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:	January 29, 2021

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

Munich, January 29, 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 builds the foundation of many modern database systems. Since its first definition in [BM72] variations have been developed. More complex operations such as deletion are famously spared by many standard references. The original definition and modern variants are examined and compared with mechanically verified versions of the data structure. A functional version that is promising to analyse in the Isabelle/HOL framework is then implemented followed by a proof of its correctness with respect to refining a set on linearly ordered elements. A proof of the logarithmic relationship between height and number of nodes based on the original paper was given to point at the efficiency of operations on the tree.

From the functional specification an imperative specification derived that stores and retrieves nodes from an abstracted heap. The specification is shown to again refine the functional program using the separation logic utilities for the Isabelle Refinement Framework from [Lam19]. This way an imperative implementation of the B-Tree data structure is obtained that is proven to correctly handle insertions and element checks.

Contents

1 Introduction

An important topic in computer science is the study of the *functional correctness* of an algorithm. It states whether an algorithm actually solves a problem in the intended way. This topic becomes especially interesting when it can be applied to directly executable code and is machine checked. Unfortunately with both desirable additions, the tasks becomes significantly more complex. In an abstract specification we may abstract away many consideration an actual program has to be concerned with and are still left with the obligatino to show non-trivial properties. Furthermore for the concrete code we have to reason about specific low level decisions such as memory allocation or the eligibility of reusing variables. In this thesis we provide a computer assisted proof 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 introduce B-Trees and common variations on them. The variation that is most promising for our approach is chosen to be implemented. 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 is equivalent with the implementation of an abstract Set-interface, that requires retrieval, insertion and deletion operations. From there, 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 gives freedom 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

We realize that the verification of B-Trees or the related B^+-Trees is a hard problem. The first rigorous approaches at it to our knowledge are the works by Fielding [Fie80] and Sexton [ST08]. Both were not machine checked but shed light on both techniques applied in this work. Fielding approaches the verification by iterative refinement of an abstracted specification which we employ in chapter 2. Sexton uses the sophisticated tool of separation logic to reason directly on imperative specification, in a similar manner to our specification in chapter 3. In the work of[ESR15] an imperative implementation is also directly verified by combining interactive theorem proving with shape analysis. The main recursive procedures interactively 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 can be found in [Mal+10], with the YNOT extension to the interactive theorem prover Coq. To our knowledge, the preceding works comprise all published attempts at verifying implementations of B-Trees or the related data structure as B$^+$-Tree. No such verification has been found that explicitly covers the implementation of functional B-Trees.

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 ML, which influences the syntax of functional programs written in it. Isabelle code is portioned in so called theories, roughly being equal to a module in common languages. A theory consists of the specification of datatypes, typed functional programs and theorems with proofs.

1.1.1 Notation and proofs in Isabelle

Functions, predicates and the like are all expressed in a functional manner. **definition** denotes classical definitions and **fun** is used for recursive definitions. The latter is only a valid definition if it incorporates a proof of termination. Usually this proof is done automatically by the system. If a function evaluation does not terminate for all inputs it is denoted with **partial_function**. The term **abbreviation** is used to define simple shorthands for more complex expressions, similar to a macro.

Lemmas, theorems and the like can be expressed in a similar manner as they would be written in mathematical textbook. They begin with i.e. **lemma**, followed by an expression or predicate and a proof that it holds. Wherever possible and readable, we will try to reflect the actual syntax of the Isabelle system. 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. 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. In this work we will not present the proofs as written in the actual proof files but rather outline the structure of the proofs.

The system provides a number of proof methods, based on different manipulation tools such as logical reasoning or simplification. When mentioning *automatic* proofs, we mean a proof that comprises very few ($\leq$ 5) apply style invocations of proof methods is meant. 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.

