A Verified Implementation of B$^+$-Trees in Isabelle/HOL

Niels Mündler[1(✉)] and Tobias Nipkow[2]

[1] Department of Computer Science, ETH Zürich, Zurich, Switzerland
nmuendler@ethz.ch
[2] Department of Informatics, Technical University of Munich, Munich, Germany
https://in.tum.de/ nipkow/

Abstract. In this paper we present the verification of an imperative implementation of the ubiquitous B$^+$-tree data structure in the interactive theorem prover Isabelle/HOL. The implementation supports membership test, insertion and range queries with efficient binary search for intra-node navigation. The imperative implementation is verified in two steps: an abstract set interface is refined to an executable but inefficient purely functional implementation which is further refined to the efficient imperative implementation.

Keywords: Separation logic · Verification · Refinement

1 Introduction

B$^+$-trees form the basis of virtually all modern relational database management systems (RDBMS) and file systems. Even single-threaded databases are non-trivial to analyse and verify, especially machine-checked. Meanwhile it is important to verify various properties like functional correctness, termination and runtime, since RDBMS are ubiquitous and employed in critical contexts, like the banking sector and realtime systems. The only work in the literature on that topic that we are aware of is the work by Malecha *et al.* [10]. However, it lacks the commonly used range query operation, which returns a pointer to the lower bound of a given value in the tree and allows to iterate over all successive values. This operation is particulary challenging to verify as it requires to mix two usually strictly separated abstractions of the tree in order to reason about its correctness. We further generalize the implementation of node internal navigation. This allows to abstract away from its implementation and simplifies proofs. It further allows us to supply an implementation of efficient binary search, a practical and widespread runtime improvement as nodes usually have a size of several kilobytes. We provide a computer assisted proof in the interactive theorem prover Isabelle/HOL [13] for the functional correctness of an imperative implementation of the B$^+$-tree data-structure and present how we dealt with the resulting technical verification challenges.

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2022
H. Seidl et al. (Eds.): ICTAC 2022, LNCS 13572, pp. 324–341, 2022.
https://doi.org/10.1007/978-3-031-17715-6_21

2 Contributions

In this work, we specify the B^+-tree data structure in the functional modeling language higher-order logic (HOL). The tree is proven to refine a finite set of linearly ordered elements. All proofs are machine-checked in the Isabelle/HOL framework. Within the framework, the functional specifications already yield automatic extraction of executable, but inefficient code.

The contributions of this work are as follows

- The first verification of genuine range queries, which require additional insight in refinement over iterating over the whole tree.
- The first efficient intra-node navigation based on binary rather than linear search.

The remainder of the paper is structured as follows. In Sect. 2.1, we present a brief overview on related work. The definition of B^+-tree and our approach is introduced in Sect. 3. In Sects. 4 and 5, we refine a functionally correct, abstract specification of point, insertion and range queries as well as iterators down to efficient imperative code. Finally, we present learned lessons and evaluate the results in Sect. 6.

The complete source code of the implementation referenced in this research is accessible via the Archive of Formal Proofs [11].

2.1 Related Work

There exist two pen and paper verifications of B^+-tree implementations via a rigorous formal approach. Fielding [5] uses gradual refinement of abstract implementations. Sexton and Thielecke [16] show how to use separation logic in the verification. These are more of a conceptual guideline on approaching a fully machine checked proof.

There are two machine checked proofs of imperative implementations. In the work of Ernst *et al.* [4], an imperative implementation is directly verified by combining interactive theorem proving in KIV [14] with shape analysis using TVLA [15]. The implementation lacks shared pointers between leaves. This simplifies the proofs about tree invariants. However, the tree therefore also lacks iterators over the leaves, and the authors present no straightforward solution to implement them. Moreover, by directly verifying an imperative version only, it is likely that small changes in the implementation will break larger parts of the proof.

Another direct proof on an imperative implementation was conducted by Malecha *et al.* [10], with the Ynot extension to the interactive theorem prover Coq. Both works use recursively defined shape predicates that describe formally how the nodes and pointers represent an abstract tree of finite height. The result is both a fairly abstract specification of a B^+-tree, that leaves some design decisions to the imperative implementation, and an imperative implementation that supports iterators.

Due to the success of this approach, we follow their example and define these predicates functionally. One example of the benefits of this approach is that we were able to derive finiteness and acyclicity only from the relation between imperative and functional specification. In contrast to previous work, the functional predicates describing the tree shape are kept completely separated from the imperative implementation, yielding more freedom for design choices within the imperative refinement. Both existing works rely on linear search for intra-node navigation, which we improve upon by providing binary search. We extend the extraction of an iterator by implementing an additional range query operation.

3 B$^+$-trees and Approach

The B$^+$-tree is a ubiquitous data structure to efficiently retrieve and manipulate indexed data stored on storage devices with slow memory access [3]. They are k-ary balanced search trees, where k is a free parameter. We specify them as implementing a set interface on elements of type $'a$, where the elements in the leaves comprise the content of an abstract set. The inner nodes contain separators. These have the same type $'a$ as the set content, but are only used to guide the recursive navigation through the tree by bounding the elements in the neighboring subtrees. Further the leaves usually contain pointers to the next leaf, allowing for efficient iterators and range queries. A more formal and detailed outline of B$^+$-trees can be found in Sect. 3.2.

