Efficient Ordered Sets and Maps in a Functional Setting

Esben Bjerre (201709479)
June 2020

Bachelor Report (15 ECTS) in Computer Science
Department of Computer Science, Aarhus University

Advisors:
Magnus Madsen
Anders Møller

Abstract

An advantage of functional programming is that it allows for easy modularization: we can compose simpler types and functions to build more complex ones. Applying pure functions to immutable values provides referential transparency, which makes programs easier to write, use and reason about. In a functonal setting, data structures are values and inherits the benefits of referential transparency. In this report, we focus on the implementation of efficient immutable ordered sets and maps. Sets and maps are foundational data structures with intrinsic value. We show how immutable sets and maps can be defined using a red-black tree. We implement the red-black tree, set and map data structures as well as a collection of composable higher-order functions in the Flix programming language and evaluate the performance.

Acknowledgements

I wish to recognize the invaluable support of my advisor, Magnus Madsen. Without his guidance and encouragement, this report would not have been realized. Magnus has been a wise and patient advisor and I hope to learn more from him in the future.

Thanks to Anders Møller for his course on programming languages, which sparked my interest in functional programming.

Thanks to Sif for her tolerance and patience during the past three years. You always remind me, that there is more to life than university. I look forward to picking the first strawberries in our garden.

Lastly, thanks to my family for their support throughout the years.

Contents

1

Introduction

Introductory programming courses usually focus on the imperative programming paradigm. While imperative programming is undeniable closer to how the hardware works (CPU registers are mutable, after all), mutability can make it hard to reason about the behaviour of programs. In imperative programming, changes in the internal state of a program may modify the behaviour of functions, such that `f(x) = y` now, but it may not be later. This is not true for functional programming. Pure functions applied to the same values yield the same result each time they are invoked. This property, known as referential transparency, makes it possible to conduct equational reasoning on the code. For instance, if `f(x) = y` and `g(y, y) = h` then we can be sure that `g(f(x), f(x)) = h`. In functional programming, values are immutable. Instead of modifying existing values, modified copies are created. The same philosophy and benefits applies to immutable data structures.

In this report, we present an implementation of immutable ordered sets and maps in the Flix programming language[1]. Sets and maps are foundational data structures which are often used as part of modelling other data types and algorithms. The Flix standard library missed efficient sets and maps, which have been added as a result of this report. The set and map implementations are rich data structures, supporting a combined total of 93 distinct functions.

The sets and maps are implemented using immutable red-black trees (Okasaki, 1998; Germane and Might, 2014). Red-black trees are a well-known type of balanced binary search tree, which support search, insert and delete in $O(\log n)$ time where n is the size of the tree (Cormen et al., 2009).

The implementations have been benchmarked, showing that the performance of the red-black tree in practice, is in the order of $O(\log^2 n)$ where n is the size of tree. The benchmarks also shows that the sorting performance of the red-black tree is within a factor $1.5x$ of a sorting algorithm for immutable lists found in the Flix standard library. Lastly, the benchmarks shows that the search performance of the immutable map is comparable to Scala's `TreeMap`.

The main contributions of this report are the immutable red-black tree, set and map implementations spanning more than 1300 lines of code[2].

[1] https://flix.dev

[2] https://github.com/flix/flix/tree/master/main/src/library

1.1 Trees vs. Hashing: A Question of Order

In a functional context, sets and maps are typically implemented using either balanced binary search trees or hash tries. A balanced binary search tree supports search, insert and delete in $O(\log n)$ time, where n is the size of the tree (Cormen et al., 2009). The $O(\log n)$ time complexity guarantees that the time used to search for, insert or delete an element in the tree is bounded above by the logarithm of the size of the tree.

A hash trie supports search, insert and delete in $O(1)$ amortized time (Bagwell, 2001). However, hashing destroys the ordering of the keys. In contrast, binary search trees maintains their elements in sorted order. Given the lack of order in hash tries and the simplicity of binary trees, we opt for the latter.

1.2 Source Language

All source code will be presented in Flix. However, the implementation can easily be translated into any other functional language. Flix is a functional-first programming language with support for algebraic data types, pattern matching, parametric polymorphism, higher-order functions and tail call elimination (Madsen, Yee, and Lhoták, 2016; Madsen, Zarifi, and Lhoták, 2018). Flix compiles to JVM bytecode and runs on the Java Virtual Machine. A distinct feature of Flix is that the type and effect system supports effect polymorphism[3]. In short, the type and effect system allows us to seperate pure and impure code, restrict the purity of function arguments and specify that the effect of a higher-order function depends on the effects of its function arguments.

1.3 Goal

The goal of this report is twofold:

(i) Implement efficient immutable ordered sets and maps in Flix.

(ii) Benchmark and evaluate the performance of the implementations.

[3] https://flix.dev//blog/taming-impurity-with-polymorphic-effects

2

Implementation

This chapter is about the implementation of the immutable red-black tree, set and map. It starts with the implementation of the underlying data structure, namely the red-black tree (2.1). Then it moves on to the definition of a select subset of generic higher-order functions for the red-black tree. Lastly, it shows how the set (2.2) and map (2.3) are implemented using the red-black tree.

2.1 Red-Black Tree

A red-black tree is a binary tree which satisfies the following invariants:

(i) For all nodes with key x, x is greather than all keys in its left subtree and smaller than all keys in its right subtree.

(ii) All nodes are red or black.

(iii) All leaves are black.

(iv) All red nodes has black children.

(v) Every path from the root to a leaf contains the same number of black nodes.

Together, these invariants ensures that the longest path in the tree is no more than twice as long as the shortest (Cormen et al., 2009).

We will define red-black trees as an algebraic data type. In Flix, the `enum` keyword introduces the definition of a data type. We define the color of red-black tree node as

```
enum Color {
  case Red
  case Black
  case DoubleBlack
}
```

The third color is a special "double-black" color introduced by Germane and Might (2014) which simplifies the deletion algorithm. We subsequently define a red-black tree as

```
enum RedBlackTree[k, v] {
  case Leaf
  case DoubleBlackLeaf
  case Node(Color, RedBlackTree[k, v], k, v, RedBlackTree[k, v])
}
```

Note that `RedBlackTree` is a recursive data type, since it refers to itself. In addition, `RedBlackTree` is polymorphic, with keys of type `k` and values of type `v`. A `Leaf` has no key or value. A `Node` has a color, left subtree, key, value and right subtree. We use `k -> v` as a shorthand for a node with key `k` and value `v`.

2.1.1 Search

Binary search trees supports search in $O(\log n)$ time, where n is the size of the tree (Cormen et al., 2009). The ordering of the keys means we can use binary search and skip about half of the remaining tree at each step. We will implement two search functions. The first is `memberOf` which solely checks if a given key exists in the tree. The second is `get` which checks if a given key exists in the tree and returns its associated value. In order to avoid duplicate code, we implement the binary search in the `get` function only.

The `get` function takes a key `k` and returns `Some(v)` if the tree contains `k -> v`. Otherwise it returns `None`. We define `get` as

```
def get(k: k, tree: RedBlackTree[k, v]): Option[v] = match tree {
  case Node(_, a, k1, v, b) =>
    let cmp = k <=> k1;
    if (cmp < 0)
      get(k, a)
    else if (cmp == 0)
      Some(v)
    else
      get(k, b)
  case _ => None
}
```

The `<=>` syntax denotes a three-way comparison. We can subsequently define `memberOf` as

```
def memberOf(k: k, tree: RedBlackTree[k, v]): Bool = get(k, tree) != None
```

2.1.2 Insert and Delete

The insert and delete functions are adapted from Okasaki (1998) and Germane and Might (2014). We will not discuss the implementation details here. However, the general idea is the same as the imperative version: modifications such as insert and delete may break the invariants of the tree, so these must be restored after each modification.