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 on another list. As usual in ML like languages, we may deconstruct an argument to a function by pattern matching to this construction. 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*. 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)
```

It is possible to specify functions, predicates and theorems with respect to some abstracted function of which only certain properties are known, resulting in so-called **locales**.

Further details on notation, proof techniques and more in Isabelle/HOL may be found in [NPW02].

1.2 The B-Tree Data Structure

B-Trees were first proposed by Bayer et al. in [BM72], as a data-structure to efficiently store and retrieve indices stored on storage devices with slow memory access. They are n-ary balanced search trees, where each node contains many indices. The number of subtrees and indices in each node is defined by the *order* of the tree. B-Trees are a generalization of 234-Trees and a specialization of (a,b)-Trees A common variation is the B^+-Tree, where the inner nodes only contain separators to guide the recursive search but all data is stored in the leaves. [Com79] Note that usually B-Trees are supposed to

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

implement a map-like structure, where each element is an index together with a pointer or some arbitrary additional data. However for the sake of keeping the implementation and proofs simple, and first showing the validity of the approach in this case, we say that B-Trees store some linearly ordered values that themselves form the elements of an concretized set.

1.2.1 Definitions

Every node contains a list of *keys* (also *separators*, *index elements*), and *subtrees* (*children*), that refer to further B-Trees. 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 [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. This is possible to accomplish 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.

Order In general terms, the property of *order* ensures a certain minimum and maximum number of subtrees for each node. However, as pointed out in [FZR92], the property is defined differently in the literature. For the purpose of this work, the original definition by Bayer et al. 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 alternative of defining the nodes to have between $\lceil \frac{k}{2} \rceil$ and k children (as proposed in [Knu98]), involves cumbersome *real* arithmetic that unnecessarily complicates mechanized proofs. Sticking to the original definition is further supported 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.

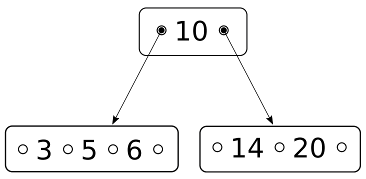

Figure 1.1: A small balanced, sorted B-Tree of order 2 and height 2 containing 3 nodes and 6 elements. Circles represent nodes, where empty circles are leafs and filled circles point to the corresponding subtree.

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 without children. This is close to the usual approach in functional programming, and will yield 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.

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. However as it should not have more than $2k = 0$ elements, this constraint cannot be satisfied. Requiring positive k is consistent with the definitions in the literature consulted [BM72; Com79; Cor+09]

A simple example B-Tree may be seen in Figure 1.1. Nodes and Leafs are represented as circles, were the former points to the subtree that a node represents and the latter is left empty.

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, the tree in between is the correct for recursion. 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 case that there is enough space left, the element is simply placed at the correct position in the list of separators. However if 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 subsection 2.3.4 how this can be elegantly expressed in a functional specification.

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*. However, 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 out for increasing the size of 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 are merged to form a new, bigger node of valid order.

Following the description of [BM72], as done by [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 [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 item, the probability for another underflow at the next deletion is higher than if several elements are copied over. Further, the node to merge with has to be read from memory anyways. The cost of moving around up to k elements that already reside in memory is hence negligible compared to the cost of another underflow.

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 required 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 due to the fact that every node reserves the storage of $2k$ keys and separators but by definition always contains at least k elements.

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

2 Functional B-Trees in Isabelle

Proving higher level properties of data structures tends to be easier on a functional level than on the 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 or non-persistence.

2.1 Basic Definitions

As discussed in chapter 1, we define B-Trees recursively to either hold a list subtrees and keys or be a completely empty Leaf. The only room for interpretation is how to actually store the subtrees and separators. For Trees of variable but bounded size, an explicit constructor may be given as for the 234-Trees in [Nip16]. However since the number of elements in B-Trees is technically unbounded (as k is not bounded) we need to use an intermediate data structure, a list, to store them.

Inspired by the design in [Mal+10], we will store subtrees and keys in pairs, such that the left subtree of a key is the left element in a tuple and the key is the right element. By interleaving the subtrees and separators, expression of invariants on the tree is possible without the explicit use of indices into lists. Since the list of trees is exactly one longer than the list of keys, the last tree is added to the node in a special position. The definition follows and a visualization may be seen in Figure 2.1.

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

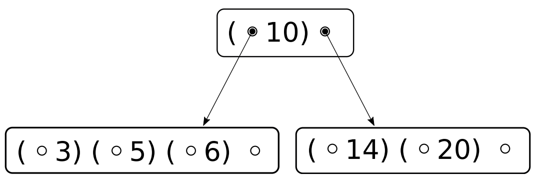

Figure 2.1: The B-Tree from figure Figure 1.1. Subtrees and separators that lie in pairs in the implementation are surrounded by brackets.

The construction leads to the fact that it is easiest to relate any separator to the subtree on the left, as this subtree is in the same tuple. If a subtree on the right is required we may either match the list to the right with the list constructor or choose the last tree.

The balancedness of a B-Tree is defined recursively, making use of an intuitively defined height.[1] There is a choice regarding how to obtain the number to which the height of all subtrees has to be equal. Either we fix the balancedness of a tree on 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 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 + (fold max (map height (subtrees ts)) (height t))

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

Further the order of the trees needs to be formally defined. As discussed in subsection 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 "order$_k$ t" for convenience (likewise for "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) ∧
```