The goal of this work is to define this data structure and implement and verify efficient heap-based imperative operations on them. For this purpose, we introduce a functional, algebraic definition and specify all invariants on this level that can naturally be expressed in the algebraic domain. It is important to note that this representation is not complete, as aliased pointers are left out on the algebraic level. However, important structural invariants, such as sortedness and balancedness can be verified.

In a second step an imperative definition is introduced, that takes care of the refinement of lists to arrays in the heap and introduces (potentially shared) pointers instead of algebraic structures. Using a refinement relationship, we can prove that an imperative refinement of the functional specification preserves the structural invariants of the imperative tree on the heap. The only remaining proof obligation on this level is to ensure the correct linking between leaf pointers.

The above outlined steps are performed via manual refinement in Imperative HOL [2]. We build on the library of verified imperative utilities provided by the Separation Logic Framework [9] and the verification of B-trees [11], namely list interfaces and partially filled arrays. The implementation is defined with respect to an abstract imperative operation for node-internal navigation. This means that within each node, we do not specify how the correct subtree for recursive queries is found, but only constrain some characteristics of the result. We provide one such operation that employs linear search, and one that conducts binary search. All imperative programs are shown to refine the functional specifications

using the separation logic utilities from the Isabelle Refinement Framework by Lammich [8].

3.1 Notation

Isabelle/HOL conforms to everyday mathematical notation for the most part. For the benefit of the reader who is unfamiliar with Isabelle/HOL, we establish notation and in particular some essential datatypes together with their primitive operations that are specific to Isabelle/HOL. We write $t :: 'a$ to specify that the term t has the type $'a$ and $'a \Rightarrow 'b$ for the type of a total function from $'a$ to $'b$. The type for natural numbers is *nat*. Sets with elements of type $'a$ have the type $'a$ *set*. Analogously, we use $'a$ *list* to describe lists, which are constructed as the empty list *[]* or with the infix constructor *#*, and are appended with the infix operator *@*. The function *concat* concatenates a list of lists. The function *set* converts a list into a set. For optional values, Isabelle/HOL offers the type *option* where a term *opt :: 'a option* is either *None* or *Some a* with *a* :: $'a$.

3.2 Definitions

We first define an algebraic version of B$^+$-trees as follows:

datatype $'a$ *bplustree* =
Leaf ($'a$ *list*) |
Node (($'a$ *bplustree* × $'a$) *list*) ($'a$ *bplustree*)

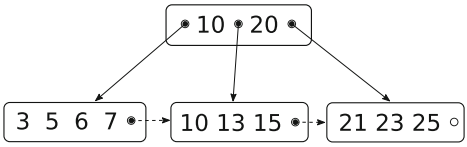

Fig. 1. Nodes contain several elements, the internal list/array structure is not depicted. The dotted lines represent links to following leaf nodes that are not present in the algebraic formulation.

Every node *Node* [(t_1,a_1), ..., (t_n,a_n)] t_{n+1} contains an interleaved list of *keys* or *separators* a_i and *subtrees* t_i. We write as t_i the subtree to the left of a_i and t_{i+1} the subtree to the right of a_i. We refer to t_{n+1} as the *last* subtree. The leaves *Leaf* [v_1, ..., v_n] contain a list of *values* v_i. The concatenation of lists of values of a tree t yields all elements contained in the tree. We refer to this list as *leaves* t. A B$^+$-tree with the above structure must fulfill the invariants *balancedness*, *order* and *alignment*.

Balancedness requires that each path from the root to a leaf has the same length. In other words, the height of all trees in one level of the tree must be equal, where the height is the maximum path length to a leaf.

The *order* property ensures a minimum and maximum number of subtrees for each node. A B$^+$-tree is of order k, if each internal node has at least $k+1$ subtrees and at most $2k+1$. The root is required to have a minimum of 2 and a maximum of $2k+1$ subtrees. We require that k be strictly positive, as for $k=0$ the requirements on the tree root are contradictory.

Alignment means that keys are sorted with respect to separators: For a separator k and all keys l in the subtree to the left, $l < k$, and all keys r in the subtree to the right, $k \leq r$. (where $\leq$ and $<$ can be exchanged).

For the values within the leaves, *sortedness* is required explicitly. We require the even stronger fact that *leaves* t is sorted. This is a useful statement when arguing about the correctness of set operations.

3.3 Implementation Definitions

Proofs about the correctness of operations with respect to implementing an abtract set interface and preserving these invariants are only done on the abstract level, where they are much simpler and many implementation details can be disregarded. It will serve as a reference point for the efficient imperative implementation.

The more efficient executable implementation of B$^+$-trees is defined on the imperative level. Each imperative node contains non-null pointers (*ref*) rather than the algebraic subtree. We refine lists with partially filled arrays of capacity $2k$. A partially filled array (a, n) with capacity c is an array a of fixed size c. The array consists of the elements at indices 0 to $n-1$. Element accesses beyond index n are undefined. Unlike dynamic arrays, partially filled arrays are not expected to grow or shrink. Each imperative node contains the equivalent information to an abstract node. The only addition is that leaves now also contain a pointer to another leaf, which will form a linked list over all leaves in the tree. This was not implemented in the algebraic version as it requires pointer aliasing.

