Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 218 (2008) 355–369

www.elsevier.com/locate/entcs

Reasoning about B+ Trees with Operational Semantics and Separation Logic

Alan Sexton and Hayo Thielecke[1]

School of Computer Science, University of Birmingham, UK

Abstract

The B+ tree is an ordered tree structure with a fringe list. It is the most widely used data structure for data organisation and searching in database systems specifically, and, probably, computing in general. In this paper, we apply two techniques from programming language theory to B+ trees: operational semantics, in the form of an abstract machine, and separation logic. We use an abstract machine to give a precise and tractable formalisation of the operations on B+ trees. Separation logic is then used to formalise a data structure invariant for B+ trees and to establish correctness by showing that the invariant is preserved by the operations. As usual in separation logic, a frame property is essential for keeping the reasoning local. In our setting, that means that we concentrate on the subtree reached from the top of the stack of the abstract machine, while the remainder of the B+ tree stays invariant. A particularly attractive feature of this approach is the smooth way that proofs can cope with algorithms that begin with a tree descent and switch to fringe list traversal.

Keywords: B+ Trees, Separation Logic, Abstract Machines, Data Structure, Invariant.

1 Introduction

The B+ tree [1,4] is the most widely used data structure for data organisation and searching in database systems today. In this paper we present a formalisation of the B+ tree and its insert and search algorithms that succinctly captures its behaviour. We present data structure invariants for the tree and use these to prove correctness of insertion. We also prove correctness of a general find algorithm that, given lower- and upper-bound key values, returns a list of matching data items by descending the tree to find the first data item in the range, and then traversing the fringe list to obtain all the remaining required items.

Separation logic: To reason about B+ trees rigorously, we will express a data structure invariant for them in a fragment of separation logic [3,6,9]. The central feature here is the separating conjuction $*$, which splits a store into disjoint parts, and allows reasoning to be local. While the literature on separation logic has largely

[1] Email: a.p.sexton@cs.bham.ac.uk and h.thielecke@cs.bham.ac.uk

doi:10.1016/j.entcs.2008.10.021

used separation logic predicates as the assertions of a Hoare logic, we do not use Hoare logic here. That is, rather than formally deriving Hoare triples $\{P\}\ c\ \{Q\}$ for some code c, we reason semantically about stores σ satisfying some formula, $\sigma \models P$.

Abstract machines: To formalize the operations on B+ trees, we use an abstract machine based operational semantics to give a somewhat more abstract and concise specification than pure code. Abstract machines were pioneered by Landin with his SECD machine [7]. Since then, a large variety of abstract machines have been used to formalise programming languages and features, mainly for functional and logic programming languages. A central feature of many abstract machines is the stack. Our machine uses its stack not as a function call stack, but for traversal of the B+ tree. The tree itself is held in what we call the store. In B+ tree implementations, the nodes of the tree are stored in disk pages. The way these pages are linked together is analogous to the way pointers refer across the heap in separation logic. (We call that part of the machine the "store" rather than "heap" to avoid clashes with database terminology). In related work [10,11], we have used this approach to specify the BV-tree [5], a structure that is significantly more complex than the B+ tree, and used it to solve some open problems on this structure as well as obtaining an implementation by hand-translating the abstract machine rules into Java.

Local reasoning and footprint: Our correctness statements will be formulated in terms of the operational semantics and predicates on stores. They typically take the following form: If the initial store σ satisfies $\sigma \models P$, then there is a sequence of machine transitions

$$\langle \dots, \sigma \rangle \rightsquigarrow \cdots \rightsquigarrow \langle \dots, \sigma' \rangle$$

such that the new store σ' satisfies $\sigma' \models Q$. Here P and Q are separation logic predicates using our B+ tree invariant. A key advantage of the separating conjunction $*$ is that it gives rise to local reasoning. In a Hoare triple $\{P\}\ c\ \{Q\}$ for a command c, another formula R can be added via the frame rule:

$$\begin{array}{c} \{P\}\ c\ \{Q\} \\ \{P * R\}\ c\ \{Q * R\} \end{array}$$

Intuitively, anything that is not mentioned in the specification of the command c cannot be altered by it, so we can assume that R stays invariant. Informally, one speaks of the command having a certain "footprint" to which all possible changes are confined. In the present machine setting it is not obvious what the footprint should be, or how a local reasoning is to be achieved (compare the situation in the presence of code pointers [12]). Since we do not reason about code in some given language, but about machine transitions, there is no ready-made frame rule that we could appeal to in proofs. Rather, we have to find some analogue of the footprint that enables us to reason locally. Roughly speaking, in the abstract machine transitions, we have a current B+ tree pointer that identifies the footprint, and such pointers can be pushed onto the stack as we descend the tree. Our reasoning stays local by focusing on transition sequences that restore the stack π after a series of push and pop operations:

$$\langle \dots, \pi, \sigma \rangle \rightsquigarrow \cdots \rightsquigarrow \langle \dots, \pi, \sigma' \rangle$$

Outline: We formalize B+ trees as a data structure invariant, formulated in separation logic, in Section 2. We then introduce abstract machine semantics for B+ tree operations and show their correctness: insertion (Section 3) and finding elements in a range query (Section 4). Deletion of elements is discussed in Section 5. Finally, we remark on some issues raised by this work, discuss our conclusions and indicate our planned future work in Section 6.

2 B+ trees in separation logic

The B+ tree is a ordered, n-ary branching, balanced, search tree supporting destructive updates. In order to preserve the B+ tree invariant, updates can trigger further updates at higher levels of the tree. Data items are stored only in the leaves and all nodes except the root are guaranteed to be kept at least half full. Leaf pages are maintained in a singly-linked list.

We describe our term structure for B+ tree nodes and for abstract machine commands with a BNF grammar extended with sequence constructors: thus $[X]$ means a sequence of terms of type X.

We assume that there is function $\texttt{key}\colon \mathit{Entry} \to \mathit{Key}$ and that Key is totally ordered. For entries a and a', we write $a \sqsubseteq a'$ for $\texttt{key}\,(a) \leq \texttt{key}\,(a')$, and analogously for $\sqsubset$. This notation is extended both to sets of entries, by quantification, and to keys: thus $S \sqsubseteq k$, where S is a set of entries and k is a key, stands for $\forall a \in S.\, \texttt{key}\,(a) \leq k$.

For simplicity, we assume that the maximum number of entries in a leaf page and that the maximum number of child pages (i.e., the maximum fanout) of an internal page are the same and given by a parameter MaxN.

The grammar for a node of a B+ tree is as follows:

$$\begin{array}{rcl} \mathit{Node} & ::= & \mathit{INode} \mid \mathit{LNode} \\ \mathit{INode} & ::= & \texttt{I}\,([\mathit{Key}]\,;\,[\mathit{PageID}]) \\ \mathit{LNode} & ::= & \texttt{L}\,([\mathit{Entry}]\,;\,\mathit{PageID}) \end{array}$$

The intention is that an *INode* represents an internal node of the tree (i.e., a disk page containing a sequence of keys and a sequence of child page pointers), whereas an *LNode* corresponds to a leaf node of the tree, i.e., a disk page containing a sequence of data entries and a forward page pointer.

We assume a store, defined as a finite partial map from page identifiers (or locations) to nodes $\sigma : \mathit{PageID} \rightharpoonup \mathit{Node}$.

