge\, R = inner(0, w, current, t, q, u)) \\ \vee \\ (\exists S, p''', v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s', v''', q) \wedge level > 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = current : xs' \wedge h' = h'' \wedge q \mapsto B \\ \wedge\, B = inner(0, v''', p''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup \{(v''', p''')\} \cup s'' * Locked(q) \\ \wedge\, current \mapsto A \wedge w = v' \wedge t = v'' \wedge u = v'' \wedge r \mapsto inner(0, \_, \_, \_, \_) \\ \wedge\, R = inner(0, w, current, t, q, u))\end{array}\right.$$

$$\left\{\begin{array}{l}(\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s', v''', q) \wedge level = 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = current : xs' \wedge h' = h'' \wedge q \mapsto B \\ \wedge\, B = leaf(0, v''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup s'' * Locked(q) \\ \wedge\, current \mapsto A \wedge w = v' \wedge t = v'' \wedge u = v'' \wedge r \mapsto inner(0, \_, \_, \_, \_) \\ \wedge\, R = inner(0, w, current, t, q, u)) \\ \vee \\ (\exists S, p''', v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s', v''', q) \wedge level > 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = current : xs' \wedge h' = h'' \wedge q \mapsto B \\ \wedge\, B = inner(0, v''', p''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup \{(v''', p''')\} \cup s'' * Locked(q) \\ \wedge\, current \mapsto A \wedge w = v' \wedge t = v'' \wedge u = v'' \wedge r \mapsto inner(0, \_, \_, \_, \_) \\ \wedge\, R = inner(0, w, current, t, q, u))\end{array}\right\}$$

PUT(R, r);

$$\left\{\begin{array}{l}(\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s', v''', q) \wedge level = 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = current : xs' \wedge h' = h'' \wedge q \mapsto B \\ \wedge\, B = leaf(0, v''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup s'' * Locked(q) \\ \wedge\, current \mapsto A \wedge w = v' \wedge t = v'' \wedge u = v'' \wedge rR \\ \wedge\, R = inner(0, w, current, t, q, u)) \\ \vee \\ (\exists S, p''', v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s', v''', q) \wedge level > 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = current : xs' \wedge h' = h'' \wedge q \mapsto B \\ \wedge\, B = inner(0, v''', p''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup \{(v''', p''')\} \cup s'' * Locked(q) \\ \wedge\, current \mapsto A \wedge w = v' \wedge t = v'' \wedge u = v'' \wedge r \mapsto R \\ \wedge\, R = inner(0, w, current, t, q, u))\end{array}\right\}$$

ADDROOT(PB, r);

$$\left\{\begin{array}{l}(\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s', v''', q) \wedge level = 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = r : xs' \wedge h' = h'' \wedge q \mapsto B \\ \wedge\, B = leaf(0, v''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup s'' * Locked(q) \\ \wedge\, current \mapsto A \wedge w = v' \wedge t = v'' \wedge u = v'' \wedge rR \\ \wedge\, R = inner(0, w, current, t, q, u)) \\ \vee \\ (\exists S, p''', v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s', v''', q) \wedge level > 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = r : xs' \wedge h' = h'' \wedge q \mapsto B \\ \wedge\, B = inner(0, v''', p''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup \{(v''', p''')\} \cup s'' * Locked(q) \\ \wedge\, current \mapsto A \wedge w = v' \wedge t = v'' \wedge u = v'' \wedge r \mapsto R \\ \wedge\, R = inner(0, w, current, t, q, u))\end{array}\right\}$$

PUTPRIMEBLOCK(h, PB);

$$\left\{\begin{array}{l}(\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s', v''', q) \wedge level = 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = xs' \wedge h' = h'' \wedge q \mapsto B \\ \wedge\, B = leaf(0, v''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup s'' * Locked(q) \\ \wedge\, current \mapsto A) \\ \vee \\ (\exists S, p''', v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s', v''', q) \wedge level > 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = xs' \wedge h' = h'' \wedge q \mapsto B \\ \wedge\, B = inner(0, v''', p''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup \{(v''', p''')\} \cup s'' * Locked(q) \\ \wedge\, current \mapsto A)\end{array}\right\}$$

$$\left\{\begin{array}{l}(\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s', v''', q) \wedge level = 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = xs' \wedge h' = h'' \wedge q \mapsto B \\ \wedge\, B = leaf(0, v''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup s'' * Locked(q) \\ \wedge\, current \mapsto A) \\ \vee \\ (\exists S, p''', v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s', v''', q) \wedge level > 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = xs' \wedge h' = h'' \wedge q \mapsto B \\ \wedge\, B = inner(0, v''', p''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup \{(v''', p''')\} \cup s'' * Locked(q) \\ \wedge\, current \mapsto A)\end{array}\right\}$$

UNLOCK($current$);

$$\left\{\begin{array}{l}(\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false \\ \wedge\, stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s', v''', q) \wedge level = 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = xs' \wedge h' = h'' \wedge q \mapsto B \\ \wedge\, B = leaf(0, v''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup s'' * Locked(q)) \\ \vee \\ (\exists S, p''', v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false \\ \wedge\, stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s', v''', q) \wedge level > 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = xs' \wedge h' = h'' \wedge q \mapsto B \\ \wedge\, B = inner(0, v''', p''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup \{(v''', p''')\} \cup s'' * Locked(q))\end{array}\right\}$$

UNLOCK(q);

$$\left\{\begin{array}{l}(\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s', v''', q) \wedge level = 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge completed = false) \\ \vee \\ (\exists S, p''', v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s', v''', q) \wedge level > 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge completed = false)\end{array}\right\}$$