The `insert` function takes a key `k` and a value `v` and returns a new tree where `k -> v` has been added to the given tree.

```
def insert(k: k, v: v, tree: RedBlackTree[k, v]): RedBlackTree[k, v]
```

The delete function takes a key `k` and returns a new tree where `k -> v` has been removed from the given tree.

```
def delete(k: k, tree: RedBlackTree[k, v]): RedBlackTree[k, v]
```

2.1.3 Fold and Reduce

The `fold` function encapsulates a pattern of recursion for processing a data structure. `fold` is very generic and many other functions can be expressed in terms of it. After defining `fold` we will demonstrate its power by using it to define the `reduce` function.

`fold` is a higher-order function that takes a function `f` and a seed value `s`. The function `f` itself takes an accumulator, a key and a value and returns an accumulator. `fold` recursively applies `f` to the accumulator and each key-value pair in the given tree, starting with the seed value `s`. `fold` returns the accumulated value if the tree is non-empty. Otherwise it returns `s`. We define `foldLeft` as

```
def foldLeft(f: (b, k, v) -> b & e, s: b, tree: RedBlackTree[k, v]): b & e =
  match tree {
    case Node(_, a, k, v, b) => foldLeft(f, f(foldLeft(f, s, a), k, v), b)
    case _ => s
  }
```

Note that the definition contains two recursive calls, one for each subtree. The first call recurses on the left subtree, then we apply `f` and lastly we recurse on the right subtree. `foldLeft` traverses the tree in ascending sorted order of the keys. The definition is not tail-recursive, but this is not a problem since the number of recursive calls are bounded by the height of the tree (i.e., $O(\log n)$ where n is the size of the tree). The `& e` syntax denotes that the effect of `fold` depends on the effect of the function `f`. The `foldRight` function is symmetric and traverses the tree in descending sorted order of the keys.

`reduce` is a higher-order function that takes a function `f`. The function `f` itself takes two key-value pairs and returns a single key-valur pair. `reduce` applies `f` to all key-value pairs in the given tree until a single key-value pair is obtained. `reduce` returns `Some(k, v)` if the tree is non-empty. Otherwise it returns `None`. We define `reduceLeft` as

```
def reduceLeft(f: (k, v, k, v) -> (k, v) & e, tree: RedBlackTree[k, v]):
Option[(k, v)] & e =
  foldLeft((x: Option[(k, v)], k1: k, v1: v) -> match x {
      case Some((k2, v2)) => Some(f(k2, v2, k1, v1))
      case None => Some(k1, v1)
  }, None, tree)
```

Note that the seed value given to `fold` is `None`. We demonstrate the `reduce` function with an example. Consider a tree `t` containing `0 -> 1`, `2 -> 3` and `4 -> 5`. We can then compute the sum of all keys and values in `t` with

```
RedBlackTree.reduceLeft((k1, v1, k2, v2) -> (k1 + k2, v1 + v2), t)
```

which returns `Some((6, 9))`.

2.1.4 Find

The `find` function finds (as the name implies) the first key-value pair in the given tree which satisfies a predicate. Given that we would like `find` to terminate as soon as a key-value pair satisfying the predicate is found, it cannot readily be expressed efficiently in terms of `fold`, as `fold` is unable to break termination early.

`find` is a higher-order function that takes a function `f`. The function `f` itself is a predicate that takes a key-value pair and returns a boolean. `find` returns `Some(k, v)` if the tree is non-empty. Otherwise it returns `None`. We define `findLeft` as

```
def findLeft(f: (k, v) -> Bool, tree: RedBlackTree[k, v]): Option[(k, v)] =
  match tree {
    case Node(_, a, k, v, b) => match findLeft(f, a) {
        case None => if (f(k, v)) Some((k, v)) else findLeft(f, b)
        case Some((k1, v1)) => Some((k1, v1))
    }
    case _ => None
  }
```

We recurse on the left subtree until we reach a leaf, then we check whether `f(k, v)` holds, returning `Some((k, v))` if it is. Otherwise we recurse on the right subtree. Note that `findLeft` traverses the tree in ascending sorted order of the keys and returns the smallest element satisfying `f`. The `findRight` function is symmetric and traverses the tree in descending sorted order of the keys. The predicate `f` is required to be pure.

2.1.5 Forall and Exists

The `forall` and `exists` functions are akin to the $\forall$ and $\exists$ quantifiers in first-order logic.

`forall` is a higher-order function that takes a function `f`. The function `f` itself is a predicate that takes a key-value pair and returns a boolean. `forall` returns `true` if and only if all the key-value pairs in the tree satisfies `f`. We define `forall` as

```
def forall(f: (k, v) -> Bool, tree: RedBlackTree[k, v]): Bool = match tree {
  case Node(_, a, k, v, b) => f(k, v) && forall(f, a) && forall(f, b)
  case _ => true
}
```

Note the use of short-circuit evaluation with `&&` which ensures that `forall` terminates as early as possible, in case some key-value pair does not satisfy `f`. The predicate `f` is required to be pure.

`exists` is a higher-order function that takes a function `f`. The function `f` itself is a predicate that takes a key-value pair and returns a boolean. `exists` returns `true` if and only if at least one key-value pair in the tree satisfies `f`. We define `exists` as

```
def exists(f: (k, v) -> Bool, tree: RedBlackTree[k, v]): Bool = match tree {
  case Node(_, a, k, v, b) => f(k, v) || exists(f, a) || exists(f, b)
```

```
    case _ => false
}
```

Note the use of short-circuit evaluation with `||` which ensures that `exists` terminates as early as possible, in case some key-value pair satisfies `f`. The predicate `f` is required to be pure.

2.1.6 Foreach

The `foreach` function is special: it is the only implemented function which is inherently impure.

`foreach` is a higher-order functions that takes a function `f`. The function `f` itself takes a key-value pair and returns the unit value `()` with some side effect. We define `foreach` as

```
def foreach(f: (k, v) ~> Unit, tree: RedBlackTree[k, v]): Unit & Impure =
  match tree {
    case Node(_, a, k, v, b) => foreach(f, a); f(k, v); foreach(f, b)
    case _ => ()
  }
```

Note the `~>` syntax, denoting that `f` is required to be impure. We demonstrate the `foreach` function with an example. Consider again the tree `t` containing `0 -> 1`, `2 -> 3` and `4 -> 5`. We can then print out all values in `t` to the console with

```
RedBlackTree.foreach((_, v) -> Console.printLine(Int32.toString(v)), t)
```

which returns `()` and prints out

```
1
3
5
```

to the console.

2.2 Set

We will now utilize the red-black tree to implement immutable sets. We will define sets as an algebraic data type consisting of a red-black tree, where the keys in the tree corresponds to the elements in the set. We will use the unit value `()` as a dummy value for all values in the tree. We define the set type as

```
enum Set[t] {
  case Set(RedBlackTree[t, Unit])
}
```

Note that `Set` is polymorphic with elements of type `t`, which is also the type of the keys in the underlying red-black tree.

We can define the `memberOf`, `insert` and `delete` functions as facades which calls the corresponding function on the underlying red-black tree.

```
def memberOf(x: a, xs: Set[a]): Bool =
  let Set(es) = xs;
  RedBlackTree.memberOf(x, es)

def insert(x: a, xs: Set[a]): Set[a] =
  let Set(es) = xs;
  Set(RedBlackTree.insert(x, (), es))

def delete(x: a, xs: Set[a]): Set[a] =
  let Set(es) = xs;
  Set(RedBlackTree.delete(x, es))
```

The `let Set(es) = xs` syntax extracts the underlying red-black tree and assigns it to the variable `es`.