[1]Note that the actual implementation of height is slightly different, however this is of no importance in any proofs but rather notationally convenient.

```
    (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) = (
  (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 concatenation of all elements of the tree in in-order traversal. The concatenation function *concat* on lists of strings is employed to express the resulting string. Note that 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, we can only later cover up this expression by using abbreviations. In mathematical terms in the following, we will not distinguish between lists, pairs or trees when talking about their inorder representation. The matching function 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 resprect to $<$) in ascending order. This definition is very compact and brings for another benefit pointed out by Nipkow et al in [Nip16]. Many properties of search trees follow intuitively by considering the inorder view on the tree. Usually, it invariant for all operations (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 $k > 0 \Longrightarrow \mathrm{btree}_k\, t := \mathrm{bal}\, t \wedge \mathrm{order}^r_k\, t \wedge \mathrm{sorted}(\mathrm{inorder}\, t)$

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 in the first paper describing B-Trees, the height of B-Trees is logarithmic with respect to the number of nodes of the tree. [BM72] The 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 $\mathrm{order}_k\, t \wedge \mathrm{bal}\, t \longrightarrow$

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

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

From Theorem 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 bounds obtained by Bayer et al. in [BM72], except for the fact that we have generalized the equation, incorporating whether t is a tree or not.[2]

Theorem 2.2.2 $\mathrm{order^r}_k\, t \wedge \mathrm{bal}\, t \wedge k > 0 \longrightarrow$

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

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

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.

These bounds are sharp. We prove this by providing functions that generate exactly those trees that satisfy the requirements of B-Trees, have a given height and satisfy the inequality with equality in Listing 2.1. 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 2.2.3 $t_f := \text{full\_node}\, k\ a\ h \wedge t_s := \text{slim\_node}\, k\ a\ h \longrightarrow$

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

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

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

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

Theorem 2.2.4 $k > 0 \wedge t_f := \text{full\_tree}\, k\ a\ h \wedge t_s := \text{slim\_tree}\, k\ a\ h \longrightarrow$

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

2.3 Set operations

With the definition of B-Trees come a number of operations that allow for set-like operations on the trees - queries whether an element is contained, 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

An implementation *'a t* of the sets of elements of type *'a* is required to provide the following operations

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)
```

- *empty :: 'a t*
- *isin :: 'a t ⇒ 'a ⇒ bool*
- *insert :: 'a ⇒ 'a t ⇒ 'a t*
- *delete :: 'a ⇒ 'a t ⇒ 'a t*

Additionally, an *invariant :: 'a t ⇒ bool* has to be supplied that stays unchanged by all above operations and is satisfied by the *empty* element. For this work, using the definition from Listing 2.1, we consider *'a t = 'a btree*.

The standard approach is to show that we can provide functions for the B-Tree that have the same effect as insertion, deletion or testing on contained elements in abstract sets obtained by *set :: 'a t ⇒ 'a set* that contains the same elements. 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 [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 in [Nip16]. That work is promising, as their approach yielded automatic proofs for 2-3-Trees and 2-3-4-Trees, which are specializations of B-Trees. Their modified Set interace requires instead of the *set* abstraction a *inorder :: 'a t ⇒ 'a list* abstraction.

The invariant and inorder functions for this set interface follow directly from the definition of B-Trees. Hence, we specify $k > 0 \wedge \mathrm{order}^r_k\, t \wedge \mathrm{bal}\, t$ as the invariant of B-Trees use as inorder invariant the concatentation function from section 2.1. The interface is then implemented by providing set operations and the proofs of invariant preservation.

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. The following sections will describe the implementation of the non-trivial set operations.

2.3.2 The split-Function

Naturally the set operations are defined recursively on the nodes of the tree. Since each node contains a non-trivial number of elements, a function to navigate to the correct separator and subtree is central to all operations.

