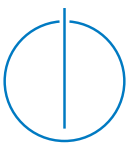
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

A Verified Imperative Implementation of B-Trees

Niels Mündler



DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

A Verified Imperative Implementation of B-Trees

Eine Verifizierte Imperative Implementierung von B-Bäumen

Author:	Niels Mündler
Supervisor:	Prof. Dr. Tobias Nipkow
Advisor:	Dr. Peter Lammich
Submission Date:	February 7, 2021

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, February 7, 2021 Niels Mündler

Acknowledgments

I want to thank my supervisor Tobias Nipkow for introducing me to Isabelle and the world of computer-assisted proofs through his lectures. Further I am very grateful to Dr. Peter Lammich for his patient and thorough introduction into the framework for separation logic proofs in Isabelle.

A very special thank you goes to my parents for their unconditional support, and to my father specifically for sparking and nurturing my interest in the natural science and never ending technical discussions.

Abstract

The B-Tree data structure is a classical data structure that lays the foundation of many modern database systems. Since its first definition by Bayer [BM72] variations have been developed. The original definition and modern variants are examined as well as common variations of the deletion operation, which varies strongly among implementations. We derive our own definitions based on the original specification and specifications that had been verified previously, whether mechanically or with a pen and paper proof. According to this definition, a functional version of the B-Tree is implemented in the Isabelle/HOL framework, an interactive automated theorem prover. The implementation is complemented by a proof of its correctness with respect to refining a set on linearly ordered elements and a proof of the logarithmic relationship between height and number of nodes. This reproduces results of Bayer [BM72] supporting claims on the efficiency of operations on the tree.

From the functional specification an imperative version is derived in Isabelle/HOL using refinement. The main difference to the functional specification is the efficient usage of a heap. The implementation is kept general to support different searching algorithms within nodes. We provide a final version that works with a linear search and one that supports binary search within the node. These imperative programs are shown to again refine the functional specifications using the separation logic utilities from the Isabelle Refinement Framework by Lammich [Lam19]. For this purpose, some additional imperative data structures and operations on variants of arrays are introduced. The result is a proof of the functional correctness of an imperative implementation of the B-Tree data structure that supports set membership and insertion queries and uses efficient binary search for intra-node navigation. The whole proof can be found in the appendix [Mün21].

As with every specification in Imperative/HOL, automatic executable code extraction to several functional programming languages is supported. This approach compares well to other approaches at mechanized verifications of B-Tree implementations considering development time and effort in lines of code.

Contents

1 Introduction

An important topic in computer science is the study of the *functional correctness* of an algorithm. It states whether an algorithm satisfies specified input/output behavior. Proving an algorithm correct becomes especially interesting when a) it can be applied to directly executable code and b) it is machine checked. Unfortunately with both desirable additions, the task becomes significantly more complex. Even in an abstract specification, where topics such as memory management may be abstracted away, machine checked proofs of non-trivial properties are hard. Furthermore for concrete implementations, low-level decisions about memory allocation or the eligibility of reusing variables need to be reasoned about. In this thesis we provide a computer assisted proof in the interactive theorem prover Isabelle/HOL for the functional correctness of an imperative implementation of the B-tree data-structure and present how we dealt with the above mentioned issues.

In Chapter 1, we have a brief overview on related work and introduce common variations of the definition of B-trees. We choose to implement the definition that is most promising for our approach. We first design a functional, abstract implementation. Together with a proof of its functional correctness, it is presented in Chapter 2. On this level, functional correctness means that we show the specifications implement an abstract set interface for linearly ordered element. This interface supports membership queries as well as insertion and deletion operations. From the functional specification, an imperative implementation is derived in Chapter 3. Its functional correctness is shown by proving that it refines the functional specification. Thus the proof obligation for the imperative implementation is reduced to a proof of equivalence between the output of the functional and the imperative implementation. This allows for low-key optimizations with regard to a naive translation. Finally, we present learned lessons, compare the results with related work and suggest potential future research in Chapter 4.

1.1 The Isabelle Proof Assistant

Isabelle/HOL is an interactive theorem prover that allows to reason, among other things, in Higher Order Logic [NPW02]. It is built in ML, which influences the syntax of functional programs written in it. Isabelle source files are divided in so called

theories, corresponding to code modules in common programming languages. A theory consists of the specification of datatypes, typed definitions and theorems with proofs.

1.1.1 Notation and proofs in Isabelle

Functions, predicates and the like are all expressed in a functional manner. The keyword **definition** denotes classical definitions. The term **abbreviation** is used to define simple shorthands for more complex expressions, similar to a macro. The word **fun** precedes recursively defined total functions. It supports definition by pattern matching and is only a valid definition if it incorporates a proof of termination. Usually this proof is derived automatically by the system. If a termination can not be guaranteed for all inputs, a function may be defined via **partial_function**. As circle free-ness of heap pointers cannot be guaranteed for all inputs on imperative programs, many imperative definitions are given as partial functions. Partial function definitions come at the inconvenience of no supported pattern matching [Wen20].

Lemmas, theorems and the like can be expressed in a similar manner as they would be written in mathematical textbook. They begin with **lemma**, **theorem** or **corollary**, followed by an expression or predicate and a proof. Wherever possible and readable, we will try to reflect the actual syntax of the Isabelle system, however compromise in order to keep the notation of obtained lemmata and theorems close to conventional notation. To prove a theorem correct, the system basically provides two different proof styles.

One option is to write structured proofs in the isar language. The user outlines a proof with intermediate goals and tells the system which proof methods to apply to resolve each step. This method is usually preferred as it is more readable and usually faster on the system side. All complex proofs in Chapter 2 are hence written in this style.

In the apply style, the user tells the system which proof method to apply to modify the current goal. Examples are the application of a rule or simplification using equivalences. This is practical if the number of assumptions is high or the goal is large and writing the full terms out would be impractical. Since this is the case for the proofs of the imperative programs, almost all proofs in Chapter 3 are written in this style.

The system provides a number of proof methods, based on different manipulation tools such as logical reasoning or simplification. In this work we will not present the proofs as written in the actual proof files but rather outline the structure of the proofs. When mentioning *automatic* proofs, we mean a proof that comprises very few (≤ 5) apply style invocations of proof methods. A method that commonly allows for such proofs is the *auto* method, a combination of logical reasoning and

repeated simplification, but also allowing automatic case distinctions and destructive rule application.

It is possible to specify functions, predicates and theorems with respect to some abstracted function or constant on which only certain assumptions are made, resulting in so-called **locales**. By providing functions or constants that satisfy the assumptions, we may *instantiate* the locale. We implement the set interface concretely by instantiating a set locale from the standard library.

1.1.2 Examples and basics of the Isabelle language

Datatypes may be defined recursively. The following shows as an example the internal definition of the list datatype.[1]

datatype *'a list = [] | 'a # 'a list*

In natural language this means that either a list is the empty list, or it is an element prepended to another list. As usual in ML like languages, we may apply pattern matching to function argument. In addition, the type $'t$ of any expression e can be made explicit by writing *e :: 't*. Functions that take values of type $'a$ and return values of type $'b$ have type *'a ⇒ 'b*. For functions that take several arguments, currying is applied. The Isabelle internal list catenation function "@" serves as an example for a function definition with explicit type.

fun *(@) :: 'a list ⇒ 'a list ⇒ 'a list* **where**
[] @ ys = ys |
(x#xs) @ ys = x # (xs @ ys)

Further details on notation, proof techniques and more in Isabelle/HOL may be found in Chapter 1 of [NK14].

1.2 The B-Tree Data Structure

B-trees were first proposed by Bayer in [BM72], as a data-structure to efficiently retrieve and manipulate indexed data stored on storage devices with slow memory access. They are n-ary balanced search trees. As such B-trees are generally said to implement a map interface, mapping indices to data and supporting map updates and deletions. They may also be specified as implementing a set interface, where the indices form the actual content of the set. B-trees are a generalization of 234-trees and a specialization of (a,b)-trees.

[1] The actual definition is worded slightly different but this is of no importance here.

A commonly implemented variation is the B^+-tree, where the inner nodes only contain separators to guide the recursive navigation through the tree. In B^+-trees, all data is stored in the leaves [Com79]. Further the leaves are often implemented so as to contain pointers to the next leaf in order, allowing for efficient range queries.

Node are generally thought to be implemented by containing a number of indices, corresponding data and children.

1.2.1 Definitions

Every node contains a list of *keys* (also *separators*, *index elements*), and *subtrees* (*children*), that represent further B-tree nodes. The separators and subtrees may be considered interleaved within a node, such that we can speak of a subtree left of a separator and a subtree right of a separator, where for a separator at index i we mean the subtree in the respective subtree list at index i and $i+1$ respectively. Note that this already implies that the list of subtrees is one longer than the list of separators - we refer to the last subtree as the *last* or *dangling* subtree. In the original definition by Bayer [BM72], a B-tree with above structure must fulfill the three properties *balancedness*, *order* and *sortedness*.

Balancedness *Balancedness* requires that each path from the root to any leaf has the same length h. In other words, the height of all trees in one level of the tree must be equal, where the height is the maximum path length to any leaf. It is only possible to maintain this property due to the flexible amount of subtrees in each node.

Sortedness Further the indices must be *sorted* within the tree which means that all indices stored in the subtree left of a separator are smaller than the value of the separator and all indices on the right are greater. Further all indices within a node should maintain a sorted order, that is the list of separators in a node must be sorted.

Order In general terms, the property of *order* ensures a certain minimum and maximum number of subtrees for each node. However, as pointed out by Folk and Zoellick [FZR92], the property is defined differently in the literature. For the purpose of this work, the original definition by Bayer was chosen as most suitable. A B-tree is of order k, if each internal node has at least $k+1$ subtrees and at most $2k+1$. The root is allowed to have a minimum of 2 subtreess and a maximum of $2k+1$.

An alternative definition proposed by Knuth [Knu98], is to allow between $\lceil \frac{k}{2} \rceil$ and k children. However this involves cumbersome *real* arithmetic that unnecessarily complicates mechanized proofs. Sticking to the original definition is further supported

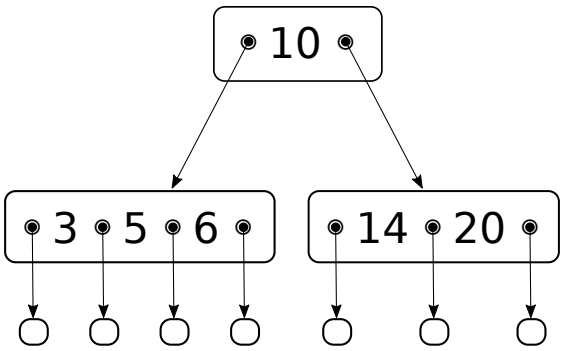


Figure 1.1: A small balanced, sorted B-tree of order 2 and height 2 containing 3 nodes and 6 elements. In subsequent depictions, leafs will be depicted by empty circles.

by the fact that nodes are supposed to fill memory pages which are usually of even size (usually some power of 2). An even number of separators and trees plus one dangling last right tree maximizes the usage of such a page.

The same ambiguity exists for the term *Leaf* which we will define consistently with Knuth's definition [Knu98] to be an empty node, carrying no information, rather than a node that contains data but no children. This is close to the usual approach in functional programming, and will yield more elegant recursive equations for our B-tree operations. The lowest level of nodes hence contains a list of separators and list of pointers to leaves as can be seen in Figure 1.1.

Note that the B-tree definition is only meaningful for positive k. For the case that k is equal to 0, all elements of the tree would have exactly one subtree and no internal elements. For the root node this even leads to a contradictory state: It is required to contain at least 1 element or 2 subtrees. However as it should not have more than $2k + 1 = 1$ elements, this constraint cannot be satisfied. Requiring positive k is consistent with the definitions in the consulted literature [BM72; Com79; Cor+09].

1.2.2 B-Tree operations

The B-tree is a dynamic data-structure and provides a number of operations to inspect or modify the stored data. Generally, the operations are defined recursively on the nodes. The correct subtree may be found by inspecting the separators stored in the currently visited node. If the value that is being searched for is in the range of two adjacent separators and equal to neither of them, we recurse into the tree in between. The obvious corner cases are if the value is less than the minimal element or greater than the maximum element stored. In that case we recurse in either the first or the last

subtree. The exact manner of inspection should in practice of course be efficient and will hence be kept abstract until Chapter 3.

Retrieval Since the whole tree is sorted, checking whether certain elements are contained in the tree is simply conducted by recursing into the correct node in each level. Either the element is found directly or found at a lower level. If we reach a leaf node we know that the element is not contained in the tree. There is little variation on this algorithm so there is no need for comparison.

Insertion There is also much consensus in the literature on how to conduct insertion into a B-tree [Com79]. Generally, an element is inserted into the nodes on the lowest level. In a first step, the element is simply placed at the correct position in the list of separators. If the node had enough space left prior to this operation, we are done. If however the node has more than $2k$ elements after this insertion, we need to split it and, passing the median to the parent node, recurse back upwards. We will see in Section 2.3.4 how this can be elegantly expressed in a functional specification. More detailed descriptions and examples of insertion and deletion may be found in [Cor+09].