A total of 37 functions were implemented for sets. However, many of them are defined as the facades above and will not be discussed. Instead, we once again demonstrate the power of `fold` by using it to implement the three functions `union`, `partition` and `subsets`. The signature of `Set.foldLeft` is

```
def foldLeft(f: (b, a) -> b & e, s: b, xs: Set[a]): b & e
```

`union` takes two sets `xs` and `ys` and returns a set containing the union of the elements in `xs` and `ys`. We define `union` as

```
def union(xs: Set[a], ys: Set[a]): Set[a] =
  foldLeft((acc, x) -> insert(x, acc), xs, ys)
```

`partition` takes a function `f`. The function `f` itself is a predicate that takes an element an returns a boolean. `partition` returns a pair of sets `(ys, zs)` where `ys` contains all the elements in the given set that satisfy `f` and `zs` contains all the elements that does not satisfy `f`. We define `partition` as

```
def partition(f: a -> Bool, xs: Set[a]): (Set[a], Set[a]) =
  foldLeft((acc, x) ->
    let (a, b) = acc;
    if (f(x))
      (insert(x, a), b)
    else
      (a, insert(x, b)),
  (empty(), empty()), xs)
```

`subsets` returns a set of all the subsets in the given set. We define `subsets` as

```
def subsets(xs: Set[a]): Set[Set[a]] =
  foldLeft((acc, x) ->
    union(map(y -> insert(x, y), acc), acc),
  insert(empty(), empty()), xs)
```

The `map` function used above is also defined in terms of `fold` and has the signature

```
def map(f: a -> b & e, xs: Set[a]): Set[b] & e
```

We demonstrate the `subsets` function with an example. Consider a set `s` containg `1`, `2` and `3`. We can then compute all 2^3 subsets of `s` as

```
Set.subsets(s)
```

which returns

```
Set(Set(), Set(1), Set(2), Set(1, 2), Set(3), Set(1, 3), Set(2, 3), Set(1, 2,
3))
```

We conclude this section with a dependency graph which shows the relations between the set functions and the red-black tree. A function with `*` indicates that it exists in multiple versions (e.g., `fold*` covers `foldLeft` and `foldRight`). Note that not all set functions are shown here for brevity. For a complete list we refer the reader to the Flix Standard Library[1]. Many functions depends on more than one other function, but for clarity we only show the primary dependency here.

[1] https://flix.dev/api/#/Set

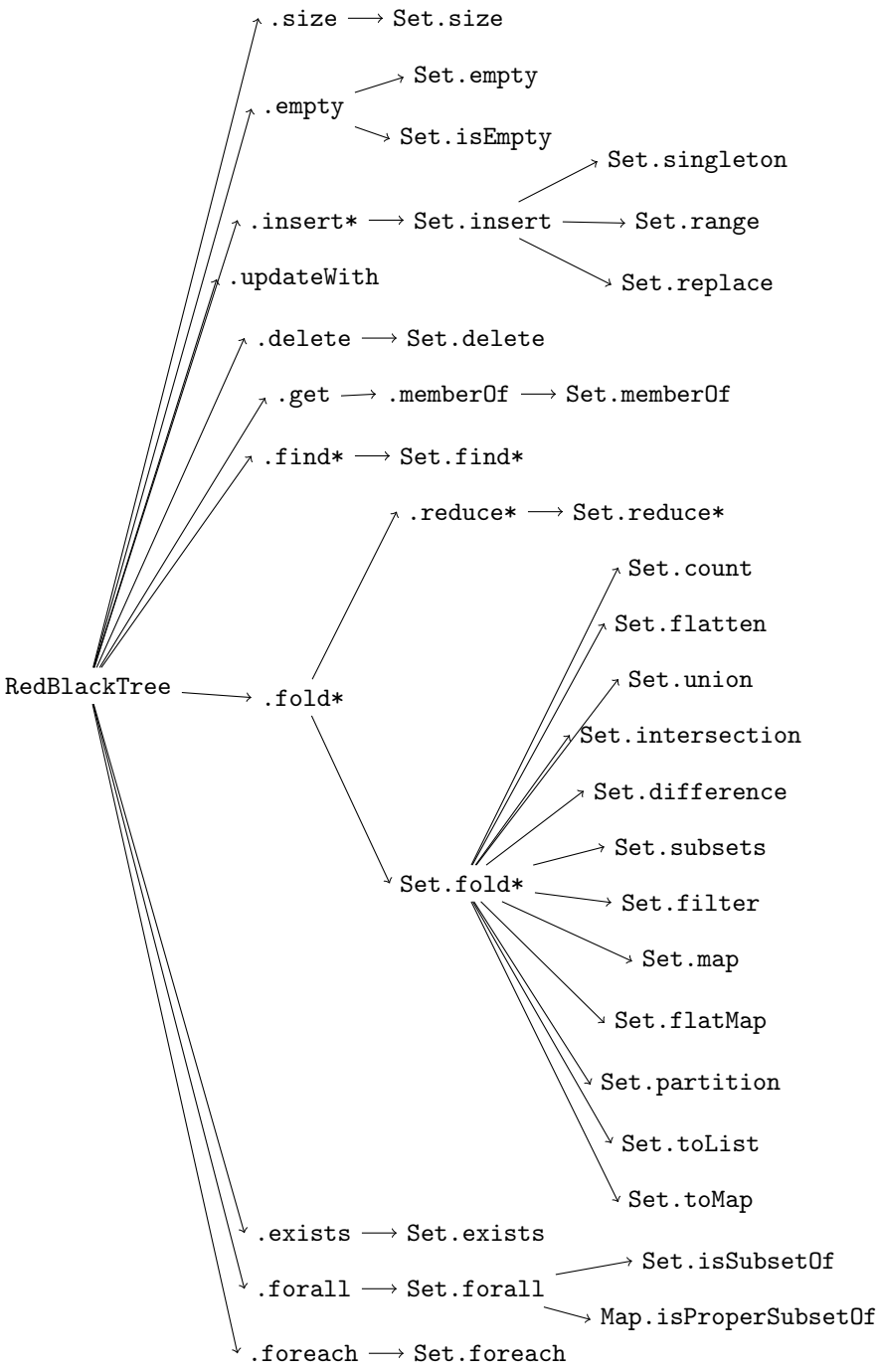


Figure 2.1: Dependency graph of `RedBlackTree` and `Set`

2.3 Map

We now define immutable maps in a manner similar as we did with sets. We will define maps as an algebraic data type consisting of a red-black tree, where the keys and values in the tree corresponds to the keys and values in the map. We define the map type as

```
enum Map[k, v] {
  case Map(RedBlackTree[k, v])
}
```

Note that `Map` is polymorphic with keys of type `k` and values of type `v`.

A total of 56 functions were implemented for maps. The `memberOf`, `insert` and `delete` functions are identical to the facades we implemented in `Set`, except the `Map.insert` function takes and inserts both a key and value. Due to their dull nature, we will not further discuss the facade definitions for maps.

We conclude this section with a dependency graph which shows the relations between the map functions and the red-black tree. A function with $*$ indicates that it exists in multiple versions. Note that not all map functions are shown here for brevity. For a complete list we refer the reader to the Flix Standard Library[2]. Many functions depends on more than one other function, but for clarity we only show the primary dependency here.