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

In order to use the algebraic data structure as a reference point, we introduce a refinement relation. The correctness of operations on the imperative node can then be shown by relating imperative input and output and to the abstract input and output of a correct abstract operation. In particular we want to show that if we assume $R\ t\ t_i$, where R is the refinement relation and t and t_i are the abstract and the imperative version of the same conceptual tree, $R\ o(t)\ o_i(t_i)$ should hold, where o_i is the imperative refinement of operation o. The relation is expressed as a separation logic formula that links an abstract tree to its imperative equivalent.

The notation for separation logic in Isabelle is quickly summarized in the list below.

– *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
- $a \mapsto_r x$ holds if the heap at location a is reserved and contains value x
- $\exists_A\ x.\ P\ x$ holds if there exists some x such that Px holds on the heap.
- $P_1 * P_2$ denotes the separating conjunction and holds if each assertion P_1 and P_2 hold on non-overlapping parts of the heap
- *is_pfa c xs xsi* holds if *xsi* is a partially filled array with capacity c and *xs[i] = xsi[i]* holds for all $i \leq |xs| = |xsi|$.
- *list_assn R xs ys* holds if *R xs[i] ys[i]* holds for all $i \leq |xs| = |ys|$.

Separation Logic formulae express assertions that can be made about the state of the heap. They are therefore just called *assertion* in the following. The assertion P describes all heaps for which the formula P evaluates to *true*. The entailment $P \Longrightarrow_A Q$ holds iff Q holds in every heap in which P holds. For two assertions P and Q, $P = Q$ holds iff $P \Longrightarrow_A Q \wedge Q \Longrightarrow_A P$. For proving imperative code correct, assertions are used in the context of Hoare triples. We write $\langle P\rangle\ c\ \langle\lambda r.\ Q\ r\rangle$ if, for any heap where P holds, after executing imperative code c that returns value r, formula $Q\ r$ holds on the resulting heap. $\langle P\rangle\ c\ \langle\lambda r.\ Q\ r\rangle_t$ is a shorthand for $\langle P\rangle\ c\ \langle\lambda r.\ Q\ r * true\rangle$ More details can be found in the work of Lammich and Meis [9].

The assertion *bplustree_assn* expresses the refinement relation. It relates an algebraic tree (*bplustree*) and a non-null pointer to an imperative tree a (*btnode ref*), pinning its first leaf r and the first leaf of the next sibling z. The formal relation is shown in Fig. 2.

```
fun bplustree_assn :: nat ⇒ 'a bplustree ⇒ 'a btnode ref
      ⇒ 'a btnode ref option ⇒ 'a btnode ref option where
    bplustree_assn k (Node ts t) a r z = ∃A tsi ti tsi' rs.
      a ↦r Btnode tsi ti
      −− Obtain list with array contents for folding list_assn
    * is_pfa (2*k) tsi' tsi
    * ↑(length tsi' = length rs)
      −− Recursively apply the assertion to subtree pointers
    * list_assn ((λ t (ti,r',z'). bplustree_assn k t (the ti) r' z') ×a id_assn) ts (
      −− Pointers to left/right sibling are obtained by offset zipping
        zip (zip (map fst tsi') (zip (butlast (r#rs)) rs))) (map snd tsi')))
    * bplustree_assn k t ti (last (r#rs)) z)
  | bplustree_assn k (Leaf xs) a r z = ∃A xsi fwd.
        a ↦r Btleaf xsi fwd * is_pfa (2*k) xs xsi * ↑(fwd = z) * ↑(r = Some a)
```

Fig. 2. The B$^+$-tree is specified by the split factor k, an abstract tree, a pointer to its root a, a pointer to its first leaf r and a pointer to the first leaf of the next sibling z. The pointers to first leaf and next first leaf are used to establish the linked leaves invariant.

The main structural relationship between abstract and imperative tree is established by linking abstract list and array via the *is_pfa* predicate. We then

fold over the two lists using *list_assn*, which establishes a refinement relation for every pair of list elements.

In addition to the refinement of data structures, the first leaf r and next leaf z are used to express the structural invariant that the leaves are correctly linked. There is no abstract equivalent for the forwarding pointers in the leaves, therefore we only introduce and reason about their state on the imperative layer. The invariant is ensured by passing the first leaf of the right neighbor to each subtree. The pointer is passed recursively to the leaf node, where it is compared to the actual pointer of the leaf. All of this happens in the convoluted *list_assn*, by folding over the list of the leaf pointer list *rs* zipped with itself, offset by one. The linking property is required for the iterator on the tree in Sect. 5.1.

3.4 Node Internal Navigation

In order to define meaningful operations that navigate the node structure of the B$^+$-tree, we need to find a method that handles search within a node. Ernst *et al.* [4] and Malecha *et al.* [10] both use a linear search through the key and value lists. However, B$^+$-trees are supposed to have memory page sized nodes [3], which makes a linear search impractical.