Deletion On deletion, elements are removed from the leaves only. If the element to be deleted resides in an inner node, it is replaced by the maximal lesser or minimal greater element in the tree, which always resides in a leaf to the left or right of the element to be removed.

After deletion, the nodes may need rebalancing in order to ensure the order property. A node having less than k elements is called to have *underflow*. Opposing to the insertion function, the exact procedure to handle underflow varies strongly in the literature. Since only one element is removed from the node, the most intuitive remedy to underflow is to *steal* or *borrow* it from the left or right sibling [Cor+09]. If the left or right sibling has more than k elements, one of the neighboring elements may be moved into the current node. Only in case that both siblings have left only k elements, a kind of reversal of the insertion split is conducted: One of the siblings and the node itself, together with the separating element of the parent node are merged and the result split again to form a new, bigger node of valid order.

Following the description of Bayer [BM72], as done by Fielding [Fie80], the two cases can be treated identically. If a node has less than k elements, merge it with one of its siblings. If the resulting catenated node has an overflow again, it will be split in half, just as with insertion related overflows. According to Comer [Com79] this may even be more efficient than stealing single single keys from siblings. First of, the resulting node is less likely to underflow again. If it had stolen only one element, the

probability for another underflow at the next deletion from this node is 1, higher than if several elements are copied over. Further, the node to merge with been completely read from memory at the point of stealing an element. Merging does therefore not incur additional memory accesses and the cost of inter-memory copy of up to k elements is negligible.

1.2.3 Properties

B-trees are assumed to be stored on external memory, such that each node roughly matches a page in main memory. Due to the potentially large branching factor and the balancing, the number of requi memory accesses for retrieving data stays small even for large amounts of data stored. This is due to the fact that the overall number of memory accesses is bounded by the depth of the tree, which again is logarithmic in the number of indices - where the base of the logarithm is closely proportional to the order k of the tree.

Further, by design, the storage usage of B-trees is at a minimum close to 50%, where the average usage is usually higher [BM72]. This is achieved by ensuring that every node reserves the storage of $2k$ keys and separators. By definition, the nodes (except for the root) always contain at least k elements, yielding a storage usage of close to 50%.

B-trees lay the foundation to most modern relational database implementations. The above mentioned properties are key to the widespread popularity of B-trees in databases and are hence preserved in the given implementation.

1.3 Related Work

All found related work considers B^+-trees rather than B-trees, however some parallels can be found.

There exist two pen and paper proofs via the rigorous approach by Fielding [Fie80] and Sexton [ST08]. Even though not machine checked, they shed light on techniques applied in this work.

Fielding approaches the verification by refinement. First, B^+-trees are viewed as nested sets of subtrees or as leafs that are sets of key-value pairs. Obtaining the correct subtree for recursion is in the abstract setting only defined by appropriate pre and post-condition and an actually executable function is provided in the second refinement step. On this level, only arguments on the invariants are made. In a second step B-trees are considered more concretely to contain sorted lists of children and keys. Here the argument for invariant preservation is informally that the refined implementation has the same structure as the abstract implementation. Final imperative PASCAL code is

given, derived by hand imitating the functional style implementations and extended by assertions.

This approach is similar to what will be done in this work, however mostly concerning the methodology rather than the actual steps of refinement. In this work, we will start at the definition using lists in Chapter 2 and refine to an imperative implementation with pointers and arrays in Chapter 3. The code we obtain will be exported automatically to a functional language of choice.

We reason about the validity of this refinement via separation logic, the same tool as employed by Sexton [ST08]. They show in their work that this tool allows to reason based on some kind of locality, in particular that operations on subtrees are really only affecting subtrees. We will make implicit use of this property in Chapter 3 for proofs on our imperative implementation.

The specification by Sexton is given as an abstract machine rules, somewhat more abstract than pure code but operating on a stack much like we expect imperative programs to operate with pointers on a heap. We therefore categorize this as a direct verification of an imperative implementation.

Among imperative implementations, two machine checked proofs exist too. In the work of Ernst [ESR15], an imperative implementation is directly verified by combining interactive theorem proving with shape analysis. The main recursive procedures are interactively verified in KIV. Data structure properties such as circle-freeness are then proven by shape-analysis. Another direct proof on an imperative specification was conducted by Malecha [Mal+10], with the YNOT extension to the interactive theorem prover Coq. Both works use recursively defined "shape" predicates that describe how the nodes and pointers describe an abstract tree of finite height. However, we know of no verification that explicitly covers the implementation of functional B-trees. In addition to providing one, this work aims to show the benefits of taking this indirection for imperative implementations. That is that the analysis of invariants of an abstract shape may be spared or at least significantly simplified when restricting the concrete pointer structures to be refinements of an abstract algebraic type.

2 Functional B-trees in Isabelle

Proving higher level properties of data structures tends to be easier on a functional level than on an imperative level. This is mostly due to the fact that many details of implementation can be abstracted away or expressed in a simpler manner. The work therefore begins with a functional specification of B-trees that is not aware of the existence of heaps and uses persistent data structures.

2.1 Basic Definitions

As discussed in Chapter 1, we define B-trees recursively to either be a *Node*, comprising a list of subtrees and keys or to be a *Leaf* that contains no information. The only room for interpretation is how to actually store the subtrees and separators. For trees of variable but constantly bounded size, an explicit constructor for each possible size may be given. An example may be found for the 234-trees analysed by Nipkow[Nip16], where the number of subtrees is maximally 5. However in B-trees, the number of subtrees is bounded only by k, which is neither constant nor bounded by a constant.

One possible way out is the choice to make nodes extensible by their definition. That is, a node is does either have one subtree, one element and a continuation of the node or it is the last entry and has only one subtree. This approach was implemented by Ernst [ESR15]. However the approach yields a number of downsides. First, all operations on the tree were defined co-recursively, which adds a layer of complexity to the project. Further, general lemmas for lists or similar datatypes can not be used directly for this structure and would need to be transferred manually or derived newly.

Therefore we use an intermediate data structure to store the subtrees. This approach is called nested recursive definition and will become clearer in the actual code. The intuitive idea would be to choose sets as an abstract datatype to hold subtrees and keys, as was the first step done by Fielding [Fie80]. However here we are limited as recursive nested definitions make only sense for finitely bounded subtypes.[1] Sets themselves are potentially of infinite cardinality and hence not bounded and working with finite sets introduces additional complexity that we want to circumvent in the abstract definition.

[1] More information on this topic and potential remedies may be found in the tutorial to datatype definition in Isabelle [Bie+20].

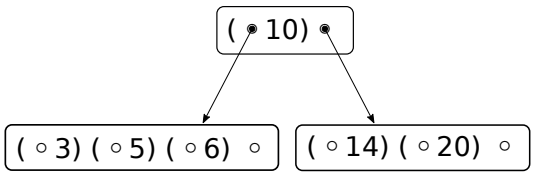


Figure 2.1: The B-tree from figure Figure 1.1 as concretely specified. Subtrees and separators that lie in pairs in the implementation are surrounded by brackets.

Therefore we resort to the list datatype, defining each node to hold a list of key-node pairs. By storing subtrees and keys next to each other, properties that relate subtrees to their immediately neighboring separators (i.e. sortedness) can be defined locally on each pair in the list. This also implicitly enforces the relationship of the number of subtrees and separators and makes the explicit use of length and index relationships unnecessary.

A similar approach was chosen by Malecha [Mal+10]. There, the B-tree datatype was defined directly on the heap and comprised the height of the tree, a pointer to the last subtree and an array of fixed size that contains pairs of value and pointers. We will store subtrees and keys in pairs too, however with a different in-pair order, such that the left subtree of a key is the left element in a tuple and the key is the right element. Moreover in a functional context, we will not make use of arrays but rather resort to lists. In [Mal+10] the height attribute was used to inductively define a shape predicate that is only valid if the height decreases for subtrees. However, the finite size of the abstract tree is an implicit result of its recursive definition. We therefore also do not explicitly store the height of the tree. A visualization of the final definition may be seen in Figure 2.1.

datatype *'a btree* = Leaf | Node *(('a btree * 'a) list) ('a btree)*

As said, this definition makes it easy to relate any separator to the subtree on the left, as this is subtree in the same tuple. If we need access to the subtree on the right, we may match the remaining list the list constructor and take the subtree of the first tuple. In case the remaining list is empty, we need to choose the last tree.

The balancedness of a B-tree is defined recursively, making use of an intuitively defined height.[2] The height of all subtrees must be equal. However it is not completely intuitive what would be the best way to express this property. Either we fix the height of all trees to some existentially quantified number or to a known value. Since we know the last tree in the node exists and further know that if all subtrees of the node have

[2]The operator ' denotes the image of all elements of a set for a given function. In this case we map all subtrees to their respective height.

the same height, then their height is equal to the last tree, we simply choose the height of the last tree as an anchor point.

```
fun height :: 'a btree ⇒ nat where
  height Leaf = 0 |
  height (Node ts t) = 1 + Max (height ' (set (subtrees ts@[t]))

fun bal :: 'a btree ⇒ bool where
    bal Leaf = True |
    bal (Node ts t) = (
      (∀sub ∈ set (subtrees ts). height sub = height t) ∧
      (∀sub ∈ set (subtrees ts). bal sub) ∧ bal t
    )
```

Another option, demonstrated by Fielding, is to define balancedness as defined by Bayer to be that all paths from the root to the node are of equal length, or that the set of lengths of all paths be of size 1. However reasoning on set cardinality is likely to be more difficult in a mechanized proof than on paper and the definition for the mechanized proof by Ernst resembles ours more closely, so we did not experiment much with this option.

Further the order of the trees needs to be formally defined. As discussed in Section 1.2.1, the most useful choice here is to allow for at least k and at most $2k + 1$ subtrees. Since the last subtree is fixed, and subtrees and children are residing as pairs in the same list, we simply require the length of that list to be between k and $2k$. Note that we also need a special property $order^r$ for the root of the tree, which has between one and $2k$ elements. In mathematical equations, we will denote "order k t" as "$\text{order}_k\ t$" for convenience (likewise for "$\text{order}^r\ k\ t$"), which is to be read as "tree t is of order k".

```
fun order :: nat ⇒ 'a btree ⇒ bool where
    order k Leaf = True |
    order k (Node ts t) = (
    (length ts ≥ k) ∧
    (length ts ≤ 2*k) ∧
    (∀sub ∈ set (subtrees ts). order k sub) ∧ order k t
  )

fun order^r where
  order^r k Leaf = True |
  order^r k (Node ts t) = (
```

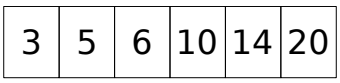


Figure 2.2: The inorder view on the B-tree from figure Figure 2.1.

```
  (length ts > 0) ∧
  (length ts ≤ 2*k) ∧
  (∀s ∈ set (subtrees ts). order k s) ∧ order k t
)
```

We define the sortedness of a B-tree based on the *inorder* of the tree, which is the catenation of all elements in the tree in in-order traversal. An example can be seen in Figure 2.2. The library function *concat* on lists of lists is employed here to express catenation of the inorder of subtrees and the separators. This definition reads a bit inconveniently as the node internal list is first mapped and then concatenated. However since we make recursive use of the inorder-function inside the mapping expression, mapping the subtree list is inconvenient to externalize. We only later cover up this expression by using abbreviations. In the following, we will not distinguish between lists, pairs or trees when talking about their inorder representation. The function with the correct type from the below listing is meant instead.

```
fun inorder :: 'a btree ⇒ 'a list where
    inorder Leaf = [] |
    inorder (Node ts t) =
        concat (map (λ (sub, sep). inorder sub @ [sep]) ts) @ inorder t

abbreviation inorder_pair ≡ λ(sub,sep). inorder sub @ [sep]
abbreviation inorder_list ts ≡ concat (map inorder_pair ts)
```

That way we can express sortedness of the tree t as a simple $\text{sorted}(\text{inorder}\, t)$, where *sorted* is the property of being sorted strictly (with respect to $<$) in ascending order. This definition is very compact and brings for another benefit pointed out by Nipkow [Nip16]: Many properties of search trees follow intuitively by considering the inorder view on the tree. Often, it is supposed to be invariant (i.e. stealing from the right neighbor node) or only deviates in a manner we expect it to deviate given the inorder view (i.e. insert an element at the correct position). We will see later how the use of this fact comes in handy for proving important properties of the implementation.

Since our B-tree definition really only makes sense for positive k, we obtain the following overall invariant for B-trees.

Definition 2.1.1 $k > 0 \Longrightarrow \text{btree}_k\, t = \text{bal}\, t \wedge \text{order}_k^r\, t \wedge \text{sorted}(\text{inorder}\, t)$

All trees that satisfy this invariant have a very small height considering the number of inserted elements. This fact is examined closer in the following section.

2.2 Height of B-Trees

As pointed out by Bayer [BM72], the height of B-trees is logarithmic with respect to the number of nodes of the tree. The original paper even gives a precise lower and upper bound, which will be quickly sketched in the following.