We call this function *split*-function, determining the position in the list of separators and subtrees. At the "correct" 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. At this position, either the separator is equal to the desired

```
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 []
```

Figure 2.2: A function implementing the split function specifications. It scans linearly through the list, returning the first tuple where the separator or subtree could potentially contain the value *x*

value or we need to recurse into the left subtree. This approach is similar to the one found in the work of Malecha and Fielding [Mal+10; Fie80].[3] Usually however, it is integrated into the set-operation by a 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 *some* kind of function is required that correctly splits the key-value list and an example is given in Figure 2.2. 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 a very efficient splitting function, e.g. based on binary search.

Therefore all set functions are defined based the following requirements:

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

[3]In Fieldings approach the corresponding function is called *index*. Malecha calls it *findSubtree*.

Listing 2.2: The *isin* function

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

Described in natural language, the split function should return two sublists 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.

The benefit of restricting the sortedness of the list only to its separators simplifies the proofs of showing that a certain function fulfils the split abstraction. We really only need to consider the separators on each level and not the subtrees themselves. Showing that the function works on a sorted inorder list would only introduce an additional step to follow that the separators are sorted as well. Having fixated this abstraction we only need to follow this fact once and can relieve the writer of split functions of the need to do so.

The abstraction was formulated in a locale. Inside the locale we have access to exactly the specified split function, without knowing anything about the composition of that function.

2.3.3 The *isin* operation

The simplest operation required in the Set interface is the *isin* function. The definition in Listing 2.2 also shows exemplary usage of the split function. In case the right part of the split list is non-empty, we check the element at that level or recurse in the given subtree. Otherwise, we may directly recurse to the last tree in the node.[4]

[4]Since this function recurses on the split function, in order to show that this function terminates, we need to tell 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.

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 kind of lemmas are of special interest as they specialize an idea proposed in [Nip16] and similar lemmas are used for the correctness proofs of insert and delete alike.

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 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 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. However, rather than manually checking whether the lowest level was reached, we recurse until we reach a Leaf node. Inserting into it will lead to an overflow that can be handled by the lowest internal nodes in the same manner as for other internal nodes.

To modularize insertion, the check for the overflow of a nodes list has been extracted to the function $node_i$. It returns data in a new datatype called up_i, which 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. This construction is useful for handling overflow in a functional context and appears similarly in [Nip16].

The *ins* function is a recursive helper function, that recurses down into the correct leaf node for inserting the desired element. The recursive call may have caused an overflow in a lower node. If an overflow occured, an additional element and new subtree is inserted into the current nodes list with the help of $node_i$, the result of which is again of type up_i to indicate to the parent whether overflow occurred. If no overflow occured we may directly return T_i of the current node to indicate no overflow.

Finally, the *insert* function calls the *ins* helper function and transforms the result into a valid B-Tree.

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 tree.

Note that for the above proofs, a separate notion of height, balancedness and order was introduced for the up_i datatype. Height and balancedness are defined such that the property is invariant to splitting and merging of nodes. For example, "height $ts\ t =$ height_up$_i$(node$_i$ $k\ \ ts\ \ t$)". Note that this is not the same as applying the original functions to the tree obtained by tree$_i$. Order of up$_i$ elements however 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 the balancedness invariant requires as additional lemma that the height is 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 and with it 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. After all, the operations are never directly affecting the height of any tree, and the all trees generated by splitting a node comprise trees that had been there before, now simply distributed along to node in the same level.