$completed$:= TRUE;

$$\left\{\begin{array}{l}(\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s', v''', q) \wedge level = 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge completed = true) \\ \vee \\ (\exists S, p''', v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s', v''', q) \wedge level > 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge completed = true\end{array}\right\}$$

}

$$\left\{\begin{array}{l}(\exists S, v', v'', s, h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s', v'', q) \wedge level = 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge completed = true) \\ \vee \\ (\exists S, p''', v', v'', s, h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', s, v'', q) \wedge level > 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge completed = true)\end{array}\right\}$$

and the proof for the helper function INSERTINTOUNSAFE:

$$\left\{\begin{array}{l}(\exists S, q, v', v'', s, h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_{1} \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s, v'', \_) \wedge level = 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge |s| = 2k \wedge PB = h'' : xs' \wedge h' \neq h'') \\ \vee \\ (\exists S, v', v'', s, h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_{1} \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s, v'', \_) \wedge level > 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge |s| = 2k \wedge PB = h'' : xs' \wedge h' \neq h'')\end{array}\right\}$$

INSERTINTOUNSAFE $\triangleq$ {

$$\left\{\begin{array}{l}(\exists S, q, v', v'', s, h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_{1} \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s, v'', \_) \wedge level = 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge |s| = 2k \wedge PB = h'' : xs' \wedge h' \neq h'') \\ \vee \\ (\exists S, v', v'', s, h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_{1} \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s, v'', \_) \wedge level > 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge |s| = 2k \wedge PB = h'' : xs' \wedge h' \neq h'')\end{array}\right\}$$

$q :=$ NEW();

$$\left\{\begin{array}{l}(\exists S, q, v', v'', s, h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_{1} \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s, v'', \_) \wedge level = 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge |s| = 2k \wedge PB = h'' : xs' \wedge h' \neq h'' \wedge q \mapsto leaf(0, \_, \_, \_, \_)) \\ \vee \\ (\exists S, v', v'', s, h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_{1} \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s, v'', \_) \wedge level > 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge |s| = 2k \wedge PB = h'' : xs' \wedge h' \neq h'' \wedge q \mapsto inner(0, \_, \_, \_, \_, \_))\end{array}\right\}$$

$B :=$ REARRANGE(A, v, p, q);

$$\left\{\begin{array}{l}(\exists S, q, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_{1} \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s', v''', \_) \wedge level = 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = h'' : xs' \wedge h' \neq h'' \wedge q \mapsto leaf(0, \_, \_, \_, \_) \\ \wedge\, B = leaf(0, v''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup s'') \\ \vee \\ (\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_{1} \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s', v''', \_) \wedge level > 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = h'' : xs' \wedge h' \neq h'' \wedge q \mapsto inner(0, \_, \_, \_, \_) \\ \wedge\, B = inner(0, v''', p''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup \{(v''', p''')\} \cup s'')\end{array}\right\}$$

PUT(B, q);

$$\left\{\begin{array}{l}(\exists S, q, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_{1} \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s', v''', \_) \wedge level = 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = h'' : xs' \wedge h' \neq h'' \wedge q \mapsto B \\ \wedge\, B = leaf(0, v''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup s'') \\ \vee \\ (\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_{1} \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s', v''', \_) \wedge level > 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = h'' : xs' \wedge h' \neq h'' \wedge q \mapsto B \\ \wedge\, B = inner(0, v''', p''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup \{(v''', p''')\} \cup s'')\end{array}\right\}$$

$$\left\{\begin{array}{l}(\exists S, q, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,q) \notin S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s', v''', \_) \wedge level = 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = h'' : xs' \wedge h' \neq h'' \wedge q \mapsto B \\ \wedge\, B = leaf(0, v''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup s'') \\ \vee \\ (\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s', v''', \_) \wedge level > 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge PB = h'' : xs' \wedge h' \neq h'' \wedge q \mapsto B \\ \wedge\, B = inner(0, v''', p''', s'', v'', \_) \wedge s \cup \{(v,p)\} = s' \cup \{(v''', p''')\} \cup s'')\end{array}\right\}$$

PUT(A, $current$);

$$\left\{\begin{array}{l}(\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s', v''', \_) \wedge level = 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true) \\ \vee \\ (\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, completed = false * Locked(current) \\ \wedge\, stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s', v''', \_) \wedge level > 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true)\end{array}\right\}$$

UNLOCK($current$);

$$\left\{\begin{array}{l}(\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s', v''', \_) \wedge level = 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge completed = false) \\ \vee \\ (\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s', v''', \_) \wedge level > 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge completed = false)\end{array}\right\}$$

$p := q$;

$$\left\{\begin{array}{l}(\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s', v''', \_) \wedge level = 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge p = q \wedge completed = false) \\ \vee \\ (\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s', v''', \_) \wedge level > 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge p = q \wedge completed = false)\end{array}\right\}$$

$v :=$ HIGHVALUE(A);

$$\left\{\begin{array}{l}(\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s', v''', \_) \wedge level = 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge p = q \wedge v = v''' \wedge completed = false) \\ \vee \\ (\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s', v''', \_) \wedge level > 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge p = q \wedge v = v''' \wedge completed = false)\end{array}\right\}$$

