The Theory and Verification of B+ Tree Cursor Relations

Brian McSwiggen

Advisor: Professor Andrew Appel

Submitted in partial fulfillment
of the requirements for the degree of
Bachelor of Science in Engineering

Department of Computer Science
Princeton University
May 2018

Abstract

In this thesis, I make progress toward a fully verified functional B+ tree program in the Coq proof assistant. A verified functional B+ tree would serve as the basis for the future verification of an imperative B+ tree program through proving equivalence, which would have applications in databases and other systems. The primary focus of the properties and abstractions covered in this thesis is to examine and describe the theory of the B+ tree cursor, which is central to the operations of a B+ tree both functionally and imperatively. The functional implementation presented here covers the essential cursor operations as well as B+ tree lookups and insertions, but does not include a delete function. The progress made toward verification includes the development of a complete abstract specification, the structural correctness of cursor creation and positioning, and the relation of cursors and B+ tree operations to an element-list abstract cursor representation.

Acknowledgements

As my time at Princeton comes rushing to an end, I would like to give a heartfelt thanks to the many, many people who helped me make it, who carried me through this wild place, and who sometimes had to drag me kicking and screaming. I couldn't possibly fit all of you on this page, but I'm going to make the font small and try my best.

First, to Professor Appel, I wouldn't have any of this thesis or even know what formal verification is without having you as my advisor and my teacher. Thank you, for believing in me, for meeting with me every week, for giving me help and feedback as often as I asked for it, and for showing me all the cool things these tools can do. Also to Professor Walker, my second reader, thanks for teaching me functional programming in the first place.

To Aurèle and Tosin, thank you both for taking the time to explain things to me and answer my questions. Thank you also for letting an undergrad contribute as much as he can, even though you probably could have done my whole thesis, better, in half the time. I hope what I've done is useful to you anyways.

Thank you to all the professors and teachers and mentors and tutors and TAs that I have had throughout all my time in school, even before Princeton: Dr. McCall, Mr. Plummer, Madame Hecker, Dr. Kairet, Mrs. Mentor, and Mrs. Lewis, to name just a few.

To my two crazy families on campus—Theatre Intime and Quad—you may have driven me crazy from stress but you also made it all worth it. I can't imagine a better place to be stuck working until 2am. I'm so proud of what we accomplished, and I have so much faith in the new officers to do even better.

To the Q33, forever in my heart, forever in my mind—but *never* in the TV room.

Kika, you made my senior year wonderful, and you've been the best companion I could have asked for on this wild ride. I have full confidence that your senior year will be even more wonderful, more fruitful, and produce a better thesis than this one.

Some other people deserve a special shout-out for making sure I didn't literally fail. Katie, for dragging me out of bed with Nutella for that god-awful 9am freshman physics precept. Julie, for placing bets on how long I could go without missing a class. Zach, for getting breakfast with me whenever I was worried I might sleep through a morning meeting. Swebb and Taylor, for always picking up the phone, even at ungodly hours of the night (or morning). Lydia, who has heard me rant about more stupid problems than anyone. Jillian, for knowing me since middle school, and for somehow remaining my friend even after finding out that I'm basically still a child.

I of course have to thank all the people who are already PTL, and all my friends in the Class of 2017, for proving that it can in fact be done. I'm not sure I would believe it otherwise.

Finally, to my family. My parents, for supporting me and guiding me (and even proofreading my thesis!). My brothers, for being role models, mentors, and friends. My Oma and Opa, for making it all possible.

Thank you all. I love you all so much.

This thesis represents my own work in accordance with University regulations.

Brian McSwiggen

May 7, 2018

Contents

1 Introduction and Motivation

B+ trees are an extension of the more commonly known B-trees, which are N-ary trees often used to organize data in file systems. The property that makes B+ trees unique is that the *values*, that is the data actually being stored in the tree, are kept exclusively at the leaves. The interior nodes hold only keys and pointers to their children.

Databases and file systems often use B+ trees, especially for indexing, because they can efficiently lookup a value and do a range query. Because B+ trees don't have to store as much data in the interior nodes, a single node can fit more pointers, meaning that the fanout can be larger and the tree needs fewer levels, so data can be accessed more quickly [1]. A well-designed B+ tree will store enough values per leaf that a single leaf node just fits into a single storage block, allowing the maximum possible amount of data retrieval with the fewest possible I/O operations. Additionally, because all of the values are in the leaves and not buried in higher nodes of the tree, it is relatively easy to do a query for all keys within a certain range. Some implementations augment the tree with a linked list between leaf nodes to make range queries even easier.

Because this is a data structure commonly used by databases and file systems [1], it is a critical component underlying a great portion of the systems and software we rely on every day. On the one hand, that means that B+ tree implementations used in practice are likely to be very well tested; on the other hand, bugs or failures have the potential to be wide-reaching and catastrophic.

That risk of failure is why rigorous testing practices are widely recognized as crucial to making good software that works correctly. However, in the most critical cases, more certainty may be necessary. The development of “formal methods” has provided an option for a greater degree of confidence in a range of critical applications. Formal methods encompass a range of approaches to applying strict mathematical rigor to software and hardware design, and “formal verification” in particular is the practice of rigorously proving algorithmic correctness. Because of the high certainty attained through formal methods, they have been applied to three categories of software: safety-critical, security-critical, and systems-critical programs.

Safety-critical software (or hardware) is a category of systems with direct impact on human life and safety. For example, if an airplane processor fails, if a nuclear reactor control has a bug, or if a defibrillator misfires, the consequences could very likely include death or serious injury. For that reason, software development for these systems has sometimes turned to formal methods to ensure correctness. In one example from as far back as the 1970s, NASA commissioned an aircraft control computer called SIFT which was subject to very strict failure-rate limits. Formal verification was used to ensure that the SIFT software for using redundant systems to detect and recover from hardware failures would work correctly [2].

Security-critical software, while it does not always carry the same risk to human life, does carry the risk of compromising crucial cryptographic tools, data privacy and integrity, and more. A lot of security-critical software is incredibly important to modern technology, but is also very intricate, hard to implement correctly, and hard to keep secure. Formal verification provides a means of reasoning about these tricky tools. In 2015, the Verified Software Toolchain group completed a full verification of an OpenSSL HMAC implementation, meaning that as long as SHA-256 is in fact cryptographically secure (which remains unproven), the HMAC is as well [3].

Systems-critical software carries a great amount of risk because many other systems and pieces of software rely on it. While it may not itself carry risk to human life, or to cryptographic security, if any safety-critical or security-critical software runs on a system then it carries that risk indirectly. This has led a push to develop fully verified system stacks, such as the CLInc stack of compiler, assembler, kernel, and microprocessor, which were all verified correct in the 1980s [2]. A more relevant example to this project is the Verified Software Toolchain developed by Andrew Appel at Princeton, which also includes the CompCert verified C-language compiler developed by Xavier Leroy at INRIA [4].

Because of the many systems that directly or indirectly depend on B+ trees, the verification of such an implementation falls under the systems-critical category. Formally proving a B+ tree implementation correct increases our trust in the storage systems we use underneath any other piece of software, including any safety- or security-critical applications that may be built on top.

2 Previous Work

Formal verification is a rich field of study, and a large body of research has focused both on verifying software and on tools to make the verification of software easier. Search trees, as an important and complex class of data structures, have been the subject of previous verification. Recent work at Princeton and other universities has focused on the science and tools of specification and verification.

Various interesting search tree variants, in both functional[1] and imperative[2] languages, have been the subject of verification work. An educational specification and proof of basic binary search trees can be found in Verified Functional Algorithms [5] by Professor Andrew Appel of Princeton University, who also published an efficient verified functional implementation of Red-Black trees in 2011 [6]. Appel's paper in turn builds upon the work of Filliâtre and Letouzey in 2004, in which they verified functional implementations of finite sets in three ways: using ordered lists, AVL trees, and Red-Black trees [7]. This paper was a powerful demonstration of the use of Coq modules to contain both definitions and proven properties about those definitions, and this approach has been used here as well in the abstract specification.

Functional program verification such as the work done by Appel, Filliâtre, and Letouzey serves a practical purpose both because it can help find bugs in current functional libraries (as Filliâtre and Letouzey did with the OCaml Set module based on AVL trees [7]) and because it can be extracted from Coq into live OCaml code to be used in practice. However, many practical applications rely on imperative

[1] *Functional* programming refers to a programming paradigm by which execution is the evaluation of a mathematical function, and all data are immutable (i.e. there is no program state). Well-known functional programming languages include Haskell, Lisp, and OCaml.

[2] *Imperative* programming refers to a programming paradigm by which state-based programs are built up from individual statements, or commands. Well-known imperative programming languages include C, Java, and Python.

software. As an example in the domain of search trees, Xiwen Chen of York University provided a verified concurrent imperative binary search tree which he had proven linearizable, i.e. that any operation performs correctly as if it executed immediately [8]. Although it is more difficult to reason about imperative programs than about purely functional ones, doing imperative verifications is crucial for practical application of formal verification.

There are many approaches and tools which have been developed and can be used to reason about imperative software. Although this thesis focuses on the verification of a purely functional program, such a functional verification can serve as a crucial intermediate step in proving properties of an imperative program. The functional B+ trees discussed here will be used to ultimately prove correctness of a B+ tree implementation written in C. The tools and techniques that will be used for the *imperative* proof provide a framework for what a useful *functional* B+ tree verification is, and are described in the following sections.

2.1 Separation Logic

A group at Princeton University and the National University of Singapore very recently included a demonstration of a verified binary search tree as part of a paper on separation logic and the "magic wand" connective (separating implication) [9]. *Separation logic* is a set of logic rules extending Hoare logic that facilitate proofs about variably sized imperative data structures (such as arrays or trees) by allowing the program state to be separated into disjoint parts about which properties can be proven [10].

Separation logic is crucial because having to reason about the entirety of the program state together quickly gets very complicated. On top of the desired properties, there have to be assertions for exactly what parts of the program state any given command affects, and assertions that no other part of the program is affected, and assertions that any two portions of a program intended to be affected differently are in fact disjoint [10]. It is easy to see how the whole structure does not scale well within Hoare logic, which is why separation logic is so important for the verification of many imperative programs.

2.2 Verified Software Toolchain

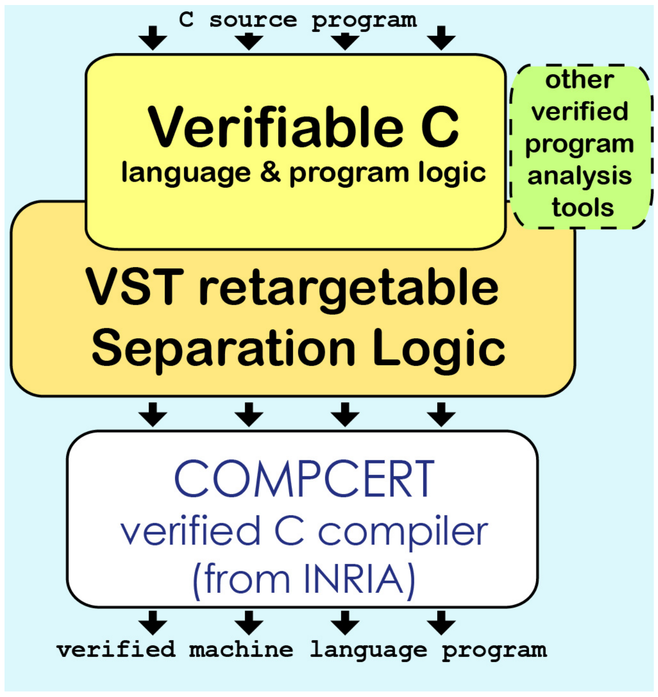
C source program
Verifiable C
language & program logic
other verified program analysis tools
VST retargetable Separation Logic
COMPCERT
verified C compiler
(from INRIA)
verified machine language program

Figure 1: Diagram of VST components [11]

Separation logic is crucial not just to a proof about a particular program, such as this B+ tree implementation, but also to the entire verification stack that it is based on. Since the mid-2000s, a group at Princeton University led by Professor Andrew Appel has been developing the *Verified Software Toolchain* [11], a stack of verified components that allow proofs to be written about source programs and apply to the compiled program as well. Each component of the toolchain, from the Verifiable C program logic to Xavier Leroy's CompCert C compiler to the machine language operational semantics (and all of the interfaces between them), is designed for verification and proven correct [12].

Although the scope of this thesis does not extend into connecting the B+ tree specification to an imperative C program that it represents, the functional specification produced could be useful for the verification of a C program. The strategy of functional specification used in this thesis is intended to work with the Verified Software Toolchain, so that a C program could be proved via the VST separation logic to be equivalent to this functional specification.

2.3 DeepSpec

The Verified Software Toolchain itself is part of a larger research group called DeepSpec. Since 2016, the DeepSpec group at Princeton, the University of Pennsylvania, Yale, and MIT have been collaborating under an NSF Expedition in Computing grant to explore the science of deep specifications. This thesis is in part an attempt to extend that work by applying deep specification to a B+ tree database program. Deep specifications, as described by the DeepSpec group, are program specifications with four special properties [13]:

1. They are **rich**, that is, they can describe the complex behavior of real programs.

2. They are **two-sided**. Specifications function analogously to interfaces in modular programming: they describe the boundary between two independent parts. A two-sided specification is one that is exercised from both sides of

the boundary, i.e. it matches both the program it describes and the programs that use it.

3. They are **formal**, meaning they are written in exact mathematical terms. Specifications that are provided only as vague or intuitive definitions may have holes or errors that are hard to spot. Formal specifications can be machine-checked and tested.

4. They are **live**, that is, they are connected directly to the implementation and the client code. If either changes, then the machine-checked proofs may no longer pass; this ensures that when the proofs are verified we can be sure they describe the actual program.

In the course of their work, the DeepSpec group and their collaborators are developing an extensive network of systems-related programs connected at deep specification interfaces. The goal is to be able to construct a verified stack from the operating system and applications like the B+ tree database all the way down to the machine language and transistors at the core of the computer.

The previously mentioned Verified Software Toolchain being developed by Appel is one part of this stack. It also includes a verified LLVM (VeLLVM) developed by Steve Zcancewic at UPenn, a verified operating system (CertiKOS) developed by Zhong Shao at Yale, and many other tools, systems, and applications.

3 Project Approach and Scope

The strategy that this thesis targets for verifying imperative software is through a *functional model*. A functional model describes the actual behavior of the operations. In this case the model is an equivalent piece of software written in a functional programming language, but other functional models are possible. When reasoning about a programming language, for example, the functional model might be the small-step operational semantics of the language. For the purposes of this thesis, since functional programs are much easier to formally reason about, the functional model provides a bridge: we can prove the equivalence of the functional program and the imperative program, and then any proofs of correctness for the functional program are also valid proofs of the imperative one.

One step beyond the functional model is the *abstract specification*. This describes, in the most pure and abstract terms possible, the goal for the program. The abstract specification defines what it *means* to be a "correct implementation" of the desired program. Proving these properties of the functional model proves that the model is correct, and therefore any imperative program equivalent to the model is also correct.

The scope of this thesis covers a B+ tree abstract specification, a functional implementation, and a partially formalized proof of the implementation's correctness relative to the specification. Once completely formally verified, the functional program could be used as the functional model for an imperative program, but that step is not covered in this work.

The particular style of B+ tree and set of operations which are targeted in this work are derived from a subset of the SQLite commands. SQLite uses B+ trees to store information in relational databases. One particularity of SQLite is that it represents a cursor into the B+ tree as a list of pointers to the split point at each level. This makes it easier to change the position of the cursor without needing to retraverse the tree.

4 B+ Tree Details

A B+ tree is a type of ordered tree with three important properties:

1. Any B+ tree has a *max fanout* value b which represents the maximum number of children that any node can hold. Every node in the B+ tree except the root must have between $\lceil \frac{b}{2} \rceil$ and b children; the root may have as few as two (if it is an interior node) or one (if it is a leaf node) [1].

2. All *values* are stored in leaf nodes. Interior nodes only store keys and pointers to the child nodes. This means that an interior node with n children only needs to store $n-1$ keys, and also that keys may repeat in the tree [1].

3. As new key-value pairs are inserted, the tree only grows up at the root, not down at the leaves: when a node gets too big and splits, both halves become children of the original parent node, and no new levels are added to the tree. The only way for a new level to be added is for the root node to split and create a new root node above it. Similarly, as key-value pairs are deleted, the tree shrinks at the root rather than at the leaves. This ensures that the whole tree always stays balanced, that is, the distance from any leaf to the root is the same as the distance from any other leaf to the root [1].

An interior node in a B+ tree might look like the following:

P_1	K_1	P_2	K_2	...	K_{n-1}	P_n

Table 1: B+ tree interior node with $n-1$ keys and n subtrees

Where each P_i is a pointer to a child node, and each K_i is a key. The fanout n is constrained by $\lceil \frac{b}{2} \rceil \leq n \leq b$, the keys K must be sorted $(\forall i, j : K_i < K_j)$, and each key K constrains the values in the child nodes to either side: for every key X in the node at P_i, $K_{i-1} \leq X < K_i$. Of course, for the end pointers, only one side of that inequality will hold [1].

A leaf node holds values rather than child pointers, and must have exactly one key per value. Therefore, a leaf node might look like the following:

(K_1, V_1)	(K_2, V_2)	...	(K_n, V_n)

Table 2: B+ tree leaf node with n keys and n values

In this case, each V_i is some value stored by the B+ tree (often, but not necessarily, a pointer to a block of data in memory), and each K_i is once again a key. The keys must be sorted just as in the interior nodes, and it still holds that the fanout n is constrained by $\lceil \frac{b}{2} \rceil \leq n \leq b$ [1].

With this structure, we now consider how to do database operations on this structure. In particular, we describe Lookup (finding a key/value in a b+tree), Insert (adding a key/value pair), and Delete (removing a key/value pair).

For all three operations, we will consider the following b+tree for examples:

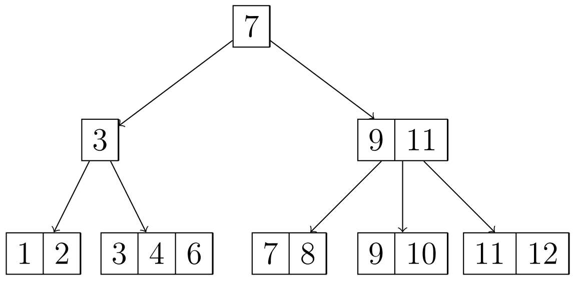

Figure 2: A simple B+ tree of height 3. Each interior node has 2 or 3 children.

Notice that all keys and subtrees are properly ordered, that all leaves are at depth 2 (where the root is depth 0), and that keys may appear multiple times within the tree. The fanout is $b = 3$, meaning each node may have two or three children (this simple example is actually a variant of a 2-3 tree). For these examples, the values at the leaves have been left out, since they are irrelevant to the tree operations.

4.1 Lookup

Finding a key-value pair in the tree proceeds from the root. At each level, we search through the list of keys to find the split where the value we want falls. Since every node has a sorted list of keys, we can do binary search within a list if desired. However, the number of children is often small enough that a simpler linear search works as well, especially when the fanout is optimized for the size of CPU cache lines, rather than full disk pages.

If we were to search for 10 in the example tree, we would first note that $7 < 10$ and go to the second child of the root. We would then note that $9 \leq 10 < 11$, and go to the middle child, where we would find 10.

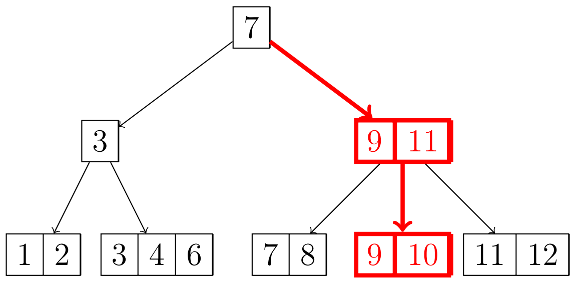

Figure 3: To find 10, we search for 10 at each level and recursively descend.

4.2 Insert

Insertion is simple if the leaf node has less than b children: we simply find the correct place according to the list of keys, and add the value there. However, we must maintain that any given node has at most b children. Therefore, if a key is inserted which would go into a leaf that has b children already, then we have to split that node and add the middle key to the node above [1].

If that node is also full, then we have to split it in turn, and so on up to the root where, if necessary, the root will split and the tree will grow by one level [1].

For a leaf node, since every key must appear mapped to its value, we keep the middle key in the leaf as well. However, for any interior node that splits, the middle key which goes up to the next level is removed from the level that split [1].

If we were to insert the key-value pair $(5, V_5)$ in the example tree, we would have to first find where in a leaf node 5 should go (between 4 and 6, on the left). Then, seeing that the leaf would have 4 key-value pairs, we split it in two and copy 5 up to the next level.

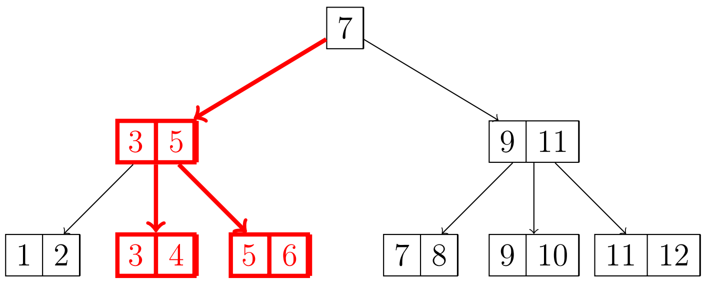

Figure 4: To add a mapping for 5, we have to first find where 5 goes in the tree, then add the new key-value pair and split nodes as necessary.