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)
```

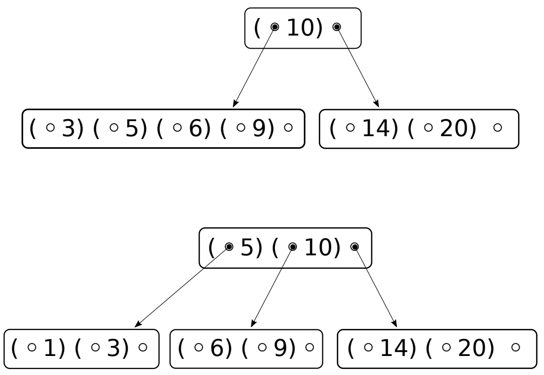

Figure 2.3: The tree from Figure 2.1 after successive insertion of 9 (top) and 1 (bottom).

Lemma 2.3.4

$$\text{height}\, t = \text{height}(\text{ins}_k\, x\; t)$$
$$\text{bal}\, t \Longrightarrow \text{bal}(\text{ins}_k\, x\; t)$$

Now for the proof of preserving the order 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* 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. The argument hardly changes comparing the upper bounds $2k + 1$ and $4k + 1$, but with the former upper bound it 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 $\text{order}_k\, t \Longrightarrow \text{order}_k(\text{ins}_k\, x\; t)$

Since the invariant on B-Trees only requires a root order of k on the root, we additionally derive versions of the invariants for root order. They are straightforward however as all trees in the inductive case have order k, for which preservation was already proven. Putting things together, we obtain preservation of the invariants.

Theorem 2.3.7 $\text{order}^r_k\, t \wedge \text{bal}\, t \Longrightarrow \text{order}^r_k(\text{insert}_k\; x\; t) \wedge \text{bal}(\text{insert}_k\; xt)$

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 in case of 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 2.3.8 $\text{sorted}(\text{inorder}(\mathit{Node}\; ts\; t)) \wedge \text{split}\, ts\; x = (ls, rs) \Longrightarrow$

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

Lemma 2.3.9 $\text{sorted}(\text{inorder}(\mathit{Node}\; ts\; t)) \wedge \text{split}\, ts\; x = (ls, (sub, sep)\#rs) \wedge sep \neq x \Longrightarrow$

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

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 2.3.10 $\text{sorted}\, xs \wedge x \in \text{set}\, xs \Longrightarrow \text{ins}_{\text{list}}\, x\; xs = xs$

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

Theorem 2.3.11 $\text{order}_k\, t \wedge \text{bal}\, t \wedge \text{sorted}(\text{inorder}\, t) \Longrightarrow$

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

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
    )
```

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.

The other detail is regarding the *split_max* function defined in Listing 2.5. Here some freedom exists whether to swap with the maximum lesser or the minimum greater element in the tree. Since 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, it is significantly easier to obtain the maximum of the left subtree than the minimum of the right subtree. Therefore we will always swap inner separators with the former element.

The most interesting property for the rebalancing operations is the order property, which is meant to be restored after potential underflows. It is important to note here that we cannot guarantee that at least one element remains in the node that is passed back upwards. However 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. 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* of k, order^a_k.

Lemma 2.3.12 *If all subtrees in ls, rs are of order k and of sub and t, only one may have order^a k and the other has order k, and all subtrees have equal height, then*

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

For this proof, we re-use Theorem 2.3.5, as two nodes of order k will always contain a maximum of $4k + 1$ separators. The case that the subtree and separator list becomes zero is exactly when a root with minimal size underflows. 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.

From this fact follows further, that for the results of all intermediate functions in the deletion process, $\text{order}^a\ k$ of the results can be guaranteed. Since the last subtree in the result of deletion has order k, and for more than one subtree the node has root order, the result of *delete* always has at least $\text{order}^r\ k$.

The proofs for balancedness invariance follow similarly to the ones in subsection 2.3.4 by first showing height invariance. The properties follow using the associativity and