For a store σ, we define $\mathrm{dom}\,\sigma \;::=\; \{p \mid \exists n \in \mathit{Node}.\, p \mapsto n \in \sigma\}$ and the store update notation $\sigma[p \mapsto v]$ to mean $\{q \mapsto n \mid q \mapsto n \in \sigma \wedge q \neq p\} \cup \{p \mapsto v\}$. We further define $\sigma[p_0 \mapsto v_0, p_1 \mapsto v_1, \ldots]$ to mean $\sigma[p_0 \mapsto v_0][p_1 \mapsto v_1]\ldots$. Finally, to accommodate release of a location r in a store σ, where $r \in \mathrm{dom}\,\sigma$, we define $\sigma \setminus r$ to mean the restriction of σ with r removed, $\sigma\,|_{\mathrm{dom}\,\sigma\setminus\{r\}}$

For example, consider a store initialised to contain an empty B+ tree with root page pointer r. It maps r to a leaf node containing an empty sequence of entries and a null forward pointer, that is, it is of the form:

$$\{r \mapsto \mathtt{L}\,([\,]\ ;\mathtt{null})\}$$

The invariants on stores containing B+ trees will be expressed in predicate logic; but, crucially, the logic is augmented with the spatial (or separating) conjunction of Separation Logic. We briefly recall some relevant definitions from the literature [6,9]. The partial operation $*$ on stores is defined only if σ_0 and σ_1 are disjoint, that is, $\operatorname{dom}\sigma_0 \cap \operatorname{dom}\sigma_1 = \emptyset$. If so, then $\sigma_0 * \sigma_1 = \sigma_0 \cup \sigma_1$ as partial functions.

$$\boxed{\begin{array}{l}
\sigma \models \mathtt{true} \text{ iff } \sigma \text{ is any store}\\
\sigma \models \mathtt{emp} \text{ iff } \operatorname{dom}(\sigma) = \emptyset\\
\sigma \models r \mapsto N \text{ iff } \sigma = \{r \mapsto N\} \text{ for a node } N\\
\sigma \models Q_0 * Q_1 \text{ iff } \sigma = \sigma_0 * \sigma_1 \text{ where } \sigma_0 \models Q_0 \text{ and } \sigma_1 \models Q_1\\
\sigma \models Q_0 \wedge Q_1 \text{ iff } \sigma \models Q_0 \text{ and } \sigma \models Q_1\\
\sigma \models Q_0 \vee Q_1 \text{ iff } \sigma \models Q_0 \text{ or } \sigma \models Q_1
\end{array}}$$

Fig. 1. Semantics of some separation logic connectives

Figure 1 gives the semantics of the separation logic connectives in terms of what it means for a store σ to satisfy a formula Q, written as $\sigma \models Q$. We also use equality of integers and standard predicate logic quantification. For instance, $\sigma \models \exists x.Q$ iff, for some v, we have $\sigma \models Q[x \mapsto v]$. We elide the type of x, as it is typically clear from the context.

The B+ trees contain two data structures superimposed on each other: a tree and a linked list at the fringe of the tree. In terms of separation logic, the tree is handled by a spatial conjunction $*$ among each internal node and its subtrees. Such a data structure consisting of a tree with a list at the leaves is one of the cases studied by Bornat, Calcagno and O'Hearn [3]; B+ trees are a more complex version of this common situation. In our version, the list is not specified directly, but as the linking of the first and last nodes of subtrees to make the induction go through. It specialises to a list at the leaves, and this is what searching for entries relies on.

Formally, we define a predicate $\mathsf{Btree}_h(r, S, a, z, n)$ on stores. Intuitively, it means that r points to a B+ tree of height h, which contains n immediate children if $h > 1$ or n entries if $h = 1$, and whose set of entries is S, such that a is the address of the first leaf node and the last leaf node has z as its forward pointer.

Definition 2.1 The predicate $\mathsf{Btree}_h(r, S, a, z, n)$ is defined by induction over h as follows:

$$\begin{array}{rcl}
\mathsf{Btree}_1(r, S, a, z, n) & \Longleftrightarrow & \exists e_1, \ldots, e_n.\ n \leqslant \mathit{MaxN}\\
& \wedge & r \mapsto \mathtt{L}\,(e_1 \ldots e_n\ ;\ z)\\
& \wedge & S = \{e_1, \ldots, e_n\} \wedge a = r \wedge e_1 \sqsubset \cdots \sqsubset e_n
\end{array}$$

$$\begin{array}{rcl}
\mathsf{Btree}_{h+1}(r,S,a,z,n) & \iff & \exists d_1,\dots,d_{n-1},q_1\dots q_n,m_1,\dots,m_n.\ n\leqslant \mathit{MaxN} \\
& \wedge & (r\mapsto \mathtt{I}\,(d_1\dots d_{n-1}\ ;\ q_1\dots q_n) \\
& & \quad *\ \mathsf{Btree}_h(q_1,S_1,a_1,a_2,m_1) \\
& & \quad *\ \mathsf{Btree}_h(q_2,S_2,a_2,a_3,m_2) \\
& & \quad *\ \cdots \\
& & \quad *\ \mathsf{Btree}_h(q_n,S_n,a_n,a_{n+1},m_n)) \\
& \wedge & a_1=a\wedge a_{n+1}=z \\
& \wedge & S=S_1\cup\cdots\cup S_n \\
& \wedge & (\forall j.\ 1<j<n-1\Rightarrow d_j\sqsubseteq S_j\sqsubset d_{j+1}) \\
& \wedge & (\forall j.\ 1<j\leqslant n\Rightarrow \lceil \mathit{MaxN}/2\rceil\leqslant m_j) \\
& \wedge & (S_1\sqsubset d_1) \\
& \wedge & (d_{n-1}\sqsubseteq S_n)
\end{array}$$

For a complete B+ tree, we define that the list of leaves is `null`-terminated:

$$\mathsf{ComBtree}(r,S)\equiv\exists h,a,n.\mathsf{Btree}_h(r,S,a,\mathtt{null},n)$$

Note that the Btree predicates directly limit the maximum occupancy of a node to its maximum capacity, *MaxN*. Also, the Btree_{h+1} predicate limits the minimum occupancy of each child node to be at least half its maximum capacity. The usual occupancy guarantee of B+ trees, namely that all except the root node are at least half full, then follows from the obvious inductive argument.

3 Insertion as abstract machine rules

We present a number of ancillary definitions and notations in Figure 2. While most of these are fairly standard, we make them precise here because of ongoing work on automatic generation of implementations of index structures from the abstract machine specification.

In the rules for the transition relation $\rightsquigarrow$, we draw a distinction between the conditions and the definitions for the purpose of generation of efficient code: the conditions must be tested before a transition can be triggered, however, only those definitions that are used in the source configuration or in the condition should be substituted in before the condition test succeeds. The rules are ordered so that if multiple rule heads and conditions match an abstract machine configuration, only the first is triggered. This ordering allows the removal of non-determinism without requiring tedious repetition of negated conditions of previous rules with the same heads.

Insertion configurations are tuples of the form $\langle C\,,\,r\,,\,\pi\,,\,\sigma\rangle$ where C is a command, r is a page identifier, π is a stack of pairs of the form (q,i), where q is a page identifier, i is an integer, and σ is a page store.