4.3 Delete

Deletion works intuitively as the inverse of insertion. It is simple if the leaf node has at least $\lceil \frac{b}{2} \rceil + 1$ entries; then we can just remove the entry and leave the rest of the tree as it is. However, if the leaf node would be left with less than $\lceil \frac{b}{2} \rceil$ entries after deletion, then we need to redistribute entries in order to maintain the fanout. That can happen by moving over entries from adjacent nodes (and changing the keys in the parent node), or by merging a node with adjacent nodes to reduce the total number of children. In the latter case, this reduction in the number of children may cause the parent to have to redistribute or merge as well, potentially propagating up to the root and decreasing the depth of the tree by one [1].

If we were to delete the key 9 from our example tree, we would see that its leaf now has only 1 entry. Since both adjacent nodes are too small to redistribute, we instead have to merge them.

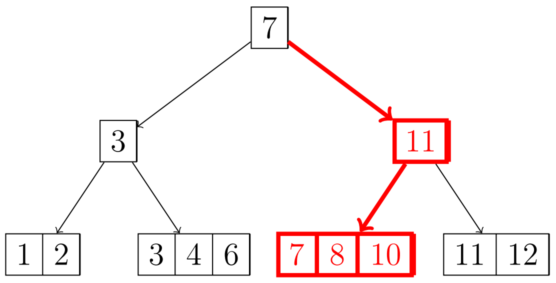

Figure 5: When we delete 9, we may need to merge nodes together.

5 Abstract Specification

Our abstract specification of a B+ tree is that of a lookup table. That is, it stores data indexed by keys, and allows that data to be arbitrarily fetched, changed, added to, or removed. However, we add one more layer to this particular table abstraction. B+ trees rely on a cursor, which could be any way of pointing to a particular place in the table. We build this notion into our abstract specification, allowing us to prove correctness relative to this abstract notion of a cursor in addition to a table.

Figure 6 and **Figure** 7 show the specification as written in Coq. The `Module Type` describes the specification. An instance of `CURSOR_TABLE` must provide a definition for every `Parameter` and a proof for every `Axiom`. The implementation developed in this thesis is described in **Section** 6 and **Section** 7, and its relation to `CURSOR_TABLE` is given in **Section** 8.

A cursor table is a relation that uses positionable cursors to access and update stored values of type `V`. As such, any cursor table implementation must have both a `table` type to represent an entire relation and a `cursor` type to act as a pointer into a relation.

Notice that any cursor-table must also have a `key` type which is mapped to values. For our purposes we have hard-coded this as Coq's built-in `Z` type for integers. Perhaps it would be better to abstract even further by allowing the key to be any type that has a compare function and satisfies some particular set of axioms; this was in fact the approach taken by Filliâtre and Letouzey in their verification of finite set implementations [7]. However, the simplifying assumption that the keys will be integers is not an unreasonable one, and in particular it fits well with the imperative

```
Module Type CURSOR_TABLE.
 Parameter V: Type.
 Definition key := Z.
 Parameter table: Type.
 Parameter cursor: Type.
 Parameter empty_t: table.

 (* Functions of the implementation *)
 Parameter make_cursor: key -> table -> cursor.
 Parameter get_table: cursor -> table.
 Parameter get_key: cursor -> option key.
 Parameter get: cursor -> option V.
 Parameter insert: cursor -> key -> V -> cursor.
 Parameter next: cursor -> cursor.
 Parameter prev: cursor -> cursor.
 Parameter first_cursor: table -> cursor.
 Parameter last_cursor: table -> cursor.

 (* Predicates of the implementation *)
 Parameter abs_rel: table -> cursor -> Prop.
 Parameter key_rel: key -> cursor -> Prop.
 Parameter eq_cursor : cursor -> cursor -> Prop.
 Parameter cursor_correct: cursor -> Prop.
 Parameter table_correct: table -> Prop.
```

Figure 6: The parameters in the first half of the Coq module type.

```
 (* Theorems about the predicates *)
 Axiom make_cursor_rel: forall t k,
   abs_rel t (make_cursor k t).
 Axiom get_table_rel: forall t c,
   abs_rel t c <-> get_table c = t.
 Axiom first_rel: forall t,
   abs_rel t (first_cursor t).
 Axiom last_rel: forall t,
   abs_rel t (last_cursor t).
 Axiom next_rel: forall t c,
   abs_rel t c -> abs_rel t (next c).
 Axiom prev_rel: forall t c,
   abs_rel t c -> abs_rel t (prev c).
 Axiom correct_rel: forall t c,
   abs_rel t c -> (cursor_correct c <-> table_correct t).

 (* Correctness preservation *)
 Axiom insert_correct: forall k v c,
   cursor_correct c -> key_rel k c -> cursor_correct (insert c k v).

 (* Get/insert correctness *)
 Axiom glast: forall t,
   get (last_cursor t) = None.
 Axiom gis: forall k v c,
   cursor_correct c -> key_rel k c ->
   get (make_cursor k (get_table (insert c k v))) = Some v.
 Axiom gio: forall j k v c,
   cursor_correct c -> key_rel k c -> ~ key_rel j c ->
   get (make_cursor j (get_table (insert c k v))) =
   get (make_cursor j (get_table c)).

 (* Cursor movement correctness *)
 Axiom next_prev: forall c t,
   cursor_correct c -> abs_rel t c -> ~ (c = last_cursor t) ->
   eq_cursor c (prev (next c)).
 Axiom prev_next: forall c t,
   cursor_correct c -> abs_rel t c -> ~ (c = first_cursor t) ->
   eq_cursor c (next (prev c)).
 Axiom cursor_order: forall c k1 k2,
   cursor_correct c -> get_key c = Some k1 ->
   get_key (next c) = Some k2 -> lt_key k1 k2 = true.
End CURSOR_TABLE.
```

Figure 7: The axioms in the second half of the Coq module type.

implementation that this functional specification could be used to verify.

As suggested by the name of the module type, cursors are central to the operation of any cursor table. Appropriately, `make_cursor` is the most central operation to the cursor table. All the other operations listed in **Figure 6** rely on a cursor produced by either `make_cursor` or the related `first_cursor` and `last_cursor`. The functions `first_cursor` and `last_cursor` are needed to creating a cursor pointing at the very beginning or very end of a list, and these are primarily needed to define later `Axiom`s about the operation of a cursor table. The inverse of `make_cursor` is `get_table`: it extracts from a cursor the table that it points into.

Once we have set a cursor, a cursor-table should be able to use it to do any of the desired operations: `get` to retrieve the value stored at the cursor, `get_key` to retrieve the key mapped to that value, and `insert` to update the tree with a new key-value pair. Deletion is not included as part of this specification, but could be axiomatized similarly to insertion. The cursor should also be movable from its position: `next` should move it to the next key greater than the cursor's current position, while `prev` should move it to the last key lesser than its current position.

Beyond the definition of the implementation itself, a series of predicates about cursors and tables must also be defined. The first, `abs_rel`, defines what it means for a particular cursor to be a reference into a particular table, i.e. for the table and cursor to represent the same abstract relation. Of course, this could be as simple as the predicate that `get_table` on the cursor returns the table, but we leave room for the particular implementation to provide a more complex cursor-table correspondence that makes later proofs easier (as is the case for this implementation, as described in **Section 7**). The second, `key_rel`, defines what it means for a cursor to be positioned for a particular key k, i.e. that the key to the right of the cursor is greater than k and the key to the left is at most k. The third, `eq_cursor`, is a predicate for two cursors being abstractly equivalent even if they are not structurally equal.

The fourth and fifth predicates describe what it means for a cursor or table to be correct: since we don't require every instance of the `cursor` and `table` types to be correct, it is possible to have a cursor or table that does not actually correspond to a valid instance of a `CURSOR_TABLE`. For example, with B+ trees it is possible to construct an unbalanced or unsorted tree, in which case the operations are not guaranteed to be correct. In this implementation, it would have been possible to make a more complicated B+ tree type than the one described in **Section 6** and enforce the correctness properties on every tree instance. However, this would have made working with that type more difficult.

Given the operation and predicate `Parameters`, we now want to ensure they satisfy certain desirable properties. These are the `Axioms` of the module type shown in **Figure 7**. The first set of axioms correspond to the predicates themselves, and how they should behave. For example, `make_cursor` and `get_table` should function correctly relative to our table-cursor correspondence `abs_rel`, as should the functions for the ends, `first_cursor` and `last_cursor`. Cursor repositioning (`next` and `prev`) should stay within the same table. Finally, correctness should translate between cursors and tables—if a cursor is correct, then so should its corresponding table be correct; and for any correct table, all cursors into that table should be correct.

The axiom `insert_correct` dictates that inserting a key-value pair into a correct cursor should in general return a new correct cursor. As a consequence of this, by `correct_rel` and `get_table_rel` above, there must exist a new table (that is, `get_table` of the new cursor) that is itself correct. However, notice that we add an additional restriction: the new cursor only has to be correct if it satisfies `key_rel` `k c`, i.e. it was positioned properly for the key k that was inserted. We don't actually specify what should happen when inserting a key-value pair into a cursor in the wrong position. Depending on the implementation, it could work correctly, return the original cursor back unchanged (as our implementation does), or modify the cursor completely incorrectly. It doesn't ultimately matter—we only *require* the behavior to be correct when the cursor is positioned correctly.

The next three axioms in **Figure 7** cover how `get` should behave in conjunction with `insert`. Specifically, we build up the behavior of `get` through modifications to the table. This reflects the idea that the behavior of `get` for a certain key should only be affected by certain modifications, and should ignore others. If we look up a particular key k for which we have inserted some value v, we should get back that value v. On the other hand, if we look up a key j into a table where we last inserted k, and j and k are not in the same range for the cursor c, then the behavior of `get` should be as if k had never been inserted.

Finally, we want to ensure that for any valid cursor except the last one, moving to the next position and then back to the first gives us back a cursor equivalent to the original; this is `next_prev`. In the reverse direction, for any valid cursor except the first one, moving to the previous position and back gives us an equivalent cursor to the original cursor; this is `prev_next`. We also expect `next` and `prev` to position the cursor relative to the ordering of the table. If we call `next`, we expect the "next" key that it moves to to be the key immediately greater than the one the cursor was previously on.

Note that while all of these axioms do specify *what* the cursor-table should do, they don't specify *how* it should do it. We implement this specification with B+ trees, but it could as well be implemented with a simple binary tree, a linked list, an array, or many other data structures. That level of abstraction is crucial: we simplify our correctness properties to only the general results that we want to prove.

5.1 A Small Example

To demonstrate the application of the abstract specification as described above, consider a small example. Suppose we have a small relation built up by inserting $(3, V_3)$, then $(1, V_1)$, and finally $(2, V_2)$. Functionally, this would look like the following:

```
insert (insert (insert (make_cursor 0 empty_t) 3 V_3) 1 V_1) 2 V_2
```

Let the cursor returned by the final insert be c. If we now try to retrieve the value for key 1 from c, we want to be able to prove that we get V_1 back. This might look like the following:

```
get (make_cursor 1 (get_table c))
```

If we peel off the outer `insert` of c, we see that we can apply `gio`:

```
get (make_cursor 1 (get_table (insert c 2 V_2)))
=
get (make_cursor 1 (get_table c))
```

Peeling off the next `insert`, we can now apply `gis` and prove that we return V_1:

```
get (make_cursor 1 (get_table (insert c 1 V_1))) = Some V_1
```

In other words, without any knowledge of the specific implementation of any of the operations, and using only the theorems defined by the abstract specification, we have proved that the value returned by get is the intended value placed into the table.

6 B+ Tree Type Specification

The first step of writing an effective functional model is determining how to translate the imperative concepts of the B+ tree data structure into functional terms. Pointers and mutable data have to be turned into immutable data that can be recursively destructed. In making decisions about how to design the functional type, we have to achieve two goals: first, creating a simple type that will be easy to prove correctness properties of, and second, creating a type that will be easy to prove equivalent to the imperative program.

6.1 Tree Data Structure Type

We represent a B+ tree as a mutually inductive type of `tree` and `treelist`.

```
Inductive tree : Type :=
 | node : key -> treelist -> tree
 | final : treelist -> tree
 | val : key -> V -> tree
with treelist : Type :=
 | tl_nil : treelist
 | tl_cons : tree -> treelist -> treelist
```

Figure 8: The tree and treelist type definitions.

Intuitively, `tree` holds the key-subtree mapping: each *element* within a node is a tree. The `final` tree type holds the single subtree not paired with a key, since interior nodes have $n + 1$ subtrees for n keys. Then, an entire node is a `treelist`. An immediate similarity is apparent between the `treelist` type and Coq's standard list type. This makes it convenient to destruct a treelist to iterate over it.

It also makes the single treelist type work for both interior nodes (which map keys to subtrees without values, and use the `node` and `final` tree types) and leaf nodes (which map keys to values, and use the `val` tree type). Since both of those nodes can be expressed by the single type `tree`, a treelist can hold either one. This in turn makes the structure of the B+ tree simpler, but at the cost of added complexity and assumptions when proving properties of a tree. The type itself does not enforce the structural property that a node is *either* a leaf node with `val`s or an interior node with `node`s and exactly one `final`, so we have to assume or prove this whenever needed.

The choice of putting a final tree at the end, rather than having an initial keyless subtree at the beginning, is somewhat out of sync with the imperative implementation. However, it makes both correctness proofs and the functional implementation much simpler.

Imagine that we had an initial keyless subtree at the beginning. We want to say that a treelist is structurally correct if it has exactly one keyless subtree. Once we destruct a correct treelist and remove that initial subtree, the remainder suddenly only has nodes, and is no longer structurally correct! By placing final at the end instead, a destructed treelist is still structurally correct.

Additionally, putting final at the end makes searching for a particular key (e.g. when making a cursor) easier. At each step, we need to compare the key we are looking for against the key for this subtree. With an initial keyless subtree, we would have to look at the key held in the *next* tree at each step. With a final at the end, we only have to compare against the key held in the current node; once we get to final we know that any remaining keys have to be there.

When relating this representation to the imperative representation, it suffices to shift subtrees by one. In other words, the imperative implementation's keyless "first pointer" is related to the first key's subtree in the functional implementation; the imperative first key's subtree is the functional second key's subtree. The functional final subtree, which is keyless, is related to the imperative implementation's last key's subtree.

6.2 B+ Tree Cursor Type

Imperatively a cursor is a pointer to a particular entry in a B+ tree. Once a cursor points to a part of a B+ tree, it can be used to retrieve the value stored there, update it to store something else, or insert an entirely new key-value pair to the tree. In a functional B+ tree there are no pointers per se, and we cannot update the tree in-place in the same way since all data is immutable. However, it is still useful to represent the idea of a cursor "pointing at" a particular position in a B+ tree.

```
Definition cursor : Type := list nat * list treelist.
```

Figure 9: The cursor type definition.

In this representation, a pointer into a treelist is structured as the combination of that treelist and the index of the cursor's position within it. The `cursor` type is then a list of indices combined with a list of treelists. This cursor structure was chosen to make proofs more straightforward, because it contains the original tree structure unmodified, and the `make_cursor` operation structurally matches the `treelist` type (this is explained in more detail in **Section 7.1**). However, other operations are slightly complicated because of the choice to use indices rather than pulling out the relevant tree or encoding the pointer in some other structural way. Every step in any operation requires destructing the treelist to find the right position before we can get or update the keys and values stored there.

Other possible representations of a cursor were also considered when designing the

B+ tree types for the functional model.[3] The choice of this particular representation was made to prioritize simplicity of proofs over simplicity of implementation. The purpose of this functional model is *not* to be efficient—it exists as a tool to be proven about; with that in mind, it is almost always better to prioritize proofs over code when there is a trade-off to be made.

Note that this representation of a cursor is as a pair of *lists* of individual treelist pointers. This might seem strange, since an imperative implementation only needs to store a single pointer to a single position. However, representing a cursor as a list of pointers reflects the structure of the SQLite cursor mentioned earlier, which is similarly a list of pointers to the relevant location at each level. It also greatly simplifies the functional implementation and proofs to be able to destruct and reconstruct the B+ tree on a per-level basis, rather than needing to start from the root for any operation.

Also note that the `cursor` type as defined here has no way to distinguish between pointing *at* an element versus pointing *in between* two elements. In fact, in an abstract sense, a cursor is always considered to represent a split between two keys. In situations where we are considering the element that the cursor is on (e.g. when looking up a value), by default we assume it is the first element to the right of the cursor's position. However, it is important to note that the cursor does not need to have any element to its right. For example, a cursor into an entirely empty table is completely valid, and is represented as index 0 into the treelist `tl_nil`.

[3]The primary other representation considered was to think of a cursor as a *split* in a tree: it "points" to a particular part of the B+ tree by actually separating it into the trees and treelists to the left of that part vs those to the right. The cursor type is then a list of treelist splits. Representing a cursor this way makes operations on that cursor simpler: getting the (key,value) pair at the cursor only requires looking at the entry immediately on the right in the first treelist split; updating it can be easily done by updating the value, gluing the split back together, and recursively inserting it back into the next level up. Unfortunately, this representation of a cursor complicates the task of cursor creation, and by consequence complicates the process of proofs about that cursor.

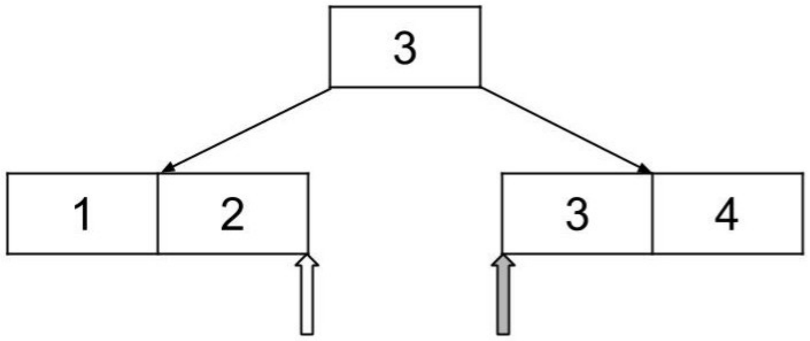

Figure 10: The white and grey cursors should be equivalent.

This adds a potential point for confusion: two *structurally* different cursors could *abstractly* represent the same split. Consider the example in **Figure 10**. Both arrows represent the split between keys 2 and 3 in this B+ tree. Abstractly they are equivalent, and any operations on them should do the same thing. But in this implementation they are not actually equal structures, so we have to ensure (and prove) that all of our B+ tree operations can detect this and treat it appropriately.

7 Functional Implementation

7.1 Cursor Operations

A typical implementation of a relational table would have a lookup function that would find a key in the table, such as the operation discussed in **Section 4.1**. In an array that function might be a linear or binary search. In a binary search tree it would require descending into the left or right subtree of a node depending on whether the key being looked up is less than or greater than that node's key. In this B+ tree implementation, `make_cursor` fills that role. However, rather than simply returning the value held at that key it instead returns a cursor that points to the part of the B+ tree where the key is or would be located.

```
Function make_cursor_rec (x: key) (f : treelist) (ci : list nat)
  (ct : list treelist) (n : nat) : cursor :=
  match f with
  | tl_nil => (n::ci,ct)
  | tl_cons (node k f') f =>
    if lt_key k x then make_cursor_rec x f ci ct (S n)
    else make_cursor_rec x f' (n::ci) (f'::ct) 0
  | tl_cons (final f') tl_nil =>
    make_cursor_rec x f' (n::ci) (f'::ct) 0
  | tl_cons (val k v) f =>
    if lt_key k x then make_cursor_rec x f ci ct (S n)
    else (n::ci,ct)
  | _ => ([],[])
  end.

Definition make_cursor (x : key) (f : treelist) : cursor :=
  make_cursor_rec x f [] [f] 0.
```

Figure 11: The cursor creation function.

The structure of the function is similar to what lookup would be in a binary search tree, but as it destructs the tree to find the subtree to descend into it constructs a record of where it went. Creating a cursor to point at a particular key x happens in two stages. Within a particular treelist, we scan across each node, comparing the node's key k against the target key x. This is what is happening in the `match` section: the treelist is decomposed horizontally into its component trees. As we scan across, we update a counter n that represents the index within the treelist where the function currently is. Once we find the first key greater than or equal to x (as in the `else` portion), we add that index onto the cursor that we are building and restart from 0 in the treelist below that index. If we ever hit a `final` tree, then we can assume this is the last tree in the treelist and descend into it without scanning the treelist further. For leaves, we will either hit a `val` that matches or we will hit `tl_nil` because we have passed the end of the treelist. Either way, we can just append the index and return the cursor that has been built up.

Decomposing the creation of a cursor in this way helps to make induction and termination of this function easy. At each step we are destructing the treelist given and recursing on one of the two results. Since functional objects have no pointers and must be constructed from the data they hold, it is impossible to have an infinitely large tree, so it is easy for Coq to automatically prove the termination of this function. Similarly, when we are proving properties of `make_cursor`, having a function that destructs the treelists naturally fits precisely with our induction scheme, shown in **Figure 12**.

This works well for making a cursor, but we still have the issue of multiple structurally distinct cursors which are abstractly the same. In order to do any operations with a cursor, we need a way to handle these two cursors equivalently. For this purpose, we wrote two normalization functions. The first, `next_node`, normalizes a cursor so it points directly to a `val` tree; this would be the position pointed to by the grey arrow in **Figure 10** from **Section 6.2**. The second, `prev_node`, is

```
treelist_tree_rec :
forall (P : tree -> Prop) (P0 : treelist -> Prop),
(forall (k : key) (t : treelist), P0 t -> P (node k t)) ->
(forall t : treelist, P0 t -> P (final t)) ->
(forall (k : key) (v : V), P (val k v)) ->
P0 tl_nil ->
(forall t : tree, P t ->
  forall t0 : treelist, P0 t0 -> P0 (tl_cons t t0)) ->
forall t : treelist, P0 t.
```

Figure 12: The induction scheme requires a predicate `P` that is inductively assumed about every tree, and a predicate `P0` that is assumed and proved about every treelist. Having functions destruct a treelist in both directions matches this scheme well.

the reverse: it normalizes a cursor so it has a val tree directly behind it, i.e. the white arrow in **Figure 10**.

