Background paper: verifying concurrent indexes

James Fisher

2011, May 3

Abstract

The *index* is a fundamental abstract data type. A large number of algorithms exist that implement it, varying greatly in approach and complexity. As multicore computing becomes standard, indexes allowing concurrent access are called for. A number of concurrent index algorithms exist. Their approaches build on the sequential algorithms, but their complexity significantly exceeds them, and their semantics are less clear.

The importance, complexity and ambiguity of concurrent indexing algorithms make them candidates for formal verification. One success here is da Rocha Pinto *et al.*'s verification [4] of Sagiv's B-Link tree [15] (a well-known concurrent index algorithm). The reasoning here uses separation logic [14] and concurrent abstract predicates [5], which provide the tools to specify the behaviour of concurrent indexes and to reason about algorithms implementing this.

This project applies the same tools to concurrent index algorithms building on the Red-Black tree.

Contents

1 Indexes

1.1 What is an index?

An index[1] stores values, where each value has a distinct name. Many data sets fit this description. For example, your mobile's contact list stores *phone numbers*, and each number is accessed with a person's *name*. At a larger scale, the Web can be seen as a store of *pages*, where each page has a *URL*. Even a language can be seen as an index: a store of *concepts*, where each concept can be accessed by a *word*. More fundamentally, any function stores elements from a *range*, where each element is accessed by a unique number from a *domain*.

1.1.1 The index API

It isn't surprising, then, that almost every programming language has an index data type. [17] (Indeed, in some languages, such as Javascript[2], all objects are indexes.) With its appealingly intuitive interface, the programmer can memoize the results of a function, model the houses on her street, or describe a directed graph. She might, for another example, cache the Web, simply with:[3]

```
typedef Index<Url, Page> Webcache;
Webcache cache;

Page get(Url u) {
  Page p = cache.search(u);

  if (p != null && !p.expired()) return p;

  p = http_get(u);

  if (p == null) cache.remove(u);
  else           cache.insert(u, p);
```

[1] The same data type also goes by the names *map*, *dictionary*, *associative array*, *table*, and *hash*.
[3] [12], p. 2.
[3] I use a syntax similar to C++ in my examples.

```
  return p;
}
```

This small example illustrates the entire API. Let's look at it a bit more formally. The index is an *abstract* data type,[4] and cannot be used as-is; the *key* and the *value* must first be given a type. The resulting *concrete data type* can then be instantiated. (In the above, `Index` is the abstract data type, and `Webcache` is the concrete data type, with `Urls` as keys and `Pages` as values.) We say that an index 'maps `K`s onto `V`s' if its key type is `K` and its value type is `V`. Given an index `i` that maps `K`s onto `V`s, we can specify the semantics of operations on it. In the following, the variable `key` is of type `K`, and the variable `value` is of type `V`:

`value = i.search(key);`[5] If there is a value associated with `key` in `i`, `value` will be that value. Otherwise, `value` will be `null`.[6]

`i->insert(key, value);`[7] Subsequent calls to `i->search(key)` will return `value` (until subsequent calls to `i->insert(k, )` or `i->remove(k)`, where `k == key`).

`i->remove(key);`[8] Subsequent calls to `i.read(key)` will return `null` (until subsequent calls to `i->insert(k, )` or `i->remove(k)`, where `k == key`).

1.1.2 Formalizing behavior: a Set ADT

A correct index implementation will satisfy the behavior described above. To *prove* that one does so, however, we shall have to describe the behavior more precisely.

The basic tool we use to do this is the concept of of *pre- and post-conditions*.[9]In the context of writing a specification, these let us specify the possible valid states of the program before executing a command, and what the state of the program will be after execution of it. For example, we might state that if $x = 2$ prior to executing `x=square(x)`, then $x = 4$ afterwards. This would be written as the "Hoare triple" $\{x = 2\}$ `x=square(x)` $\{x = 4\}$.[10]More complex "commands", such as the operations on an index, can be specified in the same way: $\{...\}$ `i->insert(k, v)` $\{...\}$. For an operation like this, the precondition must capture the valid states of the variables `i`, `k` and `v`.

However, `i`, a pointer to a valid index in memory, can be a problematic thing to capture. First, we don't actually *know* what a valid index looks like, because, as in the above descriptions, we do not want to concern ourselves with these low-level details when describing behavior. Second, even if we did want to describe memory layout, the details of this would be too complex to be practicable in a specification.

The solution is simply to skip over this problem at this stage, and instead use an *abstract predicate* to describe what `i` represents to the programmer using it, namely, "a pointer to something representing an index containing…". This predicate over `i` is written as index(`i`, I).

[4] Also called a *container type*.

[6] The `search` function is also called `find`, `fetch`, `read`, and `get`.

[6] Note that with a `null` type, an index can be seen as a *total* function.

[7] The `insert` function is also called `store`, `set`, `save`, and `add`.

[8] The `remove` function is also called `delete`. It may also be absent, in favour of `insert(k, null)` (suggesting the 'total function' interpretation).

[10] This is a central concept of Hoare logic. [11] There is a lot more to basic Hoare logic, but here I am only specifying behaviour, not proving that anything satisfies it.

[10] Actually, the triple makes no assertions as to whether the command terminates, so the assertion made is strictly that the postcondition will hold *if and when* the command terminates. A second syntax is used for assertions that the command terminates: $[x = 2]$ `x=square(x)` $[x = 4]$. I do not concern myself with this here or in the specification.