The grammar for the B+ tree insertion command terms is as follows:

$$\mathit{InsertionCommand}\ ::=\ \mathtt{Insert}\,(\mathit{Entry})\mid \mathsf{S}\mid \mathtt{D}\,(\mathit{Key},\mathit{PageID})\mid \mathtt{Ret}$$

For sequences $\boldsymbol{s}$ and $\boldsymbol{t}$, element a, and for $1 \leqslant i, j \leqslant |\boldsymbol{s}|$ and $1 \leqslant k \leqslant |\boldsymbol{s}|+1$ and predicate P, we define:

$$
\begin{aligned}
\operatorname{dom} \boldsymbol{s} &::= \{ i \in \mathbb{N} \mid 1 \leqslant i \leqslant |\boldsymbol{s}| \} \\
\operatorname{elems} \boldsymbol{s} &::= \{ \boldsymbol{s}_i \mid i \in \operatorname{dom} \boldsymbol{s} \} \\
\boldsymbol{s} \oplus \boldsymbol{t} &::= \begin{cases} |\boldsymbol{s}| = 0 & \boldsymbol{t} \\ |\boldsymbol{t}| = 0 & \boldsymbol{s} \\ |\boldsymbol{s}| \neq 0 \wedge |\boldsymbol{t}| \neq 0 & [\boldsymbol{s}_1, \ldots, \boldsymbol{s}_{|\boldsymbol{s}|}, \boldsymbol{t}_1, \ldots, \boldsymbol{t}_{|\boldsymbol{t}|}] \end{cases} \\
a :: \boldsymbol{t} &::= [a] \oplus \boldsymbol{t} \\
\boldsymbol{s}_{i \cdot\cdot j} &::= \begin{cases} i > j & [\,] \\ i \leqslant j & [\boldsymbol{s}_i, \boldsymbol{s}_{i+1}, \ldots, \boldsymbol{s}_j] \end{cases} \\
\texttt{ins}\,(a, i, \boldsymbol{s}) &::= \boldsymbol{s}_{1 \cdot\cdot i-1} \oplus [a] \oplus \boldsymbol{s}_{i \cdot\cdot |\boldsymbol{s}|} \\
\texttt{del}\,(i, \boldsymbol{s}) &::= \boldsymbol{s}_{1 \cdot\cdot i-1} \oplus \boldsymbol{s}_{i+1 \cdot\cdot |\boldsymbol{s}|} \\
\texttt{replace}\,(a, i, \boldsymbol{s}) &::= \boldsymbol{s}_{1 \cdot\cdot i-1} \oplus [a] \oplus \boldsymbol{s}_{i+1 \cdot\cdot |\boldsymbol{s}|} \\
\texttt{append}\,(a, \boldsymbol{s}) &::= \boldsymbol{s} \oplus [a] \\
\texttt{first}\,(\boldsymbol{s}, P) &::= \begin{cases} \nexists x \in \operatorname{elems} \boldsymbol{s}.\ P(x) & |\boldsymbol{s}| + 1 \\ \exists x \in \operatorname{elems} \boldsymbol{s}.\ P(x) & \min \{ i \in \operatorname{dom} \boldsymbol{s} \mid P(\boldsymbol{s}_i) \} \end{cases} \\
\texttt{test}\,(k, \boldsymbol{s}, P) &::= \begin{cases} k \in \operatorname{dom} \boldsymbol{s} & P(\boldsymbol{s}_k) \\ k \notin \operatorname{dom} \boldsymbol{s} & \texttt{false} \end{cases}
\end{aligned}
$$

Fig. 2. Definitions

The initial configuration for an insert of an entry a into some B+ tree $\langle r, \sigma \rangle$, where σ is a (page) store and r is the page identifier of the root page of the B+ tree, is $\langle \texttt{Insert}\,(a)\,,\, r\,,\, [\,]\,,\, \sigma \rangle$. A terminal configuration is $\langle \texttt{Ret}\,,\, r'\,,\, [\,]\,,\, \sigma' \rangle$, where the resulting B+ tree is $\langle r', \sigma' \rangle$.

We need two ancillary definitions for use in the insertion rules. These define the policy for splitting the contents of leaf and internal nodes when such a split in necessary.

The first, $\texttt{splitL}\,(i, a, \boldsymbol{e})$, defines the components that are produced when a sequence of entries, $\boldsymbol{e}$, has another entry, a, inserted into it at position i. The components of the result are $\langle \boldsymbol{e}', k, \boldsymbol{e}'' \rangle$, where $\boldsymbol{e}' \oplus \boldsymbol{e}'' = \texttt{ins}\,(a, i, \boldsymbol{e})$, $k = \texttt{key}\,(\boldsymbol{e}''_0)$ and $||\boldsymbol{e}'| - |\boldsymbol{e}''|| \leqslant 1$.

The second, $\texttt{splitI}\,(i, k, q, \boldsymbol{d}, \boldsymbol{p})$, defines the components, $\langle \boldsymbol{d}', \boldsymbol{p}', k', \boldsymbol{d}'', \boldsymbol{p}'' \rangle$, that are produced when a key, k, and page identifier, q, are inserted into the paired key and page identifier sequences, $\langle \boldsymbol{d}, \boldsymbol{p} \rangle$ with $|\boldsymbol{p}| = |\boldsymbol{d}| + 1$, of an internal node at position i, which must then be split.

The components of the result, $\langle \boldsymbol{d}', \boldsymbol{p}', k', \boldsymbol{d}'', \boldsymbol{p}'' \rangle$, satisfy the conditions: $\boldsymbol{d}' \oplus [k'] \oplus \boldsymbol{d}'' = \texttt{ins}\,(k, i, \boldsymbol{d})$, $\boldsymbol{p}' \oplus \boldsymbol{p}'' = \texttt{ins}\,(q, i+1, \boldsymbol{p})$, $|\boldsymbol{p}'| = |\boldsymbol{d}'| + 1$, $|\boldsymbol{p}''| = |\boldsymbol{d}''| + 1$, and $||\boldsymbol{p}'| - |\boldsymbol{p}''|| \leqslant 1$.

The abstract machine transition rules for insertion are described in figure 3.