The central functionality of `next_node` is to travel up the treelist as far as necessary to find a treelist that has a next tree to descend down into. It then positions itself before the very first element of that tree. If it is able to find such a modified cursor, it returns both that cursor and the pivot key at the highest node it reached (i.e., the key that separates the original cursor position from the new one). If it is not able to find such a cursor—in other words, if the cursor was already at the last possible position in the tree—then it returns `None`.

Here, `point n f` is a function that decomposes a treelist f into three parts: the treelist before index n, the tree at index n (if it exists, `None` otherwise), and the treelist after index n. We assume `next_node` is called from a leaf node, so if there is a `val` that the cursor is already pointing to then we are done, and no modification is necessary. If there is something next to point to that is a `node` or a `final`, then we can assume we are in the middle of a recursive call. Since there is something next, this is the pivot tree, so we move the index up by one and attach the associated key k.

```
Fixpoint next_node (cn : list nat) (cf : list treelist)
  : option (cursor * key) :=
  match (cn,cf) with
  | (n::cn',f::cf') =>
    (match point n f with
     | (_,Some (node k _),tl_cons (node _ f') _) =>
       Some (((S n)::cn',f::cf'),k)
     | (_,Some (node k _),tl_cons (final f') _) =>
       Some (((S n)::cn',f::cf'),k)
     | (_,Some (val k v), _) => Some ((n::cn',f::cf'),k)
     | _ =>
       (match next_node cn' cf' with
        | Some ((n'::cn,f'::cf),k) =>
          (match point n' f' with
           | (_,Some (node _ f1),_) =>
             Some ((O::n'::cn,f1::f'::cf),k)
           | (_,Some (final f1),_) =>
             Some ((O::n'::cn,f1::f'::cf),k)
           | _ => None
           end)
        | _ => None
        end)
    end)
  | (_,_) => None
  end.
```

Figure 13: The normalization to point directly to a `val` tree. The structure of `prev_node` is similar.

If we match `point n f` and find something else, for example `None` or a `final` tree, then we are at the end of a node and cannot point to the next thing. Therefore, we must recursively call `next_node` to get the node adjacent to this one. Then we can move down into the tree pointed to by that partial cursor, and position a new (potentially partial) cursor at the first index of that tree.

The structure of `prev_node` is very similar, but operating in reverse it destructs the cursor index to examine the tree immediately *before* the cursor's position. If it is at the beginning of a node and there is nothing directly preceding it to point to, then it recursively calls `prev_node` at the next level up in the cursor. The partial cursor it gets back is then grown with the treelist f' pointed to and a pointer to the last element of f'.

```
Fixpoint move_to_next (c : cursor) : cursor :=
  match c with (cn,cf) =>
  (match next_node cn cf with
   | Some ((n::cn',cf'),_) => (S n::cn',cf')
   | _ => c
   end)
  end.

Fixpoint move_to_prev (c : cursor) : cursor :=
  match c with (cn,cf) =>
  (match prev_node cn cf with
   | Some ((S n::cn',cf'),_) => (n::cn',cf')
   | _ => c
   end)
  end.
```

Figure 14: The Coq code for moving the cursor. With the normalization functions, actual cursor movement is simple.

With these normalization functions, actually moving the cursor becomes trivial. We normalize to the next or prev position to ensure there is an adjacent element to move the cursor to, then we change the cursor to point to it.

For example, consider the first (white) cursor in **Figure 15**. `move_to_next` would first normalize it to the position of the grey cursor, then increment the counter to position it at the black cursor. If we then called `move_to_prev` from the black cur-

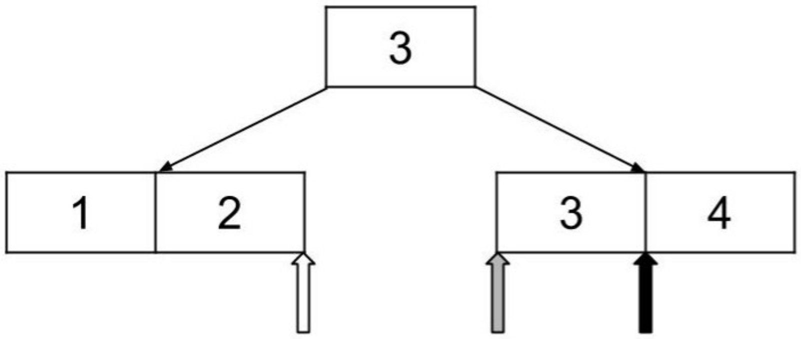

Figure 15: An example of moving the cursor.

sor, either the white or the grey cursor could be a correct result; in this implementation it would return the grey cursor.

7.2 Search

To simplify proofs of search properties, we have combined the core of both `get` and `get_key` into a single function `get_tree` from which the key and value can easily be retrieved. This allows us to prove properties of `get_tree` once, and easily relate them to both `get` and `get_key`.

```
Fixpoint get_tree (c : cursor) : option tree :=
  match c with (cn,cf) =>
  (match next_node cn cf with
   | Some ((n::_,f::_),_) => lin_search n f
   | _ => None
   end)
  end.
```

Figure 16: The generalized search function.

Similarly to the `move_to_next` and `move_to_prev` operations, the `get_tree` operation is vastly simplified by having set up `next_node`. We normalize with `next_node` to ensure that the cursor points directly at an element, and then we can return that element. If `next_node` doesn't return a valid cursor, then we must be at the end of the tree, so there is no element for `get` to return.

7.3 Insertion

Insertion into a functional B+ tree is significantly more complicated than cursor manipulation or search operations. There are two possibilities for insertion. If the key is not already in the tree then we want to insert the key-value pair and position the cursor immediately after it. (Positioning the new cursor after the pair inserted makes it easy to quickly create a new B+ tree or insert a range of key-value pairs). If the key is already in the tree we want to simply update the value and leave the cursor positioned as is. Both of these assume that the cursor is properly positioned to insert the key; if it is not then the cursor should simply be returned unchanged.

Additionally, the insertion implementation has to maintain several B+ tree invariants. The most complicated of these is maintaining the fanout value b, as explained in **Section 4**, so that every treelist contains between $\lceil \frac{b}{2} \rceil$ and b subtrees (except for potentially the root). This means that inserting a key-value pair might require splitting the leaf it was inserted into, and inserting the new leaf into its parent node might require splitting that node, and so on up to the root. In order to handle this, we introduce a new type `splitpair`:

```
Inductive splitpair : Type :=
| One : treelist -> splitpair
| Two : treelist -> key -> treelist -> splitpair.
```

Figure 17: The type used for splitting too-large treelists during insertion.

A treelist that is the correct size and is not split will result in a `One`. If a treelist is too large and must be split then it results in a `Two` containing each half of the split treelist as well as the key differentiating them that will be inserted up into the parent node.

At a high level, inserting into a treelist is a recursive process that destructs the cursor going up and recreates it going back down. For each level of the treelist, we receive a splitpair from the level below. As shown in **Figure 18**, we use the function `insert_across` to place that splitpair's treelist(s) into the position indicated by the cursor, then the function `insert_up` (potentially) splits the new treelist and passes it up to be inserted in the next level of the cursor. The result it gets back

```
Fixpoint insert_across (s : splitpair) (f' : treelist) (n : nat)
  : treelist :=
  match f' with
  | tl_cons (node k f) f' =>
    (match n with
     | O =>
       (match s with
        | One f => tl_cons (node k f) f'
        | Two f1 k' f2 => tl_cons (node k' f1) (tl_cons (node k f2) f')
        end)
     | S n' => tl_cons (node k f) (insert_across s f' n')
     end)
  | tl_cons (final f) tl_nil =>
    (match n with
     | O =>
       (match s with
        | One f => tl_cons (final f) tl_nil
        | Two f1 k' f2 => tl_cons (node k' f1) (tl_cons (final f2) f')
        end)
     | S n' => tl_cons (final f) tl_nil
     end)
  | _ => tl_nil (* Shouldn't ever be hit. *)
  end.

Fixpoint insert_up (s : splitpair) (cn : list nat) (cf : list treelist)
  : cursor :=
  match (cn,cf) with
  | (n::cn,f::cf) =>
    (match decide_split (insert_across s f n) with
     | One f => (match insert_up (One f) cn cf with (cn,cf) =>
                 (n::cn,f::cf) end)
     | Two f1 k f2 =>
       (match insert_up (Two f1 k f2) cn cf with (cn,cf) =>
        if (Nat.leb (treelist_length f1) n)
        then ((n-(treelist_length f1))::cn, f2::cf)
        else (n::cn,f1::cf) end)
     end)
  | (_,_) => ([],[])
  end.
```

Figure 18: Some of the more important sections of the insertion code.

then has the appropriate index and treelist appended back onto it. `decide_split` is a function that compares the length of the treelist to the fanout, and splits it in half if needed. Note that since the cursor carries with it the whole structure of the B+ tree, the whole cursor must be destructed and reconstructed with the updated B+ tree, even if no node needs to be split.

8 Proofs

This section runs through the application of the functional implementation from **Section 7** to the abstract specification from **Section 5**, as well as the intermediate lemmas used to get there. It includes all of the most important lemmas and theorems stated in the Coq files. Not every lemma or theorem stated here has been fully proved; the breakdown of what has and hasn't been proved can be found in **Appendix A** along with the associated Coq theorem statement. Additionally, not every lemma or theorem is included here; the more minor ones used can be found in the complete Coq files in **Appendix B**.

8.1 Correctness Predicates

Various predicates about the correctness of cursors and trees are useful for proving the correctness of operations on those trees. The statement of these predicates in Coq is presented here, along with a description of what the predicate entails.

Figure 19 shows the inductive `Prop` for the keys of a treelist being sorted in a certain range, which is `treelist_sorted`. It also shows the more high-level `sorted`, which represents the idea that there is *some* range in which a treelist is sorted. The four cases of the inductive proposition follow the four possible treelist-tree combinations. Any treelist that is just `tl_nil` is sorted within any range. For a `node` or a `final` tree within a `tl_cons` treelist, we build up sortedness based on the sortedness of the subtree, the sortedness of the remainder of the treelist, and (for `node`) the key of the tree. For a `val` tree, the tree's key must be within the new range and outside the range of the remainder of the treelist.

```
Inductive treelist_sorted : key -> key -> treelist -> Prop :=
| ts_nil : forall ki kf, treelist_sorted ki kf tl_nil
| ts_node : forall (ki ki' kf k : key) (f f' : treelist),
    treelist_sorted ki' kf f -> (* forall x in f, ki' < x <= kf *)
    treelist_sorted ki k f' -> (* forall x in f', ki < x <= k *)
    lt_key ki' k = false -> (* k <= ki' *)
    lt_key ki k = true -> (* ki < k *)
    treelist_sorted ki kf (tl_cons (node k f') f)
| ts_final : forall (ki ki' kf : key) (f f' : treelist),
    treelist_sorted ki' kf f -> (* forall x in f, ki' < x <= kf *)
    treelist_sorted ki ki' f' -> (* forall x in f', ki < x <= ki' *)
    lt_key ki ki' = true -> (* ki < ki' *)
    treelist_sorted ki kf (tl_cons (final f') f)
| ts_val : forall (ki ki' kf k : key) (v : V) (f : treelist),
    treelist_sorted ki' kf f -> (* forall x in f, ki' < x <= kf *)
    lt_key ki' k = false -> (* k <= ki' *)
    lt_key ki k = true -> (* ki < k *)
    treelist_sorted ki kf (tl_cons (val k v) f).

Definition sorted (f : treelist) : Prop :=
  exists ki kf, treelist_sorted ki kf f.
```

Figure 19: The inductive property that all the keys of a treelist are ordered.

```
Inductive balanced_treelist : nat -> treelist -> Prop :=
| bf_nil : balanced_treelist 1 tl_nil
| bf_val : forall k v f,
    balanced_treelist 1 f -> (* f is a val treelist *)
    balanced_treelist 1 (tl_cons (val k v) f)
| bf_node : forall n k f f',
    balanced_treelist n f -> (* f is balanced with n levels *)
    balanced_treelist (S n) f' -> (* f' is balanced with n+1 levels *)
    balanced_treelist (S n) (tl_cons (node k f) f')
| bf_final : forall n f,
    balanced_treelist n f -> (* f is balanced with n levels *)
    balanced_treelist (S n) (tl_cons (final f) tl_nil).

Definition balanced (f : treelist) : Prop :=
  exists n, balanced_treelist n f.
```

Figure 20: The inductive property that all leaves in a treelist are the same distance from the root.

Intuitively, the property of a B+ tree being balanced shown in **Figure 20** means that every leaf node is the same distance away from the root. However, since functional treelists are built from the bottom up, we flip that property: a balanced B+ tree is when the treelist is the same distance n from all leaves in its subtrees. We encode this property by giving any `val` tree a height of 1. Then any treelist `tl_cons t f` has one more than the height of t, provided that f also has that same height.

```
Inductive fanout_restr : nat -> treelist -> Prop :=
| fr_nil : fanout_restr O tl_nil
| fr_val : forall n f k v,
    n < b ->
    fanout_restr n f ->
    fanout_restr (S n) (tl_cons (val k v) f)
| fr_node : forall n n' k f f',
    n' > div_two b false ->
    fanout_restr n' f' ->
    n < b ->
    fanout_restr n f ->
    fanout_restr (S n) (tl_cons (node k f') f)
| fr_final : forall n f,
    n > div_two b false ->
    fanout_restr n f ->
    fanout_restr 1 (tl_cons (final f) tl_nil).

Definition fanout (f : treelist) : Prop :=
  exists n, fanout_restr n f.
```

Figure 21: The property that any node in a treelist has between $\lceil \frac{b}{2} \rceil$ *and* b *subtrees.*

The last treelist property is that of the fanout restriction shown in **Figure 21**: every node in a B+ tree with fanout b must have between $\lceil \frac{b}{2} \rceil$ and b subtrees. The `nat` parameter inductively builds up the length of the treelist, and can grow to any size between 0 and b. However, we restrict it to being at least $\frac{b}{2}$ long when it is used as a subtree in a higher treelist. This makes the inductive `Prop` easy to construct, and also easily encodes the property that the root is allowed to have very few children.

```
Inductive cursor_correct_struct : cursor -> Prop :=
| cc_nil : cursor_correct_struct ([],[])
| cc_first : forall n f, cursor_correct_struct ([n],[f])
| cc_node : forall n n' k f f' ci ct,
    cursor_correct_struct (n::ci,f::ct) ->
    lin_search n f = Some (node k f') ->
    cursor_correct_struct (n'::n::ci,f'::f::ct)
| cc_final : forall n n' f f' ci ct,
    cursor_correct_struct (n::ci,f::ct) ->
    lin_search n f = Some (final f') ->
    cursor_correct_struct (n'::n::ci, f'::f::ct).
(* add that it ends at a leaf *)
(* first f should be correct *)

Inductive rec_prop (P : treelist -> Prop) : cursor -> Prop :=
| rp_nil : rec_prop P ([],[])
| rp_next : forall n f cn cf,
    P f -> rec_prop P (cn,cf) -> rec_prop P (n::cn,f::cf).

Definition cursor_correct (c : cursor) : Prop :=
  cursor_correct_struct c /\
  rec_prop balanced c /\
  rec_prop sorted c /\
  rec_prop fanout c.
```

Figure 22: Structural and treelist correctness properties of cursors.

Figure 22 shows the various cursor correctness predicates. For a cursor to be *structurally* correct means that each level of the lists is properly built from the previous one. If one level gives a treelist f and an index n, then the treelist f' at the next level should actually be contained at that index n in the previous treelist f.

A correct cursor also encapsulates the correctness of the B+ tree it references. The inductive predicate `rec_prop` allows any treelist predicate to be applied over all treelists in the cursor. This allows us to easily say that a correct cursor should not only be structurally correct, but should also reference a treelist that is balanced, sorted, and fanout-restricted.

8.2 Element-List Abstraction

The task of proving that our B+ tree operations treat cursors appropriately relies on an intuitive understanding of how cursors should behave that we now need to formalize. Within our implementation, a B+ tree cursor is a list of indices and treelists. In the abstract specification, a B+ tree cursor is only defined by how our operations use it. This leaves a gap in formalization that makes it difficult to prove our operations correct or prove that they use cursors appropriately.

To solve this problem, we introduce an intermediate abstraction of cursors as an element-list. An element, in this sense, is any key-value pair that appears in the B+ tree. The entire tree can be thought about as a list of these elements in order by increasing key. A cursor splits that list into the parts before the cursor's position and the parts after it. Finally, to simplify our operations we reverse the order of the first list. In other words, the *element-list abstraction* of a cursor is a pair of lists, one to the "left" and one to the "right", where the first element in each list is the element immediately adjacent to the cursor on each side, and the last element in each list is the extreme first or last element of the tree.

It is clear that under this abstraction, functionally equivalent cursors are represented the same. If two structurally different cursors in the tree point to the same split, then they must have the same element-list abstraction. Furthermore, our operations on these element-lists are very simple. To lookup the value at a cursor, we simply pop an item off the right list and return its value. To insert a new key-value pair we just add it to the left list (placing the cursor just after it); to update the value for a key already in the list we just replace the value at the beginning of the right list. Moving the cursor position is just popping items off of one list and concatenating them to the other.

```
Fixpoint cursor_right (cn : list nat) (cf : list treelist)
  (base : list (key * V)) : list (key * V) :=
  match (cn,cf) with
  | (n::cn,f::cf) =>
    (match point n f with (_,_,f') =>
    cursor_right cn cf (right_el f' base) end)
  | (_,_) => base
  end.

Fixpoint cursor_left (cn : list nat) (cf : list treelist)
  (base : list (key * V)) : list (key * V) :=
  match (cn,cf) with
  | (n::cn,f::cf) =>
    (match point n f with (f',_,_) =>
    cursor_left cn cf (left_el f' base) end)
  | (_,_) => base
  end.

Fixpoint cursor_elements' (cn : list nat) (cf : list treelist)
  (l : list (key * V)) (r : list (key * V))
  : (list (key * V)) * (list (key * V)) :=
  match (cn,cf) with
  | (n::cn,f::cf) =>
    (match point n f with (f1,_,f2) =>
    cursor_elements' cn cf (left_el f1 l) (right_el f2 r) end)
  | (_,_) => (l,r)
  end.
```

Figure 23: The functions used to generate the element-list abstraction of a cursor.

In order to actually work with this abstraction, we have to create a functional way to translate between a cursor and its abstraction. The element-lists can be generated by the function `cursor_elements` shown in **Figure 23**, but since it is easier to reason in proofs about one side of the element-list at a time, `cursor_left` and `cursor_right` do the two sides separately. The following lemma describes this fact:

Lemma 1 (`cursor_elements'_sides_equiv`). *The pair that `cursor_elements` returns is equal to the pair of `cursor_left` and `cursor_right`.*

Both the right and left functions are called recursively and gradually build up a "base" of elements already added. For both directions, we process elements from the inside outward. For this, we have the following lemmas:

Lemma 2 (`left_rec_interior`; `right_rec_interior`). *For both the left and right functions, the "base" they are passed ends up at the beginning of the list, i.e. if the base is `b` and the end result is `l`, then there is some `l'` such that `l = b++l'`.*

We also want to show that cursors are processed from the bottom up, and that the things reached from the bottom (which are consequently closer to the cursor position) end up in the base first.

Lemma 3 (`cursor_right_elements1`; `cursor_left_elements1`). *For both the left and right functions, if we split a cursor into two parts, then the result of computing the elements of the whole cursor is equal to the result of first computing the elements of the first part, then using that as the base for computing the elements of the second part.*

Finally, we need to prove that the element-lists preserve the sortedness of the tree. (B+ tree balance and fanout are irrelevant to the element-list, since this abstraction flattens the tree.)

Lemma 4 (`cursor_right_el_sorted`; `cursor_left_el_sorted`). *The element list for a correct cursor (i.e. of an in-order tree) is in order.*

8.3 Cursor Normalization

The normalization functions `next_node` and `prev_node` allow the implementation to treat abstractly equivalent cursors the same. That relies on the properties that they don't change the cursor's abstraction and that they do generate correct cursors, which are formalized in the following lemmas.

Lemma 5 (`next_node_correct`). *`next_cursor` preserves cursor correctness.*

Lemma 6 (`prev_node_correct`). *`prev_cursor` preserves cursor correctness.*

Lemma 7 (`cursor_elements_next`; `cursor_elements_prev`). *If c is a correct cursor, then `next_cursor` of c, `prev_cursor` of c, and c all have the same element-list representation.*

8.4 Cursor Creation

In order to create a cursor, we recursively build up a list of indices into treelists, starting from the root and going down to the leaf that contains (or would contain) the key the cursor is pointing at. We need to prove primarily two things: first, that this produces a correct cursor; and second, that the positioning of the cursor (relative to the abstract element-list representation) is correct.

Lemma 8 (`make_cursor_correct`). *`make_cursor` produces a correct cursor.*

Lemma 9 (`make_cursor_right`; `make_cursor_left`). *The cursor c returned by `make_cursor` for a given key k has the property that the immediate right element has key greater than or equal to k, and the immediate left element has key less than k.*

8.5 Cursor Movement

The correctness of the `move_to_next` and `move_to_prev` functions follows directly from the correctness of the normalization functions. Given proofs that `next_node` and `prev_node` both produce correct cursors, then `move_to_next` and `move_to_prev` clearly produce correct cursors as well because the only modification they make is in the last index of the cursor.

Lemma 10 (`move_to_next_correct`; `move_to_prev_correct`)**.** *Both functions, given a correct cursor, produce a correct cursor.*

We stated in **Section 8.3** that both `next_node` and `prev_node` preserve the element-list, so that they position the cursor before or after an element without changing its abstract representation. Then it is intuitive to prove that the element-list of `move_to_next` of a cursor c is the original cursor's element-list, but with the first element of the "right" list moved to the "left" list. Similarly, `move_to_prev` is the original element-list, but with the first element of the "left" list moved to the "right" list. If there is no right (or respectively left) element, this implies that the cursor is at the very end (or very beginning) of the tree, and we prove that `next_node` and `prev_node` return `None` in that case, so that `move_to_next` and `move_to_prev` just return the original cursor.