A bit more notation is necessary to describe the mappings in an index. However, at this point and with familiar set notation, we can already fully specify a similar abstract data type: the *set*. This will be useful, because a data structure representing an index can be constructed by minimal modification of a set data structure.

The following functions for a Set ADT, and their specifications, should be intuitive. We can `search` for, `insert`, and `remove` elements from the set, just as we can do with mappings in an index. The pre- and postconditions simply describe how and whether the functions mutate the set, and how information is extracted from the structure.

$\{\texttt{set(s,}S\texttt{)} \wedge \texttt{el} \in S\}$	`exists = s->search(el);`	$\{\texttt{set(s,}S\texttt{)} \wedge \texttt{el} \in S \wedge \texttt{exists} = \mathbf{T}\}$
$\{\texttt{set(s,}S\texttt{)} \wedge \texttt{el} \notin S\}$	`exists = s->search(el);`	$\{\texttt{set(s,}S\texttt{)} \wedge \texttt{el} \notin S \wedge \texttt{exists} = \mathbf{F}\}$
$\{\texttt{set(s,}S\texttt{)}\}$	`s->insert(el);`	$\{\texttt{set(s,}S \cup \{\texttt{el}\}\texttt{)}\}$
$\{\texttt{set(s,}S\texttt{)}\}$	`s->remove(el);`	$\{\texttt{set(s,}S \setminus \{\texttt{el}\}\texttt{)}\}$

1.1.3 An index API specification

Specifying an index ADT is only marginally more complex. As with the set$S$, the index I is now purely logical, and we can describe it mathematically without concern for memory layout or local variables. I is a mapping from a set of keys (which is a subset of the universe of keys) to a universe of values. In other words it is a *finite partial function*, written as $I : \mathsf{Keys} \rightharpoonup_{\text{fin}} \mathsf{Values}$.

As Keys is just a set, we can express whether or not a key is in the index using ordinary set notation: $x \in \mathsf{Keys}(I)$ states that there is a mapping in the index from x to some value. As I is a function, we can express a specific mapping using ordinary function notation: $I(x) = y$ states that there is a mapping in the index from x to y.

With just a couple more shorthand notations, we can then express the desired behavior of the index. First, the *function update* notation, $I[k \mapsto v]$, denotes the finite partial function I with the modification that $I(k) = v$. Second, the *set difference* notation $I \setminus \{k\}$ is a shorthand for I with the removal of the key k from $\mathsf{Keys}(I)$.

The specification of the desired index behavior follows.

$\{\mathsf{index(i,}I\mathsf{)} \wedge I(\mathsf{k}) = v\prime\}$	`v = i->search(k);`	$\{\mathsf{index(i,}I\mathsf{)} \wedge I(\mathsf{k}) = v\prime \wedge \mathsf{v} = v\prime\}$
$\{\mathsf{index(i,}I\mathsf{)} \wedge \mathsf{k} \notin \mathsf{Keys}(I)\}$	`v = i->search(k);`	$\{\mathsf{index(i,}I\mathsf{)} \wedge \mathsf{k} \notin \mathsf{Keys}(I) \wedge \mathsf{v} = \textbf{\textit{nil}}\}$
$\{\mathsf{index(i,}I\mathsf{)} \wedge I(\mathsf{k}) = v\prime\}$	`i->insert(k, v);`	$\{\mathsf{index(i,}I\mathsf{)} \wedge I(\mathsf{k}) = v\prime\}$
$\{\mathsf{index(i,}I\mathsf{)} \wedge \mathsf{k} \notin \mathsf{Keys}(I)\}$	`i->insert(k, v);`	$\{\mathsf{index(i,}J\mathsf{)} \wedge I[\mathsf{k} \mapsto \mathsf{v}])\}$
$\{\mathsf{index(i,}I\mathsf{)}\}$	`i->remove(k);`	$\{\mathsf{index(i,}I \setminus \{\mathsf{k}\}\mathsf{)}\}$

1.2 Index and set algorithms

In the previous section, the function calls, and their specifications, did not reveal anything about what `i` looks like "inside": we don't know how memory is laid out, or even what *parts* of memory are allocated for this. Nor did we specify what a call to `search` (for example) actually *does*, and nor did we specify how long it might take (nor even if it halts at all!).

This was necessary and desirable. It was desirable because it enables the programmer to reason about the index as a mathematical construct.[11] It was necessary in order to separate specification from implementation. This is particularly important in the case of indexes and sets, where a large number of data structures exist that satisfy our API. The following is a very partial list:

Array	Association list	Radix tree	van Emde Boas tree
Skip list	Hash table	Binary tree	Splay tree
AVL tree	Bloomier filter	Red-Black Tree	AA-tree
LLRB-tree	B-tree	B^+-tree	B^*-tree

After examining of a few of these, it will become quickly evident that most *index* data structures are simply *set* structures with values "plugged in" next to the key. An index $\rightharpoonup_{\text{fin}} IKV$ can be represented by a set of tuples (k, v) (where $k \in K$ and $v \in V$), with the restriction that no two elements have the same k value. Where the algorithms take this approach, for simplicity I instead describe them as implementing a set.