$level := level + 1$;

$$\left\{\begin{array}{l}(\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s', v''', \_) \wedge level = 2 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge p = q \wedge v = v''' \wedge completed = false) \\ \vee \\ (\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s', v''', \_) \wedge level > 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge p = q \wedge v = v''' \wedge completed = false)\end{array}\right\}$$

$$\left\{\begin{array}{l}(\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s', v''', \_) \wedge level = 2 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge p = q \wedge v = v''' \wedge completed = false) \\ \vee \\ (\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s', v''', \_) \wedge level > 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge p = q \wedge v = v''' \wedge completed = false)\end{array}\right\}$$

if (isEmpty($stack$)) {

$$\left\{\begin{array}{l}(\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, stack = [] \wedge current = h' \wedge A = leaf(t_{id}, v', s', v''', \_) \wedge level = 2 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge p = q \wedge v = v''' \wedge completed = false) \\ \vee \\ (\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, stack = [] \wedge current = h' \wedge A = inner(t_{id}, v', \_, s', v''', \_) \wedge level > 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge p = q \wedge v = v''' \wedge completed = false)\end{array}\right\}$$

PB := getPrimeBlock(h);

$$\left\{\begin{array}{l}(\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, stack = [] \wedge current = h' \wedge A = leaf(t_{id}, v', s', v''', \_) \wedge level = 2 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge p = q \wedge v = v''' \wedge PB = xs \wedge completed = false) \\ \vee \\ (\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, stack = [] \wedge current = h' \wedge A = inner(t_{id}, v', \_, s', v''', \_) \wedge level > 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge p = q \wedge v = v''' \wedge PB = xs \wedge completed = false)\end{array}\right\}$$

$current$:= getNodeLevel(PB, $level$);

$$\left\{\begin{array}{l}(\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, stack = [] \wedge current = h' \wedge A = leaf(t_{id}, v', s', v''', \_) \wedge level = 2 \wedge completed = false) \\ \vee \\ (\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, stack = [] \wedge current = h' \wedge A = inner(t_{id}, v', \_, s', v''', \_) \wedge level > 1 \wedge completed = false)\end{array}\right\}$$

} **else** {

$$\left\{\begin{array}{l}(\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, stack = xs \wedge current = h' \wedge A = leaf(t_{id}, v', s', v''', \_) \wedge level = 2 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge p = q \wedge v = v''' \wedge completed = false) \\ \vee \\ (\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, stack = xs \wedge current = h' \wedge A = inner(t_{id}, v', \_, s', v''', \_) \wedge level > 1 \wedge found = true \\ \wedge\, v \leq v'' \wedge r = true \wedge p = q \wedge v = v''' \wedge completed = false)\end{array}\right\}$$

$current$:= pop($stack$);

$$\left\{\begin{array}{l}(\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, stack = [] \wedge current = h' \wedge A = leaf(t_{id}, v', s', v''', \_) \wedge level = 2 \wedge completed = false) \\ \vee \\ (\exists S, v', v'', v''', s, s', s'', h', h'', xs, xs' \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \\ \wedge\, stack = [] \wedge current = h' \wedge A = inner(t_{id}, v', \_, s', v''', \_) \wedge level > 1 \wedge completed = false)\end{array}\right\}$$

}

$$\left\{\begin{array}{l}\exists S, v', v'', s, h', xs \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false \\ \wedge\, stack = xs \wedge current = h' \wedge A = inner(\_, v', \_, s, v'', \_) \wedge level > 1\end{array}\right\}$$

}

$$\left\{\begin{array}{l}\exists S, v', v'', s, h', xs \,.\, \boxed{BTree(h,S) \wedge (v,p) \in S}^{r'}_{Def(r',h)} * [CHANGE'(v)]^{r'}_1 \wedge completed = false \\ \wedge\, stack = xs \wedge current = h' \wedge A = inner(\_, v', \_, s, v'', \_) \wedge level > 1\end{array}\right\}$$

Appendix B

RGSep

B.1 Correctness of insert

We wish to prove the correctness of the insert algorithm, formally we wish to show that:

$$r := \text{INSERT}(val, p) \models \left\{ \exists pb, S \,.\, \boxed{BTree(pb, S) * true} \right\}, R', G', \left\{ \begin{array}{c} (\exists pb, S \,.\, \boxed{BTree(pb, S) * true} \\ \wedge \\ (val, p) \in S \wedge r = true) \\ \vee \\ (\exists pb, S \,.\, \boxed{BTree(pb, S) * true} \\ \wedge \\ (val, p) \in S \wedge r = false) \end{array} \right\}$$

The rely set R' is the set of actions described in the section before, except any actions that insert or delete, at the key value *val* of the insert. This would result in a non-deterministic race for the data. The guarantee set G' is the set of actions lock, unlock, insert at leaf node, insert at intermediate node, insert at root node and creating a new root.

We define loop invariants, which will be used in the correctness proof of insert, as follows:

$$\begin{array}{rcl} loopInvariant'' & \stackrel{\text{def}}{=} & \exists pb, S, M', xs \,.\, \boxed{BTree(pb, S) * true} \\ & \wedge & v = val \\ & \wedge & completed = false \\ & \wedge & stack = xs \\ & \wedge & A = M \\ & \wedge & niceNode(M, v', v'') \\ & \wedge & v' \leq v \\ & \wedge & \boxed{n \mapsto M' * true} \\ & \wedge & M' = node(\_, v', \_, \_, \_) \end{array}$$