Lemma 11 (`move_to_next_el`)**.** *If c has the element lists $(l, (k, v) :: r)$, then next of c has element lists $((k, v) :: l, r)$.*

Lemma 12 (`move_to_prev_el`)**.** *If c has the element lists $((k, v) :: l, r)$, then prev of c has element lists $(l, (k, v) :: r)$.*

Lemma 13 (`move_to_next_none`; `move_to_prev_none`)**.** *If c has no element to its right (or to its left), then next of c (or prev of c) is just c.*

8.6 Correctness of Get

The correctness property of `get` relative to the element-list abstraction is simple, and can be easily applied to the properties of the abstract specification. Specifically, if (k, v) is the element at the front of the right element-list of a (correct) cursor, then `get_tree` should return `val k v`.

Lemma 14 (`get_correct`). *`get` returns the key-value pair at the front of the right list in the element-list abstraction.*

If the cursor is correct—including that it points into a leaf node—then `next_node` must be correct as well (from **Section 8.3**), so the treelist that `next_node` returns will necessarily be a leaf node (this is part of the definition of correctness), and the tree that `get_tree` returns must necessarily be a `val` tree. Furthermore, given that `next_node` positions the cursor to point a tree without changing the element-list for the cursor, we know both that `get_tree` will return that tree and that it must be at the front of the right element-list of `next_node` of the cursor (and therefore of the original cursor as well). Therefore, the tree returned by `get_tree` and the element at the beginning of the right element-list must be the same.

8.7 Correctness of Insert

By comparison to the functional implementation, the element-list abstraction of insertion is quite simple. Let l be the left element-list, and let $(k, v) :: r$ be the right element-list. Insertion has three distinct cases to consider when inserting a key-value pair (x, v'):

1. If the cursor is not correctly positioned for x, then it should return the cursor unchanged.

2. If the cursor is correctly positioned, but x is not in the tree, then it should result in element-lists of $(x, v') :: l$ and $(k, v) :: r$.

3. If the cursor is correctly positioned, and x is in the tree (i.e. $x = k$), then the resulting element-lists should be l and $(k, v') :: r$.

We can prove a helpful lemma with regards to the first point:

Lemma 15 (`bad_cursor_insert_same`). *If the key x is not in the range of keys represented by this cursor, then* `insert` *returns back the original cursor.*

Now consider insertion of a key x into a cursor that does represent x. We first want to prove that insert is structurally correct, i.e. a correct cursor, once inserted into, gives back a correct cursor.

Lemma 16 (`insert_correct`). *If c is a correct cursor into a correct B+ tree, then for any key x and value v, inserting (x, v) into c produces a correct cursor into a correct B+ tree.*

Then, we want to prove what it does to the element-list abstraction, i.e. that it either replaces the next key-value pair or inserts this key-value pair at the cursor's position.

Lemma 17 (`insert_eq_elements`). *If the next key is equal to the key x being inserted, then let l be the left element list and $(x, v') :: r$ be the right element list. Then inserting (x, v) into the cursor produces a cursor with element list equal to: $(l, (x, v) :: r)$.*

Lemma 18 (`insert_neq_elements`). *If the next key is not equal, then let l be the left element list and r be the right element list. Then insertion produces a cursor with element list equal to $((x, v) :: l, r)$.*

8.8 Complete Correctness Theorem

We need one final theorem that pulls all of the other results together and states that the functional model satisfies all of the properties in the abstract specification.

In Coq this is completed by filling out an instance of the `CURSOR_TABLE` module type shown in **Figure 6** and **Figure 7**. We create a `Module` of type `CURSOR_TABLE`:

```
Module BT_Table <: CURSOR_TABLE.
  ...
End BT_Table.
```

Figure 24: The Coq structure for creating an instance of a `Module Type`.

This `Module` contains all of the implementations of the `Parameters` and proofs of the `Axioms` in `CURSOR_TABLE`.

Many of the theorems in the module follow directly from properties proved above about the B+ tree operations and the element-list abstraction. For example, consider `next_prev`, which was stated in Coq terms in **Figure 7**. In English, the theorem is the following:

Theorem 19 (`next_prev`)**.** *For any cursor c that points into some table t, if c is not in the last position of t then `prev` of `next` of c is equivalent to c.*

```
Theorem next_prev: forall c t,
   abs_rel t c -> ~ (c = last_cursor t) -> eq_cursor c (prev (next c)).
```

Figure 25: The formal Coq statement of `next_prev`.

Note that "equivalent" uses the predicate `eq_cursor`. In the definitions section of the `Module`, we define `eq_cursor` as having equal element-list representations.

```
Definition eq_cursor c1 c2 : Prop :=
  cursor_elements V c1 = cursor_elements V c2.
```

Figure 26: The formal Coq statement of `next_prev`.

Therefore, the proof of `next_prev` follows immediately from lemmas about `next` and `prev`: **Lemma 11** (`move_to_next_el`), **Lemma 12** (`move_to_next_el`), and **Lemma 13** (`move_to_next_none`; `move_to_prev_none`). If the cursor is not the last cursor, it must have an element to its right in its element-list; then `move_to_next` moves that element to the left list, and `move_to_prev` moves it back to the right.

Many other proofs in the module would follow from the lemmas listed above. For example, `insert_correct` (**Lemma 16**) in the implementation is a more general version of the `insert_correct` required in the abstract specification. Others that don't follow as exactly are still intuitive to understand how they would fit together. However, the module cannot be completed without first completing the various auxiliary lemmas listed above and others in the Coq files. Although this thesis makes significant progress toward connecting the implementation and the specification in a complete correctness proof, the verification is still only partial.

9 Reflections on Specification

As described in **Section** 2.3 from the DeepSpec group, the goal of a deep specification has four parts: it should be rich, two-sided, formal, and live. All of these properties together ensure a high-quality specification that is both correct and useable. Since that is a goal of this thesis, it is worth reflecting on how well this particular specification project achieved those properties.

This specification certainly attempts to be both rich and formal. By modelling the desired program as an equivalent functional program in Coq's functional programming language Gallina, and by proving its correctness properties using Coq, we have both described the desired functionality in detail and in mathematically rigorous terms. Nonetheless, this specification has room to improve in both richness and formality. The difficulty of a functional programming language to represent naturally imperative concepts such as a cursor makes the implementation at times inelegant and complicated. In this way, the specification struggles to accurately and in detail describe the imperative implementation being targeted. Furthermore, the complicated implementation makes formal proofs more difficult, and as a result this verification relies on informal intuitive proofs in some areas to fill in gaps that could not be proved rigorously in Coq. However, both of these shortcomings would be overcome in a complete implementation and verification. In that sense, the verification *approach* taken is both rich and formal. The specification would also certainly be live, were it attached to an imperative program through machine-checked proofs using the Verified Software Toolchain.

The specification was not initially two-sided, since there is no current client for it, and this was the cause of some difficulty in composing it. However, this thesis does demonstrate a certain client side, even without an explicit client program.

Two-sidedness is important for two reasons: first, it provides a clear goal for the specification; and second, it ensures that the specification is actually useful. Without it, there is the risk that the specification's theorems will be irrelevant to how a client would want to use it, or worse, that a future client program not directly connected to the original specification could misunderstand it and rely on properties that may not actually be true.

The issue of two-sidedness was able to be addressed through the abstract specification. Once the abstract specification was fairly well-developed, then this actually motivated some of the later changes that were made in the functional specification. For example, the decision to build the cursor into the module type resulted in a decision to change the inputs to the `get` and `insert` functions and consider cursor operations separately from search and insertion operations. Additionally, the need to formalize abstract axioms required consideration of the exact properties each function needed to have, and how they could best be implemented. In this way, the completed abstract specification functioned as an approximation for the process of exercising the specification from the client side. It helped to simplify the properties that needed to be proved into a concise description that would hopefully be clear and useful to any future client of the specification. Furthermore, the very brief example in **Section 5.1** demonstrates some of the applicability of the abstract specification developed.

Although this specification is not a completely deep specification, and could not be without a full imperative verification outside the scope of this thesis, the approach presented here makes significant progress toward a true deep specification. For all four of richness, formality, liveness, and two-sidedness, attempting to achieve these properties in this specification led to a better, more rigorous, and more useful specification.

10 Conclusion

This thesis describes the uses and complexities of B+ trees and their verification. B+ trees as a search tree variant are both very useful and very complicated, and the formalization and specification of their functional operation is an important step toward an applicable complete verification of an imperative B+ tree implementation.

The detailed examination of the mechanics of B+ tree cursors is an important development for formally understanding and describing B+ trees. The abstract specification in this thesis presents one possible complete axiomatization of the operation of cursor tables such as a B+ tree. The element-list abstraction gives an intuitive and workable simplification of a B+ tree cursor that serves as a crucial intermediary for proving properties of B+ tree cursors. It acts as a firewall between the complicated functioning of the cursor operations and the desired properties to prove: once the complex cursor operations have been proved equivalent to the simple element-list operations, it is much easier to work with the element-list and prove high-level properties about it without worrying about complicated analysis of the underlying functions and the large number of cases they introduce.

The formal proofs of the cursor and B+ tree operations demonstrate the correctness and usefulness of these varying levels of abstractions. They also serve as evidence for this approach to specification and how it can be successfully applied to complex programs for useable verification interfaces.

11 Future Directions

This thesis has both practical applications and broader implications about specification in general. The functional specification developed in this thesis can be further developed and used as the basis for the verification of an imperative B+ tree program, and such a B+ tree database is currently being developed in C for verification at Princeton University by Oluwatosin Adewale and Aurèle Barrière. The process and approach to specification in this thesis is based on the deep specification framework being studied by the DeepSpec group, and this thesis highlights the successes and complications encountered with that approach. Further work with the specification approach used in this thesis could extend and improve specification techniques for complex imperative software and specifications without an explicit client side.

Appendix A Proof Status Summary

A.1 Element-List Abstraction

Lemma 1 (`cursor_elements'_sides_equiv`)**.** *The pair that `cursor_elements` returns is equal to the pair of `cursor_left` and `cursor_right`.*

```
Theorem cursor_elements'_sides_equiv : forall cn cf l r,
  cursor_elements' cn cf l r =
  (cursor_left cn cf l, cursor_right cn cf r).
```

Proof. This has been formally proved in Coq. □

Lemma 2 (`left_rec_interior`; `right_rec_interior`)**.** *For both the left and right functions, the "base" they are passed ends up at the beginning of the list, i.e. if the base is `b` and the end result is `l`, then there is some `l'` such that `l = b++l'`.*

```
Theorem left_rec_interior : forall cn cf l,
  exists l', cursor_left cn cf l = (l++l').

Theorem right_rec_interior : forall cn cf r,
  exists r', cursor_right cn cf r = (r++r').
```

Proof. This has been formally proved in Coq. □

Lemma 3 (`cursor_right_elements1`; `cursor_left_elements1`)**.** *For both the left and right functions, if we split a cursor into two parts, then the result of computing the elements of the whole cursor is equal to the result of first computing the elements of the first part, then using that as the base for computing the elements of the second part.*

```
Theorem cursor_right_elements1 : forall cn1 cf1 cn2 cf2 b,
  length cn1 = length cf1 -> length cn2 = length cf2 ->
  cursor_right (cn1++cn2) (cf1++cf2) b =
  cursor_right cn2 cf2 (cursor_right cn1 cf1 b).

Theorem cursor_left_elements1 : forall cn1 cf1 cn2 cf2 b,
  length cn1 = length cf1 -> length cn2 = length cf2 ->
  cursor_left (cn1++cn2) (cf1++cf2) b =
  cursor_left cn2 cf2 (cursor_left cn1 cf1 b).
```

Proof. This has been formally proved in Coq. □

Lemma 4 (cursor_right_el_sorted; cursor_left_el_sorted)**.** *The element list for a correct cursor (i.e. of an in-order tree) is in order.*

```
Theorem cursor_right_el_sorted : forall cn cf,
  cursor_correct (cn,cf) -> sorted_less (cursor_right cn cf []).

Theorem cursor_left_el_sorted : forall cn cf,
  cursor_correct (cn,cf) -> sorted_more (cursor_left cn cf []).
```

Proof. This has not been formally proved in Coq.

Informally, this follows from the order implied by cursor_correct. We can introduce an invariant that as it processes the cursor, the list created so far is sorted and within certain bounds, and then the remainder of the cursor (whose element list, by the previous lemmas, will be placed after this first list) will be sorted and outside of those bounds. □

A.2 Cursor Normalization

Lemma 5 (next_node_correct)**.** *next_cursor preserves cursor correctness.*

```
Theorem next_node_correct : forall cn cf c k,
  cursor_correct (cn,cf) -> next_node cn cf = Some (c,k) ->
  cursor_correct c.
```

Proof. This has been formally proved in Coq. □

Lemma 6 (prev_node_correct)**.** *prev_cursor preserves cursor correctness.*

```
Theorem prev_node_correct : forall cn cf c k,
  cursor_correct (cn,cf) -> prev_node cn cf = Some (c,k) ->
  cursor_correct c.
```

Proof. This has not been formally proved in Coq. However, the structure of the proof would closely follow the proof of next_node_correct. □

Lemma 7 (cursor_elements_next; cursor_elements_prev)**.** *If c is a correct cursor, then next_cursor of c, prev_cursor of c, and c all have the same element-list representation.*

```
Theorem cursor_elements_next : forall cn cf c k,
  cursor_correct_struct (cn,cf) ->
  next_node cn cf = Some (c,k) ->
  cursor_elements (cn,cf) = cursor_elements c.

Lemma cursor_elements_prev : forall cn cf c k,
  cursor_correct_struct (cn,cf) ->
  prev_node cn cf = Some (c,k) ->
  cursor_elements (cn,cf) = cursor_elements c.
```

Proof. `cursor_elements_next` has been partially proved; `cursor_elements_prev` has not been proved. □

A.3 Cursor Creation

Lemma 8 (`make_cursor_correct`)**.** *`make_cursor` produces a correct cursor.*

```
Lemma make_cursor_correct : forall x f,
  cursor_correct_struct (make_cursor x f).
```

Proof. This has been formally proved in Coq. □

Lemma 9 (`make_cursor_right`; `make_cursor_left`)**.** *The cursor c returned by `make_cursor` for a given key k has the property that the immediate right element has key greater than or equal to k, and the immediate left element has key less than k.*

```
Lemma make_cursor_right : forall cn cf x f k v l,
  make_cursor x f = (cn,cf) ->
  cursor_right cn cf [] = (k,v)::l ->
  lt_key k x <> true.

Lemma make_cursor_left : forall cn cf x f k v l,
  make_cursor x f = (cn,cf) ->
  cursor_left cn cf [] = (k,v)::l ->
  lt_key k x = true.
```

Proof. This has not been formally proved in Coq. □

A.4 Cursor Movement

Lemma 10 (`move_to_next_correct`; `move_to_prev_correct`)**.** *Both functions, given a correct cursor, produce a correct cursor.*

```
Lemma move_to_next_correct : forall c,
  cursor_correct c ->
  cursor_correct (move_to_next c).

Lemma move_to_prev_correct : forall c,
  cursor_correct c ->
  cursor_correct (move_to_prev c).
```

Proof. `move_to_next_correct` has been formally proved in Coq.

`move_to_prev_correct` has not been formally proved. □

Lemma 11 (`move_to_next_el`)**.** *If c has the element lists $(l, (k, v) :: r)$, then next of c has element lists $((k, v) :: l, r)$.*

```
Lemma move_to_next_el : forall c l r k v,
  cursor_correct c ->
  cursor_elements c = (l,(k,v)::r) ->
  cursor_elements (move_to_next c) = ((k,v)::l,r).
```

Proof. This has not been formally proved in Coq. □

Lemma 12 (`move_to_prev_el`)**.** *If c has the element lists $((k, v) :: l, r)$, then prev of c has element lists $(l, (k, v) :: r)$.*

```
Lemma move_to_prev_el : forall c l r k v,
  cursor_correct c ->
  cursor_elements c = ((k,v)::l,r) ->
  cursor_elements (move_to_next c) = (l,(k,v)::r).
```

Proof. This has not been formally proved in Coq. □

Lemma 13 (`move_to_next_none`; `move_to_prev_none`)**.** *If c has no element to its right (or to its left), then next of c (or prev of c) is just c.*

```
Lemma move_to_next_none : forall cn cf,
  cursor_correct (cn,cf) ->
  (cursor_right cn cf [] = [] <-> move_to_next (cn,cf) = (cn,cf)).

Lemma move_to_prev_none : forall cn cf,
  cursor_correct (cn,cf) ->
  (cursor_left cn cf [] = [] <-> move_to_prev (cn,cf) = (cn,cf)).
```

Proof. This has not been formally proved in Coq. □

A.5 Correctness of Get

Lemma 14 (`get_correct`). *`get` returns the key-value pair at the front of the right list in the element-list abstraction.*

```
Lemma get_correct : forall cn cf k v l,
  cursor_correct (cn,cf) ->
  cursor_right cn cf [] = (k,v)::l ->
  get_tree (cn,cf) = Some (val k v).
```

Proof. This has not been formally proved in Coq. The informal proof laid out in **Section 8.6** describes how it would be proved from various auxiliary lemmas. □

A.6 Correctness of Insert

Lemma 15 (`bad_cursor_insert_same`). *If the key x is not in the range of keys represented by this cursor, then `insert` returns back the original cursor.*

```
Lemma bad_cursor_insert_same : forall c k v,
  key_rel k c <> true -> insert k v c = c.
```

Proof. This has not been formally proved in Coq. □

Lemma 16 (`insert_correct`). *If c is a correct cursor into a correct B+ tree, then for any key x and value v, inserting (x, v) into c produces a correct cursor into a correct B+ tree.*

```
Lemma insert_correct : forall c k v,
  cursor_correct c -> cursor_correct (insert k v c).
```

Proof. This has not been formally proved in Coq. □

Lemma 17 (`insert_eq_elements`)**.** *If the next key is equal to the key x being inserted, then let l be the left element list and (x, v') :: r be the right element list. Then inserting (x, v) into the cursor produces a cursor with element list equal to: $(l, (x, v) :: r)$.*

```
Lemma insert_eq_elements : forall c k v' v l r,
  cursor_elements c = (l,(k,v')::r) ->
  cursor_elements (insert k v c) = (l,(k,v)::r).
```

Proof. This has not been formally proved in Coq. □

Lemma 18 (`insert_neq_elements`)**.** *If the next key is not equal, then let l be the left element list and r be the right element list. Then insertion produces a cursor with element list equal to $((x, v) :: l, r)$.*

```
Lemma insert_neq_elements : forall c k v l r,
  get_key c <> Some k -> cursor_elements c = (l,r) ->
  cursor_elements (insert k v c) = ((k,v)::l,r).
```