We introduce a context (*locale* in Isabelle) in which we assume that we have access to a function that correctly navigates through the node internal structure. *Correct* in this context meaning that the selected subtree for recursive calls will lead to the element we are looking for. We call this function *split*, and define it only by its behavior. The specification for *split* is given in Fig. 3 (where $'b = 'a$ *bplustree* $\times$ $'a$). A corresponding function *split_list* is defined on the separator-only lists in the leaf nodes.

```
locale split_tree =
  fixes split :: 'b list ⇒ 'a ⇒ 'b list × 'b list
  split xs p = (ls,rs) ⟹ xs = ls @ rs
  split xs p = (ls@[(sub,sep)],rs); sorted_less (separators xs) ⟹ sep < p
  split xs p = (ls,(sub,sep)#rs); sorted_less (separators xs) ⟹ p ≤ sep
```

Fig. 3. Given a list of separator-subtree pairs and a search value x, the function should return the pair (s, t) such that, according to the structural invariant of the B$^+$-tree, t must contain x or will hold x after a correct insertion.

In the following sections, all operations are defined and verified based on *split* and *split_list*. When approaching imperative code extraction, we provide a binary search based imperative function, that refines *split*. Thus we obtain imperative code that makes use of an efficient binary search, without adding complexity to the proofs. The definition and implementation closely follows the approach described in detail in the verification of B-trees [11].

4 Set Operations

B^+-trees refine sets on linearly ordered elements. For a tree t, the refined abstract set is computed as *set (leaves t)*. The set interface requires that there should be query, insertion and deletion operations o_t such that *set (leaves (o_t t)) = o (set (leaves t))*. Moreover, the invariants described in Sect. 3 can be assumed to hold for t and are required for o_t. We provide these operations and show their correctness on the functional layer first, then refine the operations further to the imperative layer. For point queries and insertion, we follow the implementation suggested by Bayer and McCreight [1].

4.1 Functional Point Query

For an inner node t and a searched value x, find the correct subtree s_t such that if a leaf of t contains x, a leaf of s_t must contain x. Then recurse on s_t. Inside the leaf node, we search directly in the list of values. We make use of the *split* and *isin_list* operation, as described in Sect. 3.4.

```
fun isin:: 'a bplustree ⇒ 'a ⇒ bool where
  isin (Leaf ks) x = (isin_list x ks) |
  isin (Node ts t) x = (case split ts x of
     (_,(sub,sep)#rs) ⇒ isin sub x
   | (_,[]) ⇒ isin t x
  )
```

Since this function does not modify the tree involved at all, we only need to show that it returns the correct value.

```
theorem assumes sorted_less (leaves t) and aligned l t u
  shows isin t x = (x ∈ set (leaves t))
```

In general, these proofs on the abstract level are based on yet another refinement relation suggested by Nipkow [12]. We say that the B^+-tree t refines a sorted list of its leaf values, *leaves t*, the concatenated lists of values in leafs visited in in-order traversal of the tree. We argue that recursing into a specific subtree is equivalent to splitting this list at the correct position and searching in the correct sublist. The same approach was applicable for proving the correctnes of functional operations on B-trees [11].

The proofs on the functional level can therefore be made concise. We go on and define an imperative version of the operation that refines each step of the abstract operation to equivalent operations on the imperative tree.

4.2 Imperative Point Query

The imperative version of the point query is a partial function. Termination cannot be guaranteed anymore, at least without further assumptions. This is inevitable since the function would not terminate given cyclic trees. However, we will show that if the input refines an abstract tree, the function terminates

and is correct. The imperative $isin_i$ refines each step of the abstract operation with an imperative equivalent. The result can be seen in Fig. 4.

```
partial_function (heap) isin_i :: 'a btnode ref ⇒ 'a ⇒ bool Heap where
  isin_i p x = do {
  node ← !p;
  (case node of
     Btleaf xs _ ⇒ isin_list_i x xs |
     Btnode ts t ⇒ do {
       i ← split_i ts x;
       tsl ← length ts;
       if i < tsl then do {
         s ← get ts i;
         let (sub,sep) = s in
           isin_i (the sub) x
       } else
           isin_i t x
     }
  )}
```

Fig. 4. The imperative refinement of the *isin* function. As a partial function, its termination is not guaranteed for all inputs. Additionally it implicitly makes use of the heap monad.

Again, we assume that $split_i$ performs the correct node internal search and refines an abstract *split*. Note how $split_i$ does not actually split the internal array, but rather returns the index of the pair that would have been returned by the abstract split function. The pattern matching against an empty list is replaced by comparing the index to the length of the list l. In case the last subtree should be recursed into, the whole list l is returned.

In order to show that the function returns the correct result, we show that it performs the same operation on the imperative tree as on the algebraic tree. This is expressed in Hoare triple notation and separation logic.

lemma assumes $k > 0$ **and** *root_order* k t **and** *sorted_less* (*inorder* t) **and** *sorted_less* (*leaves* t) **shows**
$\langle$*bplustree_assn* k t ti r $z\rangle$
  $isin_i$ ti x
$\langle\lambda y.$ *bplustree_assn* k t ti r $z * \uparrow(isin\ t\ x = y)\rangle_t$

The proof follows inductively on the structure of the abstract tree. Assuming structural soundness of the abstract tree refined by the pointer passed in, the returned value is equivalent to the return value of the abstract function. We must explicitly show that the tree on the heap still refines the same abstract tree after the operation, which was implicit on the abstract layer. It follows directly, since no operation in the imperative function modifies part of the tree.

4.3 Insertion and Deletion

The insertion operation and its proof of correctness largely line up with the one for point queries. But since insertion modifies the tree, we need to additionally show on the abstract level that the modified tree maintains the invariants of B^+-trees.