$$
\begin{array}{|rcl|}
\hline
\langle \mathtt{Insert}\,(a)\,,\, r\,,\, \pi\,,\, \sigma \rangle & \leadsto & \langle \mathtt{Insert}\,(a)\,,\, \boldsymbol{p}_i\,,\, (r,i) :: \pi\,,\, \sigma \rangle \\
 & & \text{if } \sigma(r) = \mathtt{I}\,(\boldsymbol{d}\,;\boldsymbol{p}) \\
 & & \text{where } i = \mathtt{first}\,(\boldsymbol{d}, \lambda x.\, x > \mathtt{key}\,(a)) \\
\hline
\langle \mathtt{Insert}\,(a)\,,\, r\,,\, \pi\,,\, \sigma \rangle & \leadsto & \langle \mathtt{S}\,,\, r\,,\, \pi\,,\, \sigma[r \mapsto \mathtt{L}\,(\mathtt{replace}\,(a,i,\boldsymbol{e})\,;f)] \rangle \\
 & & \text{if } \mathtt{test}\,(i, \boldsymbol{e}, \lambda x.\, \mathtt{key}\,(x) = \mathtt{key}\,(a)) \\
 & & \text{where } \sigma(r) = \mathtt{L}\,(\boldsymbol{e}\,;f) \\
 & & \text{and } i = \mathtt{first}\,(\boldsymbol{e}, \lambda x.\, \mathtt{key}\,(x) \geqslant \mathtt{key}\,(a)) \\
\langle \mathtt{Insert}\,(a)\,,\, r\,,\, \pi\,,\, \sigma \rangle & \leadsto & \langle \mathtt{S}\,,\, r\,,\, \pi\,,\, \sigma[r \mapsto \mathtt{L}\,(\mathtt{ins}\,(a,i,\boldsymbol{e})\,;f)] \rangle \\
 & & \text{if } |\boldsymbol{e}| < \mathit{MaxN} \\
 & & \text{where } \sigma(r) = \mathtt{L}\,(\boldsymbol{e}\,;f) \\
 & & \text{and } i = \mathtt{first}\,(\boldsymbol{e}, \lambda x.\, \mathtt{key}\,(x) \geqslant \mathtt{key}\,(a)) \\
\langle \mathtt{Insert}\,(a)\,,\, r\,,\, \pi\,,\, \sigma \rangle & \leadsto & \langle \mathtt{D}\,(k,q)\,,\, r\,,\, \pi\,,\, \sigma[r \mapsto \mathtt{L}\,(\boldsymbol{e}'\,;q)\,,\, q \mapsto \mathtt{L}\,(\boldsymbol{e}''\,;f)] \rangle \\
 & & \text{where } \sigma(r) = \mathtt{L}\,(\boldsymbol{e}\,;f) \\
 & & \text{and } i = \mathtt{first}\,(\boldsymbol{e}, \lambda x.\, \mathtt{key}\,(x) \geqslant \mathtt{key}\,(a)) \\
 & & \text{and } \langle \boldsymbol{e}', k, \boldsymbol{e}'' \rangle = \mathtt{splitL}\,(i, a, \boldsymbol{e}) \\
 & & \text{and } q \notin \mathrm{dom}(\sigma) \\
\hline
\langle \mathtt{S}\,,\, r\,,\, (t,i) :: \pi\,,\, \sigma \rangle & \leadsto & \langle \mathtt{S}\,,\, t\,,\, \pi\,,\, \sigma \rangle \\
\langle \mathtt{D}\,(k,q)\,,\, r\,,\, (t,i) :: \pi\,,\, \sigma \rangle & \leadsto & \langle \mathtt{S}\,,\, t\,,\, \pi\,,\, \sigma[t \mapsto \mathtt{I}\,(\mathtt{ins}\,(k,i,\boldsymbol{d})\,;\mathtt{ins}\,(q,i+1,\boldsymbol{p}))] \rangle \\
 & & \text{if } |\boldsymbol{p}| < \mathit{MaxN} \\
 & & \text{where } \sigma(t) = \mathtt{I}\,(\boldsymbol{d}\,;\boldsymbol{p}) \\
\langle \mathtt{D}\,(k,q)\,,\, r\,,\, (t,i) :: \pi\,,\, \sigma \rangle & \leadsto & \langle \mathtt{D}\,(k',q')\,,\, t\,,\, \pi\,,\, \sigma[t \mapsto \mathtt{I}\,(\boldsymbol{d}'\,;\boldsymbol{p}')\,,\, q' \mapsto \mathtt{I}\,(\boldsymbol{d}''\,;\boldsymbol{p}'')] \rangle \\
 & & \text{where } \sigma(t) = \mathtt{I}\,(\boldsymbol{d}\,;\boldsymbol{p}) \\
 & & \text{and } \langle \boldsymbol{d}', \boldsymbol{p}', k', \boldsymbol{d}'', \boldsymbol{p}'' \rangle = \mathtt{splitI}\,(i,k,q,\boldsymbol{d},\boldsymbol{p}) \\
 & & \text{and } q' \notin \mathrm{dom}(\sigma) \\
\hline
\langle \mathtt{S}\,,\, r\,,\, [\,]\,,\, \sigma \rangle & \leadsto & \langle \mathtt{Ret}\,,\, r\,,\, [\,]\,,\, \sigma \rangle \\
\langle \mathtt{D}\,(k,t)\,,\, r\,,\, [\,]\,,\, \sigma \rangle & \leadsto & \langle \mathtt{Ret}\,,\, q\,,\, [\,]\,,\, \sigma[q \mapsto \mathtt{I}\,([k]\,;[r,t])] \rangle \\
 & & \text{where } q \notin \mathrm{dom}(\sigma) \\
\hline
\end{array}
$$

Fig. 3. B+ tree insertion rules

They are split into 4 sections. The first contains the single rule for descending down the correct path of internal nodes in the tree, while pushing the path location information on the stack at each step. The second section describes the three cases that can occur when a leaf page is encountered: (i) The entry to be inserted has the same key value as an existing entry in the page and so replaces the existing entry and no further action is necessary (a single page result has occurred). (ii) The entry fits into the page so the page is updated. Again no further action is necessary as a single page result has occurred. (iii) The entry did not fit and the page had to be split between the original leaf page and a new one. As a double page result has occurred, a new key/page pointer pair has to be inserted into the parent level

The third section describes the possible ripple in post operations up the tree. Again there are three cases, each popping the parent location off the stack: (i) If the result from the level below was a single page, there is no change necessary in this level so we pass on a single page result to the level above. (ii) If the result from below was a double page, we have to insert a new pair in this level. If it fits then we return a single page result to the level above. (iii) If the result from below was a double page, and there is insufficient space in this node to insert the new key/pointer pair, then we have to split this page and return a double page result to the level above.

The final section specifies the behaviour when the upward rippling finds the stack to be empty. At this point the system is trying to return a result from the root page level: Either the old root has not been split, in which case the root of the new tree is the old root itself, or the root page has been split, in which case a new root page has to be constructed and made to point to the two sub-trees.

The main result for insertion (Theorem 3.2 below) states that the insert command, if run on a well-formed B+ tree in the store, leaves a tree in the store that also contains the new entry. To be more precise, it could happen that an old entry with the same key could have been overwritten; we introduce the notation

$$S + e = S \setminus \{e' \mid \texttt{key}\left(e'\right) = \texttt{key}\left(e\right)\} \cup \{e\}$$

to state this insertion of entries. We need to show that when the machine starts with an insertion $\left\langle \texttt{Insert}\left(e\right), r, [\,], \sigma\right\rangle$, it transforms its initial store satisfying $\sigma \models \mathsf{ComBtree}(r, S)$ into a new store σ' satisfying $\sigma' \models \mathsf{ComBtree}(q, S + e)$. In the proof, we reason, not about individual machine steps, but about longer transition sequences that process a whole subtree (if the subtree is a leaf, then the transition sequence is actually a single step). This is essential, as it allows us to treat the rest of the store with a sort of frame property as we descend the tree to focus on smaller subtrees. We need the following lemma to make the induction go through:

Lemma 3.1 *Let R be any predicate on stores and assume $\sigma \models \mathsf{Btree}_h(r, S, a, z, n) * R$. Then one of the following holds:*

(i) *There is a transition sequence*

$$\left\langle \texttt{Insert}\left(e\right), r, \pi, \sigma\right\rangle \rightsquigarrow \cdots \rightsquigarrow \left\langle \texttt{S}, r, \pi, \sigma'\right\rangle$$

and $\sigma' \models \mathsf{Btree}_h(r, S + e, a, z, n') * R$ and either $n' = n$ or $n' = n + 1$.