First, we define the number of nodes in a tree:

```
fun nodes :: 'a btree ⇒ nat where
  nodes Leaf = 0 |
  nodes (Node ts t) =
    1 + ∑t←subtrees ts. nodes t + nodes t
```

We obtain bounds on the number of nodes of an subtree with respect to its height by induction on the computation of the nodes function.

Lemma 2.2.1 $\text{order}_k\, t \wedge \text{bal}\, t \longrightarrow$

$$(k+1)^{\text{height}\, t} - 1 \leq \text{nodes}\, t * k \tag{2.1}$$

$$\text{nodes}\, t * 2k \leq (2k+1)^{\text{height}\, t} - 1 \tag{2.2}$$

From Lemma 2.2.1 we can almost directly obtain the bounds on valid roots of B-trees. The only difference to the bound of internal nodes occurs on the lower bound side. The issue here is that a root node may contain less elements than a valid internal node (namely only one), which yields two subtrees with known height plus one for the node itself. Note that these are the exact same bounds as obtained by Bayer [BM72], except for the fact that we have generalized the equation, incorporating whether t is a tree or not.[3]

Theorem 2.2.1 $\text{order}^{\text{r}}{}_k\, t \wedge \text{bal}\, t \wedge k > 0 \longrightarrow$

$$2((k+1)^{\text{height}\, t-1} - 1) \operatorname{div} k + (t \neq \mathit{Leaf}) \leq \text{nodes}\, t \leq ((2k+1)^{\text{height}\, t} - 1) \operatorname{div} 2k \tag{2.3}$$

These results are very interesting, because the runtime of all further operations will more or less trivially be directly proportional to the height of the tree. Therefore, we are glad to see that the height is logarithmic with respect to the number of nodes stored in the tree.

[3] If t is a *Leaf*, $2((k+1)^{\text{height}\, t-1} - 1) \operatorname{div} k + (t \neq \mathit{Leaf})$ becomes a fancy way of writing 0.

These bounds are sharp. We prove this by providing in Listing 2.1. functions that generate exactly those trees that satisfy the requirements of B-trees, have a given height and satisfy the inequality from Theorem 2.2.1 with equality. As might be expected, these trees are simply those trees that are minimally or maximally filled in each node with respect to the order property.

The proof for internal nodes follows by induction over the creation.

Lemma 2.2.2 $t_f := \text{full\_node}\, k\ a\ h \wedge t_s := \text{slim\_node}\, k\ a\ h \longrightarrow$

$$h = \text{height}\, t_s = \text{height}\, t_f \wedge \tag{2.4}$$

$$((2k+1)^h - 1) = \text{nodes}\, t_f * (2k) \qquad \wedge \text{order}_k\, t_f \wedge \text{bal}\, t_f \tag{2.5}$$

$$((k+1)^h - 1) = \text{nodes}\, t_s * k \qquad \wedge \text{order}_k\, t_s \wedge \text{bal}\, t_s \tag{2.6}$$

The rule for the roots follows directly, making use of the lemma for the internal nodes. Note how for the root node the result is simply two times the value for trees of one height less, which are the minimally required two subtrees.

Theorem 2.2.2 $k > 0 \wedge t_f = \text{full\_tree}\, k\ a\ h \wedge t_s = \text{slim\_tree}\, k\ a\ h \longrightarrow$

$$\begin{aligned} h = \text{height}\, t_s = \text{height}\, t_f \qquad & \wedge \\ ((2k+1)^h - 1)\ \text{div}\ 2k = \text{nodes}\, t_f \qquad & \wedge \text{order}^{\text{r}}{}_k\, t_f \wedge \text{bal}\, t_f \\ 2((k+1)^{h-1} - 1)\ \text{div}\ k + (t_s \neq \mathit{Leaf}) = \text{nodes}\, t_s \qquad & \wedge \text{order}^{\text{r}}{}_k\, t_s \wedge \text{bal}\, t_s \end{aligned}$$

2.3 Set operations

We say that B-trees store linearly ordered values that could be stored in an abstract set, and the that a B-tree is a concretization of this set. With this specification of B-trees we need to provide functions that allow set-like operations on the trees - membership queries, insertion and deletion. In the Isabelle/HOL framework, there is a standard interface for data structures that provide an implementation of sets.

2.3.1 The Set Interface

As described by Nipkow [Nip16], an implementation *'a t* of sets of elements of type *'a* is required to provide the following operations:

- *empty :: 'a t* **where** *set empty* = ∅
- *isin :: 'a t ⇒ 'a ⇒ bool* **where** *inv t ⟹ isin x t* = $x \in t$

Listing 2.1: The functions generating trees with minimal size and maximal size for given height.

```
fun full_node where
  full_node k c 0 = Leaf |
  full_node k c (Suc n) = (
       Node
         (replicate (2*k) ((full_node k c n),c))
         (full_node k c n)
  )

fun slim_node where
  slim_node k c 0 = Leaf |
  slim_node k c (Suc n) = (
       Node
         (replicate k ((slim_node k c n),c))
         (slim_node k c n)
    )

definition full_tree = full_node

fun slim_tree where
  slim_tree k c 0 = Leaf |
  slim_tree k c (Suc h) =
    Node
        [(slim_node k c h, c)]
        (slim_node k c h)
```

- *insert :: 'a ⇒ 'a t ⇒ 'a t* **where** *inv t ⟹ set (insert t x) = set t ∪ x*
- *delete :: 'a ⇒ 'a t ⇒ 'a t* **where** *inv t ⟹ set (delete t x) = set t \ x*

For some abstraction function *set :: 'a t ⇒ 'a set*. Additionally, an invariant *inv :: 'a t ⇒ bool* has to be supplied. For this work, using the definition from Section 2.1, we consider *'a t = 'a btree*. The standard approach is to provide functions on B-tree and show that they have the same effect as membership test, insertion and deletion in abstract sets with the same elements (specified above after "where"). Further, the invariant needs to remain valid for the results of the operations as well as the *empty* element (i.e. $inv\ t \Longrightarrow inv\ (insert\ x\ t)$).

We know from Section 2.1 that one of the invariants of the tree is that it is always sorted. While this is expressed in a somewhat cumbersome manner in the original definition by Bayer [BM72], this requirement is nothing else but a sortedness of the inorder view of the tree. Knowing this, we resort to a specialized set interface proposed by Nipkow [Nip16]. His approach yielded automatic proofs for 2-3-trees and 234-trees, which are specializations of B-trees. The modified Set interface reasons based on the *inorder* view on the tree instead of the *set* abstraction.

- *empty :: 'a t* **where** *inorder empty = []*
- *isin :: 'a t ⇒ 'a ⇒ bool* **where**
  inv t ∧ sorted(inorder t) ⟹ isin x t = x ∈ set (inorder t)
- *insert :: 'a ⇒ 'a t ⇒ 'a t* **where**
  inv t ∧ sorted(inorder t) ⟹ inorder (insert t x) = ins_{list} (inorder t) x
- *delete :: 'a ⇒ 'a t ⇒ 'a t* **where**
  inv t ∧ sorted(inorder t) ⟹ inorder (delete t x) = del_{list} (inorder t) x

The invariant and inorder functions for this set interface follow directly from the definition of B-trees. Hence, we specify $k > 0 \wedge \text{order}_k^r\, t \wedge \text{bal}\, t$ as the invariant of B-trees and use the concatentation function from Section 2.1 to specify the inorder abstraction *inorder :: 'a t ⇒ 'a list*. Note that sortedness is not part of the invariant but will be required for all operations and preserved due to the definition of ins_{list} and del_{list}. The interface is then implemented by providing set operations and the proofs of invariant preservation with respect to this inorder abstraction.

Some parts of the set specification are trivial. In the following, *Leaf* is an empty tree and hence represents the empty set, satisfying the first part of the specification. It can be seen directly that the leafs already satisfy all three properties of the invariant. The following sections will describe the implementation and proofs of the non-trivial set operations.

```
fun linear_split_help where
  linear_split_help [] x prev = (prev, []) |
  linear_split_help ((sub, sep)#xs) x prev = (
      if sep < x then
        linear_split_help xs x (prev @ [(sub, sep)])
      else
        (prev, (sub,sep)#xs)
  )

fun linear_split:: ('a btree×'a) list ⇒ 'a ⇒ (_ list × _ list) where
  linear_split xs x = linear_split_help xs x []
```

Lemma 2.3.1

$$\text{linear\_split}\ xs\ x = (\text{takeWhile}(\lambda(\_, s).\ s < x)\ xs,\ \text{dropWhile}(\lambda(\_, s).\ s < x))$$

Figure 2.3: An implementation of the abstract split function specifications. This functions scans linearly through the list, returning the first tuple where the separator or subtree could potentially contain the value x. As discovered by Peter Lammich, it is similar to a combination of library functions, simplifying proofs on its properties.

2.3.2 The split-Function

Naturally, the set operations are defined recursively on the nodes of the tree. Since each node contains a number of elements that is not bounded by a constant, a generalized function to navigate to the correct separator and subtree is central to all operations.[4]

We call this function *split*-function. It determines the "correct" position in the list of separators and subtrees for recursion. At this position, the range spanned by the left subtree and the separator is exactly the range in which the desired element must be contained if it is contained in the tree. Hence, either the separator is equal to the desired value or we need to recurse into the left subtree. The approach of generalizing the tree navigation has surprisingly little popularity in the implementations examined. Notably, it was implemented in the work of Malecha and Fielding, but not described in detail [Mal+10; Fie80].[5] Usually however, it is integrated into the set-operation by a

[4] Opposing to that, in 234-trees determining the correct subtree and separator could be hard coded for every case, as done by Nipkow [Nip16].

[5] In Fieldings approach the corresponding function is called *index* and the final implementation is a linear search. Malecha calls it *findSubtree* and looses no more words about it.

linear search that is promised to be replaced by a more efficient binary search in the actual implementation [Cor+09; BM72] or left in the final code [ESR15].

The precise inner workings of the split function are not of interest here and actually are not supposed to be interesting on the functional level. Of course we need to know that *some* kind of function exists that correctly splits the key-value list. An example is given in Figure 2.3. In the process of implementing the set specifications, this concrete function was used to explore the provability of the set methods. However it quickly turned out that 1) only specific lemmas about the split function are useful during proofs and 2) only relying on an abstract specification of the split method would simplify integrating alternative splitting functions. Most notably, the abstraction allows to later plug in an efficient splitting function, e.g. based on binary search.

Therefore all set functions are defined based on an abstract function *split* that fulfils the following requirements:

- $\text{split}\, xs\ p = (ls, rs) \Longrightarrow xs = ls @ rs$
- $\text{split}\, xs\ p = (ls @ [(sub, sep)], rs) \wedge \text{sorted}(\text{separators}\, xs) \Longrightarrow sep < p$
- $\text{split}\, xs\ p = (ls, (sub, sep) \# rs) \wedge \text{sorted}(\text{separators}\, xs) \Longrightarrow p \leq sep$

Described in natural language, the split function should return two lists that concatenate to the original list. Further if the elements came in sorted order, the list is split such that the key to the left is strictly less than the desired value and the key to the right should be less or equal.

Note how the split function really only needs to consider the separators and not the subtrees themselves. By requiring only sortedness of the separators, we weaken the requirements to fulfil the split abstraction. However these weakened assumptions are sufficient to guarantee functional correctness. Sortedness of the whole tree is part of the established invariant of B-trees. If the whole tree is sorted, the separators are sorted too. We prove this once and from it follows that we may use the split function for sorted trees and obtain the desired results.

2.3.3 Membership tests

The simplest operation required in the set interface is the *isin* function. It should return the same value for a B-tree as a membership queries on the set abstraction of the tree. The definition in Listing 2.2 is straightforward and also shows example usage of the split function. In case the right part of the split list is non-empty, we check the element

Listing 2.2: The *isin* function

```
fun isin :: 'a btree ⇒ 'a ⇒ bool where
    isin (Leaf) x = False |
    isin (Node ts t) x = (
        case split ts x of (_,(sub,sep)#rs) ⇒ (
            if x = sep then
                True
            else
                isin sub x
        ) | (_,[]) ⇒
            isin t x
    )
```

at its head and recurse in the given subtree if necessary. Otherwise, we may directly recurse to the last tree in the node.[6]

By the standard set interface the operation is only required to work on sorted, balanced trees of a certain order, however only the first property is actually required for correctness. The following lemma shows the required property of the function

Theorem 2.3.1 $\text{sorted}(\text{inorder}\, t) \Longrightarrow \text{isin}\, t\ x = x \in \text{set}(\text{inorder}\, t)$

It follows by induction on the evaluation of the isin function. To prove it, we invoke two specialized lemmas, that simplify arguments about the choice of the node for recursion. The lemmas specialize an idea proposed by Nipkow [Nip16] and similar lemmas will be used for the correctness proofs of the *insert* and *delete* function.