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)
```

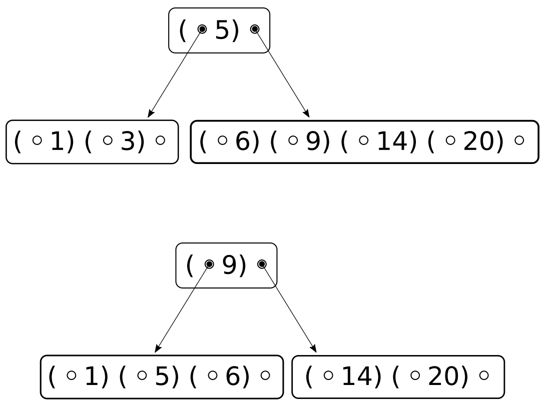

Figure 2.4: The tree from Figure 2.3 after successive deletion of 10 (top) and 3 (bottom). Note how the final result differs from approaches that would only steal single elements from neighbors to hanlde underflow.

commutativity of the maximum operation. The proof of height, balancedness and order are interleaved among the proofs for the intermediate functions, especially *split_max* and *rebalance_middle_tree*, which are only well defined for balanced trees and *split_max*, which requires at least one element in the node list, an equivalent to order^r k.

With the basic functionalities 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 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 that it might be beneficial to know the input to have normal order, this is not the case. Since the rebalancing functions do only require almost order k to return normally ordered trees, the resulting list will always be sufficient.

Lemma 2.3.13 $k > 0 \land \text{order}^r_k\, t \land \text{bal}\, t \implies \text{order}^a_k(\text{del}_k\ x\ t) \land \text{bal}(\text{delete}_k\ xt)$

Putting together the proofs on balancedness and order, we obtain the fact about the invariant preservation of the deletion function that is required by the set interface.

Theorem 2.3.14 $k > 0 \land \text{order}^r_k\, t \land \text{bal}\, t \implies$

$$\text{order}^r_k(\text{delete}_k\ x\ t) \land \text{bal}(\text{delete}_k\ xt)$$

Note that now (as opposed to insertion) $k > 0$ is required. The main reason is that, while it was not too important for most of the proofs until now, B-Trees do only work for positive k. 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 2.3.11 and induct over the execution of the *del*-function.

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

The reason the rebalancing functions 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 results can be easily simplified, even without recursive calls, the property follows automatically.

Lemma 2.3.15 *All subtrees have the same height* $\Longrightarrow$

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

Considering the function split_max, the standard approach would require showing tideously that the element returned is in fact the maximum of the whole tree 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 2.3.16 *If t has more than two subtrees, and the last two are equally high, then*

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

This fact follows easily by induction on the computation of split_max, using Theorem 2.3.15.

The reason that the proof of these functions follows so easily lies in the intention behind the definition of both functions. Together with the fact that the remaining tree is balanced and fulfils a certain order property, the inorder view is not meant to change. We obtain the same elements, in the same order, just now in a configuration from which we can obtain a new, valid, B-Tree.

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

Theorem 2.3.17 $k > 0 \wedge \text{order}^r_k\, t \wedge \text{bal}\, t \wedge \text{sorted}(\text{inorder}\, t) \Longrightarrow$

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

With this, the implementation the set interface by our implementation of B-Trees is proven correct. The next step towards imperative B-Trees is to implement an imperative refinement of the 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 do allocate memory for 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, since these facilities are extensible, code extraction is not limited to these languages.

3.1 Refinement to Imperative/HOL

The proofs of correct behavior for our imperative code are stated separation logic [Rey02], an extension of Hoare Logic. It was formalized in Isabelle/HOL by [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 specified as formulae that hold for specific heap states. The basic assertions used in this work follow.

- *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

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* $\langle\ P\ \rangle\ c\ \langle \lambda r.\ Q\ r\ \rangle$
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 modified heap

A simple example Hoare Triple is $\langle$ id_assn $a\ b\ \rangle$ return $b\ \langle \lambda r.$ id_assn $a\ r\ \rangle$. In the refinement process, we will use more complex relationship assertions on the input b to some object bi and show that the value r returned by the imperative program satisfies a relationship to the output of the refined function applied to bi. Further we use the shorthand notation $\langle\ P\ \rangle\ c\ \langle \lambda r.\ Q\ r\ \rangle_t$ which stands for $\langle\ P\ \rangle\ c\ \langle \lambda r.\ Q\ r\ * true\ \rangle$. Since *true* holds for any heap, this part of the statement 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 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[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 a chunk of data into a page of main memory, nodes of roughly this size make the most efficient use of the loading times. [BM72] Therefore the data structure to store the node content should reserve a fixed amount of data, independent of the actual number of keys and children.

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

typedef *'a pfarray = 'a array × nat*

definition is_pfa **where** *c l* ≡
    $\lambda(a,n).\ \exists_A\ l'.\ a \mapsto_a l' * \uparrow(c = \text{length}\ l' \wedge n \leq c \wedge l = (\text{take}\ n\ l'))$

where c is the *capacity* of the array, that is the actual 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 following operations to replace complex composite operations based on list construction and concatenation.

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 the starting position 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, 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

definition pfa_set ≡ λ(a,n) i x. do {
    a ← Array.upd i x a;
    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 {
  a′ ← array_shr a i 1; (∗ shifts elements to the right by 1 ∗)
  a′′ ← 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′′
}

. . .
```

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.[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. 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 which is why they are not listed listed in great detail. 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 3.1.2 $k \leq \text{length}\, s \wedge (\text{length}\, s - k) \leq dn \Longrightarrow$

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

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 that changed is the content of *di*.[3] The *take* function was refined differently. The cheapest and in our context consistent option is reducing the number of elements that are considered part of the represented list 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 3.1.3 $k \leq \text{length}\, s \Longrightarrow$

[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.

[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.