Proof. This has not been formally proved in Coq. □

Appendix B Coq Code

B.1 Implementation (BTrees.v)

```
Require Import Extract.
Require Import Coq.Lists.List.
Require Import Recdef.
Require Import FunInd.
Require Import Coq.Numbers.BinNums.
Require Import Coq.omega.Omega.
Export ListNotations.

Definition key := Z.

Section BTREES.
Variable V : Type.
Variable b : nat.

(** Basic Definitions *)

Inductive tree : Type :=
  | node : key -> treelist -> tree
  | final : treelist -> tree
  | val : key -> V -> tree
with treelist : Type :=
  | tl_nil : treelist
  | tl_cons : tree -> treelist -> treelist.

Scheme tree_treelist_rec := Induction for tree Sort Type
with treelist_tree_rec := Induction for treelist Sort Type.

(*
treelist_tree_rec :
forall (P : tree -> Type) (P0 : treelist -> Type),
(forall (k : key) (t : treelist), P0 t -> P (node k t)) ->
(forall t : treelist, P0 t -> P (final t)) ->
(forall (k : key) (v : V), P (val k v)) ->
P0 tl_nil ->
(forall t : tree, P t -> forall t0 : treelist, P0 t0 -> P0 (tl_cons t t0))
 ↪  ->
forall t : treelist, P0 t
*)

Definition cursor : Type := list nat * list treelist.

Inductive splitpair : Type :=
| One : treelist -> splitpair
| Two : treelist -> key -> treelist -> splitpair.

(** Example data *)

Definition pos_one : key := Zpos xH.
Definition neg_one : key := Zneg xH.
Definition zero : key := Z0.
Definition pos_six : key := Zpos (xO (xI xH)).
Definition default : V. Admitted.

Definition ex_treelist : treelist :=
  tl_cons (node pos_one
    (tl_cons (val neg_one default)
    (tl_cons (val pos_one default) tl_nil)))
  (tl_cons (final
    (tl_cons (val pos_six default) tl_nil))
  tl_nil).

Definition ex_treelist' : treelist :=
  tl_cons (final ex_treelist) tl_nil.

Definition ex_treelist'' : treelist :=
  tl_cons (val zero default) tl_nil.

(** Helper Functions *)

Fixpoint eq_pos (p1 : positive) (p2 : positive) :=
  match (p1,p2) with
  | (xI p1',xI p2') => eq_pos p1' p2'
  | (xO p1',xO p2') => eq_pos p1' p2'
  | (xH,xH) => true
  | (_,_) => false
  end.

Definition eq_key (k1 : key) (k2 : key) :=
  match (k1,k2) with
  | (Z0,Z0) => true
  | (Zpos p1,Zpos p2) => eq_pos p1 p2
  | (Zneg p1,Zneg p2) => eq_pos p1 p2
  | (_,_) => false
  end.

(* is p1 less than p2? *)
(* curVal carries the info of which was greater in the bit where they
↪  differed *)
Fixpoint lt_pos (p1 : positive) (p2 : positive) (curVal : bool) :=
  match p1 with
  | xH => (match p2 with
          | xH => curVal
          | _ => true
          end)
  | xO p1' => (match p2 with
              | xH => false
              | xO p2' => lt_pos p1' p2' curVal
              | xI p2' => lt_pos p1' p2' true
              end)
  | xI p1' => (match p2 with
              | xH => false
              | xO p2' => lt_pos p1' p2' false
              | xI p2' => lt_pos p1' p2' curVal
              end)
  end.

Definition lt_key (k1 : key) (k2 : key) :=
  match k1 with
  | Zpos p1 => (
    match k2 with
    | Zpos p2 => lt_pos p1 p2 false
    | _ => false
    end)
  | Z0 => (
    match k2 with
    | Zpos p2 => true
    | _ => false
    end)
  | Zneg p1 => (
    match k2 with
    | Zneg p2 => lt_pos p2 p1 false
    | _ => true
    end)
  end.

Fixpoint max_nat (m : nat) (n : nat) : nat :=
  match m with
  | O => n
  | S m' => (match n with
             | O => m
             | S n' => S (max_nat m' n')
             end)
  end.

Fixpoint unzip {A : Type} {B : Type} (l : list (A * B)) : list A * list B :=
  match l with
  | (a,b)::tl => (match unzip tl with (la, lb) => (a::la, b::lb) end)
  | [] => ([], [])
  end.

Fixpoint treelist_depth (f : treelist) : nat :=
  match f with
  | tl_nil => O
  | tl_cons t f => max_nat (tree_depth t) (treelist_depth f)
  end
with tree_depth (t : tree) : nat :=
  match t with
  | node _ f => S (treelist_depth f)
  | final f => S (treelist_depth f)
  | val _ _ => S O
  end.

Fixpoint zip (f1 : treelist) (f2 : treelist) : treelist :=
  match f1 with
  | tl_cons t f' => tl_cons t (zip f' f2)
  | tl_nil => f2
  end.

Fixpoint treelist_length (f : treelist) : nat :=
  match f with
  | tl_nil => O
  | tl_cons t f' => S (treelist_length f')
  end.

(* flip false causes floor of div *)
(* flip true causes ciel of div *)
Fixpoint div_two (n : nat) (flip : bool) : nat :=
  match n with
  | O => O
  | S n' => (match flip with
             | true => S (div_two n' false)
             | false => div_two n' true
             end)
  end.

Function lin_search (n : nat) (f : treelist) : option tree :=
  match f with
  | tl_cons t f' =>
    (match n with
     | O => Some t
     | S n' => lin_search n' f'
     end)
  | tl_nil => None
  end.

Fixpoint point (n : nat) (f : treelist): (treelist * option tree * treelist)
 ↪  :=
  match f with
  | tl_cons t f' =>
    (match n with
     | O => (tl_nil,Some t,f')
     | S n' => (match point n' f' with (f1,t',f2) => (tl_cons t f1,t',f2)
      ↪  end)
     end)
  | tl_nil => (tl_nil,None,tl_nil)
  end.

Function tl_app (f1 : treelist) (f2 : treelist) : treelist :=
  match f1 with
  | tl_nil => f2
  | tl_cons t f1' => tl_cons t (tl_app f1' f2)
  end.

(** MAKE_CURSOR section *)

(* Functions to create a cursor at a given key *)

Function make_cursor_rec (x: key) (f : treelist) (ci : list nat) (ct : list
 ↪  treelist) (n : nat) : cursor :=
  match f with
  | tl_nil => (n::ci,ct)
  | tl_cons (node k f') f =>
    if lt_key k x then make_cursor_rec x f ci ct (S n) else make_cursor_rec
     ↪  x f' (n::ci) (f'::ct) O
  | tl_cons (final f') tl_nil =>
    make_cursor_rec x f' (n::ci) (f'::ct) O
  | tl_cons (val k v) f =>
    if lt_key k x then make_cursor_rec x f ci ct (S n) else (n::ci,ct)
  | _ => ([],[])
  end.

Definition make_cursor (x : key) (f : treelist) : cursor := make_cursor_rec
 ↪  x f [] [f] O.

Function first_cursor_rec (f : treelist) (cn : list nat) (cf : list
 ↪  treelist) : cursor :=
  match f with
  | tl_nil => (cn,cf) (* Shouldn't be hit unless tree is empty *)
  | tl_cons (node _ f') _ => first_cursor_rec f' (0::cn) (f::cf)
  | tl_cons (final f') _ => first_cursor_rec f' (0::cn) (f::cf)
  | tl_cons (val _ _) _ => (0::cn,f::cf)
  end.

Definition first_cursor (f : treelist) : cursor := first_cursor_rec f [] [].

Function last_cursor_rec (f : treelist) (cn : list nat) (cf : list treelist)
 ↪  (n : nat) : cursor :=
  match f with
  | tl_nil => (n::cn,cf) (* Shouldn't be hit unless tree is empty *)
  | tl_cons t tl_nil =>
    (match t with
     | node _ f' => last_cursor_rec f' (n::cn) (f'::cf) 0 (* Shouldn't
      ↪  happen b/c always final before tl_nil *)
     | final f' => last_cursor_rec f' (n::cn) (f'::cf) 0
     | val _ _ => ((S n)::cn, cf)
     end)
  | tl_cons t f' => last_cursor_rec f' cn cf (S n)
  end.

Definition last_cursor (f : treelist) : cursor := last_cursor_rec f [] [f]
 ↪  0.

(** NEXT & PREV section *)

Fixpoint next_node (cn : list nat) (cf : list treelist) : option (cursor *
 ↪  key) :=
  match (cn,cf) with
  | (n::cn',f::cf') =>
    (match point n f with
     | (_,Some (node k _),tl_cons (node _ f') _) => Some (((S
      ↪  n)::cn',f::cf'),k)
     | (_,Some (node k _),tl_cons (final f') _) => Some (((S
      ↪  n)::cn',f::cf'),k)
     | (_,Some (val k v), _) => Some ((n::cn',f::cf'),k)
     | _ =>
       (match next_node cn' cf' with
        | Some ((n'::cn,f'::cf),k) =>
          (match point n' f' with
           | (_,Some (node _ f1),_) => Some ((0::n'::cn,f1::f'::cf),k)
           | (_,Some (final f1),_) => Some ((0::n'::cn,f1::f'::cf),k)
           | _ => None
           end)
        | _ => None
        end)
     end)
  | (_,_) => None
  end.

Fixpoint prev_node (cn : list nat) (cf : list treelist) : option (cursor *
 ↪  key) :=
  match (cn,cf) with
  | (S n::cn',f::cf') =>
     (match point n f with
      | (_,Some (node k f'),_) => Some ((n::cn',f::cf'),k)
      | (_,Some (final f'),_) => None (* Shouldn't be possible *)
      | (_,Some (val k v),_) => Some ((S n::cn',f::cf'),k)
      | (_,None,_) => None (* Shouldn't be possible *)
      end)
  | (O::cn',f::cf') =>
     (match prev_node cn' cf' with
       | Some ((n::cn, f::cf),k) =>
         (match point n f with
          | (_,Some (node _ f'),_) => Some (((treelist_length
           ↪  f')-1::n::cn,f'::f::cf),k)
          | (_,Some (final f'),_) => Some (((treelist_length
           ↪  f')-1::n::cn,f'::f::cf),k)
          | _ => None
          end)
       | _ => None
       end)
  | (_,_) => None
  end.

Fixpoint move_to_next (c : cursor) : cursor :=
  match c with (cn,cf) =>
  (match next_node cn cf with
   | Some ((n::cn',cf'),_) => (S n::cn',cf')
   | _ => c
   end)
  end.

Fixpoint move_to_prev (c : cursor) : cursor :=
  match c with (cn,cf) =>
  (match prev_node cn cf with
   | Some ((S n::cn',cf'),_) => (n::cn',cf')
   | _ => c
   end)
  end.

(** GET section *)

Fixpoint get_tree (c : cursor) : option tree :=
  match c with (cn,cf) =>
  (match next_node cn cf with
   | Some ((n::_,f::_),_) => lin_search n f
   | _ => None
   end)
  end.

Fixpoint get_key (c : cursor) : option key :=
  match get_tree c with
  | Some (val k v) => Some k
  | _ => None
  end.

Fixpoint get (c : cursor) : option V :=
  match get_tree c with
  | Some (val k v) => Some v
  | _ => None
  end.

(** INSERT section *)

(* separates the nth element to move up *)
Fixpoint split (f : treelist) (n : nat) : splitpair :=
  match n with
  | O => (match f with
         | tl_nil => One f (* can't be split that far *)
         | tl_cons t f' => (match t with
                           | node k f'' => Two (tl_cons (final f'') tl_nil) k
                            ↪  f' (* the node becomes a final *)
                           | val k v => Two (tl_cons t tl_nil) k f' (*
                            ↪  preserve the key-value pair *)
                           | final f'' => One f (* can't be split that far *)
                           end)
          end)
  | S n' => (match f with
            | tl_nil => One f (* can't be split that far *)
            | tl_cons t f' => (match split f' n' with
                              | One f1 => One (tl_cons t f1)
                              | Two f1 k' f2 => Two (tl_cons t f1) k' f2
                              end)
            end)
  end.

Fixpoint decide_split (f : treelist) : splitpair :=
  if (Nat.leb (treelist_length f) b)
  then One f
  else split f (div_two (treelist_length f) false).

Fixpoint insert_across (s : splitpair) (f' : treelist) (n : nat) : treelist
 ↪  :=
  match f' with
  | tl_cons (node k f) f' =>
    (match n with
     | O =>
       (match s with
        | One f => tl_cons (node k f) f'
        | Two f1 k' f2 => tl_cons (node k' f1) (tl_cons (node k f2) f')
        end)
     | S n' => tl_cons (node k f) (insert_across s f' n')
     end)
  | tl_cons (final f) tl_nil =>
    (match n with
     | O =>
       (match s with
        | One f => tl_cons (final f) tl_nil
        | Two f1 k' f2 => tl_cons (node k' f1) (tl_cons (final f2) f')
        end)
     | S n' => tl_cons (final f) tl_nil (* This should never happen! *)
     end)
  | _ => tl_nil (* Shouldn't ever be hit. *)
  end.

Fixpoint insert_up (s : splitpair) (cn : list nat) (cf : list treelist) :
 ↪  cursor :=
  match (cn,cf) with
  | (n::cn,f::cf) =>
    (match decide_split (insert_across s f n) with
     | One f => (match insert_up (One f) cn cf with (cn,cf) => (n::cn,f::cf)
      ↪  end)
     | Two f1 k f2 =>
       (match insert_up (Two f1 k f2) cn cf with (cn,cf) =>
        if (Nat.leb (treelist_length f1) n) then ((n-(treelist_length
         ↪  f1))::cn, f2::cf)
        else (n::cn,f1::cf) end)
     end)
  | (_,_) => ([],[])
  end.

Fixpoint insert_val (x : key) (v : V) (n : nat) (f : treelist) : nat *
 ↪  treelist :=
  match f with
  | tl_cons (val k v') f =>
    (match n with
     | O => if eq_key k x then (O, tl_cons (val k v) f)
            else (S O, tl_cons (val x v) (tl_cons (val k v') f))
     | S n => match insert_val x v n f with (n, f) => (S n, tl_cons (val k
      ↪  v') f) end
     end)
  | tl_nil => (S O, tl_cons (val x v) tl_nil)
  | _ => (S O, tl_cons (val x v) tl_nil) (* should never happen *)
  end.

(* Returns a cursor correctly positioned for insertion
   Returns None if x is not in the range for c
 *)
Fixpoint position_for_insert (c : cursor) (x : key) : option cursor :=
  match c with (cn,cf) => (
  match next_node cn cf with
  | Some (cn1,cf1,k1) =>
    (match prev_node cn cf with
     | Some (cn2,cf2,k2) =>
        if (lt_key k1 x) then Some (cn1,cf1)
        else if negb (lt_key k2 x) then Some (cn2,cf2)
        else None
     | None => if lt_key k1 x then None else Some c
     end)
  | None =>
    (match prev_node cn cf with
     | Some (cn2,cf2,k2) => if lt_key k2 x then Some c else None
     | None => None
     end)
  end)
  end.

(* Needs to point the cursor to the right place (if it's straddling a leaf
 ↪  divide) *)
(* Then, insert (x,v) in (either replacing next or inserting before it) *)
(* Needs to bump the first n up to be past what was just inserted *)
(* Finally, turns that treelist into a splitpair and calls insert_up. *)
Fixpoint insert (x : key) (v : V) (c : cursor) : cursor :=
  match position_for_insert c x with
  | Some (n::cn,f::cf) =>
    (match insert_val x v n f with (n',f') =>
     (match decide_split f' with
      | One f'' => (match insert_up (One f'') cn cf with (cn',cf') =>
       ↪  (n'::cn',f''::cf') end)
      | Two f1 k f2 =>
        (match insert_up (Two f1 k f2) cn cf with (cn',cf') =>
         if (Nat.leb (treelist_length f1) n') then ((n'-(treelist_length
          ↪  f1))::cn',f2::cf')
         else (n'::cn',f1::cf') end)
      end) end)
  | Some _ => ([S 0], [tl_cons (val x v) tl_nil]) (* tree must be empty *)
  | None => c
  end.

(** CURSOR_ELEMENTS *)

Fixpoint elements' (f : treelist) (base: list (key * V)) : list (key * V) :=
  match f with
  | tl_cons (node k f1) f2 => elements' f1 (elements' f2 base)
  | tl_cons (final f1) f2 => elements' f1 (elements' f2 base)
  | tl_cons (val k v) f2 => (k,v)::(elements' f2 base)
  | tl_nil => base
  end.

Fixpoint elements (f : treelist) : list (key * V) := elements' f [].

Fixpoint right_el (f : treelist) (base : list (key * V)) : list (key * V) :=
  match f with
  | tl_cons (node k f1) f2 => right_el f2 (right_el f1 base)
  | tl_cons (final f1) f2 => right_el f2 (right_el f1 base)
  | tl_cons (val k v) f2 => right_el f2 (base++[(k,v)])
  | tl_nil => base
  end.

Fixpoint left_el (f : treelist) (base : list (key * V)) : list (key * V) :=
  match f with
  | tl_cons (node k f1) f2 => left_el f1 (left_el f2 base)
  | tl_cons (final f1) f2 => left_el f1 (left_el f2 base)
  | tl_cons (val k v) f2 => (left_el f2 base) ++ [(k,v)]
  | tl_nil => base
  end.

Fixpoint cursor_right (cn : list nat) (cf : list treelist) (base : list (key
 ↪  * V)) : list (key * V) :=
  match (cn,cf) with
  | (n::cn,f::cf) =>
    (match point n f with (_,_,f') => cursor_right cn cf (right_el f' base)
     ↪  end)
  | (_,_) => base
  end.

Fixpoint cursor_left (cn : list nat) (cf : list treelist) (base : list (key
 ↪  * V)) : list (key * V) :=
  match (cn,cf) with
  | (n::cn,f::cf) =>
    (match point n f with (f',_,_) => cursor_left cn cf (left_el f' base)
     ↪  end)
  | (_,_) => base
  end.

Fixpoint cursor_elements' (cn : list nat) (cf : list treelist) (left : list
 ↪  (key * V)) (right : list (key * V))
  : (list (key * V)) * (list (key * V)) :=
  match (cn,cf) with
  | (n::cn,f::cf) =>
    (match point n f with (f1,_,f2) => cursor_elements' cn cf (left_el f1
     ↪  left) (right_el f2 right) end)
  | (_,_) => (left,right)
  end.
```

```
Definition cursor_elements (c : cursor) : list (key * V) * list (key * V) :=
  match c with
  | (n::cn,f::cf) =>
    (match point n f with
     | (_,Some (val k v),_) => cursor_elements' (n::cn) (f::cf) [] [(k,v)]
     | _ => cursor_elements' (n::cn) (f::cf) [] [] end)
  | (_,_) => ([],[])
  end.

(**
  * PROOFS SECTION
  *)

(** Proofs about helper functions *)

Theorem max_nat_comm : forall (x y : nat),
  max_nat x y = max_nat y x.
Proof.
  induction x; destruct y; try reflexivity.
  simpl. rewrite IHx. reflexivity.
Qed.

Theorem max_nat_largest : forall (x y : nat),
  max_nat x y = x <-> x >= y.
Proof.
  induction x; destruct y; split; intros; try omega; try reflexivity.
  - inversion H.
  - inversion H. rewrite H1. apply IHx in H1. omega.
  - simpl. assert (H': x >= y). { omega. }
    apply IHx in H'. rewrite H'. reflexivity.
Qed.

Theorem max_nat_one : forall (x y z : nat),
  z = max_nat x y -> z = x \/ z = y.
Proof.
  induction x; destruct y; intros.
  - inversion H. left. reflexivity.
  - rewrite max_nat_comm in H. simpl in H. right. apply H.
  - simpl in H. left. apply H.
  - simpl in H. remember (max_nat x y) as z'. apply IHx in Heqz'.
    inversion Heqz'.
    + left. subst. reflexivity.
    + right. subst. reflexivity.
Qed.

Theorem zip_treelist : forall t tl1 tl2 tl,
  zip tl1 tl2 = tl <-> tl_cons t tl = zip (tl_cons t tl1) tl2.
Proof. intros. split.
  - intros. simpl. subst. reflexivity.
  - intros. simpl in H. inversion H. reflexivity.
 Qed.

Theorem depth_in_treelist : forall t tl tl1 tl2,
  tl = zip tl1 (tl_cons t tl2) -> tree_depth t <= treelist_depth tl.
Proof.
  intros t. induction tl; intros.
  - simpl. destruct tl1; inversion H.
  - destruct tl1.
```

```
    + simpl in H. inversion H. subst. simpl.
      remember (max_nat (tree_depth t) (treelist_depth tl2)) as z.
      assert (H': z = tree_depth t \/ z = treelist_depth tl2). { apply
       ↪  max_nat_one. apply Heqz. }
      inversion H'.
      * subst; omega.
      * subst. rewrite max_nat_comm in H0. apply max_nat_largest in H0.
       ↪  omega.
    + simpl in H. inversion H. simpl. rewrite <- H2.
      assert (H0: tree_depth t <= treelist_depth tl). { apply IHtl with tl1
       ↪  tl2. apply H2. }
      remember (max_nat (tree_depth t1) (treelist_depth tl)) as z.
      assert (H': z = tree_depth t1 \/ z = treelist_depth tl). { apply
       ↪  max_nat_one. apply Heqz. }
      inversion H'.
      * rewrite H3. assert (H4: treelist_depth tl <= tree_depth t1).
        { apply max_nat_largest. subst. assumption. }
        omega.
      * omega.
Qed.

Theorem max_nat_least : forall x y,
  x <= max_nat x y.
Proof.
  intros. remember (max_nat x y) as z.
  assert (z = x \/ z = y). { apply max_nat_one in Heqz. apply Heqz. }
  inversion H.
  - omega.
  - rewrite H0 in Heqz. rewrite max_nat_comm in Heqz.
    assert (y >= x). { apply max_nat_largest. omega. }
    omega.
Qed.

Theorem point_lin_search : forall f n t l1 l2,
  point n f = (l1,Some t,l2) -> lin_search n f = Some t.
Proof.
  induction f; destruct n; simpl; intros.
  - inversion H.
  - inversion H.
  - inversion H. reflexivity.
  - destruct (point n f) eqn:e. destruct p. apply IHf with t2 t1.
    rewrite e. inversion H. reflexivity.
Qed.

Theorem lin_search_point : forall f n t,
  lin_search n f = Some t -> exists l1 l2, point n f = (l1,Some t,l2).
Proof.
  induction f; destruct n; simpl; intros.
  - inversion H.
  - inversion H.
  - inversion H. subst. exists tl_nil,f. reflexivity.
  - destruct (point n f) eqn:e. destruct p. apply IHf in H. inversion H.
   ↪  inversion H0.
    rewrite e in H1. inversion H1. subst.
    exists (tl_cons t x),x0. reflexivity.
Qed.

Lemma point_treelist_length : forall n f,
  n >= treelist_length f <-> exists f', point n f = (f',None,tl_nil).
```

```
Proof.
  induction n; destruct f; split; intros; simpl; try omega.
  - exists tl_nil. reflexivity.
  - simpl in H. inversion H.
  - simpl in H. inversion H. inversion H0.
  - exists tl_nil. reflexivity.
  - simpl in H. assert (n >= treelist_length f) by omega.
    apply IHn in H0. inversion H0. exists (tl_cons t x). rewrite H1.
     ↪  reflexivity.
  - simpl in H. destruct (point n f) eqn:e. destruct p. inversion H.
   ↪  inversion H0. subst.
    assert (n >= treelist_length f) by (apply IHn; exists t1; apply e).
     ↪  omega.
Qed.

Lemma point_none : forall n f f1 f2,
  point n f = (f1,None,f2) -> f2 = tl_nil.
Proof.
  induction n; intros.
  - simpl in H. destruct f; inversion H. reflexivity.
  - simpl in H. destruct f. inversion H. reflexivity.
    destruct (point n f) eqn:e. destruct p.
    inversion H. subst. apply IHn with f t1. apply e.
Qed.

Lemma point_prev : forall n f f1 f2 t f1' f2' o,
  point n f = (f1, Some t, f2) ->
  point (S n) f = (f1', o, f2') ->
  f1' = tl_app f1 (tl_cons t tl_nil).
Proof.
  induction n; intros.
  - destruct f.
    + simpl in H. inversion H.
    + simpl in H. inversion H. subst. simpl.
      simpl in H0. destruct f2; inversion H0; reflexivity.
  - simpl in H. remember (S n) as n'. simpl in H0. destruct f.
    + inversion H.
    + destruct (point n f) eqn:e. destruct p.
      destruct (point n' f) eqn:e2. destruct p.
      subst. inversion H. inversion H0. subst.
      assert (t4 = tl_app t2 (tl_cons t tl_nil)).
      { apply IHn with f f2 f2' o. apply e. apply e2. }
      simpl. rewrite H1. reflexivity.
Qed.

(** Correctness properties *)

(* Treelist in order property *)
Inductive treelist_sorted : key -> key -> treelist -> Prop :=
| ts_nil : forall ki kf, treelist_sorted ki kf tl_nil
| ts_node : forall (ki ki' kf k : key) (f f' : treelist),
    treelist_sorted ki' kf f -> (* forall x in f, ki' < x <= kf *)
    treelist_sorted ki k f' -> (* forall x in f', ki < x <= k *)
    lt_key ki' k = false -> (* k <= ki' *)
    lt_key ki k = true -> (* ki < k *)
    treelist_sorted ki kf (tl_cons (node k f') f)
| ts_final : forall (ki ki' kf : key) (f f' : treelist),
    treelist_sorted ki' kf f -> (* forall x in f, ki' < x <= kf *)
    treelist_sorted ki ki' f' -> (* forall x in f', ki < x <= ki' *)
    lt_key ki ki' = true -> (* ki < ki' *)
    treelist_sorted ki kf (tl_cons (final f') f)
| ts_val : forall (ki ki' kf k : key) (v : V) (f : treelist),
    treelist_sorted ki' kf f -> (* forall x in f, ki' < x <= kf *)
    lt_key ki' k = false -> (* k <= ki' *)
    lt_key ki k = true -> (* ki < k *)
    treelist_sorted ki kf (tl_cons (val k v) f).

Definition sorted (f : treelist) : Prop := exists ki kf, treelist_sorted ki
 ↪  kf f.

(* Balance property *)
Inductive balanced_treelist : nat -> treelist -> Prop :=
| bf_nil : balanced_treelist 1 tl_nil
| bf_val : forall k v f,
    balanced_treelist 1 f -> (* f is balanced with 1 level, i.e. f is a
     ↪  value treelist *)
    balanced_treelist 1 (tl_cons (val k v) f)
| bf_node : forall n k f f',
    balanced_treelist n f -> (* f is balanced with n levels *)
    balanced_treelist (S n) f' -> (* f' is balanced with n+1 levels *)
    balanced_treelist (S n) (tl_cons (node k f) f')
| bf_final : forall n f,
    balanced_treelist n f -> (* f is balanced with n levels *)
    balanced_treelist (S n) (tl_cons (final f) tl_nil).

(* Balance property on root *)
Definition balanced (f : treelist) : Prop := exists n, balanced_treelist n
 ↪  f.

Theorem balanced_rec : forall (t : tree) (f : treelist),
  balanced (tl_cons t f) -> balanced f.
Proof.
  induction t.
  - intros. inversion H. inversion H0. unfold balanced. exists (S n). apply
   ↪  H6.
  - intros. inversion H. inversion H0. unfold balanced. exists 1. apply
   ↪  bf_nil.
  - intros. inversion H. inversion H0. unfold balanced. exists 1. apply H3.
Qed.

(* Fanout property *)
Inductive fanout_restr : nat -> treelist -> Prop :=
| fr_nil : fanout_restr O tl_nil
| fr_val : forall n f k v,
    n < b ->
    fanout_restr n f ->
    fanout_restr (S n) (tl_cons (val k v) f)
| fr_node : forall n n' k f f',
    n' > div_two b false -> (* floor(b/2) *)
    fanout_restr n' f' ->
    n < b ->
    fanout_restr n f ->
    fanout_restr (S n) (tl_cons (node k f') f)
| fr_final : forall n f,
    n > div_two b false ->
    fanout_restr n f ->
    fanout_restr 1 (tl_cons (final f) tl_nil).
```

```
Definition fanout (f : treelist) : Prop := exists n, fanout_restr n f.

Theorem fanout_rec_node : forall (n : nat) (k : key) (f : treelist) (f' :
 ↪  treelist),
  fanout_restr n (tl_cons (node k f') f) ->
  exists (n' : nat), n' <= b /\ n' > div_two b false /\ fanout_restr n' f'.
Proof.
  intros. inversion H. subst.
  exists n'. split; try split.
  - inversion H5; omega.
  - apply H3.
  - apply H5.
Qed.

Theorem fanout_rec_final : forall (n : nat) (f : treelist) (f' : treelist),
  fanout_restr n (tl_cons (final f') f) ->
  exists (n' : nat), n' <= b /\ n' > div_two b false /\ fanout_restr n' f'.
Proof.
  intros. inversion H. subst.
  exists n0. split; try split.
  - inversion H4; try omega.
    * admit. (* Relies on b being at least 1 *)
  - apply H3.
  - apply H4.
Admitted.

Inductive cursor_correct_struct : cursor -> Prop :=
| cc_nil : cursor_correct_struct ([],[])
| cc_first : forall n f, cursor_correct_struct ([n],[f])
| cc_node : forall n n' k f f' ci ct,
    cursor_correct_struct (n::ci,f::ct) -> lin_search n f = Some (node k f')
     ↪  ->
    cursor_correct_struct (n'::n::ci,f'::f::ct)
| cc_final : forall n n' f f' ci ct,
    cursor_correct_struct (n::ci,f::ct) -> lin_search n f = Some (final f')
     ↪  ->
    cursor_correct_struct (n'::n::ci, f'::f::ct).

Inductive rec_prop (P : treelist -> Prop) : cursor -> Prop :=
| rp_nil : rec_prop P ([],[])
| rp_next : forall n f cn cf, P f -> rec_prop P (cn,cf) -> rec_prop P
 ↪  (n::cn,f::cf).

Definition cursor_correct (c : cursor) : Prop :=
  cursor_correct_struct c /\
  rec_prop balanced c /\
  rec_prop sorted c /\
  rec_prop fanout c.

Fixpoint get_root (cf : list treelist) : treelist :=
  match cf with
  | [] => tl_nil
  | [f] => f
  | f::cf => get_root cf
  end.

Fixpoint get_treelist (c : cursor) : treelist :=
  match c with (cn,cf) => get_root cf end.
```

```
Definition abs_rel (f : treelist) (c : cursor) : Prop :=
  cursor_correct_struct c /\ get_treelist c = f.

(** Abstraction of cursors as bi-directional list of key-value pairs
  * Proofs about this abstraction and the implementation
  *)

(* Lemma 1 *)
(* Proof that splitting into sides is equivalent -- now I can prove about
 ↪  the sides separately! *)
Lemma cursor_elements'_sides_equiv : forall cn cf l r,
  cursor_elements' cn cf l r = (cursor_left cn cf l, cursor_right cn cf r).
Proof.
  induction cn,cf; intros; simpl; try reflexivity.
  destruct (point a t) eqn:e. destruct p. apply IHcn.
Qed.

(* Cursor to list of elements : cons in both directions *)
(* get returns the next thing in that list! *)

Definition right_el_P (t : tree) : Prop := forall k f b,
  t = node k f \/ t = final f ->
  exists l', right_el f b = b++l'.

(* Lemma 2 *)
Lemma right_el_interior : forall f b,
  exists l', right_el f b = b++l'.
Proof.
  induction f using treelist_tree_rec with (P := right_el_P).
  - unfold right_el_P. intros. inversion H; inversion H0. subst. apply IHf.
  - unfold right_el_P. intros. inversion H; inversion H0. subst. apply IHf.
  - unfold right_el_P. intros. inversion H; inversion H0.
  - intros. unfold right_el. exists []. rewrite app_nil_r. reflexivity.
  - intros. destruct t eqn:e.
    + simpl. unfold right_el_P in IHf.
      assert (exists l', right_el t0 b0 = b0++l').
      { apply IHf with k. left. reflexivity. }
      inversion H. rewrite H0. destruct IHf0 with (b0 ++ x).
      exists (x++x0). rewrite H1. rewrite app_assoc. reflexivity.
    + simpl. unfold right_el_P in IHf.
      assert (exists l', right_el t0 b0 = b0++l').
      { apply IHf with zero. right. reflexivity. }
      inversion H. rewrite H0. destruct IHf0 with (b0 ++ x).
      exists (x ++ x0). rewrite H1. rewrite app_assoc. reflexivity.
    + simpl. destruct IHf0 with (b0 ++ [(k,v)]).
      exists ((k,v)::x). rewrite H. rewrite <- app_assoc. reflexivity.
Qed.

Definition left_el_P (t : tree) : Prop := forall k f b,
  t = node k f \/ t = final f ->
  exists l', left_el f b = b++l'.

(* Lemma 2 *)
Lemma left_el_interior : forall f b,
  exists l', left_el f b = b++l'.
Proof.
  induction f using treelist_tree_rec with (P := left_el_P).
  - unfold left_el_P. intros. inversion H; inversion H0. subst. apply IHf.
  - unfold left_el_P. intros. inversion H; inversion H0. subst. apply IHf.
  - unfold left_el_P. intros. inversion H; inversion H0.
  - intros. unfold left_el. exists []. rewrite app_nil_r. reflexivity.
  - intros. destruct t eqn:e.
    + simpl. unfold left_el_P in IHf.
      assert (exists l', left_el f b0 = b0++l'). { apply IHf0. }
      inversion H. rewrite H0. destruct IHf with k t0 (b0 ++ x).
      * left. reflexivity.
      * exists (x ++ x0). rewrite H1. rewrite app_assoc. reflexivity.
    + simpl. unfold left_el_P in IHf.
      assert (exists l', left_el f b0 = b0++l'). { apply IHf0. }
      inversion H. rewrite H0. destruct IHf with zero t0 (b0 ++ x).
      * right. reflexivity.
      * exists (x ++ x0). rewrite H1. rewrite app_assoc. reflexivity.
    + simpl. destruct IHf0 with b0. rewrite H.
      exists (x++[(k,v)]). rewrite <- app_assoc. reflexivity.
Qed.

(* Lemma 2 *)
Lemma left_rec_interior : forall cn cf l,
  exists l', cursor_left cn cf l = (l++l').
Proof.
  induction cn,cf.
  - intros. simpl. exists []. rewrite app_nil_r. reflexivity.
  - intros. simpl. exists []. rewrite app_nil_r. reflexivity.
  - intros. simpl. exists []. rewrite app_nil_r. reflexivity.
  - intros. simpl. destruct (point a t) eqn:e. destruct p.
    destruct IHcn with cf (left_el t1 l). rewrite H.
    assert (exists l', left_el t1 l = l++l') by (apply left_el_interior).
    inversion H0. rewrite H1. exists (x0++x). rewrite app_assoc.
     ↪  reflexivity.
Qed.

(* Lemma 2 *)
Lemma right_rec_interior : forall cn cf r,
  exists r', cursor_right cn cf r = (r++r').
Proof.
  induction cn,cf.
  - intros. simpl. exists []. rewrite app_nil_r. reflexivity.
  - intros. simpl. exists []. rewrite app_nil_r. reflexivity.
  - intros. simpl. exists []. rewrite app_nil_r. reflexivity.
  - intros. simpl. destruct (point a t) eqn:e. destruct p.
    destruct IHcn with cf (right_el t0 r). rewrite H.
    assert (exists r', right_el t0 r = r++r') by (apply right_el_interior).
    inversion H0. rewrite H1. exists (x0++x). rewrite app_assoc.
     ↪  reflexivity.
Qed.

(* Lemma 3 *)
(* If you split a list, the result of the elements of the first with the
 ↪  elements of the second is the same *)
Lemma cursor_right_elements1 : forall cn1 cf1 cn2 cf2 b,
  length cn1 = length cf1 -> length cn2 = length cf2 ->
  cursor_right (cn1++cn2) (cf1++cf2) b = cursor_right cn2 cf2 (cursor_right
   ↪  cn1 cf1 b).
Proof.
  induction cn1,cf1; intros; simpl.
  - reflexivity.
  - inversion H.
  - inversion H.
  - inversion H. clear H.
    destruct (point a t) eqn:e. destruct p.
    apply IHcn1. apply H2. apply H0.
Qed.

(* Lemma 3 *)
Lemma cursor_left_elements1 : forall cn1 cf1 cn2 cf2 b,
  length cn1 = length cf1 -> length cn2 = length cf2 ->
  cursor_left (cn1++cn2) (cf1++cf2) b = cursor_left cn2 cf2 (cursor_left cn1
   ↪  cf1 b).
Proof.
  induction cn1,cf1; intros; simpl.
  - reflexivity.
  - inversion H.
  - inversion H.
  - inversion H. clear H.
    destruct (point a t) eqn:e. destruct p.
    apply IHcn1. apply H2. apply H0.
Qed.

(* List in increasing order *)
Inductive sorted_less : Z -> list (key * V) -> Prop :=
| sl_nil : forall n, sorted_less n []
| sl_next : forall k v n l,
  lt_key k n = true -> sorted_less n l -> sorted_less k ((k,v)::l).

(* List in decreasing order *)
Inductive sorted_more : list (key * V) -> Prop :=
| sm_nil : sorted_more []
| sm_one : forall k v, sorted_more [(k,v)]
| sm_next : forall k1 v1 k2 v2 l,
  lt_key k2 k1 = true -> sorted_more ((k2,v2)::l) -> sorted_more
   ↪  ((k1,v1)::(k2,v2)::l).

(* Lemma 4 *)
Lemma cursor_right_el_sorted : forall cn cf n,
  cursor_correct (cn,cf) -> sorted_less n (cursor_right cn cf []).
Proof.
  induction cn,cf; intros.
  - simpl. apply sl_nil.
  - simpl. apply sl_nil.
  - simpl. apply sl_nil.
  - simpl. destruct H. destruct H0. destruct H1. inversion H1. subst.
    remember (a::cn). destruct H1 eqn:e. admit.
Admitted.

(* Lemma 4 *)
Lemma cursor_left_el_sorted : forall cn cf,
  cursor_correct (cn,cf) -> sorted_more (cursor_left cn cf []).
Proof. Admitted.

(** Theorems about next_node and prev_node *)

(* Helper for Invariants *)
Lemma point_next : forall f n f1 f1' f2 f2' o t,
  point n f = (f1, o, f2) ->
  point (S n) f = (f1', Some t, f2') ->
  f2 = tl_cons t f2'.
Proof.
```

```
  induction f.
  - intros. inversion H0.
  - intros. destruct n.
    + simpl in H. simpl in H0. destruct f.
      * inversion H0.
      * inversion H. inversion H0. reflexivity.
    + simpl in H. remember (S n) as n'. simpl in H0.
      destruct (point n f) eqn:e. destruct p. inversion H. subst. clear H.
      destruct (point (S n) f) eqn:e'. destruct p. inversion H0. subst.
       ↪  clear H0.
      apply IHf with n t2 t3 o. apply e. apply e'.
Qed.

Theorem cursor_right_elements4 : forall cn cf cn' cf' n f k f' t l1 l2 b k',
  cursor_correct_struct (cn,cf) ->
  next_node cn cf = Some (n::cn',f::cf',k') ->
  (* (cn,cf) <> (n::cn',f::cf') -> *)
  point n f = (l1,Some t,l2) ->
  t = node k f' \/ t = final f' ->
  cursor_right cn cf b = cursor_right (n::cn') (f::cf') (right_el f' b).
Proof.
  induction cn,cf; intros.
  - simpl in H0. inversion H0.
  - simpl in H0. inversion H0.
  - simpl in H0. inversion H0.
  - simpl in H0. destruct (point a t) eqn:e. destruct p. destruct o; try
   ↪  (destruct t3); destruct t1.
    + destruct (next_node cn cf) eqn:e2. destruct p. destruct c; destruct l.
     ↪  2:destruct l0.
      * inversion H0.
      * inversion H0.
      * destruct (point n0 t1) eqn:e3. destruct p.
        destruct o. destruct t6.
        { inversion H0. subst. clear H0.
          simpl. rewrite e. destruct f. inversion H1.
          rewrite e3. simpl.
          assert (cursor_right cn cf b0 = cursor_right (n0::l) (t1::l0)
           ↪  (right_el (tl_cons t6 f) b0)).
          { apply IHcn with k2 (node k2 (tl_cons t6 f)) t5 t4 k'.
            inversion H. subst. apply cc_nil. apply H4. apply H4.
            apply e2.
            apply e3.
            left. reflexivity. }
          rewrite H0. simpl. rewrite e3. simpl in H1. inversion H1. subst.
          destruct H2. rewrite H2. reflexivity. rewrite H2. reflexivity. }
        { inversion H0. subst. clear H0.
          simpl. rewrite e. destruct f. inversion H1.
          rewrite e3. simpl.
          assert (cursor_right cn cf b0 = cursor_right (n0::l) (t1::l0)
           ↪  (right_el (tl_cons t6 f) b0)).
          { apply IHcn with k (final (tl_cons t6 f)) t5 t4 k'.
            inversion H. subst. apply cc_nil. apply H4. apply H4.
            apply e2.
            apply e3.
            right. reflexivity. }
          rewrite H0. simpl. rewrite e3. simpl in H1. inversion H1. subst.
          destruct H2. rewrite H2. reflexivity. rewrite H2. reflexivity. }
        { inversion H0. }
        { inversion H0. }
      * inversion H0.
    + destruct t1.