and

$$
\begin{array}{rcl}
loopInvariant''' & \stackrel{\text{def}}{=} & \exists pb, S, M', xs \,.\, \boxed{BTree(pb, S) * true} \\
& \wedge & v = val \\
& \wedge & stack = xs \\
& \wedge & A = M \\
& \wedge & niceNode(M, v', v'') \\
& \wedge & v' \leq v \\
& \wedge & \boxed{n \mapsto M' * true} \\
& \wedge & M' = node(\_, v', \_, \_, \_) \\
& \wedge & A = node(\_, v', \_, \_, \_) \\
& \wedge & ((completed = false \wedge found = true \wedge lock(current) = true \\
& & \wedge A = node(t_{id}, \_, \_, \_, \_) \wedge v \leq highValue(A)) \\
& \vee & \\
& & (completed = false \wedge found = false \wedge lock(current) = false \\
& & \wedge A = node(t_{id}, \_, \_, \_, \_) \wedge v \leq highValue(A)) \\
& \vee & \\
& & (completed = true \wedge found = true \wedge lock(current) = false \\
& & \wedge A = node(t_{id}, \_, \_, \_, \_) \wedge isIn(A, v) = true \wedge r = false) \\
& \vee & \\
& & (completed = true \wedge found = false \wedge lock(current) = false \\
& & \wedge A = node(t_{id}, \_, \_, \_, \_) \wedge r = true) \\
& \vee & \\
& & (completed = false \wedge found = true \wedge isSafe(A) = false \\
& & \wedge isRoot(PB, current) = false \wedge r = true) \\
& \vee & \\
& & (completed = false \wedge found = true \wedge isSafe(A) = false \\
& & \wedge isRoot(PB, current) = true \wedge r = true))
\end{array}
$$

Besides that we introduce some predicates to improve the legibility of the proof. They are as follows:

$$
\begin{array}{rcl}
lock(x) & \stackrel{\text{def}}{=} & \exists A \,.\, x \mapsto A \\
& \wedge & A = node(t_{id}, \_, \_, \_, \_) \\
& \wedge & t_{id} \neq 0
\end{array}
$$

$$
\begin{array}{rcl}
isSafe(A) & \stackrel{\text{def}}{=} & \exists s, v_1, \ldots, v_i, p_1, \ldots, p_i \,.\, A = node(\_, \_, s, \_, \_) \\
& \wedge & s = [(v_1, p_1), \ldots, (v_i, p_i)] \\
& \wedge & i \leq 2k
\end{array}
$$

$$
\begin{array}{rcl}
isIn(A, v) & \stackrel{\text{def}}{=} & \exists s, p \,.\, A = node(\_, \_, s, \_, \_) \\
& \wedge & (v, p) \in s
\end{array}
$$

$$
\begin{array}{rcl}
v \leq highValue(A) & \stackrel{\text{def}}{=} & \exists v_{i+1} \,.\, A = node(\_, \_, \_, v_{i+1}, \_) \\
& \wedge & v \leq v_{i+1}
\end{array}
$$

$\{\ \exists pb, S\ .\ \boxed{BTree(pb, S) * true}\ \}$

$r :=$ INSERT(val, p) {

$\{\ \exists pb, S\ .\ \boxed{BTree(pb, S) * true}\ \}$

$v := val$;

$\{\ \exists pb, S\ .\ \boxed{BTree(pb, S) * true} \wedge v = val\ \}$

$completed :=$ FALSE;

$\{\ \exists pb, S\ .\ \boxed{BTree(pb, S) * true} \wedge v = val \wedge completed = false\ \}$

MOVEDOWNANDSTACK;

$\{\ loopInvariant'' \wedge A = leaf(\_, v', \_, \_, \_)\ \}$

$level := 1$;

$\{\ loopInvariant'' \wedge A = leaf(\_, v', \_, \_, \_) \wedge level = 1\ \}$

while $(completed =$ FALSE$)$ {

$\{\ loopInvariant''' \wedge completed = false\ \}$

$found :=$ FALSE;

$\{\ loopInvariant''' \wedge completed = false \wedge found = false\ \}$

while $(found =$ FALSE$)$ {

$\{\ loopInvariant''' \wedge completed = false \wedge found = false\ \}$

$found :=$ TRUE

$\{\ loopInvariant''' \wedge completed = false \wedge found = true\ \}$

LOCK$(current)$;

$\{\ loopInvariant''' \wedge completed = false \wedge found = true \wedge lock(current) = true\ \}$

$A :=$ GET$(current)$;

$\{\ loopInvariant''' \wedge completed = false \wedge found = true \wedge lock(current) = true \wedge A = node(t_{id}, \_, \_, \_, \_)\ \}$

$r :=$ TRUE;

$\{\ loopInvariant''' \wedge completed = false \wedge found = true \wedge lock(current) = true \wedge A = node(t_{id}, \_, \_, \_, \_)\ \}$

if $($ISIN$(A, v))$ {

$\{\ loopInvariant''' \wedge completed = false \wedge found = true \wedge lock(current) = true \wedge A = node(t_{id}, \_, \_, \_, \_)$
$\wedge\ isIn(A, v) = true\ \}$

UNLOCK$(current)$;

$\{\ loopInvariant''' \wedge completed = false \wedge found = true \wedge lock(current) = false \wedge A = node(t_{id}, \_, \_, \_, \_)$
$\wedge\ isIn(A, v) = true\ \}$

$r :=$ FALSE;

$\{\ loopInvariant''' \wedge completed = false \wedge found = true \wedge lock(current) = false \wedge A = node(t_{id}, \_, \_, \_, \_)$
$\wedge\ isIn(A, v) = true \wedge r = false\ \}$

$completed :=$ TRUE;

$\{\ loopInvariant''' \wedge completed = true \wedge found = true \wedge lock(current) = false \wedge A = node(t_{id}, \_, \_, \_, \_)$
$\wedge\ isIn(A, v) = true \wedge r = false\ \}$

} **else if** ($v >$ HIGHVALUE(A)) {

$\{\ loopInvariant''' \wedge completed = false \wedge found = true \wedge lock(current) = true \wedge A = node(t_{id}, \_, \_, \_, \_))$
$\ \ \wedge v > highValue(A)\ \}$