On the imperative layer, we show that the heap state after the operation refines the tree after the abstract insertion operation. It follows that the imperative operation also maintains the abstract invariants. Moreover, we need to show that the linked list among the leaf pointers is correctly maintained throughout the operation. This can only be shown on the imperative level as there is no abstract equivalent to the shared pointers.

lemma assumes $k > 0$ **and** *sorted_less* (*inorder t*)
and *sorted_less* (*leaves t*) **and** *root_order k t* **shows**
$\langle$*bplustree_assn k t ti r z*$\rangle$
$insert_i$ *k x ti*
$\langle\lambda u.$ *bplustree_assn k* (*insert k x t*) *u r z*$\rangle_t$

We provide a verified functional definition of deletion and a definition of an imperative refinement. Showing the correctness of the imperative version would largely follow the same pattern as the proof of the correctness of insertion. The focus of this work is not on basic tree operations, but on obtaining a (range) iterator view on the tree.

5 Range Operations

This section introduces both how the general iterator on the tree leaves is obtained and the technical challenges involved (Sect. 5.1) as well as how to obtain an iterator on a specific subset of elements efficiently (Sect. 5.2).

On the functional level, the forwarding leaf pointers in each leaf are not present, as this would require aliasing. Therefore, the abstract equivalent of an iterator is a concatenation of all leaf contents. When refining the operations, we will make use of the leaf pointers to obtain an efficient implementation.

5.1 Iterators

To obtain an iterator, recurse down the tree to obtain the first leaf. From there we follow leaf pointers until we reach the final leaf marked by a null forwarding pointer. From an assertion perspective the situation is more complex. Recall the refinement relation between abstract and implemented B^+-tree. It is important to find an explicit formulation of the linked list view on the leaf pointers. Meanwhile, we want to ensure that the complete tree does not change by iterating through the leaves. We cannot express an assertion about the linked list along the leaves and the assertion on the whole tree in two fully independent predicates as the memory described overlaps. Separation logic forces us to not make statements about the contents of any memory location twice.

We follow the approach of Malecha *et al.* [10] and try to find an equivalent formulation that separates the whole tree in a view on its inner nodes and the linked leaf node list. The central idea to separate the tree is to express that the linked leaf nodes refine *leaf_nodes t* and that the inner nodes refine *trunk t*, as depicted in Fig. 5. These are two independent parts of the heap and therefore the statements can be separated using the separating conjunction.

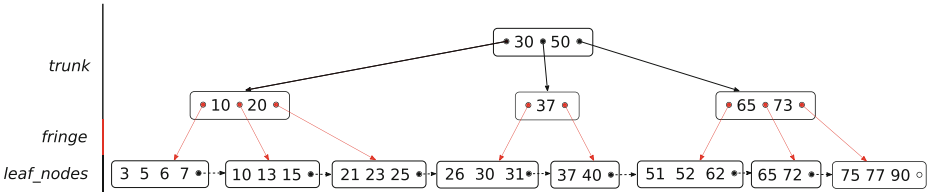

Fig. 5. In order to obtain separate assertions about the concatenated leaf list (*leaf_nodes*) and the internal nodes (*trunk*) of the tree, the structure is abstractly split along the pointers marked in red, the *fringe*. In order to be able to combine the *leaf_nodes* and the *trunk* together, the *fringe* has to be extracted and shared explicitly.

Formally, we define an assertion *trunk_assn* and *leaf_nodes_assn*. The former is the same as *bplustree_assn* (see Fig. 2), except that we remove all assertions about the content of the tree in the *Leaf* case. The latter is defined similar to a linked list refining a list of abstract tree leaf nodes, shown in Fig. 6. The list is refined by a pointer to the head of the list, which refines the head of the abstract list. Moreover, the imperative leaf contains a pointer to the next element in the list.

With these definitions, we can show that the heap describing the imperative tree may be split up into its leaves and the trunk.

lemma *bplustree_assn k t ti r z*
$\Longrightarrow_A$ *leaf_nodes_assn k* (*leaf_nodes t*) *r z* $*$ *trunk_assn k t ti r z*

However, we cannot show that a structurally consistent, unchanged B$^+$-tree is still described by the combination of the two predicates. The reason is that we cannot express that the linked leaf nodes are precisely the leaf nodes on the lowest level of the trunk, depicted in red in Fig. 5.

The root of this problem is actually a feature of the refinement approach. When stating that a part of the heap refines some abstract data structure, we make no or little statements about concrete memory locations or pointers. This is useful, as it reduces the size of the specification and the proof obligations. In this case we need to find a way around it.

We need to specifically express that the leaf pointers, and not the abstract structure they refine, are precisely the same in the two statements.

In a second attempt, the sharing is made explicit. We extract from the whole tree the precise list of pointers to leaf nodes, the *fringe* in the correct order.

```
fun leaf_nodes_assn where
  leaf_nodes_assn k ((Leaf xs)#lns) (Some r) z =
 (∃A xsi fwd.
     r ↦r Btleaf xsi fwd
   ∗ is_pfa (2∗k) xs xsi
   ∗ leaf_nodes_assn k lns fwd z
 ) |
  leaf_nodes_assn k [] r z = ↑(r = z) |
  leaf_nodes_assn _ _ _ _ = false
```

Fig. 6. The refinement relation for leaf nodes comprises the refinement of the node content as well as the recursive property of linking correctly to the next node.