Lemma 2.3.2 $\text{sorted}(\text{inorder}(\mathit{Node}\ ts\ t) \wedge \text{split}\, ts\ x = (ls, rs) \Longrightarrow$

$$x \in \text{set}(\text{inorder}(\mathit{Node}\ ts\ t)) = x \in \text{set}(\text{inorder}\, rs @ \text{inorder}\, t)$$

The idea of this fact is to argue that, if the split function has provided us with a given splitting, it is safe to limit the further search to the right side of the split. It follows directly from the requirements on the split function. If *rs* is empty, we can follow that the element has to reside in the inorder of the last tree of the node. When used in the

[6] This function recurses on results of the split function. In order to show that this function terminates, we need to show the system that the obtained subtrees are of smaller size than the current tree. Adding this fact to the default termination simplification set resolves the issue for all coming functions.

inductive proof of Theorem 2.3.1, we can then deduce that this is equal to *isin t x*, the branch taken in the isin-function (see Listing 2.2) for an empty right split result. If *rs* is not empty, we need an additional lemma.

Lemma 2.3.3 $\text{sorted}(\text{inorder}(\mathit{Node}\ ts\ t)) \wedge \text{split}\, ts\ x = (ls, (sub, sep)\#rs) \wedge sep \neq x \Longrightarrow$

$$x \in \text{set}(\text{inorder}((sub, sep)\#rs)@\,\text{inorder}\, t) = x \in \text{set}(\text{inorder}\, sub)$$

With this lemma we know that the first subtree on the right part of the split result is the correct child to recurse into. The requirement of $sep \neq x$ is because if $sep = x$, no recursion is required at all. The desired element was just found.

This operation has no effect on the tree, it only walks through it. This fact follows directly due to the persistence of functional data structures. Hence, no proof of invariant preservation is required, in contrast to the following tree-modifying operations.

2.3.4 Insertion

The implementation of the insertion function as described in Section 1.2.2 is documented in Listing 2.3. Rather than manually checking whether the lowest level was reached, we recurse until we reach a leaf. Inserting into it is defined to cause an overflow that will be handled by the lowest internal nodes in the same manner as for other internal nodes.

The *ins* function is a recursive helper function, that recurses down into the correct leaf node for insertion. As a possible intermediate result of insertion may be two trees that are the result of an obligatory split, it returns data in a new datatype called up_i. This datatype carries either a singleton valid B-tree or two trees and a separator that could correctly be part of a node when placed next to each other. The original subtree is replaced by the result of the recursive call. If an overflow occured in the lower node, the additional element and subtree are inserted into the current node list after the element that flowed over. The resulting list may have an overflow itself.

To modularize insertion, the check for the overflow of the nodes list has been extracted to the function $node_i$. In case the node that was passed in violates the order invariant, it performs the splitting and returns two trees and the additional separator. This construction is useful for handling overflow in a functional context and appears similarly in the verified functional 234-tree implementation by Nipkow [Nip16].

Finally, the *insert* function calls the *ins* helper function and transforms the result into a valid B-tree by allocating a new root if necessary.

To fulfil the set interface requirements, we need to show that this function preserves the invariants and acts the same as inserting element x into the inorder list of the

Listing 2.3: The *insert* function

```
datatype 'b up_i = T_i 'b btree | Up_i 'b btree 'b 'b btree

fun split_half where
  split_half xs = (take (length xs div 2) xs, drop (length xs div 2) xs)

fun node_i :: nat ⇒ ('a btree × 'a) list ⇒ 'a btree ⇒ 'a up_i where
  node_i k ts t = (
  if length ts ≤ 2*k then T_i (Node ts t)
  else (
    case split_half ts of (ls, (sub,sep)#rs) ⇒
      Up_i (Node ls sub) sep (Node rs t)
    )
  )

fun ins :: nat ⇒ 'a ⇒ 'a btree ⇒ 'a up_i where
  ins k x Leaf = (Up_i Leaf x Leaf) |
  ins k x (Node ts t) = (
  case split ts x of
    (ls,(sub,sep)#rs) ⇒
      (if sep = x then T_i (Node ts t)
      else
        (case ins k x sub of
          Up_i l a r ⇒
            node_i k (ls @ (l,a)#(r,sep)#rs) t |
          T_i a ⇒ T_i (Node (ls @ (a,sep) # rs) t))) |
    (ls, []) ⇒
      (case ins k x t of
          Up_i l a r ⇒
            node_i k (ls@[(l,a)]) r |
          T_i a ⇒ T_i (Node ls a)
  )
)

fun tree_i :: 'a up_i ⇒ 'a btree where
  tree_i (T_i sub) = sub |
  tree_i (Up_i l a r) = (Node [(l,a)] r)

fun insert::nat ⇒ 'a ⇒ 'a btree ⇒ 'a btree where
  insert k x t = tree_i (ins k x t)
```

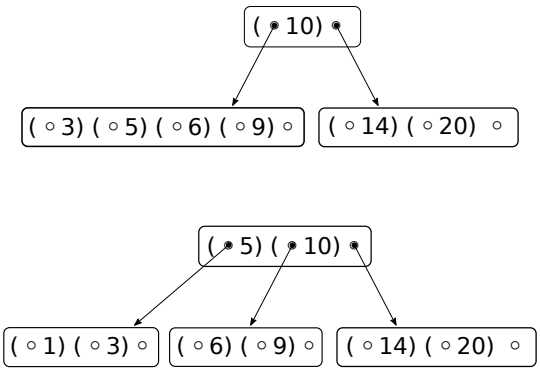


Figure 2.4: The tree from Figure 2.1 after successive insertion of 9 (top) and 1 (bottom).

tree. Note that for these proofs, a separate notion of height, balancedness, order and inorder had to be introduced for the up_i datatype. Height, balancedness and inorder are defined such that they are invariant to splitting and merging of nodes. For example, "height $ts\ t = \text{height}^{up_i}(\text{node}_i\ k\ ts\ t)$". The modified order function simply subsumes that all subtrees have the given order. Naturally, making a one-element tree of an up_i element will thus give a node of root-order, while inserting the elements will not violate the recursive order requirement.

Proving that balancedness is invariant under insertion requires an additional lemma, namely that the height is also invariant under insertion. Technically, height preservation is only required when modifying balanced trees, however we have shown the stronger statement that this is even the case for non-balanced trees. This follows by induction using the associativity and commutativity of the maximum function. With height invariance, the proof for balancedness follows much the same way. Overall, the fact that the operation does preserve balancedness and height should come at no surprise to the reader. After all, the operations are never directly affecting the height of any tree, and all trees generated by splitting a node comprise trees that had been there before, now simply distributed along to two nodes in the same level.

Lemma 2.3.4

$$\begin{aligned} \text{height}\, t &= \text{height}^{up_i}(\text{ins}_k\ x\ t) \\ \text{bal}\, t &\Longrightarrow \text{bal}^{up_i}(\text{ins}_k\ x\ t) \end{aligned}$$

The order invariant is the second property that needs to be shown to stay invariant. Since the central function to ensure order is the node_i function, we first show the following lemma:

Lemma 2.3.5 *If all children have order k and* length $ts \geq k$ *and* length $ts \leq 4k+1$ *then all trees in* $\text{node}_i\ k\ ts\ t$ *have order k.*

Even though in the case of insertion the length of the list never exceeds $2k+1$, the statement holds up to $4k+1$. The statement is proven by case distinction whether "length $ts \leq 2k$". Looking at the definition in Listing 2.3 we see that if this is the case nothing happens and the lemma is trivial. In case "length $ts > 2k$", a split occurs. The median element is passed up as a seperator, leaving $\lfloor \frac{\text{length } ts}{2} \rfloor$ and $\lceil \frac{\text{length } ts}{2} \rceil - 1$ elements for the left and right node respectively. Given the constraint on the maximum length of ts this will always be below or equal $2k$. Further, as we require a size of at least $2k+1$ elements for a split to occur, the resulting nodes each have at least k elements. This argument hardly changes comparing the upper bounds $2k+1$ and $4k+1$, but with the weaker upper bound $node_i$ can be used for merging nodes in the deletion function as well.

Using the above, the order invariant for *ins* follows quite directly by induction.

Lemma 2.3.6 $\text{order}_k\ t \Longrightarrow \text{order}_k^{up_i}(\text{ins}_k\ x\ t)$

Since the invariant on B-trees only requires root order of k on the tree, we had to additionally derive versions of these invariants for this weaker order. They are straightforward however as all trees in the inductive case have order k, for which preservation was just proven. Putting things together, we obtain preservation of the invariants.

Theorem 2.3.2 $\text{order}_k^r\ t \wedge \text{bal}\ t \Longrightarrow \text{order}_k^r(\text{insert}_k\ x\ t) \wedge \text{bal}(\text{insert}_k\ xt)$

The set interface further requires that the insertion returns a tree that has the same inorder view as the original inorder with the element inserted at the correct position.

Looking at the *ins* function in Listing 2.3, we see that, in an inductive proof, the main obligation is to argue for the choice of the subtree for recursion. Similar to the proof of Theorem 2.3.1, we use two auxiliary lemmas, that turn the remaining proof in simple case distinctions and chains of equations.

Lemma 2.3.7 $\text{sorted}(\text{inorder}(Node\ ts\ t)) \wedge \text{split}\ ts\ x = (ls, rs) \Longrightarrow$

$$\text{ins}_{\text{list}}\ x\ (\text{inorder}(Node\ ts\ t))) = \text{inorder}\ ls\ @\ \text{ins}_{\text{list}}\ x\ (\text{inorder}\ rs\ @\ \text{inorder}\ t)$$

Lemma 2.3.8 $\text{sorted}(\text{inorder}(Node\ ts\ t)) \wedge \text{split}\ ts\ x = (ls, (sub, sep)\#rs) \wedge sep \neq x \Longrightarrow$

$$\text{ins}_{\text{list}}\ x\ (\text{inorder}((sub, sep)\#rs)\ @\ \text{inorder}\ t)) =$$
$$(\text{ins}_{\text{list}}\ x\ (\text{inorder}\ sub))\ @\ sep\#\text{inorder}\ rs\ @\ \text{inorder}\ t$$

The only case not covered by the above lemmas is the case $sep = x$. In that case, the tree does not change. This is due to the fact that x is already in the set represented by the tree and hence does not need to be inserted. The same happens when inserting x into a list that already contains x (i.e. the inorder of the tree), which follows simply by induction on the list.

Lemma 2.3.9 $\text{sorted}\ xs \wedge x \in \text{set}\ xs \Longrightarrow \text{ins}_{\text{list}}\ x\ xs = xs$

The above lemmas are sufficient to show correctness of the *ins* function inductively. The theorem for *insert* follows again automatically, which concludes the proof of the insertion part.

Theorem 2.3.3 $\text{sorted}(\text{inorder}\ t) \Longrightarrow$

$$\text{inorder}(\text{insert}_k\ x\ t) = \text{ins}_{list}\ x\ (\text{inorder}\ t)$$

2.3.5 Deletion

The procedures described in Section 1.2.2 are implemented here with a certain flavor due to the setup of B-trees. The *rebalancing* as implemented in Listing 2.4 comprises merging and splitting neighboring nodes. For simplicity of the function and proofs, we always merge with the right sibling. The only exception, which is unavoidable due to the asymmetry of the data structure, is an underflow in the last subtree. It will be merged with the second-to-last subtree, the last tree in the variable length list of the node. Note also how the tree does not change when asked to rebalance a leaf. The reason is that rebalancing leafs may only happen if it is preceded by deletion from a leaf - which does not change the tree at all.

The other detail is regarding the *split_max* function employed when deleting elements from inner nodes. It is defined in Listing 2.5. Here some freedom exists whether to swap with the maximum lesser or the minimum greater element in the tree. In our setup, the last tree of the node is explicitly stored in each node and the left subtree of a separator lies within the same pair inside the node list. Using pattern matching, it is hence significantly easier to obtain the maximum of the left subtree, than than to obtain the minimum of the right subtree. Therefore we will always swap with the former, maximal lesser element.

The most interesting property for the rebalancing operations is the order property, which is meant to be restored after underflows. It is important to note here that we

Listing 2.4: The rebalancing functions

```
fun rebalance_middle_tree where
  rebalance_middle_tree k ls Leaf sep rs Leaf = (Node (ls@(Leaf,sep)#rs) Leaf) |
  rebalance_middle_tree k ls (Node mts mt) sep rs (Node tts tt) = (
    if length mts ≥ k ∧ length tts ≥ k then
        Node (ls@(Node mts mt,sep)#rs) (Node tts tt)
    else (
        case rs of [] ⇒ (
            case node_i k (mts@(mt,sep)#tts) tt of
                T_i u ⇒ Node ls u |
                Up_i l a r ⇒ Node (ls@[(l,a)]) r
            ) |
        (Node rts rt,rsep)#rs ⇒ (
            case node_i k (mts@(mt,sep)#rts) rt of
                T_i u ⇒
                    Node (ls@(u,rsep)#rs) (Node tts tt) |
                Up_i l a r ⇒
                    Node (ls@(l,a)#(r,rsep)#rs) (Node tts tt)
            )
        )
    )

fun rebalance_last_tree where
    rebalance_last_tree k ts t = (
        case last ts of (sub,sep) ⇒
            rebalance_middle_tree k (butlast ts) sub sep [] t
    )
```

cannot guarantee that at least one element remains in the node that is passed back upwards. This violates even the weakest order formulation up until now, the root order invariant. Hence before making the result of rebalancing a final tree, we need to recover root order. We can guarantee that all subtrees of the result are of order k and that there are no more than $2k$ elements in the result, which is sufficient. We say that if a tree t has at most $2k$ elements and all subtrees of t have order k, then t has *almost order k* ($\text{order}_k^a\ t$). Using this new definition we formulate the invariant for the rebalancing operation.