Each approach has its own performance and memory characteristics. Some have certain restrictions and changes in semantics: that the universe of keys must be finite, for instance; or that multiple values of the same key can exist. A good implementation of a *generic* index—one that implements the API we described, for any universe of keys with an ordering—requires $O(n)$ memory and guarantees $O(log(n))$ time, or better, for each operation (where n is the number of keys in the index).

Let's now run through the features of a few index data structures. We will then look at specifying a few more formally.

1.2.1 Array

The array is undoubtedly the simplest index algorithm: we allocate memory for all members of the universe of keys, and initialize all associated values to `null`. It has $O(1)$ complexity for all operations as the location of the key in memory is a simple offset operation. This is (heavily) paid for in terms of memory, the use of which is $O(u)$, where u is the size of the universe of keys. The array is most commonly used where the key type is `int`, but can be used with other types if each key maps to a unique integer. Additionally, the array is only possible where the universe of keys is finite–strings, for example, are off-limits. An array is sensible where there is a small allowed key range and most or all keys are expected to have a value, *e.g.* a map `char`$\rightharpoonup$`char` for use in a function `string toUpper(string)`. Otherwise, an algorithm with a memory complexity of $O(n)$ is better.

1.2.2 Hash table

The hash table attempts to solve the array's $O(u)$ memory use by choosing an array size and storing more than one key at each location. The key's location in the array (its *bucket*) is decided by a hash function h. Keys in the same bucket are typically chained in a linked list. The function h varies from extremely simple (*e.g.* $h(k) = k \mod (n)$) to complex (*e.g.* cryptographic functions). The hash table has $O(n)$ worst-case behavior in the case that all keys map to the same bucket. By design, it has poor locality of reference—a good hash function spreads similar keys evenly over buckets. There is no obvious ordered iteration algorithm.

[11] In various terminologies, our `Index` API uses "information hiding", also called "encapsulation".

1.2.3 Association list

The association list is a linked list of values of type `(K, V)`. It removes the two disadvantages of the array: its size is $O(n)$, and the key universe does not have to be finite. It pays for this with $O(n)$ time complexity (except for `insert`, where the new element may be appended to the head of the list). The association list might be desirable for its simplicity, and where the number of elements is expected to be short; it is ofted used, for example, in more complex structures including hash tables.

There are many modifications to the basic linked list, and these apply to the association list too. The "linking" can be done in both directions to allow bidirectional traversal.

1.2.4 Binary search tree

The Binary Search Tree (BST) can be seen as an improvement of the linked list, arising from the observation that the nodes of an linked list can be generalized to contain any number of links. Following a link in a linked list only reduces the set being searched by one element; following one link out of a possible two, however, potentially halves the number of elements in the candidate set.

The division made at each node is into elements smaller than the element being looked at and elements larger than it. Therefore, the key type of a BST must have an *ordering*; in practise this does not pose a real problem as arbitrary data can always be lexicographically ordered.

The division that takes place at every step gives the BST a best-case complexity of $\Omega(log(n))$ for all operations.[12]However, the worst-case scenario, in which one of the two sets at every node is the empty set, is equivalent to a linked list, and thus the BST still has $O(n)$ complexity.

1.2.5 Splay tree

One way to tackle the worst-case of the BST is simply to reduce its relevance by ensuring it is a practical corner-case. This is by no means the case with the standard BST: data is often inserted in an "almost-sorted" order; In a worst-case BST with the smallest value at the root, accessing the largest value still exhibits $O(n)$ time complexity.

The Splay tree tries to make the worst-case less relevant by ensuring it does not happen repeatedly. It does this by, on every operation, restructuring the tree to move the latest-accessed element to the root. This does not change the complexity of the BST operations,[13]and instead makes use of the same principle as caching: locality of reference. The worst-case is still exhibited in other situations: after `search`ing for all elements in order, the tree will be perfectly unbalanced.

The Splay tree has the advantage that it is (relatively) simple: the data structure is the same as the BST, and the algorithms are the same except for the addition of the `splay` method. Its disadvantages are significant though: it does not provide any worst-case improvement, and, unlike other index data structures here, its `search` algorithm mutates the tree.

1.2.6 AVL tree

The AVL tree improves on the BST, finally reducing all operations to $\Theta(log(n))$. It does this by ensuring that the height of the two subtrees of any node differ by at most one.[14] We first define the *height* of a node

[12]The skip list is another, which does this by "skipping over" more than one node at a time. "Balancing" is done by randomizing the height of nodes. I ignore it because it uses randomization, which is ugly.

[13]Unlike the BST, it does perform all operations in *amortized* $O(log(n))$; but its worst-cases are still $O(log(n))$.

[14] [3], p. 296.

n as

$$height(n) = \begin{cases} 0 & \text{if } n \text{ is null,} \\ max(height(n.left), height(n.right)) + 1 & \text{otherwise.} \end{cases}$$

We then define *balance factor* of a node n as

$$bf(n) = \begin{cases} 0 & \text{if } n \text{ is null,} \\ height(n.left) - height(n.right) & \text{otherwise.} \end{cases}$$

We can then define an AVL tree recursively as

$$AVLTree(n) = -1 \leq bf(n) \leq 1 \wedge AVLTree(n.left) \wedge AVLTree(n.right).$$

It is the node's balance factor, rather than the height, that is stored in the node.[15](As, by the definition of the AVL tree, the balance factor only ever has one of three possible values, this can take up only two bits.)