[2] https://flix.dev/api/#/Map

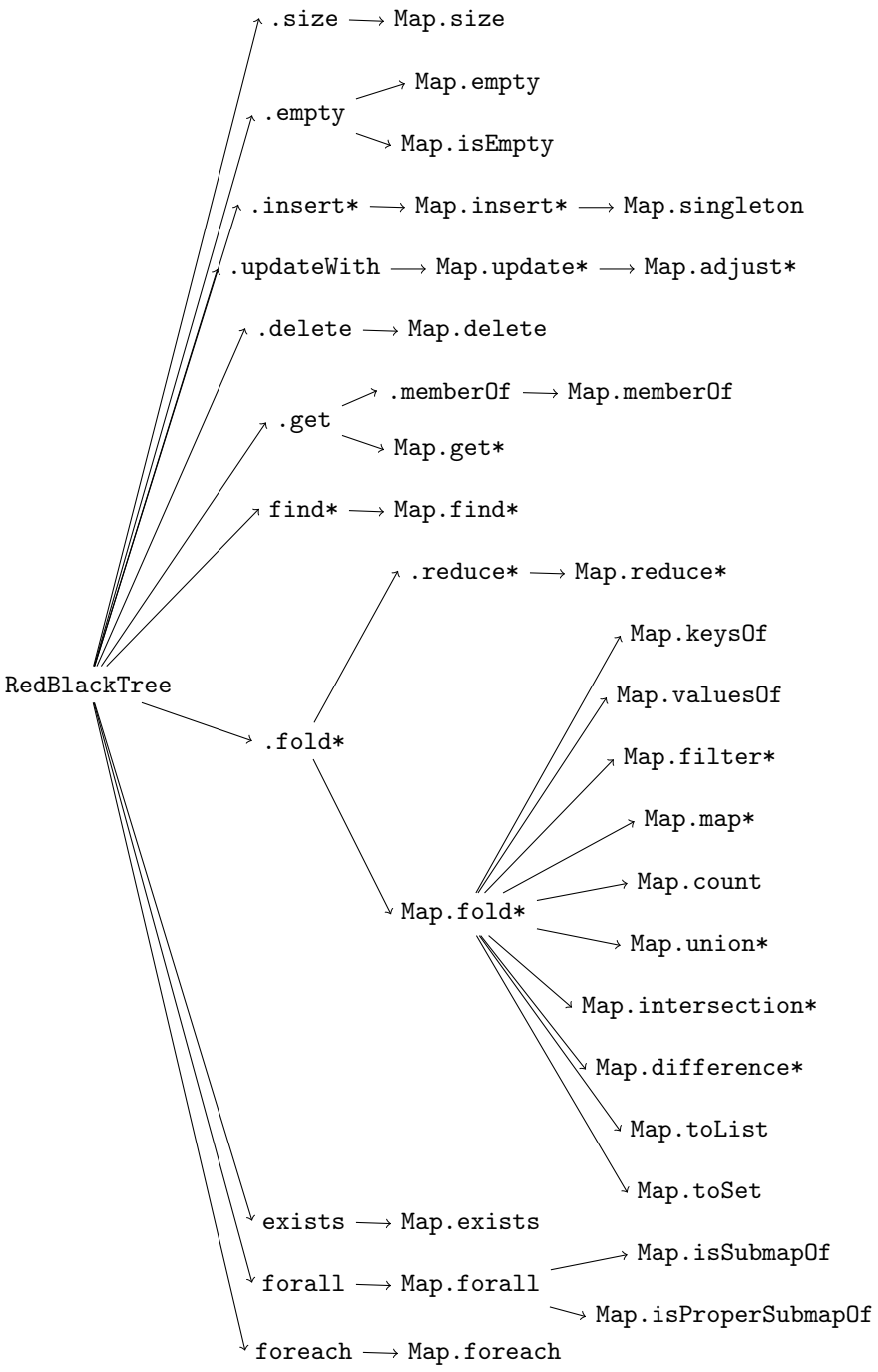


Figure 2.2: Dependency graph of `RedBlackTree` and `Map`

3

Experimental Evaluation

This chapter is about benchmarking and evaluation of the implementations. It starts by diving into the runtime characteristics of the red-black tree. Then it benchmarks and evaluates two possible optimizations. Lastly, it compares the red-black tree to two sorting algorithms and the map to a corresponding implementation from the Scala standard library. All benchmarks, except those in section 3.6, will be measured on the red-black tree. The primary type of data visualization will be scatter plots[1].

3.1 Setup

We used a machine with macOS 10.15.4 and 8 GB RAM. It has a 2.0 GHz Intel Core i7-4750HQ CPU with 32/256/6000 KB L1/L2/L3 cache and 64-byte cache lines. We used OpenJDK 64-Bit Server VM (14+36-1461) with default JVM options and a fixed heap size of 4 GB. We used Flix version 0.12.0 and Scala version 2.13.2.

Runtime performance tends to be noisy. We acknowledge two major sources of noise on the JVM: the Just-In-Time (JIT) compiler and the garbage collector. The JIT compiler may optimize the code and improve the running time. The garbage collector may temporarily slow down the running time while freeing up the memory.

To mitigate noise, all reported numbers were computed as the median over a large number of iterations. Furthermore, all benchmarks were run one at a time in seperate JVM instances. Given these considerations, we believe that the collected data is sufficiently reliable.

3.2 Performance of Search, Insert and Delete

The theoretical running time of `memberOf`, `insert` and `delete` is $O(\log n)$, where n is the size of tree. We expect the performance of our implementation to equal the theoretical running time. In other words, we expect a logarithmic increase in the running time as the tree size increases.

The three functions vary in complexity (i.e., lines of code and number of memory allocations). The implementation of `memberOf` is the least complex. In addition to consisting of fewer

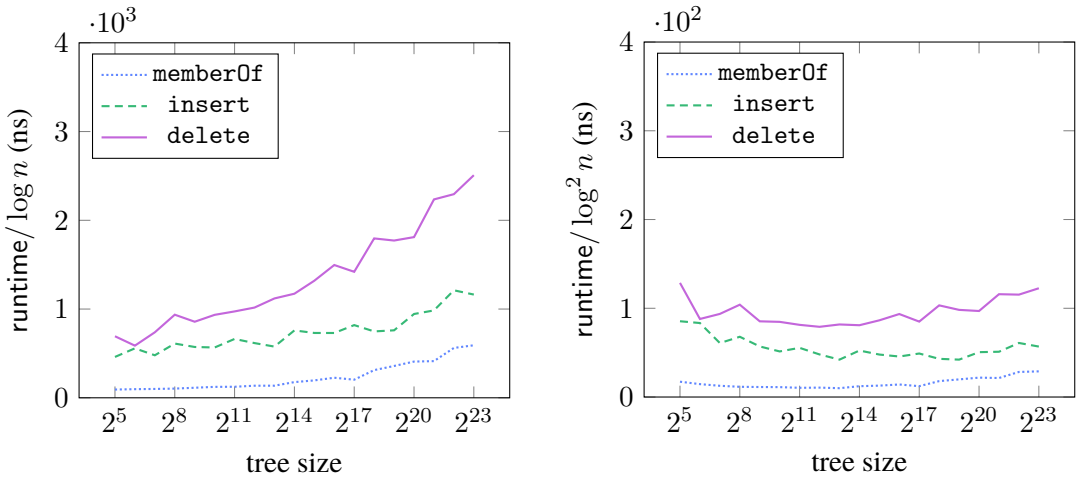
[1] https://blog.sigplan.org/2019/11/14/a-plot-twist-part-ii-in-praise-of-scatter-plots

lines of code, it makes exactly one memory allocation per call. In contrast, `insert` and `delete` makes several memory allocations due to their need of rebalancing. As a result, we expect `memberOf` to perform faster than `insert` and `delete`. In summary, the hypotheses are:

Hypothesis 1 `memberOf`, `insert` and `delete` runs in $O(\log n)$ time where n is the size of tree.

Hypothesis 2 `memberOf` is faster than `insert` and `delete` by a constant factor.

To test our hypothesis, we measure the execution time of each function on trees of sizes 2^n for $n \in [5, 23]$. The keys in the tree are 32-bit integers from a random number generator. All values in the tree is the unit value `()`. For `memberOf` and `delete` we randomly pick an element from the tree. For `insert` we pick a random element not yet present in the tree.



(a) runtime/ $\log n$ (b) runtime/ $\log^2 n$

Figure 3.1: Plot of the median runtime of `memberOf`, `insert` and `delete`

Figure 3.1a shows the median runtime of a `memberOf`, `insert` and `delete` function call divided by $\log n$ where n is the size of the tree. The x-axis shows the size of the tree which ranges from $2^5 = 32$ to $2^{23} = 8,388,608$. The y-axis shows the median runtime of a *single* `memberOf` , `insert` or `delete` function call, measured in nanoseconds. The median was computed over 300,000 iterations. We notice two distinct spikes in the running time of `delete` which will be discussed in the following section.

For hypothesis 1, we expect a constant function (i.e., a curve with slope zero) for each of the three functions, since we have divided the actual running time by the expected theoretical running time. Figure 3.1a shows that this is not the case. The running time per single function call seems to increase by more than $\log n$. For `memberOf`, the difference in running time can be observed to be more than a factor $6.3x$. For `insert`, the difference in running time can be observed to be more than a factor $2.5x$. For `delete`, the difference in running time can be observed to be more than a factor $3.6x$. We conclude that hypothesis 1 must be rejected.