      * inversion H0. subst. remember (S a) as a'. simpl. rewrite e. rewrite
       ↪  H1.
        assert (tl_cons (node k1 t1) t4 = tl_cons t0 l2).
        { apply point_next with f a t2 l1 (Some (node k' t3)). apply e.
         ↪  subst. apply H1. }
        inversion H3. subst. inversion H2; inversion H4. subst.
        simpl. reflexivity.
      * inversion H0. subst. remember (S a) as a'. simpl. rewrite e. rewrite
       ↪  H1.
        assert (tl_cons (final t1) t4 = tl_cons t0 l2).
        { apply point_next with f a t2 l1 (Some (node k' t3)). apply e.
         ↪  subst. apply H1. }
        inversion H3. subst. inversion H2; inversion H4. subst.
        simpl. reflexivity.
      * admit. (* e: node and val in same treelist *)
    + destruct (next_node cn cf) eqn:e2. destruct p. destruct c; destruct l.
     ↪  2:destruct l0.
      inversion H0.
      inversion H0.
      destruct (point n0 t1) eqn:e3. destruct p. destruct o. destruct t6.
      { inversion H0. subst. clear H0.
        simpl. rewrite e. destruct f. inversion H1.
        rewrite e3. simpl.
        assert (cursor_right cn cf b0 = cursor_right (n0::l) (t1::l0)
         ↪  (right_el (tl_cons t6 f) b0)).
          { apply IHcn with k1 (node k1 (tl_cons t6 f)) t5 t4 k'.
            inversion H. subst. apply cc_nil. apply H4. apply H4.
            apply e2.
            apply e3.
            left. reflexivity. }
          rewrite H0. simpl. rewrite e3. simpl in H1. inversion H1. subst.
          destruct H2. rewrite H2. reflexivity. rewrite H2. reflexivity. }
      { inversion H0. subst. clear H0.
        simpl. rewrite e. destruct f. inversion H1.
        rewrite e3. simpl.
        assert (cursor_right cn cf b0 = cursor_right (n0::l) (t1::l0)
         ↪  (right_el (tl_cons t6 f) b0)).
          { apply IHcn with k (final (tl_cons t6 f)) t5 t4 k'.
            inversion H. subst. apply cc_nil. apply H4. apply H4.
            apply e2.
            apply e3.
            right. reflexivity. }
          rewrite H0. simpl. rewrite e3. simpl in H1. inversion H1. subst.
          destruct H2. rewrite H2. reflexivity. rewrite H2. reflexivity. }
      { inversion H0. }
      { inversion H0. }
      { inversion H0. }
    + destruct (next_node cn cf) eqn:e2. destruct p. destruct c; destruct l.
     ↪  2:destruct l0.
      inversion H0.
      inversion H0.
      destruct (point n0 t5) eqn:e3. destruct p. destruct o. destruct t8.
      { inversion H0. subst. clear H0.
        simpl. rewrite e. destruct f. inversion H1.
        rewrite e3. simpl. admit. }
      admit. admit. admit. admit.
    + admit.
```

```
    + admit.
    + admit.
    + admit.
Admitted.

(* Helper for invariant *)
Lemma left_el_app : forall f1 f2 b,
  left_el (tl_app f1 f2) b = left_el f1 (left_el f2 b).
Proof.
  induction f1; intros.
  - reflexivity.
  - simpl. rewrite IHf1. reflexivity.
Qed.

Theorem cursor_left_elements4 : forall cn cf cn' cf' n f b f1 o f2 k,
  cursor_correct_struct (n::cn,f::cf) ->
  next_node (n::cn) (f::cf) = Some (cn',cf',k) ->
  (cn',cf') <> (n::cn,f::cf) ->
  point n f = (f1,o,f2) -> exists k f',
  (o = Some (node k f') /\ cursor_left (n::cn) (f::cf) (left_el f' b) =
   ↪  cursor_left cn' cf' b) \/
  (o = Some (final f') /\ cursor_left (n::cn) (f::cf) (left_el f' b) =
   ↪  cursor_left cn' cf' b) \/
  (o = None /\ cursor_left (n::cn) (f::cf) b = cursor_left cn' cf' b).
Proof.
  induction cn,cf; intros.
  - simpl in H0. rewrite H2 in H0. destruct o. destruct t.
    + exists k0,t. left. split. reflexivity. destruct f2. 2:destruct t0.
      * inversion H0.
      * inversion H0. remember (S n) as n'. simpl.
        destruct (point n f) eqn:e1. destruct p.
        destruct (point n' f) eqn:e2. destruct p.
        assert (t4 = tl_app t2 (tl_cons (node k t) tl_nil)).
        { apply point_prev with n f t1 t3 o0. inversion H2. subst. apply e1.
          rewrite <- Heqn'. apply e2. }
        rewrite H3. rewrite left_el_app. reflexivity.
      * inversion H0. remember (S n) as n'. simpl.
        destruct (point n f) eqn:e1. destruct p.
        destruct (point n' f) eqn:e2. destruct p.
        assert (t4 = tl_app t2 (tl_cons (node k t) tl_nil)).
        { apply point_prev with n f t1 t3 o0. inversion H2. subst. apply e1.
          rewrite <- Heqn'. apply e2. }
        rewrite H3. rewrite left_el_app. reflexivity.
      * inversion H0.
    + inversion H0.
    + inversion H0. subst. destruct H1. reflexivity.
    + inversion H0.
  - inversion H.
  - inversion H.
  - remember (a::cn) as cn1. remember (t::cf) as cf1.
    simpl in H0. rewrite H2 in H0.
    destruct (next_node cn1 cf1) eqn:e1; try (destruct p; destruct c);
    destruct o eqn:e2; destruct f2.
    + destruct l. 2:destruct l0.
      destruct t0. inversion H0. inversion H0. inversion H0. subst. destruct
       ↪  H1. reflexivity.
      destruct t0. inversion H0. inversion H0. inversion H0. subst. destruct
       ↪  H1. reflexivity.
      destruct t0. destruct (point n0 t1) eqn:e3. destruct p.
      destruct o0. destruct t4.
      { exists k1,t0. left. split. reflexivity.
        inversion H0. remember (n0::l) as cn2. remember (t1::l0) as cf2.
        simpl. rewrite H2. destruct t4. admit. (* shouldn't be possible,
         ↪  because it would be a whole nil treelist *)
        simpl. admit. }
      admit.
      admit.
      admit.
      admit.
      admit.
    + admit.
    + admit.
    + admit.
    + admit.
    + admit.
    + admit.
    + admit.
Admitted.

(* Lemma 5 *)
Theorem next_correct_struct : forall cn cf c k,
  cursor_correct_struct (cn,cf) -> next_node cn cf = Some (c,k) ->
   ↪  cursor_correct_struct c.
Proof.
  induction cn,cf.
  - intros; simpl. simpl in H0. inversion H0.
  - intros; simpl. inversion H.
  - intros; simpl. inversion H.
  - intros. inversion H.
    + subst. simpl in H0. destruct (point a t) eqn:e. destruct p.
      destruct o. destruct t2; destruct t0; try (inversion H0).
      destruct t0; inversion H0; apply cc_first.
      apply H.
      apply cc_first.
      inversion H0.
    + subst. remember (n::ci) as cn'. remember (f::ct) as cf'.
      simpl in H0. destruct (point a t) eqn:e. destruct p.
      destruct o; destruct t0.
      * destruct t2. destruct (next_node cn' cf') eqn:e2. destruct p.
       ↪  destruct c0.
        destruct l. inversion H0. destruct l0. inversion H0.
        destruct (point n0 t2) eqn:e3. destruct p. destruct o. destruct t5.
        { inversion H0. apply cc_node with k3. apply IHcn with cf' k2. apply
         ↪  H3. apply e2.
          apply point_lin_search with t4 t3. apply e3. }
        { inversion H0. apply cc_final. apply IHcn with cf' k2. apply H3.
         ↪  apply e2.
          apply point_lin_search with t4 t3. apply e3. }
        { inversion H0. }
        { inversion H0. }
        { inversion H0. }
        destruct (next_node cn' cf') eqn:e2. destruct p. destruct c0.
        destruct l. inversion H0. destruct l0. inversion H0.
        destruct (point n0 t2) eqn:e3. destruct p. destruct o. destruct t5.
        { inversion H0. apply cc_node with k2. apply IHcn with cf' k1. apply
         ↪  H3. apply e2.
          apply point_lin_search with t4 t3. apply e3. }
        { inversion H0. apply cc_final. apply IHcn with cf' k1. apply H3.
         ↪  apply e2.
          apply point_lin_search with t4 t3. apply e3. }
        { inversion H0. }
        { inversion H0. }
        { inversion H0. }
        inversion H0. apply H.
      * destruct t2; destruct t0.
        { inversion H0. inversion H. apply cc_first. apply cc_node with k3.
          apply H7. apply H10. apply cc_final. apply H7. apply H10. }
        { inversion H0. inversion H. apply cc_first. apply cc_node with k2.
          apply H7. apply H10. apply cc_final. apply H7. apply H10. }
        destruct (next_node cn' cf') eqn:e2. destruct p. destruct c0.
        destruct l. inversion H0. destruct l0. inversion H0.
        destruct (point n0 t0) eqn:e3. destruct p. destruct o. destruct t6.
        { inversion H0. apply cc_node with k4. apply IHcn with cf' k3. apply
         ↪  H3. apply e2.
          apply point_lin_search with t5 t4. apply e3. }
        { inversion H0. apply cc_final. apply IHcn with cf' k3. apply H3.
         ↪  apply e2.
          apply point_lin_search with t5 t4. apply e3. }
        { inversion H0. }
        { inversion H0. }
        { inversion H0. }
        destruct (next_node cn' cf') eqn:e2. destruct p. destruct c0.
        destruct l. inversion H0. destruct l0. inversion H0.
        destruct (point n0 t4) eqn:e3. destruct p. destruct o. destruct t7.
        { inversion H0. apply cc_node with k3. apply IHcn with cf' k2. apply
         ↪  H3. apply e2.
          apply point_lin_search with t6 t5. apply e3. }
        { inversion H0. apply cc_final. apply IHcn with cf' k2. apply H3.
         ↪  apply e2.
          apply point_lin_search with t6 t5. apply e3. }
        { inversion H0. }
        { inversion H0. }
        { inversion H0. }
        destruct (next_node cn' cf') eqn:e2. destruct p. destruct c0.
        destruct l. inversion H0. destruct l0. inversion H0.
        destruct (point n0 t4) eqn:e3. destruct p. destruct o. destruct t7.
        { inversion H0. apply cc_node with k2. apply IHcn with cf' k1. apply
         ↪  H3. apply e2.
          apply point_lin_search with t6 t5. apply e3. }
        { inversion H0. apply cc_final. apply IHcn with cf' k1. apply H3.
         ↪  apply e2.
          apply point_lin_search with t6 t5. apply e3. }
        { inversion H0. }
        { inversion H0. }
        { inversion H0. }
        destruct (next_node cn' cf') eqn:e2. destruct p. destruct c0.
        destruct l. inversion H0. destruct l0. inversion H0.
        destruct (point n0 t0) eqn:e3. destruct p. destruct o. destruct t6.
        { inversion H0. apply cc_node with k3. apply IHcn with cf' k2. apply
         ↪  H3. apply e2.
          apply point_lin_search with t5 t4. apply e3. }
        { inversion H0. apply cc_final. apply IHcn with cf' k2. apply H3.
         ↪  apply e2.
          apply point_lin_search with t5 t4. apply e3. }
        { inversion H0. }
        { inversion H0. }
        { inversion H0. }
        { inversion H0. inversion H. apply cc_first. apply cc_node with k3.
```

```
        apply H7. apply H10. apply cc_final. apply H7. apply H10. }
      { inversion H0. inversion H. apply cc_first. apply cc_node with k2.
        apply H7. apply H10. apply cc_final. apply H7. apply H10. }
      { inversion H0. inversion H. apply cc_first. apply cc_node with k3.
        apply H7. apply H10. apply cc_final. apply H7. apply H10. }
    * destruct (next_node cn' cf') eqn:e2. destruct p. destruct c0.
      destruct l. inversion H0. destruct l0. inversion H0.
      destruct (point n0 t0) eqn:e3. destruct p. destruct o. destruct t4.
      { inversion H0. apply cc_node with k2. apply IHcn with cf' k1. apply
       ↪  H3. apply e2.
        apply point_lin_search with t3 t2. apply e3. }
      { inversion H0. apply cc_final. apply IHcn with cf' k1. apply H3.
       ↪  apply e2.
        apply point_lin_search with t3 t2. apply e3. }
      { inversion H0. }
      { inversion H0. }
      { inversion H0. }
    * apply point_none in e. inversion e.
  + subst. remember (n::ci) as cn'. remember (f::ct) as cf'.
    simpl in H0. destruct (point a t) eqn:e. destruct p.
    destruct o; destruct t0.
    * destruct t2. destruct (next_node cn' cf') eqn:e2. destruct p.
     ↪  destruct c0.
      destruct l. inversion H0. destruct l0. inversion H0.
      destruct (point n0 t2) eqn:e3. destruct p. destruct o. destruct t5.
      { inversion H0. apply cc_node with k2. apply IHcn with cf' k1. apply
       ↪  H3. apply e2.
        apply point_lin_search with t4 t3. apply e3. }
      { inversion H0. apply cc_final. apply IHcn with cf' k1. apply H3.
       ↪  apply e2.
        apply point_lin_search with t4 t3. apply e3. }
      { inversion H0. }
      { inversion H0. }
      { inversion H0. }
      destruct (next_node cn' cf') eqn:e2. destruct p. destruct c0.
      destruct l. inversion H0. destruct l0. inversion H0.
      destruct (point n0 t2) eqn:e3. destruct p. destruct o. destruct t5.
      { inversion H0. apply cc_node with k1. apply IHcn with cf' k0. apply
       ↪  H3. apply e2.
        apply point_lin_search with t4 t3. apply e3. }
      { inversion H0. apply cc_final. apply IHcn with cf' k0. apply H3.
       ↪  apply e2.
        apply point_lin_search with t4 t3. apply e3. }
      { inversion H0. }
      { inversion H0. }
      { inversion H0. }
      inversion H0. apply H.
    * destruct t2; destruct t0.
      { inversion H0. inversion H. apply cc_first. apply cc_node with k2.
        apply H7. apply H10. apply cc_final. apply H7. apply H10. }
      { inversion H0. inversion H. apply cc_first. apply cc_node with k1.
        apply H7. apply H10. apply cc_final. apply H7. apply H10. }
      destruct (next_node cn' cf') eqn:e2. destruct p. destruct c0.
      destruct l. inversion H0. destruct l0. inversion H0.
      destruct (point n0 t0) eqn:e3. destruct p. destruct o. destruct t6.
      { inversion H0. apply cc_node with k3. apply IHcn with cf' k2. apply
       ↪  H3. apply e2.
        apply point_lin_search with t5 t4. apply e3. }
      { inversion H0. apply cc_final. apply IHcn with cf' k2. apply H3.
       ↪  apply e2.
        apply point_lin_search with t5 t4. apply e3. }
      { inversion H0. }
      { inversion H0. }
      { inversion H0. }
      destruct (next_node cn' cf') eqn:e2. destruct p. destruct c0.
      destruct l. inversion H0. destruct l0. inversion H0.
      destruct (point n0 t4) eqn:e3. destruct p. destruct o. destruct t7.
      { inversion H0. apply cc_node with k2. apply IHcn with cf' k1. apply
       ↪  H3. apply e2.
        apply point_lin_search with t6 t5. apply e3. }
      { inversion H0. apply cc_final. apply IHcn with cf' k1. apply H3.
       ↪  apply e2.
        apply point_lin_search with t6 t5. apply e3. }
      { inversion H0. }
      { inversion H0. }
      { inversion H0. }
      destruct (next_node cn' cf') eqn:e2. destruct p. destruct c0.
      destruct l. inversion H0. destruct l0. inversion H0.
      destruct (point n0 t4) eqn:e3. destruct p. destruct o. destruct t7.
      { inversion H0. apply cc_node with k1. apply IHcn with cf' k0. apply
       ↪  H3. apply e2.
        apply point_lin_search with t6 t5. apply e3. }
      { inversion H0. apply cc_final. apply IHcn with cf' k0. apply H3.
       ↪  apply e2.
        apply point_lin_search with t6 t5. apply e3. }
      { inversion H0. }
      { inversion H0. }
      { inversion H0. }
      destruct (next_node cn' cf') eqn:e2. destruct p. destruct c0.
      destruct l. inversion H0. destruct l0. inversion H0.
      destruct (point n0 t0) eqn:e3. destruct p. destruct o. destruct t6.
      { inversion H0. apply cc_node with k2. apply IHcn with cf' k1. apply
       ↪  H3. apply e2.
        apply point_lin_search with t5 t4. apply e3. }
      { inversion H0. apply cc_final. apply IHcn with cf' k1. apply H3.
       ↪  apply e2.
        apply point_lin_search with t5 t4. apply e3. }
      { inversion H0. }
      { inversion H0. }
      { inversion H0. }
      { inversion H0. inversion H. apply cc_first. apply cc_node with k2.
        apply H7. apply H10. apply cc_final. apply H7. apply H10. }
      { inversion H0. inversion H. apply cc_first. apply cc_node with k1.
        apply H7. apply H10. apply cc_final. apply H7. apply H10. }
      { inversion H0. inversion H. apply cc_first. apply cc_node with k2.
        apply H7. apply H10. apply cc_final. apply H7. apply H10. }
    * destruct (next_node cn' cf') eqn:e2. destruct p. destruct c0.
      destruct l. inversion H0. destruct l0. inversion H0.
      destruct (point n0 t0) eqn:e3. destruct p. destruct o. destruct t4.
      { inversion H0. apply cc_node with k1. apply IHcn with cf' k0. apply
       ↪  H3. apply e2.
        apply point_lin_search with t3 t2. apply e3. }
      { inversion H0. apply cc_final. apply IHcn with cf' k0. apply H3.
       ↪  apply e2.
        apply point_lin_search with t3 t2. apply e3. }
      { inversion H0. }
      { inversion H0. }
      { inversion H0. }
    * apply point_none in e. inversion e.
```

```
Qed.

(* Helper for Lemma 5 *)
Lemma balance_carries_one : forall n f k f',
  balanced f ->
  (lin_search n f = Some (node k f') \/ lin_search n f = Some (final f')) ->
  balanced f'.
Proof.
  induction n; intros; destruct f.
  - inversion H0; inversion H1.
  - inversion H0; inversion H1; rewrite H3 in H; inversion H; inversion H2;
   ↪  subst.
    unfold balanced. exists n. apply H7.
    unfold balanced. exists n. apply H6.
  - inversion H0; inversion H1.
  - simpl in H0. apply IHn with f k.
    + inversion H. inversion H1; subst.
      exists 1. apply H4.
      exists (S n0). apply H6.
      exists 1. apply bf_nil.
    + apply H0.
Qed.

(* Helper for Lemma 5 *)
Lemma balance_carries : forall cn cf n f,
  cursor_correct_struct (n::cn,f::cf) ->
  rec_prop balanced (cn,cf) ->
  cn <> [] ->
  rec_prop balanced (n::cn,f::cf).
Proof.
  intros. inversion H0. destruct H1. auto.
  subst. apply rp_next. 2:apply H0.
  inversion H4. inversion H.
  - subst. apply balance_carries_one with n0 f0 k. apply H4. left. apply
   ↪  H12.
  - subst. apply balance_carries_one with n0 f0 Z0. apply H4. right. apply
   ↪  H12.
Qed.

(* Lemma 5 *)
Lemma next_node_balanced : forall cn cf c k,
  cursor_correct_struct (cn,cf) -> rec_prop balanced (cn,cf) ->
  next_node cn cf = Some (c,k) -> rec_prop balanced c.
Proof.
  induction cn,cf; intros.
  - inversion H1.
  - inversion H1.
  - inversion H1.
  - inversion H0. subst.
    assert (cursor_correct_struct (cn,cf)).
    { inversion H; subst. apply cc_nil. apply H5. apply H5. }
    assert (IHpart: forall c k, next_node cn cf = Some (c,k) -> rec_prop
     ↪  balanced c).
    { intros. apply IHcn with cf k0. apply H2. apply H7. apply H3. }
    assert (cursor_correct_struct c).
    { apply next_correct_struct with (a::cn) (t::cf) k. apply H. apply H1. }
    inversion H1. destruct (point a t) eqn:e1. destruct p.
    destruct o; try (destruct t2); destruct t0.
    + destruct (next_node cn cf) eqn:e2. destruct p. destruct c0. destruct
     ↪  l. 2:destruct l0.
      inversion H6. inversion H6. destruct (point n t0). destruct p.
      2:inversion H6. destruct o. 2:inversion H6. destruct t5. 3:inversion
       ↪  H6.
      * inversion H6. apply balance_carries.
        rewrite H8. apply H3.
        apply IHpart with k1. reflexivity.
        intros Hcontra. inversion Hcontra.
      * inversion H6. apply balance_carries.
        rewrite H8. apply H3.
        apply IHpart with k1. reflexivity.
        intros Hcontra. inversion Hcontra.
    + destruct t0.
      { inversion H6. apply rp_next. apply H4. apply H7. }
      { inversion H6. apply rp_next. apply H4. apply H7. }
      destruct (next_node cn cf) eqn:e2. destruct p. destruct c0. destruct
       ↪  l. 2:destruct l0.
      inversion H6. inversion H6. 2:inversion H6.
      destruct (point n t0) eqn:e3. destruct p. destruct o. destruct t6.
      { inversion H6. apply balance_carries.
        rewrite H8. apply H3.
        apply IHpart with k2. reflexivity.
        intros Hcontra. inversion Hcontra. }
      { inversion H6. apply balance_carries.
        rewrite H8. apply H3.
        apply IHpart with k2. reflexivity.
        intros Hcontra. inversion Hcontra. }
      inversion H6. inversion H6.
    + destruct (next_node cn cf) eqn:e2. destruct p. destruct c0. destruct
     ↪  l. 2:destruct l0.
      inversion H6. inversion H6. destruct (point n t0). destruct p.
      2:inversion H6. destruct o. 2:inversion H6. destruct t5. 3:inversion
       ↪  H6.
      * inversion H6. apply balance_carries.
        rewrite H8. apply H3.
        apply IHpart with k0. reflexivity.
        intros Hcontra. inversion Hcontra.
      * inversion H6. apply balance_carries.
        rewrite H8. apply H3.
        apply IHpart with k0. reflexivity.
        intros Hcontra. inversion Hcontra.
    + destruct (next_node cn cf) eqn:e2. destruct p. destruct c0. destruct
     ↪  l. 2:destruct l0.
      inversion H6. inversion H6. destruct (point n t4). destruct p.
      2:inversion H6. destruct o. 2:inversion H6. destruct t7. 3:inversion
       ↪  H6.
      * inversion H6. apply balance_carries.
        rewrite H8. apply H3.
        apply IHpart with k0. reflexivity.
        intros Hcontra. inversion Hcontra.
      * inversion H6. apply balance_carries.
        rewrite H8. apply H3.
        apply IHpart with k0. reflexivity.
        intros Hcontra. inversion Hcontra.
    + inversion H6. apply H0.
    + inversion H6. apply H0.
    + destruct (next_node cn cf) eqn:e2. destruct p. destruct c0. destruct
     ↪  l. 2:destruct l0.
      inversion H6. inversion H6. destruct (point n t0). destruct p.
      2:inversion H6. destruct o. 2:inversion H6. destruct t4. 3:inversion
       ↪  H6.
```

```
      * inversion H6. apply balance_carries.
        rewrite H8. apply H3.
        apply IHpart with k0. reflexivity.
        intros Hcontra. inversion Hcontra.
      * inversion H6. apply balance_carries.
        rewrite H8. apply H3.
        apply IHpart with k0. reflexivity.
        intros Hcontra. inversion Hcontra.
    + destruct (next_node cn cf) eqn:e2. destruct p. destruct c0. destruct
     ↪  l. 2:destruct l0.
      inversion H6. inversion H6. destruct (point n t3). destruct p.
      2:inversion H6. destruct o. 2:inversion H6. destruct t6. 3:inversion
       ↪  H6.
      * inversion H6. apply balance_carries.
        rewrite H8. apply H3.
        apply IHpart with k0. reflexivity.
        intros Hcontra. inversion Hcontra.
      * inversion H6. apply balance_carries.
        rewrite H8. apply H3.
        apply IHpart with k0. reflexivity.
        intros Hcontra. inversion Hcontra.
Qed.

(* Lemma 5 *)
Lemma next_node_sorted : forall cn cf c k,
  cursor_correct_struct (cn,cf) -> rec_prop sorted (cn,cf) ->
  next_node cn cf = Some (c,k) -> rec_prop sorted c.
Proof. Admitted.

(* Lemma 5 *)
Lemma next_node_fanout : forall cn cf c k,
  cursor_correct_struct (cn,cf) -> rec_prop fanout (cn,cf) ->
  next_node cn cf = Some (c,k) -> rec_prop fanout c.
Proof. Admitted.

(* Lemma 5 *)
Lemma next_node_correct : forall cn cf c k,
  cursor_correct (cn,cf) -> next_node cn cf = Some (c,k) -> cursor_correct
   ↪  c.
Proof.
  intros. destruct H. destruct H1. destruct H2.
  unfold cursor_correct. split. 2:split. 3:split.
  - apply next_correct_struct with cn cf k. apply H. apply H0.
  - apply next_node_balanced with cn cf k. apply H. apply H1. apply H0.
  - apply next_node_sorted with cn cf k. apply H. apply H2. apply H0.
  - apply next_node_fanout with cn cf k. apply H. apply H3. apply H0.
Qed.

Lemma next_node_none : forall cn cf cn' cf' k,
  next_node cn cf = Some (cn',cf',k) ->
  cn' <> [] /\ cf' <> [].
Proof.
  induction cn,cf; intros.
  - simpl in H. inversion H.
  - simpl in H. inversion H.
  - simpl in H. inversion H.
  - simpl in H. destruct (point a t) eqn:e. destruct p.
    destruct t0; destruct o. destruct t0.
    * destruct (next_node cn cf) eqn:e2. destruct p. destruct c; destruct l.
     ↪  2:destruct l0.
      inversion H. inversion H.
      destruct (point n t2) eqn:e3. destruct p.
      destruct o; try (destruct t5); inversion H; split; intros Hcontra;
       ↪  inversion Hcontra.
      inversion H.
    * destruct (next_node cn cf) eqn:e2. destruct p. destruct c; destruct l.
     ↪  2:destruct l0.
      inversion H. inversion H.
      destruct (point n t2) eqn:e3. destruct p.
      destruct o; try (destruct t5); inversion H; split; intros Hcontra;
       ↪  inversion Hcontra.
      inversion H.
    * inversion H. split; intros Hcontra; inversion Hcontra.
    * destruct (next_node cn cf) eqn:e2. destruct p. destruct c; destruct l.
     ↪  2:destruct l0.
      inversion H. inversion H.
      destruct (point n t0) eqn:e3. destruct p.
      destruct o; try (destruct t4); inversion H; split; intros Hcontra;
       ↪  inversion Hcontra.
      inversion H.
    * destruct t3; destruct t0.
      inversion H. split; intros Hcontra; inversion Hcontra.
      inversion H. split; intros Hcontra; inversion Hcontra.
      destruct (next_node cn cf) eqn:e2. destruct p. destruct c; destruct l.
       ↪  2:destruct l0.
      inversion H. inversion H.
      destruct (point n t0) eqn:e3. destruct p. destruct o. destruct t6.
      inversion H. split; intros Hcontra; inversion Hcontra.
      inversion H. split; intros Hcontra; inversion Hcontra.
      inversion H. inversion H. inversion H.
      destruct (next_node cn cf) eqn:e2. destruct p. destruct c; destruct l.
       ↪  2:destruct l0.
      inversion H. inversion H.
      destruct (point n t4) eqn:e3. destruct p. destruct o. destruct t7.
      inversion H. split; intros Hcontra; inversion Hcontra.
      inversion H. split; intros Hcontra; inversion Hcontra.
      inversion H. inversion H. inversion H.
      destruct (next_node cn cf) eqn:e2. destruct p. destruct c; destruct l.
       ↪  2:destruct l0.
      inversion H. inversion H.
      destruct (point n t4) eqn:e3. destruct p. destruct o. destruct t7.
      inversion H. split; intros Hcontra; inversion Hcontra.
      inversion H. split; intros Hcontra; inversion Hcontra.
      inversion H. inversion H. inversion H.
      destruct (next_node cn cf) eqn:e2. destruct p. destruct c; destruct l.
       ↪  2:destruct l0.
      inversion H. inversion H.
      destruct (point n t0) eqn:e3. destruct p. destruct o. destruct t6.
      inversion H. split; intros Hcontra; inversion Hcontra.
      inversion H. split; intros Hcontra; inversion Hcontra.
      inversion H. inversion H. inversion H.
      inversion H. split; intros Hcontra; inversion Hcontra.
      inversion H. split; intros Hcontra; inversion Hcontra.
      inversion H. split; intros Hcontra; inversion Hcontra.
    * destruct (next_node cn cf) eqn:e2. destruct p. destruct c; destruct l.
     ↪  2:destruct l0.
      inversion H. inversion H.
      destruct (point n t3) eqn:e3. destruct p. destruct o. destruct t6.
      inversion H. split; intros Hcontra; inversion Hcontra.
```

```
      inversion H. split; intros Hcontra; inversion Hcontra.
      inversion H. inversion H. inversion H.
Qed.

(* Lemma 6 *)
Lemma prev_node_correct : forall cn cf c k,
  cursor_correct (cn,cf) -> prev_node cn cf = Some (c,k) -> cursor_correct
   ↪  c.
Proof. Admitted.

Lemma prev_node_none : forall cn cf cn' cf' k,
  prev_node cn cf = Some (cn',cf',k) ->
  cn' <> [] /\ cf' <> [].
Proof. Admitted.

(* Lemma 7 *)
Lemma cursor_elements_next : forall cn cf c k,
  cursor_correct_struct (cn,cf) ->
  next_node cn cf = Some (c,k) ->
  cursor_elements (cn,cf) = cursor_elements c.
Proof.
  intros. destruct cn,cf.
  - inversion H0.
  - inversion H0.
  - inversion H0.
  - simpl in H0. destruct (point n t) eqn:e. destruct p.
    destruct o. destruct t2.
    + admit. (* e should only have vals *)
    + admit. (* ditto *)
    + destruct t0. inversion H0. reflexivity.
      destruct t0. admit. admit. (* ditto *) inversion H0. reflexivity.
    + destruct t0. destruct (next_node cn cf) eqn:e2. destruct p. destruct
     ↪  c0.
      destruct l. 2:destruct l0. inversion H0. inversion H0.
      destruct (point n0 t0) eqn:e3. destruct p. destruct o. destruct t4.
      admit. admit. admit. admit. admit. admit.
Admitted.

(* Lemma 7 *)
Lemma cursor_elements_prev : forall cn cf c k,
  cursor_correct_struct (cn,cf) ->
  prev_node cn cf = Some (c,k) ->
  cursor_elements (cn,cf) = cursor_elements c.
Proof. Admitted.

(** Proofs about make_cursor *)

Fixpoint dec (n : nat) (f : treelist) : treelist :=
  match n with
  | O => f
  | S n' =>
    (match f with
     | tl_cons t f' => dec n' f'
     | tl_nil => tl_nil
     end)
  end.

Definition mc_correct_P (x : key) (t : tree) : Prop := forall k f n ci ct,
  t = node k f \/ t = final f ->
  cursor_correct_struct (n::ci,f::ct) ->
  cursor_correct_struct (make_cursor_rec x f ci (f::ct) O).

Lemma dec_nil : forall n,
  dec n tl_nil = tl_nil.
Proof. destruct n; reflexivity. Qed.

Lemma dec_cont : forall f n,
  dec (S n) f = dec 1 (dec n f).
Proof.
  induction f; intros.
  - simpl. destruct n; reflexivity.
  - induction n.
    + reflexivity.
    + replace (dec (S (S n)) (tl_cons t f)) with (dec (S n) f).
      replace (dec (S n) (tl_cons t f)) with (dec n f).
      apply IHf. reflexivity. reflexivity.
Qed.

Lemma dec_lin_search : forall n f t f'',
  dec n f = tl_cons t f'' -> lin_search n f = Some t.
Proof.
  induction n; intros.
  - simpl in H. rewrite H. reflexivity.
  - simpl in H. destruct f. inversion H. simpl.
    apply IHn with f''. apply H.
Qed.

(* Lemma 8 *)
Lemma make_cursor_rec_correct : forall x f' n ci ct n' f,
  cursor_correct_struct (n::ci,f::ct) ->
  dec n' f = f' ->
  cursor_correct_struct (make_cursor_rec x f' ci (f::ct) n').
Proof.
  intros x. induction f' using treelist_tree_rec with (P := mc_correct_P x);
   ↪  try unfold mc_correct_P; intros.
  - inversion H; inversion H1. subst.
    apply IHf' with O. inversion H0.
    + apply cc_first.
    + apply cc_node with k. apply H4. apply H7.
    + apply cc_final. apply H4. apply H7.
    + reflexivity.
  - inversion H; inversion H1. subst.
    apply IHf' with O. inversion H0.
    + apply cc_first.
    + apply cc_node with k0. apply H4. apply H7.
    + apply cc_final. apply H4. apply H7.
    + reflexivity.
  - inversion H; inversion H1.
  - simpl. inversion H.
    + apply cc_first.
    + apply cc_node with k. apply H3. apply H6.
    + apply cc_final. apply H3. apply H6.
  - unfold mc_correct_P in IHf'. simpl. destruct t eqn:e. destruct (lt_key k
   ↪  x). 4:destruct (lt_key k x).
    + apply IHf'0 with n. apply H. rewrite dec_cont. rewrite H0.
     ↪  reflexivity.
    + apply IHf' with k O. left. reflexivity.
      * inversion H; subst.
```

```
        { apply cc_node with k. apply cc_first. apply dec_lin_search with
         ↪  f'. apply H0. }
        { apply cc_node with k. apply cc_node with k0. apply H3. apply H6.
         ↪  apply dec_lin_search with f'. apply H0. }
        { apply cc_node with k. apply cc_final. apply H3. apply H6. apply
         ↪  dec_lin_search with f'. apply H0. }
    + destruct f'. 2:apply cc_nil. apply IHf' with x 0. right. reflexivity.
      * inversion H; subst.
        { apply cc_final. apply cc_first. apply dec_lin_search with tl_nil.
         ↪  apply H0. }
        { apply cc_final. apply cc_node with k. apply H3. apply H6. apply
         ↪  dec_lin_search with tl_nil. apply H0. }
        { apply cc_final. apply cc_final. apply H3. apply H6. apply
         ↪  dec_lin_search with tl_nil. apply H0. }
    + apply IHf'0 with n. apply H. rewrite dec_cont. rewrite H0.
     ↪  reflexivity.
    + inversion H; subst.
      * apply cc_first.
      * apply cc_node with k0. apply H3. apply H6.
      * apply cc_final. apply H3. apply H6.
Qed.

(* Lemma 8 *)
Lemma make_cursor_correct : forall x f,
  cursor_correct_struct (make_cursor x f).
Proof.
  intros. unfold make_cursor. apply make_cursor_rec_correct with 0.
  - apply cc_first.
  - reflexivity.
Qed.

(* Lemma 9 *)
Lemma make_cursor_right : forall cn cf x f k v l,
  make_cursor x f = (cn,cf) ->
  cursor_right cn cf [] = (k,v)::l ->
  lt_key k x <> true.
Proof. Admitted.

(* Lemma 9 *)
Lemma make_cursor_left : forall cn cf x f k v l,
  make_cursor x f = (cn,cf) ->
  cursor_left cn cf [] = (k,v)::l ->
  lt_key k x = true.
Proof. Admitted.

(** Proofs about move_to_next and move_to_prev *)

(* Lemma 10 *)
Lemma move_to_next_correct : forall c,
  cursor_correct c ->
  cursor_correct (move_to_next c).
Proof.
  intros. destruct c. simpl. destruct (next_node l l0) eqn:e.
  destruct p. destruct c. destruct l1.
  - apply H.
  - assert (cursor_correct (n::l1,l2)).
    { apply next_node_correct with l l0 k. apply H. apply e. }
    destruct H0. destruct H1. destruct H2.
    split. 2:split. 3:split.
    + inversion H0. apply cc_first.
      apply cc_node with k0. apply H6. apply H8.
      apply cc_final. apply H6. apply H8.
    + inversion H1. apply rp_next. apply H6. apply H8.
    + inversion H2. apply rp_next. apply H6. apply H8.
    + inversion H3. apply rp_next. apply H6. apply H8.
  - apply H.
Qed.

(* Lemma 10 *)
Lemma move_to_prev_correct : forall c,
  cursor_correct c ->
  cursor_correct (move_to_prev c).
Proof. Admitted.

(* Lemma 11 *)
Lemma move_to_next_el : forall c l r k v,
  cursor_correct c ->
  cursor_elements c = (l,(k,v)::r) ->
  cursor_elements (move_to_next c) = ((k,v)::l,r).
Proof. Admitted.

(* Lemma 12 *)
Lemma move_to_prev_el : forall c l r k v,
  cursor_correct c ->
  cursor_elements c = ((k,v)::l,r) ->
  cursor_elements (move_to_next c) = (l,(k,v)::r).
Proof. Admitted.

(* Lemma 13 *)
Lemma move_to_next_none : forall cn cf,
  cursor_correct (cn,cf) ->
  (cursor_right cn cf [] = [] <-> move_to_next (cn,cf) = (cn,cf)).
Proof. Admitted.

(* Lemma 13 *)
Lemma move_to_prev_none : forall cn cf,
  cursor_correct (cn,cf) ->
  (cursor_left cn cf [] = [] <-> move_to_prev (cn,cf) = (cn,cf)).
Proof. Admitted.

(** Proofs about GET *)

(* Lemma 14 *)
Lemma get_correct : forall cn cf k v l,
  cursor_correct (cn,cf) ->
  cursor_right cn cf [] = (k,v)::l ->
  get_tree (cn,cf) = Some (val k v).
Proof. Admitted.

(** Proofs about INSERT *)

Definition key_rel (k : key) (c : cursor) : bool :=
  match get_key c with
  | Some k1 =>
    (match get_key (move_to_prev c) with
     | Some k2 => if andb (lt_key k2 k) (negb (lt_key k1 k)) then true else
     ↪  false
     | None => if negb (lt_key k1 k) then true else false
```

```
      end)
   | None =>
     (match get_key (move_to_prev c) with
      | Some k2 => if lt_key k2 k then true else false
      | None => true
      end)
   end.

(* Lemma 15 *)
Lemma bad_cursor_insert_same : forall c k v,
  key_rel k c <> true -> insert k v c = c.
Proof. Admitted.

(* Lemma 16 *)
Lemma insert_correct : forall c k v,
  cursor_correct c -> cursor_correct (insert k v c).
Proof. Admitted.

(* Lemma 17 *)
Lemma insert_eq_elements : forall c k v' v l r,
  cursor_elements c = (l,(k,v')::r) -> cursor_elements (insert k v c) =
   ↪  (l,(k,v)::r).
Proof. Admitted.

(* Theorem 18 *)
Lemma insert_neq_elements : forall c k v l r,
  get_key c <> Some k -> cursor_elements c = (l,r) -> cursor_elements
   ↪  (insert k v c) = ((k,v)::l,r).
Proof. Admitted.

(** Tests *)

Compute (treelist_depth ex_treelist).

Compute (treelist_depth ex_treelist').

Compute (treelist_depth ex_treelist'').

Example b1: balanced ex_treelist.
Proof.
  unfold balanced. exists 2%nat.
  unfold ex_treelist.
  apply bf_node.
  - apply bf_val. apply bf_val. apply bf_nil.
  - apply bf_final. apply bf_val. apply bf_nil.
Qed.

End BTREES.
```