1.2.7 B-tree

The B-tree is primarily motivated by a different problem with the BST: the number of storage reads is $\Omega(log_2(n))$. For media with slow seek times and low granularity (such as your hard disk), this is too high, and wasteful. The B-tree's improvement starts by generalizing the binary tree to a k-ary tree, reducing the number of reads to $\Omega(log_k(n))$ (where $k > 2$). The full solution, however, naturally leads to another method to bound the height of the tree in the order of $log(n)$.

A k-ary search tree holds $k - 1$ elements at each node. But it is infeasible to demand that each node holds this maximum amount—for one thing, we could only store multiples of $k - 1$ elements! This must be relaxed so a node can be less than "full". However, this must not be relaxed so far that a node can hold just one (or zero!) elements, because the $\Omega(log_2(n))$ storage reads then reappears. The B-tree approach demands that each node[16]is between half-full[17]and full. More precisely, we can define a B-tree in terms of an integer t, such that for each node in that tree, its number of elements e is such that $t - 1 \leq e \leq 2t - 1$. [3]

To illustrate why balance naturally falls out of this, consider inserting a value. Once we have searched for its position, we cannot simply create a new node there, as this would not satisfy the restriction on the number of elements. Instead the value must be inserted into an existing leaf node. If space does not exist at the leaf, we create space by splitting the node into two at the same level, and inserting the previous median value in the parent using the same technique. If the splitting does not reach the root, the depth of the tree is unchanged. Otherwise, the root splits, and the depth increases by one for all leaf nodes. We can therefore add the additional constraint to the B-tree definition that all leaves have the same depth, from which we can show that the height of a B-tree is $\Theta(log(n))$.

Rather than bubble-up the node-splitting, most B-tree algorithms actually do this on the downwards traversal to the node at which the element will be inserted. The procedure is simple: every full node visited is split before moving down one level, and the median element inserted into the parent. (We can guarantee that the parent is non-full precisely due to the top-down order of the procedure.) The effect is

[15] [10], the most-documented and -commented AVL tree source I know of.
[17]The root node is an exception, and can be anywhere between empty and full.
[17]The B*-tree demands that nodes are at least two-thirds full.

not equivalent, as the top-down procedure splits all full nodes, where the bubble-up procedure only splits as many as necessary. However, balance is still maintained: as before, the only time that new levels are created is when the root is split.

1.2.8 Red-Black tree

The Red-Black tree inherits ideas from the AVL tree and the B-tree. It has the same goals and performance guarantees as the AVL tree, and uses the same tree mutation to achieve them: rotations. However, it can also be seen as a mapping of a B-tree of degree $t = 2$ onto a BST.[18]This has the advantage over the B-tree that adjusting a node does not require copying, and less-than-full nodes do not hold unused allocated memory.

In such a B-tree, a 3-node[19]is structured as three element-sized spaces adjacent in memory. The Red-Black tree instead uses between one and three 1-nodes. The first 1-node, which I shall call the "representative-node", is representative of the analogous B-tree 3-node. It is the target of links from parents. It is accompanied by zero, one or two more 1-nodes, which I shall call "side-nodes", and which contain the other elements in the 3-node.

The representative-node's `left` and `right` pointers have two roles. Where side-nodes exist, the pointers point to these. (Note this implies that, if the analogous 3-node is full, the representative-node will hold the middle element.) Where a side-node is absent, the corresponding pointer instead points directly to the appropriate child representative-node.[20]A side-node's pointers must point to a child representative-node. (All pointers may instead be to "nil" nodes, at which the tree terminates.)

How are we to know whether a pointer leads us to a side-node or a child representative-node? The approach of Guibas and Sedgewick was to give the pointer a *color*: pointers to representative-nodes are black, and pointers to side-nodes are red. In the literature, this has been transformed into the equivalent rule that representative-nodes are colored black, and side-nodes are colored red. This has been shortened to simply "red nodes" and "black nodes".

We can now state the properties of the Red-Black tree, and relate them to the B-tree:

1. **A node is either red or black.** (A 1-node is either a representative-node or a side-node.)
2. **The root is black,** as it is representative of the root 3-node.
3. **Leaf nodes are black.** This includes "nil" nodes (which in practise are represented by a single sentinel nil node).
4. **Red nodes cannot have red children (the "red rule").** This is because they must point to child representative-nodes (pointing to another side-node would extend the number of elements in a node to more than three).
5. **For any node, all simple paths to a descendant leaf have the same number of black nodes (the "black rule").** This number is the node's *black-height*. This is equivalent to the B-tree rule that all leaves have the same depth.

[18] *i.e.*, a B-tree where the number of elements e is $1 \leq e \leq 3$ (and therefore the number of links l is $2 \leq l \leq 4$).

[19] I use the term n-node to describe a node that can contain up to n elements.

[20] Herein lies the space-saving and copy-reducing aspects of the Red-Black tree, analogous to the benefits of the BST over an array.

From the B-tree analogy, it should already be evident that the Red-Black tree enforces balance. However, we can also see this from the last two Red-Black tree properties. Precisely, we can show that, for any node, the longest path to a descendant leaf is no more than double the length of the shortest path to a descendant leaf. If the node has black-height N, then both the shortest and longest path consist of N black nodes (by property 5), interspersed with red nodes. The shortest path will contain no red nodes (because excising one maintains the Red-Black tree properties and makes the path shorter), and therefore has length N. The longest path will contain one red node between each pair of black nodes (because introducing another violates property 4), and therefore has length $2N - 1$.