Recursively, the fringe of a tree is the concatenation of all fringes in its subtrees. The resulting assertion, taking the fringe into account, can be seen in Fig. 7. As a convenient fact, this assertion is equivalent to Fig. 2.

lemma *bplustree_extract_fringe:*
bplustree_assn k t ti r z = (∃A*fringe. bplustree_assn_fringe k t ti r z fringe*)

Using the *fringe*, we can precisely state an equivalent separated assertion. We describe the trunk with the assertion *trunk_assn*, which is the same as *bplustree_assn_fringe*, except that the *Leaf* case is changed to only ↑ (*r* = *Some a* ∧ *fringe* = [*a*]). In addition, we extend the definition of *leaf_nodes_assn* to take the *fringe* pointers into account. We now require that the *fringe* of the trunk is precisely the list of pointers in the linked list refining *leaf_nodes*.

lemma *bplustree_view_split:*
bplustree_assn_fringe k t ti r z fringe =
leaf_nodes_assn k (*leaf_nodes t*) *r z fringe* ∗ *trunk_assn k t ti r z fringe*

To obtain an iterator on the leaf nodes of the tree, we obtain the first leaf of the tree. By the formulation of the tree assertion, we can express the obtained result using the assertion about the complete tree.

lemma assumes *k* > *0* **and** *root_order k t* **shows**
⟨*bplustree_assn k t ti r z*⟩
first_leaf ti
⟨λ*u. bplustree_assn k t ti r z* ∗ ↑(*u* = *r*)⟩*t*

On the result, we can apply lemmas *bplustree_extract_fringe* and *bplustree_view_split*. The transformed expression states that the result of *first_leaf ti* is a pointer to *leaf_nodes t*. The tree root *t* remains to refine *trunk t*.

From here, we could define an iterator over the leaf nodes along the fringe, refining the abstract list *leaf_nodes*. Our final goal is to iterate over the values within each array inside the nodes. We introduce a flattening iterator for this purpose. It takes an outer iterator over a data structure *a* that returns elements of type *b*, and inner iterator over the data structure *b* that returns elements of type *c*. It returns an iterator over data structure *a* that returns the concatenated

```
fun bplustree_assn_fringe where
  bplustree_assn_fringe k (Leaf xs) a r z fringe =
  ∃_A xsi fwd.
     a ↦_r Btleaf xsi fwd
   * is_pfa (2*k) xs xsi
   * ↑(fwd = z)
   * ↑(r = Some a)
   -- In case of a singleton leaf, the leaf itself is the fringe of the tree
   * ↑(fringe = [a])
  |
  bplustree_assn_fringe k (Node ts t) a r z fringe =
  ∃_A tsi ti tsi' tsi'' rs fr_sep.
     a ↦_r Btnode tsi ti
   * is_pfa (2*k) tsi' tsi
   * ↑(length tsi' = length rs)
   -- The fringe is decomposed into the fringe of each subtree
   * ↑(concat fr_sep = fringe)
   * ↑(length fr_sep = length rs + 1)
   -- Folding over all subtrees as before, now passing each subfringe to subtrees
   * list_assn (
       (λ t (ti,r',z',fr). bplustree_assn_fringe k t (the ti) r' z' fr)
        ×_a id_assn
     ) ts (zip
        (zip (map fst tsi') (zip (butlast (r#rs)) (zip rs (butlast fr_sep))))
        (map snd tsi')
     )
   * bplustree_assn_fringe k t ti (last (r#rs)) (last (rs@[z])) (last fr_sep)
```

Fig. 7. An extended version of the B$^+$-tree assertion from Fig. 2 on imperative tree root a, first leaf r, first leaf of the next sibling z and leaf pointer list $fringe$. In order to be able to correctly relate leaf view and internal nodes, the shared pointers *fringe* are made explicit, without accessing their memory location.

list of elements of type c. The exact implementation of this iterator is left out as a technical detail.

The list iterator interface used is as defined by Lammich [7] and specifies the following function.

- An *init* function that returns the pointer to the head of the list.
- A *has_next* function that checks whether the current pointer is the null pointer.
- A *next* function that returns the the array in the current node and its forwarding pointer.
- Proofs that we can transform the *leaves_assn* statement into a leaf iterator statement and vice versa.

We implement such an iterator for the linked list of leaf nodes *leaf_nodes_iter* and combine it with the iterator over partially filled arrays using the flattening iterator to obtain the *leaves_iter*.

Finally, we want be able to express that the whole tree does not change throughout the iteration. For this, we need to keep track of both the leaf nodes assertion and the trunk assertion on t. The assertion describing the iterator therefore contains both. Most parameters to the iterator assertion are static, and express the context of the iterator, i.e. the full extent of the leaf nodes. The iterator state *it* itself is a pair of an iterator state for a partial array, the current position in that array and its size, and a pointer to the next leaf and the final leaf.

definition *bplustree_iter k t ti r vs it* = $\exists_A$ *fringe.*
leaves_iter fringe k (*leaf_nodes t*) (*leaves t*) *r vs it* ∗
trunk_assn k t ti r None fringe

Note how all notion of the explicitly shared fringe has disappeared from the client perspective as its existence is hidden within the definition of the tree iterator. We initialize the iterator using the *first_leaf* operation and obtain the singleton tree elements with the flattening iterator.