B.2 Specification (BTreesModule.v)

```
Require Import BTrees.
Require Import Coq.Numbers.BinNums.
Require Import Coq.Lists.List.
Export ListNotations.

(** Abstract module type *)
Module Type CURSOR_TABLE.
 Parameter V: Type.
 Definition key := Z.
 Parameter table: Type.
 Parameter cursor : Type.
 Parameter empty_t: table.

 (* Functions of the implementation *)
 Parameter make_cursor: key -> table -> cursor.
 Parameter get_table: cursor -> table.
 Parameter get_key: cursor -> option key.
 Parameter get: cursor -> option V.
 Parameter insert: cursor -> key -> V -> cursor.
 Parameter next: cursor -> cursor.
 Parameter prev: cursor -> cursor.
 Parameter first_cursor: table -> cursor.
 Parameter last_cursor: table -> cursor.

 (* Props defining correctness *)
 Parameter abs_rel: table -> cursor -> Prop.
 Parameter key_rel: key -> cursor -> Prop.
 Parameter eq_cursor : cursor -> cursor -> Prop.
 Parameter cursor_correct: cursor -> Prop.
 Parameter table_correct: table -> Prop.

 (* table-cursor relations *)
 Axiom make_cursor_rel: forall t k,
       abs_rel t (make_cursor k t).
 Axiom get_table_rel: forall t c,
       abs_rel t c <-> get_table c = t.
 Axiom first_rel: forall t,
       abs_rel t (first_cursor t).
 Axiom last_rel: forall t,
       abs_rel t (last_cursor t).
 Axiom next_rel: forall t c,
       abs_rel t c -> abs_rel t (next c).
 Axiom prev_rel: forall t c,
       abs_rel t c -> abs_rel t (prev c).
 Axiom correct_rel: forall t c,
       abs_rel t c -> (cursor_correct c <-> table_correct t).

 (* correctness preservation *)
 Axiom insert_correct: forall k v c,
       cursor_correct c -> key_rel k c -> cursor_correct (insert c k v).

 (* get/insert correctness *)
 Axiom glast: forall t,
       get (last_cursor t) = None.
 Axiom gis: forall k v c,
       cursor_correct c -> key_rel k c ->
```

```
      get (make_cursor k (get_table (insert c k v))) = Some v.
Axiom gio: forall j k v c,
      cursor_correct c -> key_rel k c -> ~ key_rel j c ->
      get (make_cursor j (get_table (insert c k v))) =
      get (make_cursor j (get_table c)).

 (* cursor movement *)
Axiom next_prev: forall c t,
      cursor_correct c -> abs_rel t c -> ~ (c = last_cursor t) -> eq_cursor
       ↪  c (prev (next c)).
Axiom prev_next: forall c t,
      cursor_correct c -> abs_rel t c -> ~ (c = first_cursor t) ->
       ↪  eq_cursor c (next (prev c)).
Axiom cursor_order: forall c k1 k2,
      cursor_correct c -> get_key c = Some k1 -> get_key (next c) = Some k2
       ↪  -> lt_key k1 k2 = true.
End CURSOR_TABLE.

(** B+tree specific module *)
Module BT_Table <: CURSOR_TABLE.
 Definition b : nat. Admitted.
 Definition key := Z.

 Definition V := Type.
 Definition table := treelist V.
 Definition cursor := BTrees.cursor V.
 Definition empty_t : table := (BTrees.tl_nil V).

 Definition make_cursor (k: key) (m: table) : cursor := BTrees.make_cursor V
  ↪  k m.
 Definition get_table (c : cursor) : table := BTrees.get_treelist V c.
 Definition get_key (c: cursor) : option key := BTrees.get_key V c.
 Definition get (c: cursor) : option V := BTrees.get V c.
 Definition insert (c: cursor) (k: key) (v: V) : cursor := BTrees.insert V b
  ↪  k v c.
 Definition next (c: cursor) : cursor := BTrees.move_to_next V c.
 Definition prev (c: cursor) : cursor := BTrees.move_to_prev V c.
 Definition first_cursor (m: table) : cursor := BTrees.first_cursor V m.
 Definition last_cursor (m: table) : cursor := BTrees.last_cursor V m.

 Definition abs_rel (m: table) (c: cursor) : Prop := BTrees.abs_rel V m c.
 Definition key_rel (k: key) (c: cursor) : Prop := BTrees.key_rel V k c =
  ↪  true.
 Definition eq_cursor c1 c2 : Prop := cursor_elements V c1 = cursor_elements
  ↪  V c2.
 Definition cursor_correct (c: cursor) : Prop := BTrees.cursor_correct V b
  ↪  c.
 Definition table_correct (t: table) : Prop :=
  BTrees.balanced V t /\ BTrees.fanout V b t /\ BTrees.sorted V t.

 (* table-cursor relations *)
 Theorem make_cursor_rel: forall t k,
   abs_rel t (make_cursor k t).
 Proof. Admitted.
 Theorem get_table_rel: forall t c,
   abs_rel t c <-> get_table c = t.
 Proof. Admitted.
 Theorem first_rel: forall t,
   abs_rel t (first_cursor t).
 Proof. Admitted.
 Theorem last_rel: forall t,
   abs_rel t (last_cursor t).
 Proof. Admitted.
 Theorem next_rel: forall t c,
   abs_rel t c -> abs_rel t (next c).
 Proof. Admitted.
 Theorem prev_rel: forall t c,
   abs_rel t c -> abs_rel t (prev c).
 Proof. Admitted.
 Theorem correct_rel: forall t c,
   abs_rel t c -> (cursor_correct c <-> table_correct t).
 Proof. Admitted.

 (* correctness preservation *)
 Theorem insert_correct: forall k v c,
   cursor_correct c -> key_rel k c -> cursor_correct (insert c k v).
 Proof. Admitted.

 (* get/insert correctness *)
 Theorem glast: forall t,
   get (last_cursor t) = None.
 Proof. Admitted.
 Theorem gis: forall k v c,
   cursor_correct c -> key_rel k c ->
   get (make_cursor k (get_table (insert c k v))) = Some v.
 Proof. Admitted.
 Theorem gio: forall j k v c,
   cursor_correct c -> key_rel k c -> ~ key_rel j c ->
   get (make_cursor j (get_table (insert c k v))) =
   get (make_cursor j (get_table c)).
 Proof. Admitted.

 (* cursor movement *)
 Theorem next_prev: forall c t,
   cursor_correct c -> abs_rel t c -> ~ (c = last_cursor t) -> eq_cursor c
    ↪  (prev (next c)).
 Proof. Admitted.
 Theorem prev_next: forall c t,
   cursor_correct c -> abs_rel t c -> ~ (c = first_cursor t) -> eq_cursor c
    ↪  (next (prev c)).
 Proof. Admitted.
 Theorem cursor_order: forall c k1 k2,
   cursor_correct c -> get_key c = Some k1 -> get_key (next c) = Some k2 ->
    ↪  lt_key k1 k2 = true.
 Proof. Admitted.
End BT_Table.
```