UNLOCK($current$);

$\{\ loopInvariant''' \wedge completed = false \wedge found = true \wedge lock(current) = false \wedge A = node(t_{id}, \_, \_, \_, \_))$
$\ \ \wedge v > highValue(A)\ \}$

$found := false$;

$\{\ loopInvariant''' \wedge completed = false \wedge found = false \wedge lock(current) = false \wedge A = node(t_{id}, \_, \_, \_, \_))$
$\ \ \wedge v > highValue(A)\ \}$

MOVERIGHT;

$\{\ loopInvariant''' \wedge completed = false \wedge found = false \wedge lock(current) = false \wedge A = node(t_{id}, \_, \_, \_, \_))$
$\ \ \wedge v \leq highValue(A)\ \}$

}

$\{\ (loopInvariant''' \wedge completed = true \wedge found = true \wedge lock(current) = false \wedge A = node(t_{id}, \_, \_, \_, \_)$
$\ \ \wedge isIn(A, v) = true \wedge r = false)$
$\ \ \vee$
$\ \ (loopInvariant''' \wedge completed = false \wedge found = false \wedge lock(current) = false \wedge A = node(t_{id}, \_, \_, \_, \_)$
$\ \ \wedge v \leq highValue(A))$
$\ \ \vee$
$\ \ (loopInvariant''' \wedge completed = false \wedge found = true \wedge lock(current) = true \wedge A = node(t_{id}, \_, \_, \_, \_)$
$\ \ \wedge v \leq highValue(A))\ \}$

}

$\{\ loopInvariant''' \wedge found = true\ \}$

if ($completed =$ FALSE) {

$\{\ loopInvariant''' \wedge found = true \wedge completed = false\ \}$

if (ISSAFE(A)) {

$\{\ loopInvariant''' \wedge found = true \wedge completed = false \wedge isSafe(A) = true\ \}$

INSERTINTOSAFE;

$\{\ loopInvariant''' \wedge completed = true\ \}$

} **else** {

$\{\ loopInvariant''' \wedge found = true \wedge completed = false \wedge isSafe(A) = false\ \}$

$PB :=$ GETPRIMEBLOCK();

$\{\ loopInvariant''' \wedge found = true \wedge completed = false \wedge isSafe(A) = false \wedge PB = N \wedge N = n : ps\ \}$

if (ISROOT($PB, current$)) {

$\{\ loopInvariant''' \wedge found = true \wedge completed = false \wedge isSafe(A) = false \wedge PB = N \wedge N = n : ps$
$\ \ \wedge isRoot(PB, current) = true\ \}$

INSERTINTOUNSAFEROOT;

$\{\ loopInvariant'''\ \}$

} **else** {

$\{\ loopInvariant''' \wedge found = true \wedge completed = false \wedge isSafe(A) = false \wedge PB = N \wedge N = n : ps$
$\ \ \wedge isRoot(PB, current) = true\ \}$

INSERTINTOUNSAFE;

$\{\ loopInvariant'''\ \}$

}

$\{\ loopInvariant'''\ \}$

}

$\{\ loopInvariant'''\ \}$

}

$\{\ (loopInvariant''' \wedge completed = true) \vee (loopInvariant''' \wedge completed = false)\ \}$

}

$\{\ (\exists pb, S\ .\ \boxed{BTree(pb, S) * true} \wedge (val, p) \in S \wedge r = true)$
$\ \ \vee$
$\ \ (\exists pb, S\ .\ \boxed{BTree(pb, S) * true} \wedge (val, p) \in S \wedge r = false)\ \}$

$\{\ \exists pb, S\ .\ \boxed{BTree(pb, S) * true} \wedge v = val \wedge completed = false\ \}$

moveDownAndStack $\triangleq$ {

$\{\ \exists pb, S\ .\ \boxed{BTree(pb, S) * true} \wedge v = val \wedge completed = false\ \}$

$stack :=$ newStack();

$\{\ \exists pb, S\ .\ \boxed{BTree(pb, S) * true} \wedge v = val \wedge completed = false \wedge stack = []\ \}$

$PB :=$ getPrimeBlock();

$\{\ \exists pb, S\ .\ \boxed{BTree(pb, S) * true} \wedge v = val \wedge completed = false \wedge stack = [] \wedge PB = N \wedge N = n : ps\ \}$

$current :=$ root(PB);

$\{\ \exists pb, S\ .\ \boxed{BTree(pb, S) * true} \wedge v = val \wedge completed = false \wedge stack = [] \wedge PB = N \wedge N = n : ps$
$\wedge\ current = n\ \}$

$A :=$ get($current$);

$\{\ \exists pb, S, M'\ .\ \boxed{BTree(pb, S) * true} \wedge v = val \wedge completed = false \wedge stack = []$
$\wedge\ current = n \wedge \boxed{n \mapsto M' * true} \wedge A = M \wedge niceNode(M, -\infty, \_) \wedge M' = node(\_, -\infty, \_, \_, \_)\ \}$

while (isLeaf(A) = false) {

$\{\ loopInvariant''\ \}$

if ($v \leq$ highValue(A)) {

$\{\ loopInvariant'' \wedge v \leq v''\ \}$

push($stack$, $current$);

$\{\ loopInvariant'' \wedge v \leq v''\ \}$

}

$\{\ loopInvariant''\ \}$

$current :=$ next(A, v);

$\{\ loopInvariant'' \wedge \boxed{n \mapsto M * true} \wedge current = n\ \}$

$A :=$ get($current$);