$\langle$is_pfa *sn s si*$\rangle$ pfa_shrink *si k* $\langle\lambda si'.$ is_pfa *sn* (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

With refinements for all basic components in the B-Tree set operations, 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, only storing pointers to other nodes in the element list. 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. The physical B-Tree datatype does hence only need to cover the case of actually being an inner node.

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

Note the similarity to Listing 2.1, only that *btree* was replaced by *btnode ref option* and *list* was replaced by *pfarray*.

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.

```
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*) $\times_a$ id_assn)

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 a list are refined to their physical counterpart. This fact is expressed by the list assumption term in the assumption. Second, the list itself is refined to an array. This is expressed in the term about the partially filled array relationship. Since the list assumption makes recursive use of the B-Tree assumption, this is however a very convenient way of expressing the desired relationship.

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

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. Imperative split functions should be efficient and as such not actually split an array in half and return it the way the abstract function did. Rather than that the function should return the index of the subtree in which should be recursed. The refinement condition derived from this relationship is as follows.

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

A very useful alternative statement of the relationship follows automatically.

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

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 implement the recursive function from Figure 2.2 but we embark on the exercise of using a while loop. The implementation in Listing 3.2 is very basic but good 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.

This function behaves quite different to the abstract linear split that was implemented before. Therefore, proving that it refines the abstract function requires a small detour.

Listing 3.2: The imperative linear split

```
definition lin_split :: ('a::heap × 'b::{heap,linorder}) pfarray ⇒ 'b ⇒ nat Heap where
  lin_split ≡ λ (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
  }
```

We first express and show the most basic, non-trivial result of the function. With the help of that we may then show that the result is the same as the result of applying the abstract function.

We begin with stating a simple matching Hoare Triple for the function. It follows by supplying the following loop invariant:

- $\forall j < i.\, \mathrm{snd}(xs!j) < p$: 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
- $\mathrm{is\_pfa}\, c\ xs\ (a, n)$: the array stays untouched by the operation

The measure for the loop is the difference $n - i$ which decrements by one in each iteration and stays positive. From this we obtain the Hoare Triple of the whole linear split.

Lemma 3.2.2

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

Listing 3.3: The imperative binary split

```
definition bin_split :: ('a::heap × 'b::{heap,linorder}) pfarray ⇒ 'b ⇒ nat Heap
where
  bin_split ≡ λ(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'
```

For unexciting reasons, this post condition actually suffices to imply that there exists a split relation to the result of an abstract split function. Specifically, we use the abstract split function from Figure 2.2. 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. [4] This will simplify proofs for alternative imperative split functions. With this assurance we approach the binary split.

The binary split in Listing 3.3 conducts a binary search on the list of elements. Rather than walking through the array, it narrows down a windows on the array that becomes smaller with every iteration. In the final iteration, the windows must have narrowed down to a single element, which is the result.

A detailed analysis of binary search algorithms may be found in [Mon91], which has served as an orientation for the derived algorithm. The main idea is that the algorithm iteratively narrows a window on the array in which the desired element may lie. In each step, the middle element of that window is inspected and either desired element

[4] Another, albeit more theoretical, reason for excitement is that now we can specify an imperative B-Tree specification without any reference to the abstraction. We simply always use the well known split function in the abstraction and only require a post condition similar to the one just shown on the imperative split function.

is returned or the algorithm recurses into either left or right remaining subwindow.

Since pairs as list elements add another layer of complexity for the simplifier, a binary split algorithm on lists that contain only single elements has been derived first. After several small corrections, it was proven to be correct. The split function for lists of pairs was then defined and could be tackled by the same approach, using the below 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 i are smaller than the partitioning element.
- $h < n \rightarrow \text{snd}(xs!h) < p$: If the upper bound is strictly less than the length of the array, the element it points to is greater than the partioning element
- $l \leq h$: 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
- $\text{is\_pfa}\, c\ xs\ (a, n)$: again, the array stays untouched by the operation

In this case the window that is considered by the binary split provides for the loop measure, as $h - l$ is decreased by at least one in each step. Proving that this was the case was not the main issue, but more to find 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 as the functional version of a binary split is much harder to specify and analyse than a linear split. The additional requirement for the binary split is naturally that the list of separators must be sorted.

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

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

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 3.4: 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. Pointers need to be dereferenced, modified 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 3.4 function should bare no surprises. Note that it makes use of the function *imp_split* which is an arbitrary split function that has the Hoare Triple specified in subsection 3.2.1.

The Hoare Triple to argue that this is a refinement of the abstract operation follows. The convention for variables that refer to refined datatype 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* that represents it is called *ti*. To simplify notation, we simply assume that the set operations such as *isin* are overloaded and refer to abstract or imperative version, depending on the parameter type.

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 only help that the separation automation tool required is a manually specification of the instantiation for existential quantifiers for applying the inductive hypothesis.

3.2.3 Insertion

The imperative implementation and derived heap rule of the insertion 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 ins function can be seen in Listing 3.6. The function includes some optimizations, such as not calling *node*$_i$ if the node is not overflowing. This is an optimization as otherwise *node*$_i$ in Listing 3.5 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*$_i$.

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

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. Adding a lemma on the equivalence between abstract and imperative *node*$_i$, we finally obtain a lemma for *ins* by induction on the computation of the abstract *ins* function.

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

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

Listing 3.5: 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 ← ref (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 ← ref (Btnode a'' sub);
          r ← ref (Btnode b' ti);
          return (Up_i (Some l) sep (Some r))
        }
      }
    }
```