References

[1] R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems*, 6th ed. Addison-Wesley, 2011.

[2] J. Bowen and V. Stavridou, "Safety-critical systems, formal methods and standards," *Software Engineering Journal*, vol. 8, no. 4, pp. 189–209, July 1993.

[3] L. Beringer, A. Petcher, K. Q. Ye, and A. W. Appel, "Verified correctness and security of openssl HMAC," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, 2015, pp. 207–221. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/beringer

[4] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, pp. 107–115, Jul. 2009. [Online]. Available: http://doi.acm.org/10.1145/1538788.1538814

[5] A. W. Appel, *Verified Functional Algorithms*. [Online]. Available: https://softwarefoundations.cis.upenn.edu/

[6] ——, "Efficient verified red-black trees," 2011.

[7] J.-C. Filliâtre and P. Letouzey, "Functors for Proofs and Programs," in *13th European Symposium on Programming, ESOP 2004*, ser. Lecture Notes in Computer Science, D. Schmidt, Ed. Barcelona, Spain: Springer, Feb. 2004, pp. 370–384. [Online]. Available: https://hal.archives-ouvertes.fr/hal-00150913

[8] X. Chen, E. Ruppert, and F. v. Breugel, "Formal verification of a concurrent binary search tree," Ph.D. dissertation, 2013.

[9] Q. Cao, S. Wang, A. Hobor, and A. W. Appel, "Magic wand as frame," 2017.

[10] J. C. Reynolds, "Separation logic: a logic for shared mutable data structures," in *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, 2002, pp. 55–74.

[11] "Verified software toolchain," Sep 2017. [Online]. Available: http://vst.cs.princeton.edu/

[12] A. W. Appel, "Verified software toolchain," in *Programming Languages and Systems*, G. Barthe, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 1–17.

[13] "The science of deep specification." [Online]. Available: https://deepspec.org/page/About/