$\{\ loopInvariant''\ \}$

}

$\{\ loopInvariant'' \wedge A = leaf(\_, v', \_, \_, \_)\ \}$

}

$\{\ loopInvariant'' \wedge A = leaf(\_, v', \_, \_, \_)\ \}$

$\{\ loopInvariant''' \wedge completed = false \wedge isSafe(A) = true\ \}$

insertIntoSafe $\triangleq$ {

$\{\ loopInvariant''' \wedge completed = false \wedge isSafe(A) = true\ \}$

addPair(A, v, p);

$\{\ loopInvariant''' \wedge completed = false \wedge A = node(t_{id}, \_, s, \_, \_) \wedge (v, p) \in s\ \}$

put(A, $current$);

$\{\ loopInvariant''' \wedge completed = false \wedge current = n\ \}$

unlock($current$);

$\{\ loopInvariant''' \wedge completed = false \wedge unlock(current) = true\ \}$

$completed :=$ true;

$\{\ loopInvariant''' \wedge completed = true\ \}$

}

$\{\ loopInvariant''' \wedge completed = true\ \}$

$\{\ loopInvariant''' \wedge completed = false \wedge isSafe(A) = false \wedge isRoot(PB, current) = true\ \}$

insertIntoUnsafeRoot $\triangleq$ {

$\{\ loopInvariant''' \wedge completed = false \wedge isSafe(A) = false \wedge isRoot(PB, current) = true\ \}$

$q :=$ new();

$\{\ loopInvariant''' \wedge completed = false \wedge isSafe(A) = false \wedge isRoot(PB, current) = true$
$\wedge\ q = m \wedge m \mapsto N\ \}$

$B :=$ rearrange(A, v, p, q);

$\{\ loopInvariant''' \wedge completed = false \wedge isSafe(A) = false \wedge isRoot(PB, current) = true$
$\wedge\ q = m \wedge m \mapsto N \wedge B = rearrange(A, v, p, q)\ \}$

put(B, q);

$\{\ loopInvariant''' \wedge completed = false \wedge isSafe(A) = false \wedge isRoot(PB, current) = true$
$\wedge\ q = m \wedge m \mapsto N \wedge B = rearrange(A, v, p, q) \wedge N = B\ \}$

$\{\ loopInvariant''' \wedge completed = false \wedge isSafe(A) = false \wedge isRoot(PB, current) = true \wedge q = m \wedge m \mapsto N \wedge B = rearrange(A, v, p, q) \wedge N = B\ \}$

LOCK(q);

$\{\ loopInvariant''' \wedge completed = false \wedge isSafe(A) = false \wedge isRoot(PB, current) = true \wedge q = m \wedge m \mapsto N \wedge B = rearrange(A, v, p, q) \wedge N = B \wedge N = node(t_{id}, \_, \_, \_, \_)\ \}$

PUT(A, $current$);

$\{\ loopInvariant''' \wedge completed = false \wedge isSafe(A) = false \wedge isRoot(PB, current) = true \wedge q = m \wedge \boxed{m \mapsto N'} \wedge B = rearrange(A, v, p, q) \wedge N' = node(t_{id}, \_, \_, \_, \_)\ \}$

w := MINVALUE(A);

$\{\ loopInvariant''' \wedge completed = false \wedge isSafe(A) = false \wedge isRoot(PB, current) = true \wedge q = m \wedge \boxed{m \mapsto N'} \wedge B = rearrange(A, v, p, q) \wedge N' = node(t_{id}, \_, \_, \_, \_) \wedge A = node(\_, w, \_, \_, \_)\ \}$

v := HIGHVALUE(A);

$\{\ loopInvariant''' \wedge completed = false \wedge isSafe(A) = false \wedge isRoot(PB, current) = true \wedge q = m \wedge \boxed{m \mapsto N'} \wedge B = rearrange(A, v, p, q) \wedge N' = node(t_{id}, \_, \_, \_, \_) \wedge A = node(\_, w, \_, v, \_)\ \}$

u := HIGHVALUE(B);

$\{\ loopInvariant''' \wedge completed = false \wedge isSafe(A) = false \wedge isRoot(PB, current) = true \wedge q = m \wedge \boxed{m \mapsto N'} \wedge B = rearrange(A, v, p, q) \wedge N' = node(t_{id}, \_, \_, \_, \_) \wedge A = node(\_, w, \_, \_, \_) \wedge B = node(\_, \_, \_, u, \_)\ \}$

r := NEW();

$\{\ loopInvariant''' \wedge completed = false \wedge isSafe(A) = false \wedge isRoot(PB, current) = true \wedge q = m \wedge \boxed{m \mapsto N'} \wedge B = rearrange(A, v, p, q) \wedge N' = node(t_{id}, \_, \_, \_, \_) \wedge A = node(\_, w, \_, \_, \_) \wedge B = node(\_, \_, \_, u, \_) \wedge r = o \wedge o \mapsto O\ \}$

R := NEWNODE(w, $current$, v, q, u);

$\{\ loopInvariant''' \wedge completed = false \wedge isSafe(A) = false \wedge isRoot(PB, current) = true \wedge q = m \wedge \boxed{m \mapsto N'} \wedge B = rearrange(A, v, p, q) \wedge N' = node(t_{id}, \_, \_, \_, \_) \wedge A = node(\_, w, \_, \_, \_) \wedge B = node(\_, \_, \_, u, \_) \wedge r = o \wedge o \mapsto O \wedge R = inner(0, w, current, z, u, \_) \wedge (v, q) \in z\ \}$

PB := GETPRIMEBLOCK();