(ii) *There is a transition sequence*

$$\left\langle \texttt{Insert}\left(e\right), r, \pi, \sigma\right\rangle \rightsquigarrow \cdots \rightsquigarrow \left\langle \texttt{D}\left(k, q\right), r, \pi, \sigma'\right\rangle$$

where $\sigma' \models \mathsf{Btree}_h(r, S_r, a, b, n') * \mathsf{Btree}_h(q, S_q, b, z, n'') * R$. Moreover, $S_r \cup S_q = S + e$ and $S_r \sqsubset k \sqsubseteq S_q$ and $MinN \leqslant n'$ and $MinN \leqslant n''$.

Proof. By induction over the height of the B+ tree. We sketch the induction, emphasising the spatial logic part, while eliding some straightforward checking of side conditions.

Assume that $\sigma \models \mathsf{Btree}_1(r, S, a, z, n) * R$. and consider the configuration before the transition:

$$\left\langle \texttt{Insert}\left(e\right), r, \pi, \sigma\right\rangle$$

By Definition 2.1, if the height of the tree is 1, then r points to a leaf node, that is, $\sigma \models \mathsf{Btree}_1(r, S, a, z, n) * R$ implies $\sigma = \{r \mapsto \texttt{L}\left(\boldsymbol{e}\ ;\ z\right)\} * \sigma_R$, where $\sigma_R \models R$.

If the transition leads to an S command, this gives us a sequence of length 1:

$$\left\langle \texttt{Insert}\left(e\right), r, \pi, \sigma\right\rangle \rightsquigarrow \left\langle \texttt{S}, r, \pi, \sigma'\right\rangle$$

where only r has been updated in σ', so that σ_R remains unchanged. Hence $\sigma' \models \mathsf{Btree}_1(r, S + e, a, z, n') * R$. Furthermore, the two possible transitions that lead to an S command when r points to a leaf node either replace a single entry in the node or insert one extra entry, so $n' = n$ or $n' = n + 1$.

Now suppose the transition leads to a D command:

$$\left\langle \texttt{Insert}\,(e)\,,\, r\,,\, \pi\,,\, \sigma \right\rangle \rightsquigarrow \left\langle \texttt{D}\,(k, q)\,,\, r\,,\, \pi\,,\, \sigma' \right\rangle$$

where $\sigma' = \sigma[r \mapsto \texttt{L}\,(\boldsymbol{e}'\,;\, q)\,, q \mapsto \texttt{L}\,(\boldsymbol{e}''\,;\, z)]$. Then r has been updated, q is fresh, thus not affecting σ_R. Further, $|e'| + |e''| = n + 1$ and, by the requirements of splitL, $\mathit{MinN} \leqslant |e'|$ and $\mathit{MinN} \leqslant |e''|$. We have

$$\sigma' \models \mathsf{Btree}_1(r, S_r, r, q, n') * \mathsf{Btree}_1(q, S_q, q, z, n'') * R$$

and $\mathit{MinN} \leqslant n'$ and $\mathit{MinN} \leqslant n''$ as required.

Next, suppose the tree has a height greater than 1: $\sigma \models \mathsf{Btree}_{h+1}(r, S, a, z, n) * R$. That implies that r points to an internal node, $\sigma(r) = \texttt{I}\,(\boldsymbol{d}\,;\, \boldsymbol{p})$. Hence we have a transition

$$\left\langle \texttt{Insert}\,(e)\,,\, r\,,\, \pi\,,\, \sigma \right\rangle \rightsquigarrow \left\langle \texttt{Insert}\,(e)\,,\, \boldsymbol{p}_i\,,\, (r, i) :: \pi\,,\, \sigma \right\rangle$$

Now $\sigma \models \mathsf{Btree}_{h+1}(r, S, a, z, n) * R$ implies

$$\sigma \models r \mapsto \texttt{I}\,(\boldsymbol{d}\,;\, \boldsymbol{p}) * \mathsf{Btree}_h(\boldsymbol{p}_1, S_1, a_1, a_2, m_1) * \cdots * \mathsf{Btree}_h(\boldsymbol{p}_n, S_n, a_n, a_{n+1}, m_n) * R$$

with $a_{n+1} = z$. Let R' describe the store with the subtree rooted at $\boldsymbol{p}_i$ removed:

$$\begin{aligned} R' = {} & r \mapsto \texttt{I}\,(\boldsymbol{d}\,;\, \boldsymbol{p}) \\ & * \mathsf{Btree}_h(\boldsymbol{p}_1, S_1, a_1, a_2, m_1) * \cdots * \mathsf{Btree}_h(\boldsymbol{p}_{i-1}, S_{i-1}, a_{i-1}, a_i, m_{i-1}) \\ & * \mathsf{Btree}_h(\boldsymbol{p}_{i+1}, S_{i+1}, a_{i+1}, a_{i+2}, m_{i+1}) * \cdots * \mathsf{Btree}_h(\boldsymbol{p}_n, S_n, a_n, a_{n+1}, m_n) \\ & * R \end{aligned}$$

We can then plug the subtree back in, giving $\sigma \models \mathsf{Btree}_h(\boldsymbol{p}_i, S_i, a_i, a_{i+1}, m_i) * R'$, as $*$ is commutative and associative. We can therefore apply the induction hypothesis for trees of height h and the predicate R'. There are two possible transition sequences, leading either to a S or a $\texttt{D}\,(\cdot, \cdot)$ configuration. We need to show that both these possible resulting configurations lead on further to an S or a $\texttt{D}\,(\cdot, \cdot)$ configuration that matches the pattern described in the lemma for trees of height $h + 1$:

(i) First, suppose we obtained a S configuration:

$$\left\langle \texttt{Insert}\,(e)\,,\, \boldsymbol{p}_i\,,\, (r, i) :: \pi\,,\, \sigma \right\rangle \rightsquigarrow \cdots \rightsquigarrow \left\langle \texttt{S}\,,\, \boldsymbol{p}_i\,,\, (r, i) :: \pi\,,\, \sigma' \right\rangle$$

where $\sigma' \models \mathsf{Btree}_h(\boldsymbol{p}_i, S_i + e, a_i, a_{i+1}, m_i{}') * R'$. Then the next transition pops the stack and restores r:

$$\left\langle \texttt{S}\,,\, \boldsymbol{p}_i\,,\, (r, i) :: \pi\,,\, \sigma' \right\rangle \rightsquigarrow \left\langle \texttt{S}\,,\, r\,,\, \pi\,,\, \sigma' \right\rangle$$

Combining the above transitions, we have

$$\left\langle \texttt{Insert}\,(e)\,,\, r\,,\, \pi\,,\, \sigma \right\rangle \rightsquigarrow \cdots \rightsquigarrow \left\langle \texttt{S}\,,\, r\,,\, \pi\,,\, \sigma' \right\rangle$$

where $\sigma' \models \mathsf{Btree}_{h+1}(r, S+e, a, z, n') * R$. Further, $n' = n$ as the internal node pointed to by r has not changed. This gives us a transition sequence of the required form.