Lemma 2.3.10 *If all subtrees in ls, rs are of order k and of sub and t, only one may have almost order k and the other has order k, and all subtrees have equal height, then*

$$\text{order}_k^a(\text{rebalance\_middle\_tree}_k\ ls\ sub\ sep\ sep\ rs\ t)$$

It might be suprising that we require all subtrees to have equal height for this operation. However, a close look at Listing 2.4 may clarify this requirement. In order to successfully rebalance an internal node, we need neighboring nodes that are not leafs. As this fact is already expressed in the pattern matching of the function, it is simply *undefined* if not all subtrees have equal height. But we know that B-trees are always balanced, so we may assume equal subtree height for all proofs involving *rebalance_middle_tree*. For this proof, we re-use Lemma 2.3.5, as two nodes of order k will always contain a maximum of $4k + 1$ separators.

As mentioned, the result may have an empty list as key-value list. This happens exactly when one of the two subtrees of a root with minimal size underflowed. After this operation, the tree shrinks in height. The shrinked tree then is exactly the one remaining subtree in the original node. See the operation *reduce_root* in Listing 2.5 for the exact implementation of this step. Since rebalancing always restores order k for all subtrees of the current node and *delete* employs *reduce_root*, the final result always has at least $\text{order}^r\ k$.

The proofs for balancedness invariance follow similarly to the ones in Section 2.3.4 by first showing height invariance. Both properties follow using the associativity and commutativity of the maximum operation. [7]

With the basic operations covered, the invariant preservation of the *del* function can be derived inductively. Note that the function will return a tree of almost order k, where the remaining root-underflow is coverered by *reduce_root*. Moreover we do not need an

[7] When inspecting the proof documents, one will notice that the proof of height, balancedness and order are interleaved for the intermediate functions, especially *split_max* and *rebalance_middle_tree*. This is due to the fact that these operations are only well defined for balanced trees. Moreove *split_max* requires at least one element in the node list, an equivalent to $\text{order}^r\ k$.

Listing 2.5: The *delete* function

```
fun split_max where
  split_max k (Node ts t) = (
    case t of Leaf ⇒ (
      let (sub,sep) = last ts in (Node (butlast ts) sub, sep)
    ) |
    _ ⇒ case split_max k t of (sub, sep) ⇒
          (rebalance_last_tree k ts sub, sep)
    )

fun del where
  del k x Leaf = Leaf |
  del k x (Node ts t) = (
  case split ts x of (ls,[]) ⇒
    rebalance_last_tree k ls (del k x t) |
  (ls,(sub,sep)#rs) ⇒
    if sep ≠ x then
      rebalance_middle_tree k ls (del k x sub) sep rs t
    else if sub = Leaf then
      Node (ls@rs) t
    else let (sub_s, max_s) = split_max k sub in
      rebalance_middle_tree k ls sub_s max_s rs t
  )

fun reduce_root where
  reduce_root Leaf = Leaf |
  reduce_root (Node ts t) = (case ts of
    [] ⇒ t |
    _ ⇒ (Node ts t)
  )

fun delete where delete k x t = reduce_root (del k x t)
```

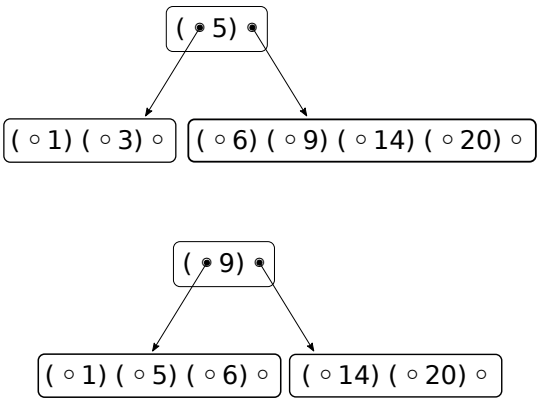


Figure 2.5: The tree from Figure 2.4 after successive deletion of 10 (top) and 3 (bottom). Note how the final result differs from approaches that would only steal single elements from neighbors to handle underflow.

additional lemma for an input with nodes that have normal order k. As $k > 0$, order k implies a root order of k. And even though it might seem to be beneficial to know whether the input has normal order or root order, this is not the case. The rebalancing functions only require almost order k and return a list of trees of valid order. We are never interested in the length of that list itself until reaching the root, so we are satisfied with statements on the order of subtrees, which are made by almost order k sufficiently.

Lemma 2.3.11 $k > 0 \wedge \text{order}_k^r\, t \wedge \text{bal}\, t \Longrightarrow \text{order}_k^a(\text{del}_k\, x\, t) \wedge \text{bal}(\text{del}_k\, xt)$

From this and our inspection of *reduce_root* follows almost directly the fact about the invariant preservation of *delete*.

Theorem 2.3.4 $k > 0 \wedge \text{order}_k^r\, t \wedge \text{bal}\, t \Longrightarrow$

$$\text{order}_k^r(\text{delete}_k\, x\, t) \wedge \text{bal}(\text{delete}_k\, xt)$$

Note that now (as opposed to insertion) $k > 0$ is required. The main reason is that, if $k = 0$, rebalancing would not always work anymore as we could not be certain to have at least two subtrees per node.

In order to prove that *delete* acts the same as $delete_{list}$ on the inorder of the tree, we need to show some inorder properties for the intermediate functions. For the *del* helper function, we use the same specialized lemmas as for Theorem 2.3.1 and Theorem 2.3.3 and induct over the execution of the *del*-function. Rather than stating the specialized

lemmas again, we would like to point out the elegance of the inorder method at this point.

The reason the rebalancing operations preserve the sortedness and set properties is plain when considering the inorder view of the tree: it does not change at all. A manual proof about the set and sortedness properties would require complicated, unnecessarily lengthy proofs. For the inorder view, since the definition can be easily simplified, not even involving recursive calls, the following property can be shown automatically.

Lemma 2.3.12 *All subtrees have the same height* $\Longrightarrow$

$$\begin{aligned}&\text{inorder } (\text{rebalance\_middle\_tree}_k\ ls\ sub\ sep\ rs\ t) = \\ &\text{inorder } (Node\ (ls@(sub, sep)\#rs)\ t)\end{aligned}$$

Considering the function split_max, the standard approach would require showing tideously that the element returned is in fact the maximum of the left subtree and showing that the remaining tree union the maximum element gives the whole original tree set. Instead we simply show the following, comprising both facts:

Lemma 2.3.13 *If t has more than two subtrees, and the last two have equal height, then*

$$\text{inorder } (\text{split\_max}_k\ t) = \text{inorder}\, t$$

This fact follows easily by induction on the computation of split_max, using Lemma 2.3.12. The reason that these proofs follow so easily is the intention behind the definition of both operations. The idea is to obtain the same elements, in the same order, just now in a configuration from which we can easily obtain a new, valid B-tree.

Finally we obtain the last important property of the set interface.

Theorem 2.3.5 $k > 0 \wedge \text{order}_k^r\, t \wedge \text{bal}\, t \wedge \text{sorted}(\text{inorder}\, t) \Longrightarrow$

$$\text{inorder } (\text{delete}_k\ x\ t) = \text{del}_{list}\ x\ (\text{inorder}\, t)$$

With this, we have proven that our implementation of B-trees is proven correct fulfils the specifications of the set interface based on an inorder view. The next step towards imperative B-trees is to implement an imperative refinement of these set operations.

3 Imperative B-Trees in Isabelle

In the previous chapter, we have seen an abstract definition of B-trees and the reasoning behind its correct implementation of the set interface. However, the specification would not yield efficiently executable code. The reason is that the abstract specification works with the persistent datatypes of nodes and lists. This way, trivially, no data is unknowingly modified or corrupted, however this is at the cost of computational efficiency as changes require allocating memory for a completely new object.

In order to obtain an efficient implementation on ephemeral data structures common to imperative languages, we specify imperative code and show that it refines the abstract specification. The imperative code can then be translated to the languages Scala [Ode+15] and SML [MTH90] using the code generation facilities in Isabelle/HOL [HB20]. However, code extraction is not limited to these specific languages since the code generator is extensible.

3.1 Refinement to Imperative/HOL

The proofs of correct behavior for our imperative code are stated in separation logic an extension of Hoare Logic invented by Reynolds [Rey02]. It was formalized in Isabelle/HOL by Lammich [Lam19] and comes with a framework that simplifies reasoning about its expressions.

3.1.1 Separation Logic

Separation logic provides a way to reason about mutable resources that lie in separated parts of an external heap, an abstracted memory device. The assumptions on the state of the heap are called *assertions*. They are stated as formulae that hold for specific heap states. The basic assertions used in this work are the following:

- *emp* holds for the empty heap
- *true* and *false* hold for every and no heap respectively
- $\uparrow(P)$ holds if the heap is empty and predicate P holds

- $a \mapsto_a as$ holds if the heap at position a is reserved and contains an array representing as.
- $a \mapsto_r x$ holds if the heap at position a is reserved and contains value x where x is of some type $'a :: heap$
- $\exists_A x.\ P\ x$ holds if there exists some x such that predicate P holds on the heap for given x.
- $P_1 * P_2$ holds if each assertion holds on its part of the heap and the areas of the heap described by each assertion are non-overlapping

The last assertion is key to enabling reasoning about local parts of the heap. With the tools provided by separation logic, we well prove our imperative implementations correct by showing that they refine the abstract implementation in Chapter 2.

In general, refinement relates a concrete data structure to an abstract structure via some refinement assertion. Examples for such a refinement assertion are the id assertion and the list assertion. The latter relates lists, given a refinement assertion between the elements of the first and second list.

definition id_assn *a b* = ↑*(a = b)*

fun list_assn :: *('a ⇒ 'c ⇒ assn) ⇒ 'a list ⇒ 'c list ⇒ assn* **where**
  list_assn *P [] [] = emp*
| list_assn *P (x#xs) (y#ys) = P x y* ∗ list_assn *P xs ys*
| list_assn _ _ _ = *false*

We can then specify Hoare Triples on assertions, to encapsulate statements about imperative programs.

Definition 3.1.1 *Hoare Triple* ⟨ *P* ⟩ *c* ⟨λ*r*. *Q r* ⟩ *holds iff*
if assertion P holds on some heap before operation c is executed on the heap, and operation c returns some r, then assertion Q holds for r and on the heap modified by c

A simple example Hoare Triple is ⟨ id_assn *a b* ⟩ return *b* ⟨λ*r*. id_assn *a r* ⟩. In the refinement process, we will use more complex relationship assertions between some abstract *b* and some refined *bi*. We then show that the value *r* returned by the imperative program *imp_op* satisfies the same relationship to *op b* where *op* is the refined abstract specification. Further we use the notation ⟨ *P* ⟩ *c* ⟨λ*r*. *Q r* ⟩$_t$ as a shorthand for ⟨ *P* ⟩ *c* ⟨λ*r*. *Q r* ∗ *true* ⟩. Since *true* holds for any heap, this part of the assertion may be used to subsume all temporary and discarded variables.

Refining all set operations from Chapter 2 will yield imperative code that operates on the heap with the option to use efficient destructive updates. We do not need to show that the operations themselves satisfy the requirements of the set interface. Instead we merely need to show that the imperative code acts on a concretized version of the B-trees the same way as the abstract operations on the abstract version.

3.1.2 Refinement of abstract lists

In the imperative collection framework by Lammich [Lam19], the abstract datatype list is usually refined to a dynamic array like data structure. It comprises an array and a natural number, where the latter denotes the current number of elements stored in the array. B-tree nodes do not always contain a constant number of elements, which is why this property should be transferred to the data structure refining key-value lists.

However, dynamic arrays are designed to grow and shrink as elements are inserted and deleted. B-trees were invented to be stored on slow hard disks. As one disk access loads as much data as fits into a page of main memory, nodes of roughly this size make the most efficient use of an access [BM72]. Therefore the data structure to store the node content should reserve arrays of fixed size, independent of the actual number of keys and children contained.

Based on the definition of dynamic arrays in the imperative collection framework, we introduce a simpler data structure, the *partially filled array*, that keeps count of currently inserted elements, but does neither grow nor shrink. A list ls is represented by a partially filled array (a, n) if the array a represents a list ls', of which the first n elements form list ls. Formally,

datatype *'a pfarray = 'a array* $\times$ *nat*

definition is_pfa :: *nat* $\Rightarrow$ *'a list* $\Rightarrow$ *'a pfarray* **where**
  is_pfa *c ls (a,n)* =
    $\exists_A$ *ls'*. *a* $\mapsto_a$ *ls'* $*$ $\uparrow$(*c* = length *ls'* $\wedge$ *n* $\leq$ *c* $\wedge$ *ls* = (take *n ls'*))

where c is the *capacity* of the array, the actually allocated size of the array on the heap. All references to lists in the abstract datatype *btree* will in the imperative version be refined to partially filled arrays.