$\{\ loopInvariant''' \wedge completed = false \wedge isSafe(A) = false \wedge isRoot(PB, current) = true \wedge q = m \wedge \boxed{m \mapsto N'} \wedge B = rearrange(A, v, p, q) \wedge N' = node(t_{id}, \_, \_, \_, \_) \wedge A = node(\_, w, \_, \_, \_) \wedge B = node(\_, \_, \_, u, \_) \wedge r = o \wedge o \mapsto O \wedge R = inner(0, w, current, z, u, \_) \wedge (v, q) \in z \wedge PB = N \wedge N = n : ps\ \}$

PUT(R, r);

$\{\ loopInvariant''' \wedge completed = false \wedge isSafe(A) = false \wedge isRoot(PB, current) = true \wedge q = m \wedge \boxed{m \mapsto N'} \wedge B = rearrange(A, v, p, q) \wedge N' = node(t_{id}, \_, \_, \_, \_) \wedge A = node(\_, w, \_, \_, \_) \wedge B = node(\_, \_, \_, u, \_) \wedge r = o \wedge o \mapsto O \wedge R = inner(0, w, current, z, u, \_) \wedge (v, q) \in z \wedge PB = N \wedge N = n : ps \wedge R = O\ \}$

ADDROOT(PB, r);

$\{\ loopInvariant''' \wedge completed = false \wedge isSafe(A) = false \wedge isRoot(PB, current) = true \wedge q = m \wedge \boxed{m \mapsto N'} \wedge B = rearrange(A, v, p, q) \wedge N' = node(t_{id}, \_, \_, \_, \_) \wedge A = node(\_, w, \_, \_, \_) \wedge B = node(\_, \_, \_, u, \_) \wedge r = o \wedge o \mapsto O \wedge R = inner(0, w, current, z, u, \_) \wedge (v, q) \in z \wedge PB = N \wedge N = r : n : ps\ \}$

PUTPRIMEBLOCK(PB);

$\{\ loopInvariant''' \wedge completed = false \wedge isSafe(A) = false \wedge q = m \wedge \boxed{m \mapsto N'} \wedge B = rearrange(A, v, p, q) \wedge N' = node(t_{id}, \_, \_, \_, \_) \wedge A = node(\_, w, \_, \_, \_) \wedge B = node(\_, \_, \_, u, \_) \wedge r = o \wedge o \mapsto O \wedge R = inner(0, w, current, z, u, \_) \wedge (v, q) \in z \wedge PB = N \wedge N = n : ps \wedge primeBlock \mapsto PB\ \}$

UNLOCK($current$);

$\{\ loopInvariant''' \wedge completed = false \wedge isSafe(A) = false \wedge q = m \wedge \boxed{m \mapsto N'} \wedge B = rearrange(A, v, p, q) \wedge N' = node(t_{id}, \_, \_, \_, \_) \wedge A = node(\_, w, \_, \_, \_) \wedge B = node(\_, \_, \_, u, \_) \wedge r = o \wedge o \mapsto O \wedge R = inner(0, w, current, z, u, \_) \wedge (v, q) \in z \wedge lock(current) = false\ \}$

UNLOCK(q);

$\{\ loopInvariant''' \wedge completed = false \wedge isSafe(A) = false \wedge q = m \wedge \boxed{m \mapsto N'} \wedge B = rearrange(A, v, p, q) \wedge N' = node(0, \_, \_, \_, \_) \wedge A = node(\_, w, \_, \_, \_) \wedge B = node(\_, \_, \_, u, \_) \wedge r = o \wedge o \mapsto O \wedge R = inner(0, w, current, z, u, \_) \wedge (v, q) \in z\ \}$

$completed$:= TRUE;

$\{\ loopInvariant''' \wedge completed = true \wedge isSafe(A) = false \wedge q = m \wedge \boxed{m \mapsto N'} \wedge B = rearrange(A, v, p, q) \wedge N' = node(0, \_, \_, \_, \_) \wedge A = node(\_, w, \_, \_, \_) \wedge B = node(\_, \_, \_, u, \_) \wedge r = o \wedge o \mapsto O \wedge R = inner(0, w, current, z, u, \_) \wedge (v, q) \in z\ \}$

}

$\{\ loopInvariant''' \wedge completed = true\ \}$

$\{\ loopInvariant''' \wedge completed = false \wedge isSafe(A) = false \wedge isRoot(PB, current) = false\ \}$

INSERTINTOUNSAFE $\triangleq$ {

$\{\ loopInvariant''' \wedge completed = false \wedge isSafe(A) = false \wedge isRoot(PB, current) = false\ \}$

$q :=$ NEW();

$\{\ loopInvariant''' \wedge completed = false \wedge isSafe(A) = false \wedge isRoot(PB, current) = false$
$\wedge q = m \wedge m \mapsto N\ \}$

$B :=$ REARRANGE(A, v, p, q);

$\{\ loopInvariant''' \wedge completed = false \wedge isSafe(A) = false \wedge isRoot(PB, current) = false$
$\wedge q = m \wedge m \mapsto N \wedge B = rearrange(A, v, p, q)\ \}$

PUT(B, q);

$\{\ loopInvariant''' \wedge completed = false \wedge isSafe(A) = false \wedge isRoot(PB, current) = false$
$\wedge q = m \wedge m \mapsto N \wedge B = rearrange(A, v, p, q) \wedge N = B \wedge N = node(0, \_, \_, \_, \_)\ \}$

PUT$(A, current)$;

$\{\ loopInvariant''' \wedge completed = false \wedge isSafe(A) = false \wedge isRoot(PB, current) = false$
$\wedge q = m \wedge \boxed{m \mapsto N'} \wedge B = rearrange(A, v, p, q) \wedge N' = node(0, \_, \_, \_, \_)\ \}$