Red-Black tree operations feel like B-tree operations. When inserting a new node, we color it red, and insert it as we would in a BST. Then, if its parent is black, we're done (analogous to there being space in a leaf 3-node). Otherwise, the parent is red, and we have violated the red rule. We know the grandparent is black. We check the color of the uncle. If the uncle is red, we do a color swap on the parent, grandparent, and uncle, and move up the tree to look for a possible red violation. This is analogous to a 3-node being full, splitting it, and moving the median up. However, if the uncle is black, we do a single rotation on the grandparent and some recoloring, and stop. This is analogous to there being free space "on the other side" of the 3-node.

1.2.9 Simplifications of the Red-Black tree

The standard Red-Black tree is sometimes seen as overly complex. More precisely, the number of violating cases that the algorithms have to rectify is seen as too large. In response, there are simplifications of the standard Red-Black tree. One is the AA-tree, which has the additional requirement that red nodes must be right children. In the B-tree analogy, they are 2-3 trees. This reduces seven cases to two.

Another is the Left-Leaning Red-Black tree, introduced by Sedgewick. In a similar spirit, it reduces the number of cases, enforcing that no node may have a black left child and a red right child. This means that, in the B-tree analogy, a 3-node only has a single representation in the RB tree. Unlike the AA-tree, this is still analogous to a 2-3-4 tree. The main advantage is code simplicity: `insert()` takes 33 lines vs 150 lines.

I find these simplifications unsatisfying. The gain in code simplicity is results in a loss of conceptual clarity. More importantly, the Red-Black tree algorithms can be drastically shortened by tackling the symmetric cases using the same code, while remaining semantically equivalent. A simple technique, that of Julienne Walker [16], uses a two-element array of child pointers instead of a `left` and a `right` pointer, like so:

```
struct node {
  int red, data;
  struct node *link[2];
};
```

This allows the direction (left or right) to be parameterized, encoded as either a `0` or `1`. The direction can thus be reversed with with the C `!` operator.

1.3 Indexes in the wild

Performance of the many index algorithms is well-covered. However, not all are commonly in use. A survey of what's-used-where should shed light on what characteristics are considered most important.

C++ Standard Template Library The GNU implementation uses Red-Black trees for sets and maps.[21]

Java Class Library `TreeMap` uses a Red-Black tree.[22]

.NET standard library This uses a Red-Black tree for its `SortedDictionary` and `SortedSet` classes, at least in the Mono Project implementation [9].

Linux kernel This uses Red-Black trees for I/O scheduling and for virtual memory. It uses a pseudo-templating style in C, so there's only one implementation: `lib/rbtree.c`.[23]

ext3 filesystem This uses a Red-Black tree for directory entries.

Python The 'CPython' implementation uses a hash.[24]

Ruby This uses a hashing algorithm for its index.

mobile telephones ...

It seems that most standard libraries and low-level codebases prefer the Red-Black tree to the many alternatives.

1.4 Some predicate definitions

Earlier, in our index API specification, we introduced the predicate index(`i`, *I*). This simply says that "`i` represents the index *I*". This was deliberately *abstract*, so that individual index implementations could define it themselves. Now we've looked at a few index data structures, how can we formally describe them?

Our main tool here is *separation logic*, an extension of Hoare logic with practical operators to describe the state of the heap. Here, I only specify the set versions of the data structures.

1.4.1 Linked list

I have chosen to start with the linked list, as it is the simplest non-trivial data structure and is easy to specify. The linked list is a recursive data structure—the structure contains itself. We might define it in C as:

```
struct list {
  int value;
  struct list * tail;
};
```

An instance of `list` is interpreted as representing a non-empty set. The `value` field is one of the members of that set, and all other members are held in the list pointed at by the `tail` field. An empty set is represented by the `null` pointer. We use a pointer to represent an arbitrary set, empty or not. Therefore, for any pointer `s`, we should be able to define a predicate set(`s`, *S*).

[21] [8]. "The insertion and deletion algorithms are based on those in Cormen, Leiserson, and Rivest, *Introduction to Algorithms*". Used to implement `std::set`, `std::multiset`, `std::map`, and `std::multimap`.

[22] [7]: "This class provides a red-black tree implementation of the SortedMap interface. […] The algorithms are adopted from Corman, Leiserson, and Rivest's *Introduction to Algorithms*."

[23] [1]. In the same directory one will find `btree.c`, implementing a B+tree, and `radix-tree.c`, implementing a radix tree.

[24] [2]. In the same directory one will also find `dictnotes.txt`, an in-depth discussion on optimization of the Python dictionary structure.