However one should note that from the set operation specification in Chapter 2 alone it is not determined which operations are supposed to be conducted in-place and which will require copying data. To obtain the most efficient code we therefore manually define the operations in Listing 3.1 to replace complex composite operations based on list construction and catenation.

Listing 3.1: Important Partially Filled Array functions for Insertion.

```
definition pfa_length ≡ λ(a,n). return n

definition pfa_get ≡ λ(a,n) i. Array.nth a i (* returns the nth element in array a *)

definition pfa_set ≡ λ(a,n) i x. do {
    Array.upd i x a; (* sets the element at position i in a to value x *)
    return (a,n)
}

definition pfa_shrink k ≡ λ(a,n). return (a,k)

definition pfa_shrink_cap k ≡ λ(a,n). do {
    a' ← array_shrink a k; (* returns an array with the given actual size, potentially reallocated*)
    return (a',min k n)
}

definition pfa_drop ≡ λ(src,sn) si (dst,dn). do {
    blit src si dst 0 (sn−si);
    return (dst,(sn−si))
}

definition pfa_insert ≡ λ(a,n) i x. do {
  array_shr a i 1; (* shifts elements from index i on to the right by 1 *)
  Array.upd i x a;
  return (a,n+1)
}

definition pfa_ensure ≡ λ(a,n) k. do {
  a' ← array_ensure a k default; (* returns an array with given minimal size, potentially reallocated *)
  return (a',n)
}

definition pfa_insert_grow ≡ λ(a,n) i x. do {
  a' ← pfa_ensure (a,n) (n+1);
  a'' ← pfa_insert a' i x;
  return a''
}

...
```

During the splitting of nodes, a part of the overflowing list is copied into another array. The operation to copy data is based on the function *blit*, that copies a slice of an array to a slice of another array. The function already existed in the standard collection and could be reused to define the *pfa_drop* function in Listing 3.1, which in this implementation copies data to a new array.

The *blit* function is however not capable of correctly copying data within the array, as required for efficient in-place insertion and deletion. A suitable function, called *sblit*, was hence added in the course of this project. The relevant addition is to differentiate between copying elements from higher to lower indices or from lower to higher indices. Depending on the direction of the copy, it is relevant in which order to copy singleton elements so that in case of overlapping ranges no elements are overwritten before being copied.

By adding a function that copies elements in reverse order and conditionally calling either copy function, loss-free in-place copying was achieved. The resulting function is conveniently agnostic to the direction of the copying, just as the respective implementations in target languages for code extraction.[1] The *pfa_insert* function in Listing 3.1 makes implicit use of in-place copying by calling a function to shift elements to the right by one. This is achieved by copying the slice of elements from position i to the end of the list to the slice $i + 1$ to the end of the list. The element to be inserted is then placed in the remaining free spot.

For all functions in Listing 3.1, appropriate Hoare Triples were derived. All triples are straightforward to derive but the reader is encouraged to look them up in the proof documents. To roughly sketch the obtained lemmas, the two examples *pfa_drop* and *pfa_shrink* are investigated below. The most important difference is whether the operations are destructive (in-place) or not. For most operations, the approach that required the least memory allocations was chosen.[2]

Lemma 3.1.1 $k \leq \text{length}\, s \wedge (\text{length}\, s - k) \leq dn \Longrightarrow$

$$\langle \text{is\_pfa}\ sn\ s\ si\ *\ \text{is\_pfa}\ dn\ d\ di \rangle$$
$$\text{pfa\_drop}\ si\ k\ di$$
$$\langle \lambda di'.\, \text{is\_pfa}\ sn\ s\ si\ *\ \text{is\_pfa}\ dn\ (\text{drop}\ k\ s)\ di' \rangle$$

The array *si* stays untouched. This fact is expressed by the fact that the term about its relationship to s is still available in the postcondition of the triple. The only thing

[1] See for example the specification of `Array.blit` in Ocaml or the `ArraySlice.copy` specification in SML.

[2] The only exception is node_i, which in the case of overflow currently requires temporary allocation of an array that holds more than $2k$ elements.

that changed is the content of *di*.[3] The *take* function was refined differently. The cheapest and in our context consistent way to implement *take* is to reduce the number of elements knowingly stored in the array from n to k. The result is that the original array gets modified and we loose any information about the content that is written beyond given k.

Lemma 3.1.2 $k \leq \text{length}\, s \Longrightarrow$

$$\langle \text{is\_pfa } sn\ s\ si \rangle \text{ pfa\_shrink } si\ k\ \langle \lambda si'.\ \text{is\_pfa } sn\ (\text{take } k\ s)\ si' \rangle$$

With these operations and suitable Hoare-Triples about them, we have all the tools required to refine the abstract set implementation. The only missing piece is an imperative split function.

3.2 Set-Operations

Having covered the refinement for the list datatype, the B-tree datatype itself can be refined. Rather than actually storing nodes within lists, the B-tree nodes are supposed to be stored in the heap, and the node list will only contain pointers to the corresponding nodes. Pointers are represented by the type *ref* in Imperative/HOL. Leaf nodes contain no information and consequently don't have to be stored on the heap, therefore we refine them using null pointers. Since the Imperative Refinement Framework does not support null pointers, B-tree nodes are refined by heap reference *options* instead. None represents leafs or empty trees. Valid heap references point to refinements of internal nodes.

datatype *'a btnode* =
Btnode *(('a btnode ref option*'a) pfarray) ('a btnode ref option)*

Note the similarity to the definition in Section 2.1, only that *btree* was replaced by *btnode ref option*, *list* was replaced by *pfarray*, and there is no leaf case anymore.

An abstract B-tree is represented by such a physical node if all references on subtrees represent the corresponding abstract subtrees and all keys are the same. The precise refinement assertion can be seen in Listing 3.2.

It seems a little bit awkward that the definition includes two relationships regarding only the list of elements in the subtree. The reason is that there are two steps of refinements going on. First, all elements in the list are refined to their imperative

[3] In the actual proof, where possible, we even show that the array itself has not changed (i.e. has not been reallocated). The correct notation does however not yield high readability and is hence abstracted here.

Listing 3.2: The refinement assertion for B-trees

```
fun btree_assn :: nat ⇒ 'a::heap btree ⇒ 'a btnode ref option ⇒ assn where
  btree_assn k Leaf None = emp |
  btree_assn k (Node ts t) (Some a) =
 (∃_A tsi ti tsi'.
       a ↦_r Btnode tsi ti
     * btree_assn k t ti
     * is_pfa (2*k) tsi' tsi
     * list_assn ((btree_assn k) ×_a id_assn) ts tsi'
     ) |
  btree_assn _ _ _ = false

abbreviation blist_assn k ≡ list_assn ((btree_assn k) ×_a id_assn)
```

counterparts. This fact is expressed by the *list_assn* term in the assumption. Second, the list itself is refined to an array. This is expressed in *is_pfa* term. Since the list assumption makes recursive use of the B-tree assertion, we again cover up usage of the term by an abbreviation.

Using the parameters of *is_pfa*, note that we can specify that every node of a B-tree should have a capacity of $2k$.

3.2.1 The split function

To implement any set operations we again need a split operation. However, what split function exactly will be used can be abstracted again. The only important thing is that it refines an abstract split function. However we have some freedom on how to specify this refinement relationship.

Imperative split functions should be efficient and as such should not actually split the array in half. The operation should rather return the index of the correct subtree-separator pair. The refinement condition derived from this relationship is as follows.

definition split_relation *xs (ls,rs) i* = $i \leq$ length $xs \wedge ls$ = take $i\ xs \wedge rs$ = drop $i\ xs$

A very useful alternative statement of the relationship follows automatically. This allows us to simplify terms that only contain xs to the two sublists when we obtain this relationship in proofs.

Lemma 3.2.1 split_relation $xs\ (ls, rs)\ i = (xs = ls@rs \wedge i = \text{length}\, ls)$

Listing 3.3: The imperative linear split

```
definition lin_split :: ('a::heap × 'b::{heap,linorder}) pfarray ⇒ 'b ⇒ nat Heap where
    lin_split (a,n) p = do {
        i ← while
            (λi. if i<n then do {
                (_,s) ← Array.nth a i;
                return (s<p)
            } else return False)
            (λi. return (i+1))
            0;
        return i
    }
```

From this, we characterize the desired imperative split function *imp_split* by its Hoare-Triple.

$$\begin{array}{l}\langle \text{is\_pfa}\, c\ tsi\ a * \text{blist\_assn}\, k\ ts\ tsi * true\rangle \\ \quad \text{imp\_split}\, a \\ \langle \lambda i.\, \text{is\_pfa}\, c\ tsi\ a * \text{blist\_assn}\, k\ ts\ tsi * \uparrow (\text{split\_relation}\, ts\ (\text{split}\, tsp)\ i)\rangle_t\end{array}$$

Just as with the abstract implementation, the imperative refinements can be built with this characterization alone. However this time we are actually interested in finding an efficient split and will spend some time finding a good split function.

In the imperative context, we prefer not to directly refine the recursive function from Figure 2.3 but we embark on the exercise of using a while loop. The implementation in Listing 3.3 not more efficient than the functional linear search, but suitable for getting to know the usage of while loops in imperative HOL. Every iteration, an index on the array is moved forward by one. The main part is done in the loop head - if we are within the array bounds, we obtain the current separator and check if it is smaller than the partitioning element. We return the first index at which the separator is greater or equal.

As mentioned, the internal behavior of this function is quite different to the abstract linear split from Figure 2.3. Therefore, proving that it refines the abstract function requires a small detour. We first express and show the most basic, non-trivial statement we can make about the result of the function. We then show that the result is the same as the result of applying the abstract function, that is that to an external observer, the functions behave the same.

We begin by stating a simple matching Hoare Triple for the function.

Lemma 3.2.2

$$\begin{aligned}&\langle \text{is\_pfa}\, c\ xs\ (a,n)\rangle\\ &\quad \text{lin\_split}\ (a,n)\ p\\ &\langle \lambda i.\, \text{is\_pfa}\, c\ xs\ (a,n)\\ &\quad * \uparrow (i \leq n \wedge (\forall j < i.\, snd(xs!j) < p) \wedge (i < n \longrightarrow snd(xs!i) \geq p))\rangle_t\end{aligned}$$

It follows by supplying the following loop invariant:

- $\forall j < i.\ \text{snd}(xs!j) < p$: All elements up to the current i are smaller than the partitioning element,
- $\uparrow (i \leq n)$: i does not exceed the length of the array and
- is_pfa $c\ xs\ (a,n)$: the array is not manipulated by the operation

In order to show that the loop also terminates, we need to provide a measure that strictly decreases in each iteration. The measure for the loop is the difference $n - i$ which decrements by one in each iteration and stays positive.

For unexciting reasons, the post condition we have thus obtained actually suffices to imply that the split relation to the result of an abstract split function is satisfied. Specifically, we use the abstract split function from Figure 2.3. For the two cases of loop termination, either $i = n$, or $i < n$, we find that the abstract function will return the exact same split.

This is great to know since the previous lemma follows directly from the computation and is closely related to the loop invariant. This will simplify proofs for alternative imperative split functions. [4] With this reassurance we approach the binary split.

A detailed analysis of binary search algorithms may be found in the work of Montague [Mon91], which has served as an orientation for the derived algorithm in Listing 3.4. Rather than walking through the array, it narrows the range in which the desired element may lie to a window that it bisected in every iteration. This bisection is implemented by inspecting the middle element of the window. If the desired element was not found, the algorithm recurses into either the left or right remaining subwindow. In the final iteration, the window has narrowed down to a single element, containing the correct separator and subtree.

[4] Another, albeit more theoretical, reason for excitement is that we can now specify imperative B-tree interfaces without any reference to the abstraction. We simply always use the abstract linear split function and only require that the imperative split function satisfies the hoare triple from Lemma 3.2.2.

Listing 3.4: The imperative binary split

```
definition bin_split :: ('a::heap × 'b::{heap,linorder}) pfarray ⇒ 'b ⇒ nat Heap
    where bin_split (a,n) p = do {
        (low',high') ← while
            (λ(low,high). return (low < high))
            (λ(low,high).
            let mid = ((low + high) div 2) in
            do {
                (_,s) ← Array.nth a mid;
                if p < s then
                    return (low, mid)
                else if p > s then
                    return (mid+1, high)
                else
                    return (mid,mid)
            })
            (0::nat,n);
        return low'
    }
```

Since pairs as list elements (i.e. pairs of trees and separators) were expected to add another layer of complexity, a binary split algorithm on normal lists has been derived first. The code required several small corrections until it could be proven correct. This supports the general call for verified programs and hints at the fragility of efficient unverified programs. The split function for lists of pairs was then defined and could be tackled using the same loop invariant. Note that the binary split works on two indices as states, the upper bound h and the lower bound l.