For hypothesis 2, we expect that `memberOf` is faster than `insert` and `delete` by a constant factor. Figure 3.1a shows that `memberOf` is faster than `insert` and `delete` as expected. We conclude that hypothesis 2 is accepted.

To investigate why the performance of `memberOf`, `insert` and `delete` is worse than the expected $\log n$, we plot the actual running time of each function divided by an expected running time of $\log^2 n$ where n is the size of the tree. Figure 3.1b shows this plot. We note that the curve has been flattened significantly. The running time of each function seems to be in the order of $O(\log^2 n)$ (i.e., polylogarithmic) where n is the size of the tree. For `memberOf`, the difference in running time can be observed to be less than a factor $1.7x$. For `insert`, the difference in running time can be observed to be less than a factor $1.5x$. For `delete`, the difference in running time can be observed to be less than a factor $1.5x$. We conclude that the observed running time of `memberOf`, `insert` and `delete` is $O(\log^2 n)$.

3.2.1 Case Study: Performance Implications of Caching

In this case study we take a little detour, to investigate the spikes in the running time of `delete` shown in figure 3.1a. We observe two distinct spikes. The first spike seems to occur when the tree reaches a size of 2^9 to 2^{10}. The second spike seems to occur when the tree reaches a size of 2^{16} to 2^{17}. We hypothesize that the spikes in the running time might be related to caching. We know that the used machine has three cache layers: L1, L2 and L3. The L2 cache is larger but slower than the L1 cache and the L3 cache is larger but slower than the L2 cache. We also know that objects on the JVM are kept in memory. We hypothesize that the spikes occur, when the tree no longer fits in a single layer of cache.

We will determine how the red-black tree is represented on the JVM and use that to estimate the memory size of the red-black tree at runtime. We are not familiar with the internals of the Flix compiler. However, we know that Flix compiles to bytecode, which we can disassemble and examine using the `javap` command[2].

Flix runs as a `.jar` file. During runtime, it creates a `target` folder which contains compiled bytecode in `.class` files. We notice a folder named `RedBlackTree` which seems to contain the relevant bytecode

```
$ ls RedBlackTree
Def$blacken$38874.class                IRedBlackTree$Int32$Obj.class
Def$empty$38768.class                  Leaf$Int32$Obj.class
Def$insert$38669.class                 Node$Int32$Obj.class
...
```

We disassemble the `Node$Int32$Obj.class` file

```
$ javap -c -p RedBlackTree/Node\$Int32\$Obj.class
Compiled from "RedBlackTree/Node$Int32$Obj"
public final class RedBlackTree.Node$Int32$Obj implements RedBlackTree.
IRedBlackTree$Int32$Obj {
  private java.lang.Object value;
```

[2]https://docs.oracle.com/en/java/javase/14/docs/specs/man/javap.html

```
  public RedBlackTree.Node$Int32$Obj(java.lang.Object);
    Code:
       0: aload_0
...
```

We observe that the disassembled `Node` class has a single field. This is inconsistent with our definition of `Node` which has 5 fields: a color, left subtree, key, value and right subtree. We go up a level and notice a file named `Tuple5$Obj$Obj$Int32$Obj$Obj.class`. Disassembling this file reveals a structure like the one we expect

```
$ javap -c -p Tuple5\$Obj\$Obj\$Int32\$Obj\$Obj.class
Compiled from "Tuple5$Obj$Obj$Int32$Obj$Obj"
public final class Tuple5$Obj$Obj$Int32$Obj$Obj implements
ITuple5$Obj$Obj$Int32$Obj$Obj {
  private java.lang.Object field0;
  private java.lang.Object field1;
  private int field2;
  private java.lang.Object field3;
  private java.lang.Object field4;
...
```

Based on the disassembled bytecode so far, we conclude that Flix compiles a red-black tree node to a class containg a single field of type `Tuple5` and that the `Tuple5` class contains the 5 fields of a tree node. We continue the analysis and learn that each red-black tree color and leafs are compiled to static classes, which means that only a single instance of each class should exist at runtime.

We will estimate the size of an object as the object header plus the size of its fields. The object header is 16 bytes (12 bytes padded to a multiple of 8). A field of type `Object` is a reference of 4 bytes. (Lindholm et al., 2020). We estimate the size of a *single* tree node as

Type	Bytes	Amount
Object	16	2
Reference	4	5
int	4	1
Total	56	

The first spike occurs when the tree reaches a size of 2^9 to 2^{10}. Using the estimated size of 56 bytes for a node, this gives us $28,672$ to $57,344$ bytes. The second spike occurs when the tree reaches a size of 2^{16} to 2^{17}, using the estimated size of 56 bytes for a node, this gives us $3,670,016$ to $7.340.032$ bytes. The size of the L1 cache is $32,000$ bytes which appears to fit with the first spike. In other words, we observe a slowdown in the performance when the L1 cache is filled. The size of the L3 cache is $6,000,000$ bytes which appears to fit with the second spike. In other words, we observe a slowdown in the performance when the L3 cache is filled. Despite the fact that the numbers are estimates, the results appears to be conclusive. We are unable to observe a slowdown in the performance when the L2 cache is filled. Using the same logic as above, we would expect a slowdown in the performance when the memory usage reaches

$256,000$ bytes, which approximately corresponds to a tree of size 2^{12}. We hypothesize that there might be a smaller difference between the speed of the L2 and L3 cache on the used machine. We conclude that caching has a noticeable impact on the performance of the tree.

3.3 Performance Implications of Adding a Size Field

Recall `Set.union` defined as

```
def union(xs: Set[a], ys: Set[a]): Set[a] =
  foldLeft((acc, x) -> insert(x, acc), xs, ys)
```

While this definition is short short and concise, it is slow when `ys` contains more elements than `xs`. We could optimize it by considering the size of each set, switching the order of the arguments accordingly

```
def union(xs: Set[a], ys: Set[a]): Set[a] =
  if size(xs) > size(ys)
    foldLeft((acc, x) -> insert(x, acc), xs, ys)
  else
    union(ys, xs)
```

However, this definition adds two calls to `Set.size`, each taking $O(n)$ time, which cancels out any performance gains. We can make `Set.size` run in $O(1)$ time by adding it as a field to `Node`. Augmenting each node with a size field of type `Int32` adds an overhead of 4 bytes. We expect that the associated performance cost is neglible (i.e., at most a constant factor). specifically, we expect that the performance cost on `memberOf`, `insert` and `delete` is less than a factor $1.05x$. In summary, the hypotheses are:

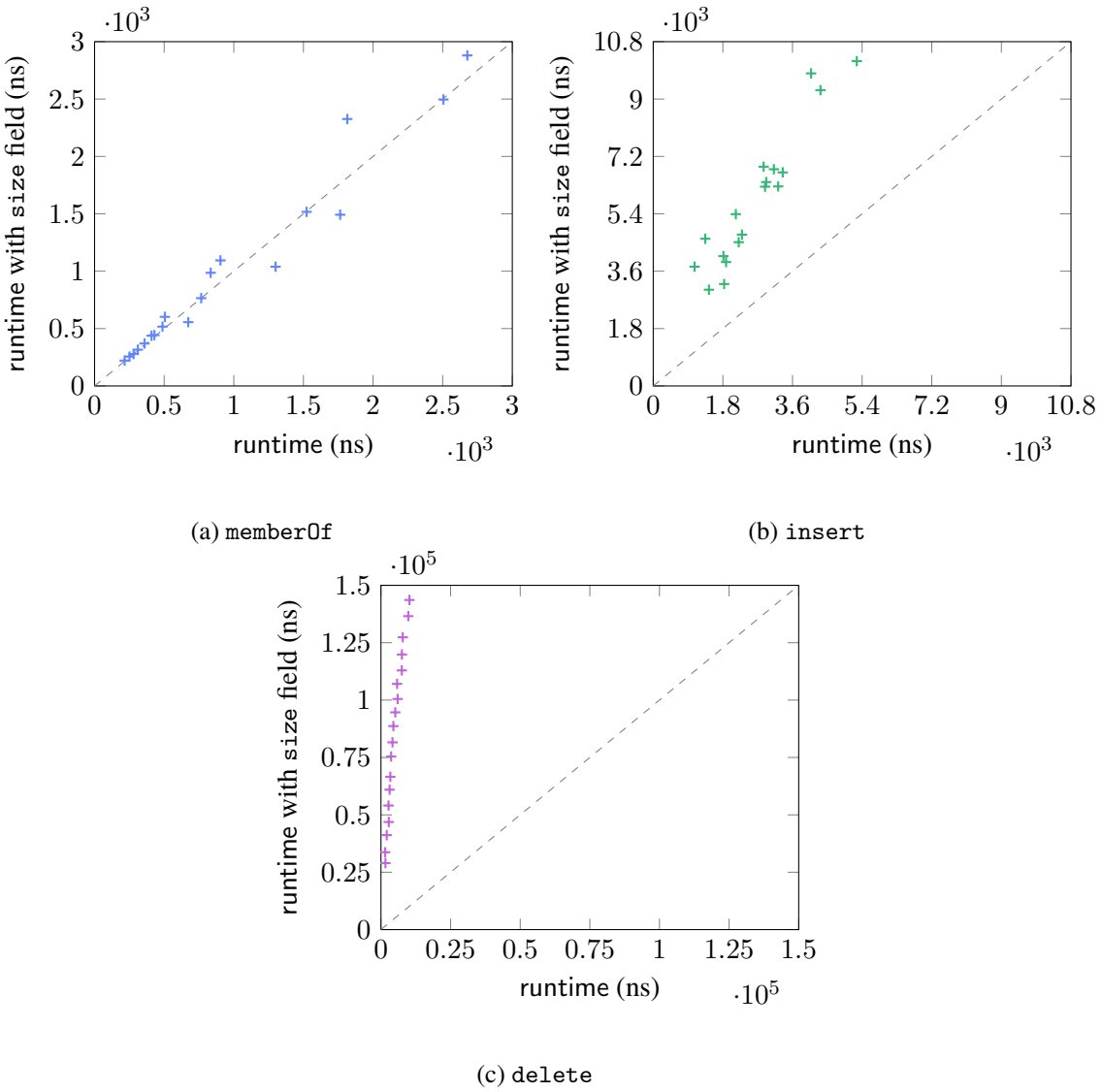
Hypothesis 3 The performance cost of adding a size field to every node is at most a constant factor.

Hypothesis 4 The performance cost is less than a factor $1.05x$.

To test our hypothesis, we augment `Node` with a size field of type `Int32`

```
enum RedBlackTree[k, v] {
  case Leaf
  case Node(Color, Int32, RedBlackTree[k, v], k, v, RedBlackTree[k, v])
}
```

We measure the execution time of `memberOf`, `insert` and `delete` as we did in section 3.2.



(a) `memberOf` (b) `insert`

(c) `delete`

Figure 3.2: Plot of the median runtime of `memberOf`, `insert` and `delete` before and after adding a size field

Figure 3.2a, 3.2b and 3.2c shows the median runtime of a `memberOf`, `insert` and `delete` function call, respectively, before and after adding a size field to each node. Each point (x, y) consists of two measurements. The x-value is the median runtime of the given function before adding the field. The y-value is the median runtime of the given function after adding the field. The dotted $x = y$ line represents the optimal case where the performance cost is 0.

For hypothesis 3, we expect that the performance cost is at most a constant factor. Figure 3.2a shows that the average performance cost for `memberOf` is a factor $1.05x$. Figure 3.2b shows that the average performance cost for `insert` is a factor $2.27x$. Figure 3.2c shows that the average performance cost for `delete` is a factor $17.82x$. The performance cost for `memberOf` seems

relatively constant. In contrast, the performance cost of `insert` and `appears` appears to decrease with the input size. We note that the performance cost for `delete` is much larger than expected. We conclude that hypothesis 3 is accepted.

For hypothesis 4, we expect that the performance cost is at most a factor $1.05x$. Figure 3.2a shows that the average performance cost for `memberOf` is a factor $1.03x$. Figure 3.2b shows that the average performance cost for `insert` is a factor $2.27x$. Figure 3.2c shows that the average performance cost for `delete` is a factor $17.82x$. We conclude that hypothesis 4 is accepted for `memberOf` but must be rejected for `insert` and `delete`.

3.4 Performance Implications of Representing Color with a Byte

The current definition of the color of a red-black tree node as an algebraic data type is semantically clear. However, clarity comes with a cost. Each of the colors becomes an object on the JVM, which means each node has a 4 byte reference to a color. We could represent the color of a node using a byte instead, replacing the `Color` type with `Int8`. We anticipate that we can improve the performance of the red-black tree by sacrificing a little readability for a reduced memory footprint. Specifically, we expect to see an increase in the performance of `memberOf`, `insert` and `delete`. The three methods vary in complexity (i.e., lines of code and number of memory allocations). The implementation of `delete` is the most complex. As a result, we expect that `delete` will show the largest increase in performance. In summary, the hypotheses are:

Hypothesis 5 The performance of `memberOf`, `insert` and `delete` is increased.

Hypothesis 6 The performance increase of `delete` is greater than the one for `memberOf` and `insert`.

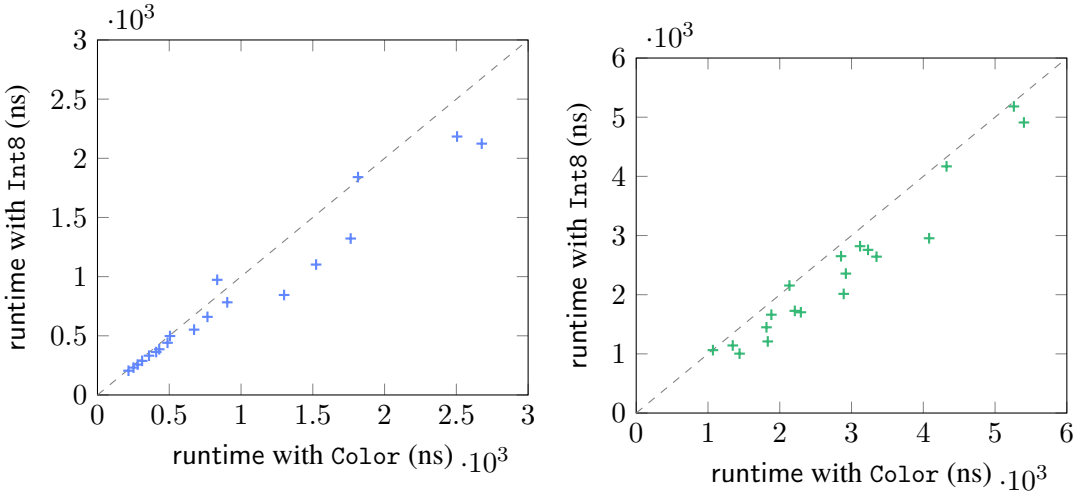
To test our hypothesis, we replace the `Color` type in `Node` by `Int8`

```
enum RedBlackTree[k, v] {
  case Leaf
  case Node(Int8, Int32, RedBlackTree[k, v], k, v, RedBlackTree[k, v])
}
```

We map each of the existing colors to a distinct integer

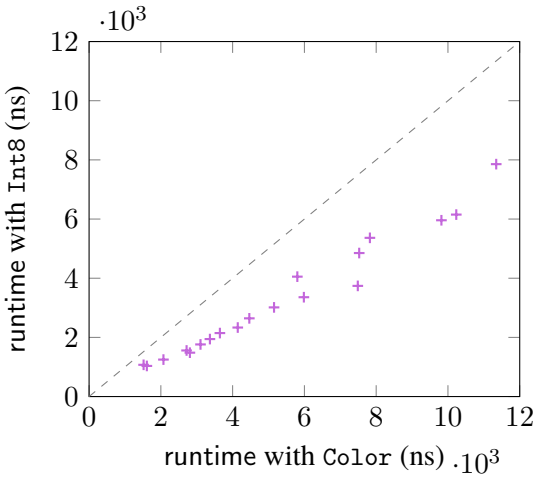
$$\begin{aligned} \texttt{Red} &\rightarrow 0 \\ \texttt{Black} &\rightarrow 1 \\ \texttt{DoubleBlack} &\rightarrow 2 \end{aligned}$$

We measure the execution time of `memberOf`, `insert` and `delete` as we did in section 3.2.