Translating these statements to separation logic is fairly straightforward. Either `s` is `null`, or it is not. If it is `null`, then $S = \emptyset$ (the empty set), and the heap pertinent to the linked list is empty (represented by emp). If `s` is not `null`, then it points to a valid `list` node: $s \mapsto \texttt{tail}, \textit{value}$. As `tail` is a list pointer, we can recursively apply the predicate to it: $\mathsf{set}(\texttt{tail}, R)$. The node description and the `tail` description are joined with the separating conjunction, because the tail does not contain the node pointed at by `s` (if it did, we would be in trouble!). Finally, `value` and the set R together make up the set S: $R \cup \{\texttt{value}\} = S$. Here's the formal definition:

$$\begin{aligned}\mathsf{set}(\mathsf{s}, S) \triangleq & (\mathsf{s} = \textbf{\textit{nil}} \ \wedge \ S = \emptyset \ \wedge \ \mathrm{emp}) \\ & \vee \ (\exists \texttt{tail}, \texttt{value}.\ s \mapsto \texttt{tail}, \texttt{value} * \mathsf{set}(\texttt{tail}, S \setminus \{\textit{value}\}))\end{aligned}$$

1.4.2 BST

Here's an unordered binary tree representing a set:

Here's the ordered BST:

$$\begin{aligned}\mathsf{set}(s, S) \triangleq & (s = \textbf{\textit{nil}} \ \wedge \ S = \emptyset \ \wedge \ \mathrm{emp}) \\ & \vee \ (\exists l, r, v.\ s \mapsto l, r, v * \mathsf{set}(l, \{v\prime \mid v\prime \in S \ \wedge \ v\prime < v\}) * \mathsf{set}(r, \{v\prime \mid v\prime \in S \ \wedge \ v\prime > v\}) \ \wedge \ v \in S)\end{aligned}$$

1.4.3 Red-Black tree

The defining feature of the Red-Black tree is the restriction that it places on the colors and black-heights of nodes. Before we even look at the tree structure, we can actually encode these restrictions on four variables: the color and black-height of a node, and the color and black-height of its parent. Let's start by defining a predicate on these (as well as making the description more intelligible, this also makes the full description shorter, and makes proofs shorter where they are unconcerned with the Red-Black tree conditions).

$$\begin{aligned}\mathsf{rbCond}(\textit{color}, \textit{pColor}, \textit{height}, \textit{pHeight}) \triangleq \quad & ((\textit{color} = \textbf{red} \ \wedge \ \textit{height} = \textit{pHeight}) \\ & \vee (\textit{color} = \textbf{black} \ \wedge \ \textit{height} = \textit{pHeight} - 1)) \\ \wedge \quad & \textit{color} = \textbf{red} \implies \textit{pColor} = \textbf{black} \\ \wedge \quad & \textit{height} > 0\end{aligned}$$

This says that:

- the color is either *constantred* or **black**
- if the node is **red**, its parent has the same black-height
- if the node is **black**, its parent has a greater black-height by 1
- if the node is **red**, its parent is **black**
- a negative black-height is disallowed

We now want to define the Red-Black tree recursively. We can then describe a node pointer. $\mathsf{tree}(n, S, pHeight, pColor)$ means the pointer n describes the set S, and its parent node has the height $pHeight$ and is **black**. I choose to not use set-builder notation as it makes the proofs easier.

$$
\begin{array}{lll}
\mathsf{tree}(n, S, pHeight, pColor) \triangleq & & \\
\quad (n = \textbf{\textit{nil}} \wedge S = \emptyset \wedge pHeight = 1 \wedge \mathrm{emp}) & & \\
\vee \quad (\exists left, right, value, color, height. & n \mapsto left, right, value, color & \\
& * \quad \mathsf{tree}(left, \{v\prime \mid v\prime \in S \wedge v\prime < value\}, height, color) & \\
& * \quad \mathsf{tree}(right, \{v\prime \mid v\prime \in S \wedge v\prime > value\}, height, color) & \\
& \wedge \quad value \in S & \\
& \wedge \quad \mathsf{rbCond}(color, pColor, height, pHeight) & \\
&) &
\end{array}
$$

Finally, we can define the predicate $\mathsf{set}(s, S)$. An entire Red-Black tree is defined as

$$\mathsf{set}(s, S) \triangleq \exists h.\mathsf{tree}(s, S, h, \textbf{black})$$

1.5 Some simple verifications

```
bool search(n, T val) {
  // rbtree(n, S).
  //  n = nil ∧ S = ∅ ∧ pH = 1 ∧ emp
  // ∨
  //  ∃l, r, v, c, h.
  //   n ↦ l, r, v, c * tree(l, Q, h, c) * tree(r, R, h, c)
  //    ∧ ∀v'.v' ∈ Q ⟹ v' < v ∧ ∀v'.v' ∈ R ⟹ v' > v
  //    ∧ Q ∪ R ∪ {v} = S
  //    ∧ rbCond(c, pC, h, pH).
  if(n == nil) {
    // n = nil ∧ S = ∅ ∧ pH = 1 ∧ emp ∧ val ∉ S.
    e = false;
    // n = nil ∧ S = ∅ ∧ pH = 1 ∧ emp ∧ val ∉ S ∧ e = F.
  }
  else {
    // ∃l, r, v, c, h.
    //  n ↦ l, r, v, c * tree(l, Q, h, c) * tree(r, R, h, c)
    //  ∧ ∀v'.v' ∈ Q ⟹ v' < v ∧ ∀v'.v' ∈ R ⟹ v' > v
    //  ∧ Q ∪ R ∪ {v} = S
    //  ∧ rbCond(c, pC, h, pH)
    rv := node->value;
    // ... ∧ rv = v
    if(val == rv) {
      // ... ∧ rv = v ∧ val = rv ∧ val = v ∧ val ∈ S
      e = true;
      // ... val ∈ S ∧ e = T
```

```
    }
    else {
      // … ∧ rv = v ∧ val ≠ rv ∧ val ≠ v
      if(val < rv) {
        // …
        // ∧ tree(l, {v′ | v′ ∈ S ∧ v′ < v}, h, c)
        // ∧ {v′ | v′ ∈ S ∧ v′ < v} ∪ {v′ | v′ ∈ S ∧ v′ > v} ∪ {v} = S
        // ∧ rv = v ∧ val ≠ v ∧ val < rv ∧ val < v ∧ v ∉ {v′ | v′ ∈ S ∧ v′ > v}
        // ∧ v ∉ {v} ∧ truthvalueof(v ∈ S) = truthvalueof(v ∈ {v′ | v′ ∈ S ∧ v′ < v})
        // ∧ search(r, val) = search(l, val)
        e = search(node->left, val);
        // …
        // ∧ rv = v ∧ val ≠ v ∧ val < rv ∧ val < v
        // ∧ tree(l, {v′ | v′ ∈ S ∧ v′ < v}, h, c)
        // ∧ tree(r, {v′ | v′ ∈ S ∧ v′ > v}, h, c)
        // ∧ v ∉ {v′ | v′ ∈ S ∧ v′ > v}
        // ∧ search(r, val) = search(l, val)
      }
      else {
        e = search(node->right, val);
      }
    }
  }
  // rbtree(node, S ∪ {v})
}
```