- $\forall j < l.\ \text{snd}(xs!j) < p$: All elements up to the current l are smaller than the partitioning element,
- $h < n \rightarrow \text{snd}(xs!h) < p$: If the upper bound h is strictly less than the length of the array, the element it points to is greater than the partioning element,
- $\uparrow (l \leq h)$: The lower bound is always less or equal to the upper bound,
- $\uparrow (h \leq n)$: the upper bound does not exceed the length of the array and
- is_pfa c xs (a, n): again, the array is not manipulated by the operation

In this case the window that is considered by the binary split provides the loop measure, as $h - l$ decreases by at least one in each step. Proving that this is the case was not the main issue, but rather finding a correct formulation of the algorithm where this is actually true.

Overall obtaining the same Hoare Triple as for the linear split required only this slightly more complex loop invariant and invocation of a number of algebraic lemmas. This was somewhat of a surprise, since the functional version of a binary split is much harder to specify and analyse than a functional linear split. The additional requirement for the binary split is naturally that the list of separators must be sorted.

Lemma 3.2.3 $\text{sorted}(\text{separators}\, xs) \Longrightarrow$

$$
\begin{aligned}
&\langle \text{is\_pfa}\, c\ xs\ (a, n)\rangle \\
&\quad \text{bin\_split}\ (a, n)\ p \\
&\langle \lambda i.\, \text{is\_pfa}\, c\ xs\ (a, n) \\
&\quad * \uparrow (i \leq n \wedge (\forall j < i. snd(xs!j) < p) \wedge (i < n \longrightarrow snd(xs!i) \geq p))\rangle_t
\end{aligned}
$$

Since we know that this implies equivalence to the abstract split function and sortedness of the separators is guaranteed by the sortedness invariant of the B-trees, we can safely use this function for specifying the set operations.

Listing 3.5: The imperative isin function

```
partial_function (heap) isin :: 'a btnode ref option ⇒ 'a ⇒ bool Heap
  where
    isin p x =
  (case p of
      None ⇒ return False |
      (Some a) ⇒ do {
        node ← !a;
        i ← imp_split (kvs node) x;
        tsl ← pfa_length (kvs node);
        if i < tsl then do {
          s ← pfa_get (kvs node) i;
          let (sub,sep) = s in
          if x = sep then
            return True
          else
            isin sub x
        } else
            isin (last node) x
    }
)
```

3.2.2 The isin function

In general, the formulation of the imperative set operations do quite directly follow from the refinement of the abstract operations. The main difference is the usage of pointers, that need to be dereferenced and updated. Functions such as length need to be called and have their result stored explicitly before use. Other than that, the definition of the imperative isin in Listing 3.5 function should bare no surprises. Note that it makes use of the function *imp_split* which is an arbitrary split function that fulfils the Hoare Triple of Lemma 3.2.3.

The convention for variables that refer to refined datatypes is that they have the same name as the abstract datatype, with the postfix *i*. For example below, the abstract *btree* is referred to by *t*, however the refined *btnode ref option* is called *ti*. To simplify notation, we simply assume that the set operations such as *isin* are overloaded and refer to abstract or imperative implementations, depending on the parameter type. The Hoare Triple to argue that the imperative *isin* refines the abstract *isin* follows.

Lemma 3.2.4 $\text{sorted}(\text{inorder}\, t) \Longrightarrow$

$$\begin{array}{l} \langle \text{btree\_assn}\, k\ t\ ti \rangle \\ \quad \text{isin}\, ti\ x \\ \langle \lambda r.\, \text{btree\_assn}\, k\ t\ ti * \uparrow (\text{isin}\, t\ x = r) \rangle_t \end{array}$$

Note how, in contrast to the abstract specification, we need to also show that the imperative function does not modify the tree residing in the heap. The proof follows by induction on the computation of the abstract isin function.

Inside a structured proof skeleton of the induction, proving the subgoals in apply style turned out to be the most appropriate. The separation automation tool only required help for the instantiation of existential quantifiers, especially after application of the inductive hypothesis.

3.2.3 Insertion

The imperative implementation and derived heap rule of the *insert* function follow the same pattern as the *isin* function. The only interesting part is whether and how often new memory is allocated. An excerpt of the refinement of the *ins* function can be seen in Listing 3.7[5]. The function includes some simple optimizations, such as not calling $node_i$ if the node is not overflowing. This is an optimization as otherwise $node_i$ in Listing 3.6 would always allocate a new B-tree node rather than updating the current node in place.

To prove that this optimization is valid, the only obligation is to show that the updated node itself is equivalent to the node returned by $node_i$. Since we are in the branch where the node contains less than $2k$ elements, this is easy to show, provided some intermediate lemmas that relate concrete node and abstract node in this context.

Lemma 3.2.5 $\text{length}\, ts \leq 2k \Longrightarrow \text{node}_i\, k\ ts\ t = T_i(Node\ ts\ t)$

[5] In the imperative language in Isabelle, allocating heap memory is abbreviated by *ref*. To clarify that this means allocation, it is named *alloc* here.

Listing 3.6: The imperative node_i function

```
definition node_i
    :: nat ⇒ ('a btnode ref option × 'a) pfarray ⇒ 'a btnode ref option ⇒ 'a btupi Heap
  where node_i k a ti ≡ do {
      n ← pfa_length a;
      if n ≤ 2*k then do {
        a' ← pfa_shrink_cap (2*k) a;
        l ← alloc (Btnode a' ti);
        return (T_i (Some l))
      }
      else do {
        b ← (pfa_empty (2*k)
            :: ('a btnode ref option × 'a) pfarray Heap);
        i ← split_half a;
        m ← pfa_get a i;
        b' ← pfa_drop a (i+1) b;
        a' ← pfa_shrink i a;
        a'' ← pfa_shrink_cap (2*k) a';
        let (sub,sep) = m in do {
          l ← alloc (Btnode a'' sub);
          r ← alloc (Btnode b' ti);
          return (Up_i (Some l) sep (Some r))
        }
      }
    }
```

Listing 3.7: Excerpt of the imperative insert function

```
partial_function (heap) ins :: nat ⇒ 'a ⇒ 'a btnode ref option ⇒ 'a btupi Heap
    where
      ins k x apo = (case apo of
    None ⇒ return (Up_i None x None) |
    (Some ap) ⇒ do { a ← !ap; i ← imp_split (kvs a) x; tsl ← pfa_length (kvs a);
      if i < tsl then do {
        s ← pfa_get (kvs a) i;
        let (sub,sep) = s in
        if sep = x then return (T_i apo)
        else do {
          r ← ins k x sub;
          case r of
            (T_i lp) ⇒ do { pfa_set (kvs a) i (lp,sep); return (T_i apo)
            } |
            (Up_i lp x' rp) ⇒ do {
              pfa_set (kvs a) i (rp,sep);
              if tsl < 2*k then do {
                  ts' ← pfa_insert (kvs a) i (lp,x');
                  ap := (Btnode ts' (last a));
                  return (T_i apo)
              } else do {
                ts' ← pfa_insert_grow (kvs a) i (lp,x');
                node_i k ts' (last a)
      } } } } else ...

definition insert :: nat ⇒ ('a::{heap,default,linorder}) ⇒ 'a btnode ref option ⇒ 'a btnode ref option Heap
 where insert ≡ λ k x ti. do {
  ti' ← ins k x ti;
  case ti' of
      T_i sub ⇒ return sub |
      Up_i l a r ⇒ do {
        ts' ← pfa_init (2*k) (l,a) 1;
        t' ← alloc (Btnode ts' r);
        return (Some t')
      }
}
```

By adding a lemma on the equivalence between abstract and imperative $node_i$, we obtain a lemma for *ins* by induction on the computation of the abstract *ins* function.

Lemma 3.2.6 $\text{sorted}(\text{inorder}\, t) \Longrightarrow$

$$\begin{array}{c} \langle \text{btree\_assn}\, k\ t\ ti \rangle \\ \text{ins}\, k t i\ x \\ \langle \lambda r.\, \text{btree\_assn}\, k\ (\text{ins}\, k\ t\ x) r \rangle_t \end{array}$$

The proof of this lemma was done in a mixed structural and apply style manner. It could have been done in a complete apply style manner. However many subgoals are generated in the proof process that are simply invalid due to the preconditions of the hoare triple. Embedding the proof in the structural skeleton makes spotting these void statements easier and allows to have a better overview over the current state of the proof.

The insertion function incorporates the function $tree_i$ in much the same way as *ins* incorporated the small $node_i$ optimizations. Here the proof of equivalence followed automatically. With the hoare triple for *ins*, we obtain the final desired lemma on the imperative insertion.

Theorem 3.2.1 $k > 0 \wedge \text{sorted}(\text{inorder}\, t) \Longrightarrow$

$$\begin{array}{c} \langle \text{btree\_assn}\, k\ t\ ti \rangle \\ \text{insert}\, k\ x\ ti \\ \langle \lambda ri.\, \text{btree\_assn}\, k\ (\text{insert}\, k\ x\ t)\ ri \rangle_t \end{array}$$

Note how we have the additional assumption that k is positive. This was only required to show correct order in the abstract setting. Here, the capacity of the root node needs to be specified at initialization. And since the capacity has to be $2k$ and the node has to hold at least one element a positive k is required such that no contradiction appears.

Until now we have only shown that the concrete implementations refine the abstract interpretations. What does this bring concretely? Do we have to finally resort to an informal argument to show that the result satisfies the invariants and the set interface? This is not the case. The final theorem that proves functional correctness of the imperative implementations with respect to the set interface follows.

Corollary 3.2.1 $k > 0 \Longrightarrow$

$$\begin{array}{c} \langle \text{btree\_assn}\, k\ t\ ti \wedge \text{btree}_k\, t \rangle \\ \quad \text{insert}\, k\ x\ ti \\ \langle \lambda ri. \exists_A r.\, \text{btree\_assn}\, k\ r\ ri \wedge \text{btree}_k\, r \wedge \text{inorder}\, r = \text{ins}_{\text{list}}\, x\ (\text{inorder}\, t) \rangle_t \end{array}$$

And indeed, this can be derived automatically with a single command invocation, unfolding Definition 2.1.1 and using Theorem 3.2.1, Theorem 2.3.3 and Theorem 2.3.2.[6]

3.2.4 Deletion

An imperative version of the deletion operation has been specified. It required some additional operations on partially filled arrays but posed no foundational challenges. Due to its length it is not shown here but may be looked up in the actual proof documents.

However the proof of its refining property was not conducted. We assume that this proof would be structured similar to the previous proofs. The only benefit would be exploring usage of the new heap rules introduced for the additional array operations. The main difference to the verification of the insertion function is that the refined functions are undefined for many cases. Hence it is highly important to strongly tie the refined and imperative version in order to discharge void cases.

[6] Technically we also need the fact that ins_{list} preserves sortedness. This was already proven by Nipkow [Nip16] and is considered trivial here.

4 Conclusions

In this work, we obtained an imperative implementation of B-trees that is capable to handle membership and insertion queries. The implementation can be supplied with different intra-node search algorithms, of which a linear search based and a binary search based version are included. It makes efficient use of in-place updates and arrays of fixed size. Concrete code has been obtained in the languages Scala and SML, where the code generator may simply be extended to support other languages. We have proven the membership and insertion operations functionally correct within the Isabelle framework with the help of an abstract specification that is refined by the imperative code. The functional specification fully implements an abstract set interface on linearly ordered elements, also supporting deletion queries.

4.1 Working in Isabelle

An overview on the resulting theory modules, their relationship to external modules and one another can be seen in Figure 4.1. The given graphic roughly resembles the actual file structure. It can be seen that in order to develop this project not many general lemmas and definitions had to be added to the library. The library of proofs considered here is the one provided by the Isabelle/HOL framework together with the *Archive of Formal Proofs*. The latter is a refereed collection of proofs that contains, among others, the separation logic framework.

Overall, proving in the Isabelle framework is quite comfortable as it supplies the user with a plethora of tools to find required or related lemmas, or even find proofs of whole theorems. However there were of course small issues in the usage of the tool that tend to be inherent to machine checked proofs.

One main source of frustration was finding the correct library functions and lemmas for smooth proofs. In informal proofs, a certain kind of function and corresponding equalities can be expected to be known by the reader. However for machine checked proofs all used functions and lemmas need to be stated and proven explicitly. Therefore using library functions that are supplied with a sufficient amount of lemmas is vital for proofs to require acceptable time and effort. The Isabelle/HOL framework provides tools like sledgehammer or **find_theorems** that usually find useful lemmas for the concrete situation. However each have their own drawbacks.