UNLOCK$(current)$;

$\{\ loopInvariant''' \wedge completed = false \wedge isSafe(A) = false \wedge isRoot(PB, current) = false$
$\wedge q = m \wedge \boxed{m \mapsto N'} \wedge B = rearrange(A, v, p, q) \wedge N' = node(0, \_, \_, \_, \_) \wedge n = node(0, \_, \_, \_, \_)\ \}$

$p := q$;

$\{\ loopInvariant''' \wedge completed = false \wedge isSafe(A) = false \wedge isRoot(PB, current) = false$
$\wedge q = m \wedge \boxed{m \mapsto N'} \wedge B = rearrange(A, v, p, q) \wedge N' = node(0, \_, \_, \_, \_) \wedge n = node(0, \_, \_, \_, \_) \wedge p = q\ \}$

$v :=$ HIGHVALUE(A);

$\{\ loopInvariant''' \wedge completed = false \wedge isSafe(A) = false \wedge isRoot(PB, current) = false$
$\wedge q = m \wedge \boxed{m \mapsto N'} \wedge B = rearrange(A, v, p, q) \wedge N' = node(0, \_, \_, \_, \_) \wedge n = node(0, \_, \_, \_, \_) \wedge p = q$
$A = node(\_, \_, \_, v, \_)\ \}$

$level := level + 1$;

$\{\ loopInvariant''' \wedge completed = false \wedge isSafe(A) = false \wedge isRoot(PB, current) = false$
$\wedge q = m \wedge \boxed{m \mapsto N'} \wedge B = rearrange(A, v, p, q) \wedge N' = node(0, \_, \_, \_, \_) \wedge n = node(0, \_, \_, \_, \_) \wedge p = q$
$A = node(\_, \_, \_, v, \_)\ \}$

if (ISEMPTY$(stack)$) {

$\{\ loopInvariant''' \wedge completed = false \wedge isSafe(A) = false \wedge isRoot(PB, current) = false$
$\wedge q = m \wedge \boxed{m \mapsto N'} \wedge B = rearrange(A, v, p, q) \wedge N' = node(0, \_, \_, \_, \_) \wedge n = node(0, \_, \_, \_, \_) \wedge p = q$
$A = node(\_, \_, \_, v, \_) \wedge stack = []\ \}$

$current :=$ GETNODELEVEL$(PB, level)$;

$\{\ loopInvariant''' \wedge completed = false \wedge isSafe(A) = false \wedge isRoot(PB, current) = false$
$\wedge q = m \wedge \boxed{m \mapsto N'} \wedge B = rearrange(A, v, p, q) \wedge N' = node(0, \_, \_, \_, \_) \wedge n = node(0, \_, \_, \_, \_) \wedge p = q$
$A = node(\_, \_, \_, v, \_) \wedge stack = [] \wedge PB = N \wedge N = n_1, \ldots, n_n\ \}$

$PB :=$ GETPRIMEBLOCK();

$\{\ loopInvariant''' \wedge completed = false \wedge isSafe(A) = false \wedge isRoot(PB, current) = false$
$\wedge q = m \wedge \boxed{m \mapsto N'} \wedge B = rearrange(A, v, p, q) \wedge N' = node(0, \_, \_, \_, \_) \wedge n = node(0, \_, \_, \_, \_) \wedge p = q$
$A = node(\_, \_, \_, v, \_) \wedge stack = [] \wedge PB = N \wedge N = n_1, \ldots, n_n \wedge current = n_{level}\ \}$

} **else** {

$\{\ loopInvariant''' \wedge completed = false \wedge isSafe(A) = false \wedge isRoot(PB, current) = false$
$\wedge q = m \wedge \boxed{m \mapsto N'} \wedge B = rearrange(A, v, p, q) \wedge N' = node(0, \_, \_, \_, \_) \wedge n = node(0, \_, \_, \_, \_) \wedge p = q$
$A = node(\_, \_, \_, v, \_) \wedge stack = x : xs\ \}$

$current :=$ POP$(stack)$;

$\{\ loopInvariant''' \wedge completed = false \wedge isSafe(A) = false \wedge isRoot(PB, current) = false$
$\wedge q = m \wedge \boxed{m \mapsto N'} \wedge B = rearrange(A, v, p, q) \wedge N' = node(0, \_, \_, \_, \_) \wedge n = node(0, \_, \_, \_, \_) \wedge p = q$
$A = node(\_, \_, \_, v, \_) \wedge current = x \wedge stack = x : xs\ \}$

}

$\{\ (loopInvariant''' \wedge completed = false \wedge isSafe(A) = false \wedge isRoot(PB, current) = false$
$\wedge q = m \wedge \boxed{m \mapsto N'} \wedge B = rearrange(A, v, p, q) \wedge N' = node(0, \_, \_, \_, \_) \wedge n = node(0, \_, \_, \_, \_) \wedge p = q$
$A = node(\_, \_, \_, v, \_) \wedge stack = [] \wedge PB = N \wedge N = n : ps \wedge current = n)$
$\vee$
$(loopInvariant''' \wedge completed = false \wedge isSafe(A) = false \wedge isRoot(PB, current) = false$
$\wedge q = m \wedge \boxed{m \mapsto N'} \wedge B = rearrange(A, v, p, q) \wedge N' = node(0, \_, \_, \_, \_) \wedge n = node(0, \_, \_, \_, \_) \wedge p = q$
$A = node(\_, \_, \_, v, \_) \wedge current = x \wedge stack = x : xs)\ \}$

}

$\{\ loopInvariant''' \wedge completed = false\ \}$