(a) `memberOf`

(b) `insert`



(c) `delete`

Figure 3.3: Plot of the median runtime of `memberOf`, `insert` and `delete` before and after replacing the `Color` type by `Int8`

Figure 3.3a, 3.3b and 3.3c shows the median runtime of a `memberOf`, `insert` and `delete` function call, respectively, before and after replacing the `Color` type by `Int8`. Each point (x, y) consists of two measurements. The x-value is the median runtime of the given function before replacing the color type. The y-value is the median runtime of the given function after replacing the color type. The dotted $x = y$ line represents the case where the performance difference is 0.

For hypothesis 5 we expect an increase in the performance of `memberOf`, `insert` and `delete` . Figure 3.3a shows an average performance increase of a factor $1.15x$ for `memberOf`. Figure 3.3b shows an average performance increase of a factor $1.21x$ for `insert`. Figure 3.3c shows an average performance increase of a factor $1.66x$ for `delete`. We note that the data is clear, except

for a single outlier in figure 3.3a. We conclude that hypothesis 5 is accepted.

For hypothesis 6 we expect that the performance increase of `delete` is greather than the one for `memberOf` and `insert`. Figure 3.3 confirms this. The average performance increase of `delete` is a factor $1.37x$ greater than the one for `memberOf` and a factor $1.44x$ greater than the one for `insert`. We conclude that hypothesis 6 is accepted.

3.5 Comparison to Sorting Algorithms

Since a red-black tree is a binary search tree, with $O(\log n)$ insertion, we can use it to sort a sequence of n comparable elements in $O(n \log n)$ time. The sorting algorithm consists of inserting each element of the sequence into the red-black tree. While red-black trees are typically not used for sorting in practice (in part due to larger memory overhead), we think a comparison against efficient sorting algorithms can serve as a performance sanity check for our implementation. We compare the performance of sorting a sequence of numbers using the red-black tree with two sorting algorithms, namely `List.sortWith` from the Flix standard library and `Arrays.sort` from the `java.util` package. `List.sortWith` uses a mergesort, ported from `sortBy` found in the libraries of Haskell[3]. `Arrays.sort` uses a dual-pivot quicksort[4] for primitive arrays such as `int[]`. Given that `Arrays.sort` has undergone extensive emperical testing and operates in-place on mutable arrays, we expect it to perform faster than both the red-black tree and `List.sortWith`. Since `List.sortWith` is an algorithm specifically designed for sorting immutable data, we expect it to perform faster than the red-black tree. In summary, the hypotheses are:

Hypothesis 7 `Arrays.sort` performs faster than `List.sortWith` and the red-black tree.

Hypothesis 8 `List.sortWith` performs faster than the red-black tree.

To test our hypothesis, we measure the time required by the red-black tree, `List.sortWith` and `Arrays.sort` to sort a sequence of 2^n random 32-bit integers for $n \in [5, 23]$. For the red-black tree, we measure the time it takes to insert 2^n elements of type `Int32`. For `List.sortWith`, we measure the time it takes to sort a `List[Int32]` of length 2^n. For `Arrays.sort`, we measure the time it takes to sort an `int[]` array of length 2^n.

[3] https://downloads.haskell.org/ ghc/latest/docs/html/libraries/containers-0.6.2.1/Data-Sequence.htmlv:sortBy
[4] https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/Arrays.htmlsort(int[])

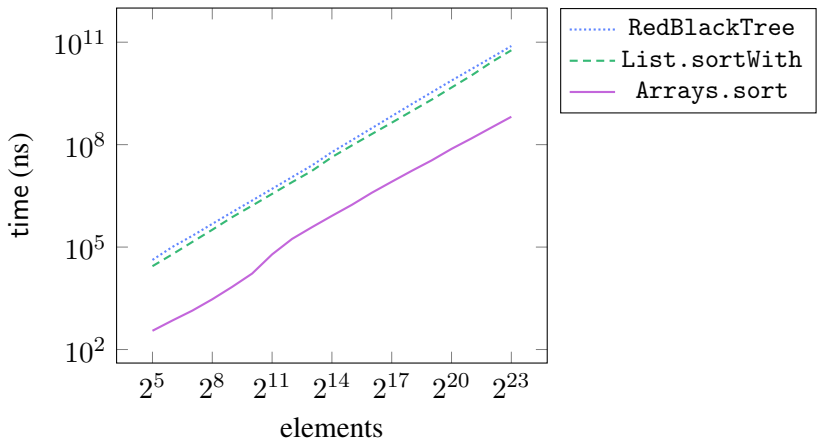


Figure 3.4: Plot of the median time required by `RedBlackTree`, `List.sortWith` and `Arrays.sort` to sort random 32-bit integers

Figure 3.4 shows the median time required by the red-black tree, `List.sortWith` and `Arrays.sort` to sort a sequence of random 32-bit integers. The x-axis shows the number of elements which ranges from $2^5 = 32$ to $2^{23} = 8,388,608$. The y-axis shows the median time required to sort the elements, measured in nanoseconds. Note that the y-axis is logarithmic. The median was computed over a number of iterations inverse proportional to the input size. To minimize noise in the measurements, we used a large number of iterations for sequences of smaller length. Due to runtime considerations, we used a smaller number of iterations for sequences of larger length. We note that the data is clean for all input sizes we consider in the range $[2^5, 2^{23}]$ and that all three algorithms appears to scale well with the input size.

For hypothesis 5, we expect that `Arrays.sort` performs faster than the red-black tree and `List.sortWith`. Figure 3.4 shows that `Arrays.sort` is faster as expected. `Arrays.sort` can be observed to be a factor $106.4x$ faster than the red-black tree. While a factor $106.4x$ sounds huge, it must be taken into account that `Arrays.sort` uses a sorting algorithm optimized for the JVM and operates on mutable arrays with excellent memory locality. In comparison, the red-black tree is immutable and has a much larger memory overhead. In addition to this, some of the slowdown can be traced to the Flix compiler. Flix currently has a factor 1-5x overhead for tail calls and a factor 1-2x overhead on uncurrying, which means up to $10x$ of the slowdown may be related to the Flix compiler. `Arrays.sort` can be observed to be a factor $71.8x$ faster than `List.sortWith` on average. We conclude that hypothesis 5 is accepted.

For hypothesis 6, we expect that `List.sortWith` performs faster than the red-black tree. Figure 3.4 shows that `List.sortWith` is faster as expected. `List.sortWith` can be observed to be a factor $1.5x$ faster than the red-black tree on average. Given that `List.sortWith` is specifically designed for sorting, the performance difference is surprisingly small. The fact that the red-black tree is only a factor $1.5x$ slower than `List.sortWith` on average, despite having a larger memory overhead, indicates that the insertion algorithm in the red-black tree is fairly efficient. We find that the sorting performance of the red-black tree is on par with `List.sortWith`. We conclude that hypothesis 6 is accepted.