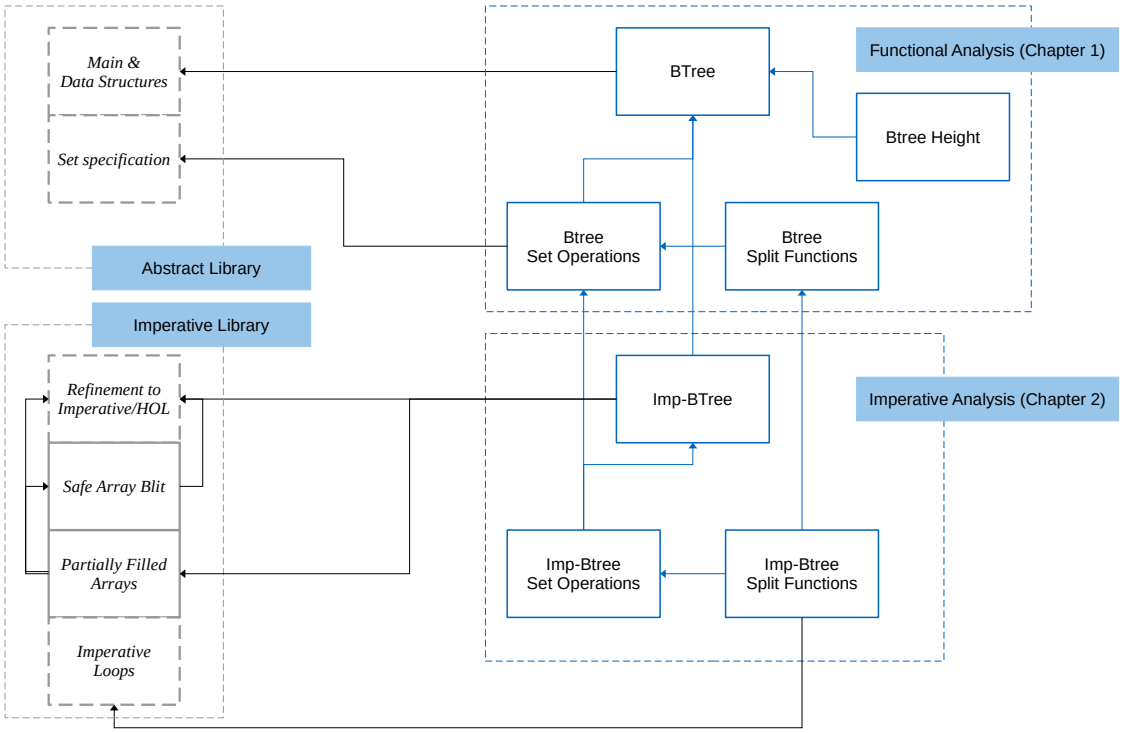


Figure 4.1: An overview on the theories comprised by this work and their approximate relationship. An arc points to a theory or package that contains required lemmas or definitions. Theories created or significantly extended in the course of this work are outlined with a continuous line.

If the goal is only to simplify a given expression but not to directly solve the result, sledgehammer is useless as it only returns the applied lemmas for a successfully finalized proof. There is no feedback on "how far" the proof got. Lemmas that could be applied to significantly simplify the goal but did not yield a complete proof are discarded. Showing what was applied would have been especially useful for the apply style proofs of the imperative implementation that was mainly comprised of simplification steps. Maybe the tools can be extended to return what lemmas lead to promising simplifications of the the goal term to provide additional input to the user.

Secondly **find_theorems** is useless if the current state is simplified too much to perfectly match existing lemmas. As a concrete example, the *inorder* of B-trees was first defined with "foldr map ... *ts []*". The proof system could not provide any useful lemmas on expressions of this form. However there exists the standard function *concat* that simplifies to the above. The standard library contains useful lemmas for applications of this function. The function was only discovered close to the finalization of this thesis, in a phase where standard terms simplifying the stated functions were looked for. Since the available lemmas were key to a practical implementation of the inorder set interface, this meant a wasted workload of almost 2000 lines of additional proof. The lesson learnt is to always look for alternative or equivalent formulations of properties as well as lemmas, to see what kind of useful lemmas may already exist in the standard library.

Another point of discomfort is in the separation logic framework. Especially when inserting elements in lists, original assumptions on the whole list have to be split into assumptions on parts of the list or lists obtained by concatenation. The aggressive simplification of the automatic separation logic prover (**sep_auto**) does often go to far in these proofs. This either results in unreadable (and unprovable) subgoals resulting from a chain of commands. Sometimes it also yields states for which a rule may not be applied anymore. For example when we apply $node_i$ on some list, this list has usually been changed to append or insert a single element. This means that *blist_assn* holds for the whole list. However the automation tools knows that the list is comprised on sublists and automatically simplifies related expressions. In order to match the resulting precondition with the $node_i$ heap rule defined on full lists, an intermediate rule has to be derived that has as precondition this simplified state.

4.2 Comparison of approaches and lessons learnt

In the abstract analysis part of this work, the proofs have been written to use only minimal requirements on the input. For example in Lemma 2.3.13, we do not require balancedness and order k for $k > 0$ on the full tree. Instead, it is only required that the

node has at least two children, the last two have equal height and that the last tree has the same, recursive property. Using this more specific assumption was beneficial during the proof, as the less restrictive requirements turn up in the induction hypothesis again and are easier to satisfy than general assumptions. However this also called for more lemmas that show that the weaker property is satisfied in valid B-trees. Nevertheless, generating the most general lemmas usually was highly useful. A good example is Lemma 2.3.5, where generalizing for up to $4k + 1$ input elements yielded the reusability of $node_i$ for merging two nodes in deletion. Another use case was the applicability of weaker properties even when temporary violations of the general invariant were observed during deletion, such as in the *del* function in Listing 2.5.

During this work, a significant difference in the effort to prove invariants using the standard set interface and the set interface by inorder has emerged. The proof of the standard set interface required a number of additional functions comprising 36 lines of code. The bigger part however is made up by the proofs required. Including additional auxiliary lemmas, the whole proof of sortedness and correct set operations required a staggering over 1900 lines of proof. The proof was complex and hardly readable due to the extensive use of efficient solvers. This is opposing 5 lines for the inorder and sorted function and 483 lines of proof required for the inorder approach, which is mostly composed of easily understandable chains of equations. Of course the invariance preservation proof was overall simplified by the detour via the functional specification.

We realize that the rigorous verification of B-trees or the related B^+-trees is a hard problem. Both Malecha [Mal+10] and Gidon [ESR15] report the validity of the tree shape and invariants such as node size or sorting to be the most difficult properties to prove. In our approach, a valid tree shape was directly given by the refinement property of the algebraic B-tree type. The other abstract invariants could be reasoned about in the context of the abstract specifications and as such required a number of intermediate, but obvious lemmas that were comparatively straightforward to obtain. The comparison between the three proof approaches in Figure 4.2 shows differences both with respect to the amount of time invested and the amount of manually provided lines of proof.

In addition to the effort related benefits, the abstract formalization allowed to verify the implementation of a binary split based node navigation algorithm that was not yet implemented in any other verified approach. It seems that the detour via a rigorously

[7] The proof integrates TVLA and KIV, and hence comprises explicitely added rules for TVLA (the first number), user-invented theorems in KIV (the second number) and "interactions" with KIV (the second number). Interactions are i.e. choices of an induction variable, quantifier instantiation or application of correct lemmas. We hence interpret them as each one apply-Style command and hence one line of proof.

	[Mal+10]$^{+}$	[ESR15]$^{+d}$	Our approach
Functional code	360	-	324
Imperative code	510	1862	170
Proofs	5190	350 + 510 + 2940[7]	2753
Timeframe (months)	-	6+	4

Figure 4.2: Comparison of (unoptimized) Lines of Code and time investment in related mechanized B-tree verifications. The marker $^{+}$ denotes implementations of B^{+}-trees and ^{d} denotes that the implementation additionally verifies deletion operations.

implemented functional implementation yields less complex proofs. It should however be noted that the B-tree implementation might in itself be simpler to reason about than B^{+}-trees, which includes several optimizations regarding storage of nodes and available pointers.

4.3 Future work

It should be noted that further, in contrast to related work and common implementations, this work implements B-trees as concretizations of sets. Usually B-trees are rather in use as map data structures, mapping the unique primary keys of database tuples to stored data. Future work could adjust the code and proofs given here to show that similar operations on a similar structure satisfies the map interface. The main difference would then be that not the indices are the main elements of the nodes but pairs of the index and the corresponding data or a similar implementation of a singleton map. We expect this to be not more than manual work, but to cause no specific new issues.

It should be noted at this point that the obtained code is not yet truly optimized with respect to minimal memory allocation and access. For that, some further refinements are necessary. For example, the imperative $node_i$ function requires temporarily allocating an array that has enough space to contain all elements that are supposed to be merged or split. Optimizing it would maybe give a function that takes as an argument two arrays and redistributes elements between them, or an array and an element that is to be inserted. This naturally increases the required amount of additional, specialized heap rules and makes the proofs more complex. How to gradually improving the efficiency of the implementation without greatly increasing the complexity of correctness proofs is a topic for further research.

However it should be noted that a transfer of the results in this work may be difficult for the related B^{+}-tree, providing another subject for future research. B+-trees usually

contain pointers to the next leaf on the lowest level, a property that is potentially harder to represent on a functional level than in imperative code. It might be necessary at that point to introduce additional invariants directly on the imperative level.

List of Figures

Bibliography

[Bie+20] J. Biendarra, J. Blanchette, M. Desharnais, L. Panny, A. Popescu, and D. Traytel. *Defining (Co)datatypes and Primitively(Co)recursive Functions in Isabelle/HOL*. 2020.

[BM72] R. Bayer and E. M. McCreight. "Organization and Maintenance of Large Ordered Indices." In: *Acta Informatica* 1 (1972), pp. 173–189. DOI: 10.1007/BF00288683.

[Com79] D. Comer. "The Ubiquitous B-Tree." In: *ACM Comput. Surv.* 11.2 (1979), pp. 121–137. DOI: 10.1145/356770.356776.

[Cor+09] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. ISBN: 978-0-262-03384-8.

[ESR15] G. Ernst, G. Schellhorn, and W. Reif. "Verification of B^+ trees by integration of shape analysis and interactive theorem proving." In: *Softw. Syst. Model.* 14.1 (2015), pp. 27–44. DOI: 10.1007/s10270-013-0320-1.

[Fie80] E. Fielding. *The Specification of Abstract Mappings and their Implementation as B+ Trees*. Tech. rep. PRG18. OUCL, Sept. 1980, p. 78.

[FZR92] M. J. Folk, B. Zoellick, and G. Riccardi. *File structures - an object-oriented approach with C++*. Addison-Wesley-Longman, 1992. ISBN: 0-201-55713-4.

[HB20] F. Haftmann and L. Bulwahn. *Code generation from Isabelle/HOL theories*. 2020.

[Knu98] D. E. Knuth. *The art of computer programming, , Volume III, 2nd Edition*. Addison-Wesley, 1998. ISBN: 0201896850.

[Lam19] P. Lammich. "Refinement to Imperative HOL." In: *J. Autom. Reason.* 62.4 (2019), pp. 481–503. DOI: 10.1007/s10817-017-9437-1.

[Mal+10] J. G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. "Toward a verified relational database management system." In: *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*. Ed. by M. V. Hermenegildo and J. Palsberg. ACM, 2010, pp. 237–248. DOI: 10.1145/1706299.1706329.

[Mon91] P. T. Montague. "A Correctness Proof for Binary Search." In: *Comput. Sci. Educ.* 2.1 (1991), pp. 81–89. DOI: 10.1080/0899340910020106.

[MTH90] R. Milner, M. Tofte, and R. Harper. *Definition of standard ML*. MIT Press, 1990. ISBN: 978-0-262-63132-7.

[Mün21] N. Mündler. *A Verified Imperative Implementation of B-Trees: Appendix*. `https://github.com/nielstron/btrees`. 2021.

[Nip16] T. Nipkow. "Automatic Functional Correctness Proofs for Functional Search Trees." In: *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings*. Ed. by J. C. Blanchette and S. Merz. Vol. 9807. Lecture Notes in Computer Science. Springer, 2016, pp. 307–322. DOI: 10.1007/978-3-319-43144-4_19.

[NK14] T. Nipkow and G. Klein. *Concrete Semantics - With Isabelle/HOL*. Springer, 2014. ISBN: 978-3-319-10541-3. DOI: 10.1007/978-3-319-10542-0.

[NPW02] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Vol. 2283. Lecture Notes in Computer Science. Springer, 2002. ISBN: 3-540-43376-7. DOI: 10.1007/3-540-45949-9.

[Ode+15] M. Odersky, P. Altherr, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. McDirmid, S. Micheloud, N. Mihaylov, M. Schinz, L. Spoon, E. Stenman, and M. Zenger. *An Overview of the Scala Programming Language (2. Edition)*. Tech. rep. Mar. 2015.

[Rey02] J. C. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures." In: *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*. IEEE Computer Society, 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817.

[ST08] A. P. Sexton and H. Thielecke. "Reasoning about B+ Trees with Operational Semantics and Separation Logic." In: *Proceedings of the 24th Conference on the Mathematical Foundations of Programming Semantics, MFPS 2008, Philadelphia, PA, USA, May 22-25, 2008*. Ed. by A. Bauer and M. W. Mislove. Vol. 218. Electronic Notes in Theoretical Computer Science. Elsevier, 2008, pp. 355–369. DOI: 10.1016/j.entcs.2008.10.021.

[Wen20] M. Wenzel. *The Isabelle/Isar Reference Manual*. 2020.