(ii) Next, suppose that the induction hypothesis tells us that the transition sequence lead to a $\mathtt{D}\,(\cdot,\cdot)$ configuration:

$$\left\langle \mathtt{Insert}\,(e)\,,\, \boldsymbol{p}_i\,,\, (r,i)\ ::\ \pi\,,\, \sigma \right\rangle \rightsquigarrow \cdots \rightsquigarrow \left\langle \mathtt{D}\,(k,q)\,,\, \boldsymbol{p}_i\,,\, (r,i)\ ::\ \pi\,,\, \sigma' \right\rangle$$

and also that the store satisfies

$$\sigma' \models \mathsf{Btree}_h(\boldsymbol{p}_i, S', a_i, b, {m_i}') * \mathsf{Btree}_h(q, S_q, b, a_{i+1}, {m_i}'') * R'$$

From this $\mathtt{D}\,(\cdot,\cdot)$ configuration, there are two possible next transitions, depending on whether the key/page pointer pair fits or the page has to be split.

Assume the former. In this case the transition is to an $\mathtt{S}$ configuration:

$$\left\langle \mathtt{D}\,(k,q)\,,\, \boldsymbol{p}_i\,,\, (r,i)\ ::\ \pi\,,\, \sigma' \right\rangle \rightsquigarrow \left\langle \mathtt{S}\,,\, r\,,\, \pi\,,\, \sigma'' \right\rangle$$

where

$$\sigma'' = \sigma'[r \mapsto \mathtt{I}\,(\mathtt{ins}\,(k, i, \boldsymbol{d})\ ;\ \mathtt{ins}\,(q, i+1, \boldsymbol{p}))]$$

Then $\sigma'' \models \mathsf{Btree}_{h+1}(r, S+e, a, z, n') * R$, where $n' = n+1$. Thus, in the updated store, r points to a B+ tree of height $h+1$ with the split subtrees inserted in their correct places.

In the latter case the transition is to a $\mathtt{D}\,(\cdot,\cdot)$ configuration:

$$\left\langle \mathtt{D}\,(k,q)\,,\, \boldsymbol{p}_i\,,\, (r,i)\ ::\ \pi\,,\, \sigma' \right\rangle \rightsquigarrow \left\langle \mathtt{D}\left(k', q'\right)\,,\, r\,,\, \pi\,,\, \sigma'' \right\rangle$$

where

$$\sigma'' = \sigma'[r \mapsto \mathtt{I}\left(\boldsymbol{d'}\ ;\ \boldsymbol{p'}\right), q' \mapsto \mathtt{I}\left(\boldsymbol{d''}\ ;\ \boldsymbol{p''}\right)]$$

and $\left\langle \boldsymbol{d}', \boldsymbol{p}', k', \boldsymbol{d}'', \boldsymbol{p}'' \right\rangle = \mathtt{splitI}\,(i, k, q, \boldsymbol{d}, \boldsymbol{p})$. By the definition of $\mathtt{splitI}$, and given that $|\boldsymbol{p}| = \mathit{MaxN}$, we have that $\lceil \mathit{MaxN}/2 \rceil \leqslant n'$ and $\lceil \mathit{MaxN}/2 \rceil \leqslant n''$, $\sigma'' \models \mathsf{Btree}_{h+1}(r, S_r, a, b, n') * \mathsf{Btree}_{h+1}(q', S_{q'}, b, z, n'') * R$ and $S_r \cup S_{q'} = S+e$. □

Given Lemma 3.1, we now prove correctness of insertion:

Theorem 3.2 *Assume that* $\sigma \models \mathsf{ComBtree}(r, S)$. *Then*

$$\left\langle \mathtt{Insert}\,(e)\,,\, r\,,\, []\,,\, \sigma \right\rangle \rightsquigarrow \cdots \rightsquigarrow \left\langle \mathtt{Ret}\,,\, q\,,\, []\,,\, \sigma' \right\rangle$$

and $\sigma' \models \mathsf{ComBtree}(q, S+e)$.

Proof. Suppose $\sigma \models \mathsf{ComBtree}(r, S)$. Then for some h and a, we have $\sigma \models \mathsf{Btree}_h(r, S, a, \mathtt{null}, n) * \mathtt{emp}$, since $\mathtt{emp}$ is the neutral element of $*$.

We apply Lemma 3.1, with $R = \mathtt{emp}$. There are two possible cases, leading to an $\mathtt{S}$ or a $\mathtt{D}\,(\cdot,\cdot)$ configuration. In the first case, we have

$$\begin{array}{rl} & \langle \mathtt{Insert}\,(e)\,,\,r\,,\,[\,]\,,\,\sigma \rangle \\ \rightsquigarrow \cdots \rightsquigarrow & \langle \mathtt{S}\,,\,r\,,\,[\,]\,,\,\sigma' \rangle \\ \rightsquigarrow & \langle \mathtt{Ret}\,,\,r\,,\,[\,]\,,\,\sigma' \rangle \end{array}$$

and $\sigma' \models \mathsf{Btree}_h(r, S+e, a, \mathtt{null}, n') * \mathtt{emp}$ where $n' = n$ or $n' = n+1$. That implies $\sigma' \models \mathsf{ComBtree}(r, S+e)$, and we are done with this case.

Now consider the case that the last node had to be split, resulting in a $\mathtt{D}\,(\cdot,\cdot)$ configuration. Then we have transitions

$$\begin{array}{rl} & \langle \mathtt{Insert}\,(e)\,,\,r\,,\,[\,]\,,\,\sigma \rangle \\ \rightsquigarrow \cdots \rightsquigarrow & \langle \mathtt{D}\,(k,q)\,,\,r\,,\,[\,]\,,\,\sigma' \rangle \\ \rightsquigarrow & \langle \mathtt{Ret}\,,\,t\,,\,[\,]\,,\,\sigma'' \rangle \end{array}$$

where

$$\sigma' \models \mathsf{Btree}_h(r, S_r, a, b, n') * \mathsf{Btree}_h(q, S_q, b, \mathtt{null}, n'') * \mathtt{emp}$$

with $S_r \cup S_q = S+e$, and furthermore $\sigma'' = \sigma'[t \mapsto \mathtt{I}\,([k]\;;\;[r,q])]$ for some $t \notin \mathrm{dom}\,\sigma'$. Then the store with the newly allocated node satisfies:

$$\sigma'' \models t \mapsto \mathtt{I}\,([k]\;;\;[r,q]) * \mathsf{Btree}_h(r, S_r, a, b, n') * \mathsf{Btree}_h(q, S_q, b, \mathtt{null}, n'')$$

hence $\sigma'' \models \mathsf{Btree}_{h+1}(t, a, S+e, \mathtt{null}, 2)$ and thus $\sigma'' \models \mathsf{ComBtree}(t, S+e)$, as required. Thus either the insertion completes by leaving the occupancy of the root page unchanged or increased by one, or the root page was already full and inserting the new subtree caused it to split into two nodes, each at least half full, and a new root node containing precisely two subtrees is grafted on top. □

4 Find as abstract machine rules

We consider a range query that takes a lower and an upper bound key value, and returns a list of entries. This can be specified in two phases, such that the first phase takes only the lower bound key value and descends the tree to find the first leaf page that can contain matching entries. We describe the transition rules for the first phase in Figure 4. The second phase simply iterates across the linked list of pages at the leaf level extracting matching entries until the upper bound is reached. As it has been well discussed in the literature, we omit the details of the iterator phase here. The operational semantics of these two phases can be verified separately, and connected only at the level of predicates. Moreover, logically the first phase is more interesting, as it starts with a tree predicate and successively "transfers" leaf nodes into a list predicate, in the sense of the "transfer of ownership" concept in separation logic.