5.2 Range Queries

A common use case of B$^+$-trees is to obtain all values within a range [6]. We focus on the range bounded from below, *lrange t x* = $\{y \in set(t) | y \geq x\}$. From an implementation perspective, the operation is similar to the point query operation. On the leaf level, it returns a pointer to the reached leaf. This pointer is then interpreted as iterator over the remaining list of linked leaves. The range bounded from below comprises all values returned by the iterator. Due to the lack of a linked leaf list in the abstract tree, the abstract definition explicitly concatenates all values in the subtrees to the right of the reached node.

```
fun lrange:: 'a bplustree ⇒ 'a ⇒ 'a list where
  lrange (Leaf ks) x = (lrange_list x ks) |
  lrange (Node ts t) x = (
    case split ts x of (_,(sub,sep)#rs) ⇒ (
        lrange sub x @ (concat (map leaves rs)) @ leaves t
    )
  | (_,[]) ⇒ lrange t x
  )
```

As before, we assume that there exists a function *lrange_list* that obtains the *lrange* from a list of sorted values.

The verification of the imperative version turns out to be not as straightforward as expected, exactly due to this recursive step. The reason is that iterators can only be expressed on a complete tree, where the last leaf is explicitly a null pointer. The linked list of a subtree is however bounded by valid leaves, precisely the first leaf of the next subtree.

In order implement and verify a refinement of this function we therefore decide to implement an intermediate abstract function *leaf_nodes_lrange*. This function returns the leaf nodes comprising the *lrange* instead of their contents.

fun *leaf_nodes_lrange*:: *'a bplustree* ⇒ *'a* ⇒ *'a bplustree list* **where**
leaf_nodes_lrange (*Leaf ks*) *x* = [*Leaf ks*] |
leaf_nodes_lrange (*Node ts t*) *x* = **case** *split ts x* **of**
(_,(*sub*,*sep*)#*rs*) ⇒
leaf_nodes_lrange sub x @ *concat* (*map leaf_nodes rs*) @ *leaf_nodes t*
| (_,[]) ⇒ *leaf_nodes_lrange t x*

fun *concat_leaf_nodes_lrange* **where**
concat_leaf_nodes_lrange t x = **case** *leaf_nodes_lrange t x* **of**
(*LNode ks*)#*list* ⇒ *lrange_list x ks* @ *concat* (*map leaves list*)

We then show that the concatenation of the contents of the leaf nodes *concat_leaf_nodes_lrange t x* = *lrange t x*. On the imperative layer $leaf\_nodes\_lrange_i$ can be obtained using only the *leaf_nodes* and *trunk* assertions as we never access the contents of the leaf nodes. We therefore avoid having to unfold any assertions about the structure of the leaf nodes. The function returns a pointer that splits the list of leaf nodes of the whole tree, terminated by the null pointer that marks the end of the complete tree. We transform the result into an iterator over the leaf nodes, as this pointer split notation aligns with the definition of *leaf_nodes_iter*. Finally we can transform this and the result of $lrange\_list_i$ to an iterator on the singleton leaf elements.

lemma assumes *k* > *0* **and** *root_order k t*
and *sorted_less* (*leaves t*) **and** *Laligned t u* **shows**
⟨*bplustree_assn k t ti r None*⟩
$concat\_leaf\_nodes\_lrange_i$ *ti x*
⟨*bplustree_iter k t ti r* (*lrange t x*)⟩$_t$

6 Conclusion

We were able to formally verify an imperative implementation of the ubiquitous B$^+$-tree data structure, featuring range queries and binary search.

6.1 Evaluation

The B$^+$-tree implemented by Ernst *et al.* [4] features point queries and insertion, however explicitly leaves out pointers within the leaves, which forbids the implementation of iterators. Our work is closer in nature to the B$^+$-tree implementation by Malecha *et al.* [10]. In addition to the functionality dealt with in their work, we extend the implementation with a missing Range iterator and supply a binary search within nodes. Our approach is modular, allowing for

the substitution of parts of the implementation with even more specialized and sophisticated implementations.

Regarding the leaf iterator, we noticed that in the work of Malecha *et al.* there is no need to extract the fringe explicitly. The abstract leaves are defined such that they store the precise heap location of the refining node. In our proposed definition, the precise heap location is irrelevant in almost every situation and can be omitted. Only when splitting the tree we obtain the memory location of nodes explicitly, because these locations are needed to guarantee structual soundness of the whole tree. It is hard to quantify or evaluate which approach is more practical. From a theoretical view point we suggest that a less strict approach restricts the implementation space less and leaves more design decisions to the specification implementing developer.

With respect to the effort in lines of code and proof as depicted in Fig. 8, our approach is similar in effort to the approach by Malecha *et al.*. The numbers do not include the newly defined pure ML proof tactics. It includes the statistics for the additional binary search and range iterator, that make up around one thousand lines of proof each.