2 Concurrent indexes

2.1 What is a concurrent index?

The previous algorithms and data structures are designed for sequential access. Performing concurrent operations on them results in undefined behavior.

One naive implementation of a concurrent index might simply enforce that, at any one time, at most one operation from {`search`, `insert`, `remove`} is running. A slightly better implementation might restrict to either any number of `reads`, or one of {`insert`, `remove`}.

This is still too restrictive: looking at the various indexing algorithms, many of them should, with minimal modification, permit multiple concurrent `insert` operations—the BST, for example. With no knowledge of implementation, what restrictions should we expect from an index? We might make the observation that, conceptually, the keys in an index are independent. The fact that key `k` is in the index or not, or that it has some value in the index, provides no information about key `l` (where `k` ≠ `l`). Therefore, we might expect restrictions to be placed only individual keys, rather than on the entire index. For example, we might have the only restriction that, for any key `k`, at most one operation from {`search(k)`, `insert(k, _)`, `remove(k)`} is running.

2.1.1 A concurrent index API specification

{in(i,k,v)}	`r = i->search(k);`	{in(i,k,v) $\wedge$ r = v}
{out(i,k)}	`r = i->search(k);`	{out(i,k) $\wedge$ r = ***nil***}
{in(i,k,v')}	`i->insert(k, v);`	{in(i,k,v')}
{out(i,k)}	`i->insert(k, v);`	{in(i,k,v)}
{in(i,k,v)}	`i->remove(k);`	{out(i,k)}
{out(i,k)}	`i->remove(k);`	{out(i,k)}

2.2 Concurrent index algorithms

The "array" approach to indexing, with one "slot" for every possible key in the universe of keys, would provide this, but carries its usual disadvantages. The hash table might provide a similar level of concurrency—one `remove` operation per hash value—but again carries its usual disadvantages.

What about the BST? We might expect highly concurrent `search`, as this does not mutate the tree. Our `insert` operations may also be, as they only mutate leaves. The `remove` operation only mutates the removed node and its in-order predecessor, and so might only lock these. It seems the BST offers good support for highly concurrent operations.

Can this support carry through to balanced tree algorithms? The basic reason the BST is promising is that the tree is highly static and operations are highly "local". We can, then, dismiss variations like the splay tree, where every operation performs rotations at every level down to the key under consideration, which must lead to horrendous locking considerations (and to my knowledge, no concurrent splay tree algorithms have been attempted). But at first glance, *all* strictly balanced trees have the same problem: maintaining the invariant that promises balance can require fixing-up the tree all the way along one path from the root to a leaf.

The way most concurrent balanced trees tackle this is to "relax" the balancing invariant: simply, the tree is allowed to be in states that break the guarantee of balance. The balancing logic is moved into a separate operation which can be performed independently of the API operations—*e.g.*, in a separate thread.

2.3 Concurrent Red-Black trees

There's more than one way to make a 'concurrent RB tree'.

2.3.1 Chromatic trees

Introduced by Nurmi and Soisalon-Soininen in 1991/95. Their tree only stores values in the leaves, and the internal nodes are merely 'routers'.[25] Their paper uses the red/black edge interpretation of the RB tree.

Instead of two states red/black, each edge is given a weight (an unsigned integer). Red=0 and black=1. Values < 0 impossible. Values > 1 termed 'overweighted'.

[25] This seems like a really important change! Is there an isomorphism between a standard RB-tree and one that stores everything in the leaves? And how important is this from a performance POV?

The "equal number of black nodes on any path" rule is transformed into: "equal sum of weights on any path". Without overweighting, this is equivalent (due to the values of red/black).

The "no two red nodes" rule is transformed into: "the parent edges of the leaves are not red".

The data structure spec now provides no guarantees of balancedness. The two rules of the sequential RB tree together guarantee that longest path length / shortest path length <= 2. The chromatic tree may have any lengths, in the case that all weights are zero (i.e. all edges are "red") except for those of the leaves (which >0).