For reasoning about the list of results of a find operation, we need a list predicate: $\mathsf{FList}(p, i, S)$ means that p points to a list of leaf nodes, such that S is the set of all the entries from the i-th position in the first node of the list and all entries in all subsequent nodes. It is essentially a standard list predicate, apart from the additional index into the first sequence of entries:

$$
\begin{aligned}
&\left\langle \texttt{Find}\,(k)\,,\, r\,,\, \sigma \right\rangle \rightsquigarrow \left\langle \texttt{Find}\,(k)\,,\, \boldsymbol{p}_i\,,\, \sigma \right\rangle \\
&\qquad\qquad\qquad\qquad \text{if } \sigma(r) = \texttt{I}\,(\boldsymbol{d}\,;\boldsymbol{p}) \\
&\qquad\qquad\qquad\qquad \text{where } i = \texttt{first}\,(\boldsymbol{d}, \lambda x.\ x > k) \\
&\left\langle \texttt{Find}\,(k)\,,\, r\,,\, \sigma \right\rangle \rightsquigarrow \langle \texttt{Ret}(r,i), \sigma \rangle \\
&\qquad\qquad\qquad\qquad \text{where } \sigma(r) = \texttt{L}\,(\boldsymbol{e}\,;f) \\
&\qquad\qquad\qquad\qquad \text{and } i = \texttt{first}\,(\boldsymbol{e}, \lambda x.\ \texttt{key}\,(x) \geqslant k)
\end{aligned}
$$

Fig. 4. B+ tree find rules

Definition 4.1

$$
\begin{aligned}
\mathsf{FList}(r,i,S) \iff\ & (r = \texttt{null} \land S = \emptyset \land \texttt{emp}) \\
\lor\ & \big(\exists f, e_1, \ldots, e_n, S_f. \\
& (r \mapsto \texttt{L}\,(e_1, \ldots, e_n\ ;\ f) * \mathsf{FList}(f, 1, S_f)) \\
& \land S = S_f \cup \{e_i, \ldots, e_n\}\big)
\end{aligned}
$$

We will also need to filter out all those entries from a set that are greater than the lower bound of a query:

Definition 4.2 For a set of entries S and key k, let

$$S \uparrow k = \{e \mid e \in S \land \texttt{key}\,(e) \geqslant k\}.$$

The correctness of find (Theorem 4.5 below) that we are aiming for states that starting the machine as $\left\langle \texttt{Find}\,(k)\,,\, r\,,\, \sigma \right\rangle$ in a store satisfying $\sigma \models \mathsf{ComBtree}(r, S)$ results in a final configuration $\texttt{Ret}(q,i)$, so that q and i identify the start of the list of results. If there are no entries with keys greater than k, $\texttt{Find}\,(k)$ returns the address of the last leaf node together with an index one past the end of the entry sequence of that page. In that case the list predicate holds only for an empty set of entries, which is the appropriate result.

In reasoning about the find operation, we need to be able to append the fringe of a tree to a list, as stated by Lemma 4.3:

Lemma 4.3 *Suppose*

$$\sigma \models \mathsf{Btree}_h(r, S_r, a, z, n) * \mathsf{FList}(z, 1, S_z) * \texttt{true}$$

Then $\sigma \models \mathsf{FList}(a, 1, S_r \cup S_z) * \texttt{true}$.

Proof. By induction on h. For $h > 1$, there is a nested induction over the number of children of the top internal node, pointed to by r. □

The correctness proof for find relies on a lemma (Lemma 4.4) that generalizes it to make the induction go through. It partitions the store into three disjoint parts: the current B+ tree, a list of leaf nodes to the right of the current B+ tree, and everything else, expressed with the catch-all predicate $\texttt{true}$. During the descent of the tree by $\texttt{Find}\,(k)$ transitions, the list acts like a data structure continuation or an accumulator, to put it in functional programming terminology. As $\mathtt{Find}\,(k)$ descends the tree, nodes are transfered into the accumulator.

Lemma 4.4 *Let* $\sigma \models \mathsf{Btree}_h(p, S_p, a, z, n) * \mathsf{FList}(z, 1, S_z) * \mathtt{true}$. *Then we have a sequence of* h *transitions*

$$\left\langle \mathtt{Find}\,(k)\,,\, p\,,\, \sigma \right\rangle \rightsquigarrow \cdots \rightsquigarrow \langle \mathtt{Ret}(q, i), \sigma\rangle$$

for some q *and* i *with* $\sigma \models \mathsf{FList}(q, i, (S_p \uparrow k) \cup S_z) * \mathtt{true}$.

Proof. Induction on the height of the B+ tree.

Suppose the height of the tree is 1, so $\sigma \models \mathsf{Btree}_1(r, S_r, a, z, n) * \mathsf{FList}(z, 1, S_z) * \mathtt{true}$. That implies $r \mapsto \mathtt{L}\,(e_1 \dots e_n\ ;\, z)$ with $S_r = \{\boldsymbol{e}_1, \dots, \boldsymbol{e}_n\}$. Then there is one transition step

$$\left\langle \mathtt{Find}\,(k)\,,\, r\,,\, \sigma \right\rangle \rightsquigarrow \langle \mathtt{Ret}(r, i), \sigma\rangle$$

such that $S_r \uparrow k = \{e_i, \dots, e_n\}$. We have $\sigma \models \mathsf{FList}(r, i, (S_r \uparrow k) \cup S_z) * \mathtt{true}$, as required.

Now suppose the height of the tree is $h + 1 > 1$, so that the store satisfies

$$\sigma \models \mathsf{Btree}_{h+1}(r, S, a, z, n) * \mathsf{FList}(z, 1, S_z) * \mathtt{true}$$

Unrolling the definition of B+ trees once, we see that there is an internal node at the top, so that

$$\begin{aligned} \sigma \models\, & r \mapsto \mathtt{I}\,(\boldsymbol{d}_1 \dots \boldsymbol{d}_{n-1}\ ;\, \boldsymbol{p}_1 \dots \boldsymbol{p}_n) \\ & * \mathsf{Btree}_h(\boldsymbol{p}_1, S_1, a_1, a_2, m_1) \\ & * \mathsf{Btree}_h(\boldsymbol{p}_2, S_2, a_2, a_3, m_2) \\ & * \cdots \\ & * \mathsf{Btree}_h(\boldsymbol{p}_n, S_n, a_n, a_{n+1}, m_n))) \\ & * \mathsf{FList}(z, 1, S_z) * \mathtt{true} \end{aligned}$$

where $a_1 = a$ and $a_{n+1} = z$. Hence the next transition step is of the form

$$\left\langle \mathtt{Find}\,(k)\,,\, r\,,\, \sigma \right\rangle \rightsquigarrow \left\langle \mathtt{Find}\,(k)\,,\, \boldsymbol{p}_i\,,\, \sigma \right\rangle$$

such that $S \uparrow k = (S_i \uparrow k) \cup S_{i+1} \cup \cdots \cup S_n$. Notice that

$$\begin{aligned} \sigma \models\ & \mathsf{Btree}_h(\boldsymbol{p}_i, S_i, a_i, a_{i+1}, m_i) \\ & * \mathsf{Btree}_h(\boldsymbol{p}_{i+1}, S_{i+1}, a_{i+1}, a_{i+2}, m_{i+1}) \\ & * \cdots \\ & * \mathsf{Btree}_h(\boldsymbol{p}_n, S_n, a_n, a_{n+1}, m_n))) \\ & * \mathsf{FList}(z, 1, S_z) * \mathtt{true} \end{aligned}$$