Listing 3.6: 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 {
                kvs' ← pfa_insert (kvs a) i (lp,x');
                ap := (Btnode kvs' (last a));
                return (T_i apo)
            } else do {
              kvs' ← pfa_insert_grow (kvs a) i (lp,x');
              node_i k kvs' (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 {
        kvs ← pfa_init (2*k) (l,a) 1;
        t' ← ref (Btnode kvs r);
        return (Some t')
      }
}
```

The insertion function incorporates the function tree_i in much the same way. Here the proof of equivalence followed automatically. With the hoare rule for *ins*, we obtain the final desired lemma on the imperative insertion.

Lemma 3.2.7 $k > 0 \wedge \text{sorted}(\text{inorder}\, t) \Longrightarrow$

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

Note how we have the additional assumption that k be 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

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.

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 and hence it is highly important to strongly tie the refined and imperative version in order to discharge impossible cases.

4 Conclusions

In this work, we obtained an imperative implementation of B-Trees that is capable to handle queries whether a key is contained and queries to insert new keys. 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 it functionally correct with the help of an abstract specification that is refined by the imperative code. The abstract specification fully implements a Set interface in Isabelle, also including deletion of elements.

4.1 Working in Isabelle

One main source of frustration is finding the correct functions and lemmas to apply for proofs. While in natural proofs a certain kind of function and corresponding equalities can be expected to be known by the reader. However for machine checked proofs, as the only meaningful way, all used lemmas need to be provided. Now usually the Isabelle/HOL provides tools like sledgehammer or **find_theorems** in order to find useful lemmas for the concrete situation. However each have their own drawbacks.

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. This 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 not perfectly match the required lemma anymore. 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 standart function *concat* that simplifies to the above and has useful lemmas for catenation and splitting. It was only discovered close to the finalization of the thesis when more standard terms for employed functions were searched 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 existing lemmas may be employed.

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.

The actual proofs in this work have been written to use only minimal requirements on the input. For example in Theorem 2.3.16, rather than requiring balancedness and order k for $k > 0$ on the full tree, 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 a 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 that way. 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 turned out to be very useful, for example in Theorem 2.3.5, yielding 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.

4.2 Future Work and Comparisons

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.

Both Malecha and Gidon report the proof for invariants to be the most difficult properties to prove. In out approach, we were able to reason about the abstract invariants in the context of an abstracted specification, significantly simplifying proof reasoning. A difference can be observed in both with respect to the amount of time invested and the amount of manually provided proofs.

Finally it should be noted that this work implements B-Trees as concretizations of sets.

	[Mal+10]$^{+}$	[ESR15]$^{+d}$	Our approach
Functional code	360	...	324
Imperative code	510	...	170
Proofs	5190	350 + 1220[1]	2753
Timeframe (months)	-	6+	3

Figure 4.1: Comparison of Lines of Code and time investment in related mechanized B-Tree verifications. $^{+}$ denotes implementations of B^{+}-Trees. ^{d} denotes that the implementation additionally verifies deletion operations.

However usually B-Trees are rather applied as concretizations of maps, for example mapping the unique primary keys of database tuples to the stored data. Future work could adjust the code and proofs given here to show that they can satisfy 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.

List of Figures

List of Tables

Bibliography

[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.

[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`.

[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`.