The comparison with Ernst *et al.* is difficult. Their research completely avoids the usage of linked leaf pointers, therefore also omitting iterators completely. The iterator verification makes up a signifant amount of the proof with at least one thousand lines of proof on its own. The leaf pointers also affect the verification of point and insertion queries due to the additional invariant on the imperative level. We conclude that the Isabelle/HOL framework provides a feature set such that verification of B^+-trees is both feasible and comparable in effort to using Ynot or KIV/TVLA. The strict separation of a functional and imperative implementation yields the challenge of making memory locations explicit

	Malecha *et al.*[10]$^+$	[4]^d	Our approach$^+$
Functional code	360	-	413
Imperative code	510	1862	1093
Proofs	5190	350 + 510 + 2940*	8663
Timeframe (months)	?	> 6	6 + 6**

Fig. 8. Comparison of (unoptimized) Lines of Code and Proof and time investment in related mechanized B^+-tree verifications. All approaches are comparable in effort, taking into account implementation specifics. The marker ^{d} denotes that the implementation verifies deletion operations, whereas $^+$ denotes the implementation of iterators.
* The proof integrates TVLA and KIV, and hence comprises explicitly added rules for TVLA (the first number), user-invented theorems in KIV (the second number) and "interactions" with KIV (the second number). Interactions are i.e. choices of an induction variable, quantifier instantiation or application of correct lemmas. We hence interpret them as each one apply-Style command and hence one line of proof.
** 6 months include the preceding work on the verification of B-trees. As they share much of the functionality with B^+-trees but required their own specifics, the time spent on them cannot be accounted for 1:1.

where needed. On the other hand, it permits great freedom regarding the actual refinement on the imperative level.

6.2 Outlook

This research may serve as a template for the implementation of B^+-trees in Isabelle-LLVM. [7] At the beginning of this work, the code generator did not yet support recursive data structures, but this functionality was added recently.

As of now, the imperative implementation provided by this research was directly exported into executable imperative code in Haskell, SML and OCaml. It may thus find applications in the development of libraries where a verified implementation of a set interface is needed.

References

1. Bayer, R., McCreight, E.M.: Organization and maintenance of large ordered indices. Acta Informatica **1**, 173–189 (1972). https://doi.org/10.1007/BF00288683
2. Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., Matthews, J.: Imperative functional programming with Isabelle/HOL. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 134–149. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71067-7_14
3. Comer, D.: The ubiquitous b-tree. ACM Comput. Surv. **11**(2), 121–137 (1979). https://doi.org/10.1145/356770.356776
4. Ernst, G., Schellhorn, G., Reif, W.: Verification of B^+ trees by integration of shape analysis and interactive theorem proving. Software & Systems Modeling **14**(1), 27–44 (2013). https://doi.org/10.1007/s10270-013-0320-1
5. Fielding, E.: The specification of abstract mappings and their implementation as b+ trees. Technical Report PRG18, OUCL (1980)
6. Graefe, G.: Modern b-tree techniques. Found. Trends Databases **3**(4), 203–402 (2011). https://doi.org/10.1561/1900000028
7. Lammich, P.: Generating verified LLVM from isabelle/hol. In: Harrison, J., O'Leary, J., Tolmach, A. (eds.) 10th International Conference on Interactive Theorem Proving, ITP 2019, Portland, OR, USA, 9–12 September 2019, vol. 141 of LIPIcs, pp. 22:1–22:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019). https://doi.org/10.4230/LIPIcs.ITP.2019.22
8. Lammich, P.: Refinement to Imperative HOL. J. Autom. Reason. **62**(4), 481–503 (2017). https://doi.org/10.1007/s10817-017-9437-1
9. Lammich, P., Meis, R.: A separation logic framework for imperative HOL. Arch. Formal Proofs **2012** (2012). https://www.isa-afp.org/entries/Separation_Logic_Imperative_HOL.shtml
10. Malecha, J.G., Morrisett, G., Shinnar, A., Wisnesky, R.: Toward a verified relational database management system. In: Hermenegildo, M.V., Palsberg, J. (eds.) Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, 17–23 January 2010, pp. 237–248. ACM (2010). https://doi.org/10.1145/1706299.1706329
11. Mündler, N.: A verified imperative implementation of B-trees. Arch. Formal Proofs **2021** (2021). https://www.isa-afp.org/entries/BTree.html

12. Nipkow, T.: Automatic functional correctness proofs for functional search trees. In: Blanchette, J.C., Merz, S. (eds.) ITP 2016. LNCS, vol. 9807, pp. 307–322. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-43144-4_19
13. Nipkow, T., Klein, G.: Concrete Semantics. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10542-0
14. Reif, W., Schellhorn, G., Stenzel, K., Balser, M.: Structured specifications and interactive proofs with kiv. In: Automated Deduction - A Basis for Applications, vol. 2 (2000). https://doi.org/10.1007/978-94-017-0435-9_1
15. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. **24**(3), 217–298 (2002). https://doi.org/10.1145/514188.514190
16. Sexton, A.P., Thielecke, H.: Reasoning about B+ trees with operational semantics and separation logic. In: Bauer, A., Mislove, M.W. (eds.) Proceedings of the 24th Conference on the Mathematical Foundations of Programming Semantics, MFPS 2008, Philadelphia, PA, USA, 22–25 May 2008, vol. 218 of Electronic Notes in Theoretical Computer Science, pp. 355–369. Elsevier (2008). https://doi.org/10.1016/j.entcs.2008.10.021