To see this, split the store into two parts $\sigma = \sigma_0 * \sigma_1$ such that σ_1 satisfies the above formula. Since $\sigma_0 \models \mathtt{true}$, and the formula contains $\dots * \mathtt{true}$, all of σ satisfies it as well. Intuitively, the internal node as well as all leaf nodes on the left (below the lower bound of the query) are thus swept into `true` to be ignored.

By Lemma 4.3, applied $(n-i)$ times to σ, we conclude that

$$\sigma \models \mathsf{Btree}_h(\boldsymbol{p}_i, S_i, a_i, a_{i+1}, m_i) * \mathsf{FList}(a_{i+1}, 1, S_{i+1} \cup \cdots \cup S_n \cup S_z) * \mathtt{true}$$

(Recall that $a_{n+1} = z$.) Intuitively, the fringes of the trees to the right are appended to the accumulator. With the preceding gerrymandering of the store, we can now apply the induction hypothesis for trees of height h to the tree rooted at $\boldsymbol{p}_i$ together with the list starting from a_{i+1}. That gives us a transition sequence of length h

$$\left\langle \mathtt{Find}\,(k)\,,\, \boldsymbol{p}_i\,,\, \sigma \right\rangle \rightsquigarrow \cdots \rightsquigarrow \langle \mathtt{Ret}(t,j), \sigma \rangle$$

where $\sigma \models \mathsf{FList}(t, j, (S_i \uparrow k) \cup S_{i+1} \cup \cdots \cup S_n \cup S_z) * \mathtt{true}$. That is, we have

$$\left\langle \mathtt{Find}\,(k)\,,\, r\,,\, \sigma \right\rangle \rightsquigarrow \cdots \rightsquigarrow \langle \mathtt{Ret}(t,j), \sigma \rangle$$

in $h+1$ steps, where $\sigma \models \mathsf{FList}(t, j, (S \uparrow k) \cup S_z) * \mathtt{true}$ as required. □

The required correctness of the find operation now arises as a special case:

Theorem 4.5 *If* $\sigma \models \mathsf{ComBtree}(r, S)$, *then*

$$\left\langle \mathtt{Find}\,(k)\,,\, r\,,\, \sigma \right\rangle \rightsquigarrow \cdots \rightsquigarrow \langle \mathtt{Ret}(q,i), \sigma \rangle$$

for some q *and* i *with* $\sigma \models \mathsf{FList}(q, i, S \uparrow k) * \mathtt{true}$.

Proof. The theorem follows from Lemma 4.4 for the special case of a complete B+ tree with an empty list: $\sigma \models \mathsf{Btree}_h(r, S, a, \mathtt{null}, n) * \mathsf{FList}(\mathtt{null}, 1, \emptyset) * \mathtt{true}$. □

5 Deletion

A full account of deletion is beyond the scope of this paper. When written out in full detail, the rules are lengthy, since they require attention to a number of corner cases. Broadly speaking, deletion is analogous to insertion. A stack is maintained while descending the tree. To maintain occupancy, nodes may need to be merged with one of their siblings (as opposed to being split for insertion). As in the case for insertion, this process may ripple up the tree and can lead to the tree shrinking in height by replacing its root page with its single remaining child page. The rules for deletion also model the deallocation of storage.

From our perspective, a central question is how we can keep reasoning about the B+ tree invariant locally in this setting. The changes to the store need not be confined to the tree pointed to by the node that the machine is currently processing (as they were for insertion). A node may "steal" entries from its left or right siblings to maintain occupancy. However, this does not mean that there is no locality. Since both the current node and its siblings are children of their common parent node on the stack, this parent node gives us a footprint to which updates are confined.This allows us to prove correctness with the same technique we used for insertion.

6 Conclusions and Future Work

We have shown that B+ trees can be formalised with our techniques building on abstract machines and separating conjunction. B+ trees are instances of the kind of data structures with disciplined sharing that separation logic is well equipped to handle [3]. Whereas the separation logic literature is mainly about Hoare logics, we do not use a formal proof system for code verification. Rather, we reason about invariants semantically. This informal (not proof-theoretic), but nonetheless rigorous approach may also be suitable for more involved index structures.

In related work [10,11], we have developed rules, in our abstract machine style, for the significantly more complex BV-tree [5]. We have also experimented with automatically translating the abstract machine rules into executable code, with the particular aim of obtaining a high performance implementation of the index structure algorithms. Our first prototype was able to successfully generate high-quality correct code for executing insertions on a B+ tree. Work on reimplementing this prototype to extend the rules to the somewhat more complex ones that were required for the BV-tree is planned.

We believe that the present work will scale up to some of the more complicated index structures. Some, such as R-trees [2], require more complex stack manipulations. By adopting a machine that manipulates its stack explicitly, we have a framework in which these trees may also be accommodated. More speculatively, separation logic may also be useful for index structures that appear to have, conceptually, holes in the store which could be expressed with the separation logic connective $-\!\!*$. Example are BV-trees and the "holey brick" (hB) trees [8].

References

[1] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[2] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *ACM SIGMOD*, 322–331, 1990.

[3] R. Bornat, C. Calcagno, and P. O'Hearn. Local reasoning, separation and aliasing. In *SPACE 04 workshop*, 2004.

[4] D. Comer. The ubiquitous B-tree. *Computing Surveys*, 11(2):121–137, June 1979.

[5] M. Freeston. A general solution of the n-dimensional B-tree problem. In *ACM SIGMOD*, 80–91, 1995.

[6] S. Ishtiaq and P. O'Hearn. BI as an assertion language for mutable data structures. In *ACM POPL*. 2001.

[7] P.J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, Jan 1964.

[8] D.B. Lomet and B. Salzberg. The hB-tree: a multiattribute indexing method with good guaranteed performance. *ACM Trans. Database Syst.*, 15(4):625–658, 1990.

[9] J.C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science (LICS)*, 55–74, 2002.

[10] A.P. Sexton and R. Swinbank. Virtual forced splitting, demotion and the BV-tree. In *British National Conference on Databases, BNCOD'08*, LNCS. Springer Verlag, June 2008. to appear.

[11] R. Swinbank. *Virtual Forced Splitting in Multidimensional Access Methods*. PhD thesis, School of Computer Science, University of Birmingham, 2008. Due for completion May 2008.

[12] H. Thielecke. Frame rules from answer types for code pointers. In *ACM POPL*, 309–319. 2006.