Note also that the B-tree isomorphism also disappears. (Or does it? In an arbitrarily-long run of reds, is there an analogous B-tree?)

Insertion/deletion perform no rebalancing, and maintain the CRB properties. However, they may introduce violations of the red rule (consecutive reds).

Rebalancing is done by a separate thread. Where the sequential RB tree balancing is done bottom-up (we insert the new node at a leaf, then rebalance moving towards the root), the chromatic rebalancing algorithm is top-down.

Concurrency control is done by three kinds of lock: r, w, x. For any given node, the following patterns are possible:

R	W	X
0	0	0
1...inf	0	0

Each thread, whether reading, writing or updating, locks nodes as it traverses the tree. Rebalancing is done purely by changing node contents. This is the locking scheme of Ellis for AVL trees [6], with modifications. E.g., Ellis proposes that writers w-lock the entire path, so that global balancing can be done; the decoupling makes w-lock coupling sufficient.

2.3.2 Larsen's tree [13]

This builds on the chromatic tree, with simplifications to the rebalancing. It requires a "problem queue". Hanke finds that Larsen's tree, due to the queue and due to bottom-up conflict handling, does not perform as well as the chromatic tree.

2.3.3 Hyperred-Black trees

Gabarro et al criticize the Chromatic tree: in a run of red nodes, only the top pair may be updated. This is because the rebalancing rules require that the grandparent is black (recall that this is a guarantee with the sequential RB tree). Runs of red nodes may well happen: sorted insertion is common, and all new nodes are colored red. (BUT in the chromatic tree paper above, leaf nodes (parent edges) are black!) The suggested change is that as well as being hyperblack ('overweighted'), nodes may be hyperred ('underweighted'), i.e., weights <= 0 are degrees of red, and weights >= 1 are degrees of black. The 'blackness' definition is maintained. The authors have proofs of correctness. However, they have experimental results that show a less-than-impressive performance vs. the chromatic tree.

3 Roadmap for individual project

Implement sequential and concurrent Red-Black tree One realization arising from this paper is the significant difference between implementations, and that each may have different implications for how

readily analyzable it is. A first step for me, then, is to produce a canonical implementation of the sequential Red-Black tree and of the concurrent Red-Black tree.

Apply separation logic and Concurrent Abstract Predicates to simple algorithms I have notes on applying simple separation logic (beyond the scope of this paper). Concurrent Abstract Predicates has considerable complexities I need to master before applying it to anything more than trivial. In this regard, work in conjunction with Thomas Dinsdale-Young will be crucial.

Apply separation logic to a sequential Red-Black tree I have sketches of how this should be applied in the case of `search` and `insert`. These need to be made more rigorous.

Apply Concurrent Abstract Predicates to concurrent Red-Black tree This is the ultimate aim of this project and constitutes its original contribution.

References

[1] Andrea Arcangeli, David Woodhouse, and The Linux Kernel authors. lib/rbtree.c. `http://git.kernel.org/?p=linux/kernel/git/next/linux-next.git;a=blob_plain;f=lib/rbtree.c;hb=refs/heads/master`, 2002.

[2] The CPython authors. dictobject.c. `http://hg.python.org/cpython/file/fafc84b45a9e/Objects/dictobject.c`.

[3] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2001.

[4] Pedro da Rocha Pinto, Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, and Mark Wheelhouse. Hiding the complexity of concurrent indexes. 2010.

[5] Thomas Dinsdale-Young. *Abstract Data and Local Reasoning*. PhD thesis, Imperial College London, 2010.

[6] C.S. Ellis. Concurrent search in avl-trees. 1980.

[7] Inc. Free Software Foundation. Treemap.java. `http://cvs.savannah.gnu.org/viewvc/classpath/java/util/TreeMap.java?root=classpath&view=markup`.

[8] Inc. Free Software Foundation. stl_tree.h. `http://gcc.gnu.org/onlinedocs/libstdc++/latest-doxygen/a01066_source.html`, 2010.

[9] Raja R Harinath and The Mono Project authors. Rbtree.cs. `https://github.com/mono/mono/blob/master/mcs/class/System/System.Collections.Generic/RBTree.cs`.

[10] Adrian Hey. Haskell data.tree.avl package. `http://hackage.haskell.org/packages/archive/AvlTree/4.2/doc/html/Data-Tree-AVL.html`, 2005.

[11] C. A. R. Hoare. An axiomatic basis for computer programming. 12:576–583, October 1969.

[12] ECMA International. Ecmascript language specification. "`http://www.ecma-international.org/publications/standards/Ecma-262.htm`", December 2009.

[13] K. Larsen. Amortized constant relaxed rebalancing using standard rotations. 1998.

[14] John C. Reynolds. *An Introduction to Separation Logic (Preliminary draft)*. Carnegie Mellon University, October 2008.

[15] Y. Sagiv. Concurrent operations on b*-trees with overtaking. 33:275–296, October 1986.

[16] Julienne S. Walker. Red black trees (tutorial). `http://eternallyconfuzzled.com/tuts/datastructures/jsw_tut_rbtree.aspx`.

[17] Wikipedia. Comparison of programming languages (mapping). `http://en.wikipedia.org/wiki/Comparison_of_programming_languages_(mapping)`, April 2011.