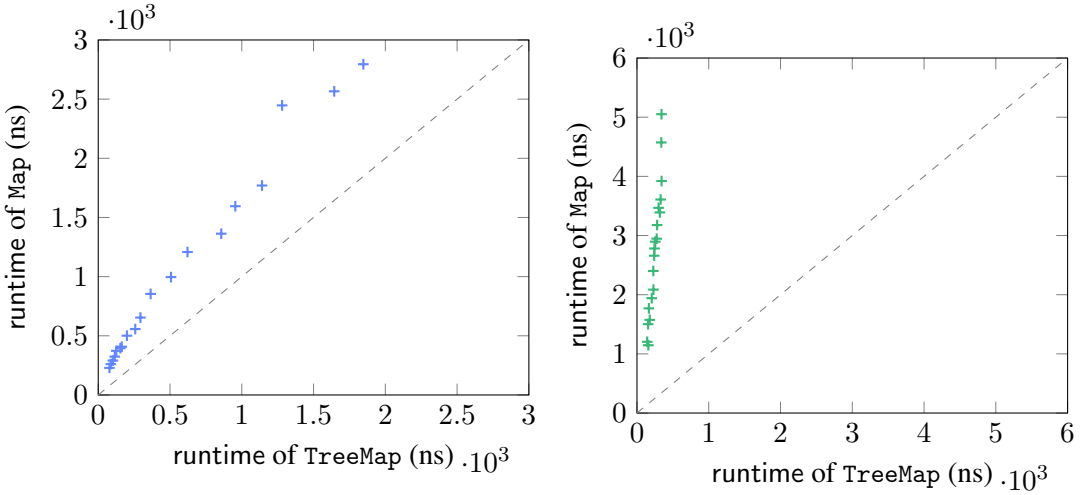
3.6 Comparison to Scala's TreeMap

The Scala standard library includes `TreeMap`, an immutable sorted map implementation, whose elements are stored in a red-black tree[5]. Both Flix and Scala compiles to JVM bytecode and runs on the Java Virtual Machine. Given that the language platform and the underlying data structure of our `Map` and Scala's `TreeMap` is identical, we expect that the performance of `Map` and `TreeMap` is comparable. Specifically, we expect that the performance of search, insert and delete is comparable. In summary, the hypothesis is:

Hypothesis 9 The performance of search, insert and delete on `Map` and `TreeMap` is comparable.

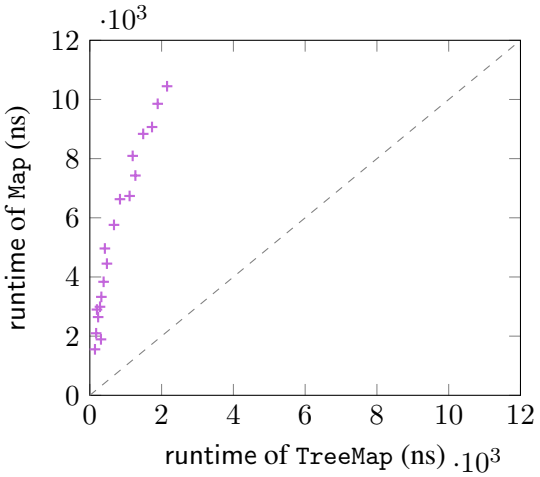
To test our hypothesis, we measure the execution time of search, insert and delete for both map implementations. We consider maps of sizes 2^n for $n \in [5, 23]$ and measure the execution time of each functions as follows. We preconstruct a map containing 2^n key-value pairs. The keys in the map are 32-bit integers from a random number generator. All values in the map is the unit value `()`. For search and delete we randomly pick an element from the map. For insert we pick a random element not yet present in the map. The search, insert and delete functions on `Map` are `memberOf`, `insert` and `delete`, respectively. The search, insert and delete functions on `TreeMap` are `contains`, `updated` and `removed`, respectively.

[5] https://www.scala-lang.org/api/current/scala/collection/immutable/TreeMap.html



(a) `memberOf` / `contains`

(b) `insert` / `updated`



(c) `delete` / `removed`

Figure 3.5: Plot of the median runtime of search, insert and delete on `Map` and `TreeMap`

Figure 3.5a, 3.5b and 3.5c shows the median runtime of search, insert and delete, respectively. Each point (x, y) consists of two measurements. The x-value is the median runtime of the given functions on Scala's `TreeMap`. The y-value is the median runtime of the given functions on our `Map`. The dotted $x = y$ line represents a performance difference of 0.

For hypothesis 9 we expect that the performance of `Map` and `TreeMap` is comparable. In other words, we expect that all points are positioned along the $x = y$ line, or close to it. Figure 3.5a shows that `Map.memberOf` is a factor $2.24x$ slower than `TreeMap.contains` on average. Figure 3.5a also shows that the performance difference seems to decrease with the map size. Figure 3.5b shows that `Map.insert` is a factor $7.21x$ slower than `TreeMap.updated` on average. Figure 3.5b also shows that the performance difference seems to increase with the map size. Figure 3.5c

shows that `Map.delete` is a factor $4.85x$ slower than `TreeMap.removed` on average. Figure 3.5a also shows that the performance difference seems to decrease with the map size. Dahse (2020) has shown that the performance of `Map` can be speed up by a factor $1.32x$-$1.96x$. Considering this, we determine that the performance of search on `Map` and `TreeMap` is comparable. We note that the performance difference on insert and delete is larger than expected. We conclude that hypothesis 9 is accepted for search and must be rejected for insert and delete.

To investigate why the performance difference is larger than expected, we conducted a short review of the red-black tree implementation used by `TreeMap`[6]. The review revealed that Scala's red-black tree is optimized by representing empty trees with `null`. As of yet, this is not possible in Flix. Scala's red-black tree also appears to be using mutable state in some parts of the implementation.

[6] https://github.com/scala/scala/blob/v2.13.2/src/library/scala/collection/immutable/RedBlackTree.scala

4

Conclusions

We have presented an efficient implementation of immutable ordered sets and maps for the Flix programming language. We have implemented 37 distinct functions for sets, including `union`, `partition` and `subsets`. We have implemented 56 distinct functions for maps, including `fold`, `find` and `reduce`. The red-black tree, set and map implementations have been merged into the Flix standard library[1].

We have evaluated the performance of the implementations, confirming theoretical results and revealing surprising discrepancies. The results shows that the observed performance of the red-black tree is in the order of $O(\log^2 n)$ where n is the size of the tree. The results also shows that the sorting performance of the red-black tree is within a factor $1.5x$ of the sorting algorithm `List.sortWith` from the Flix standard library. Finally, the results shows that the performance of searching in the immutable map is comparable to Scala's `TreeMap`.

4.1 Theory vs. Practice: Surprising Results

Our hypotheses were constructed from a theoretical and idealized world view. Given that some of the results did not match our hypotheses, some of our assumptions must have been wrong. We reflect on some of the most surprising results below.

(i) The observed performance of the red-black tree does not equal the theoretical running time of $O(\log n)$ where n is the size of the tree. We believe this is because the theoretical running time relies on the RAM model, which assumes $O(1)$ memory access (Cormen et al., 2009).

(ii) The average performance cost of adding a size field to each red-black tree node is up to $17.82x$ for the `delete` function. Given that the overhead is only 4 bytes for each red-black tree node, the observed slowdown seems excessive. We assume that the performance cost may be decreased by future improvements to the Flix compiler.

(iii) The average performance difference between using the red-black tree and `Arrays.sort` to sort a sequence of integers is $106.4x$, even though they both have a theoretical running time of $O(n \log n)$. We once again observe a clear separation between theory and practice.

[1] https://flix.dev/api

4.2 Open Problems

We conclude by describing some open problems related to this report.

- The `Map.memberOf` functions seems to be on par with Scala's `TreeMap.contains`, so why is `Map.insert` and `Map.delete` slower than `TreeMap.updated` and `TreeMap.deleted`?
- Given that Flix supports concurrency, is it practical to optimize the implementations using parallelism?
- Given that all values in `Set` currently contains a 4 byte reference to the unit value `()`, would it be practical to use another type, such as `Bool` or `Int8`?
- Given that Flix is considering adding support for `null`[2], would it be practical to represent empty trees and/or values in `Set` using `null` if and when the feature is added?

[2] https://github.com/flix/flix/issues/1154

5

References

Bagwell, P. (2001). *Ideal Hash Trees*.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. 3rd. The MIT Press. DOI: 10.5555/1614191.

Dahse, B. (2020). *Control Flow Analysis in Flix*.

Germane, K. and Might, M. (July 2014). "Deletion: The curse of the red-black tree". In: *Journal of Functional Programming* 24, pp. 423–433. DOI: 10.1017/S0956796814000227.

Lindholm, T., Yellin, F., Bracha, G., Buckley, A., and Smith, D. (2020). *The Java Virtual Machine Specification, Java SE 14 Edition*. Oracle America, Inc. URL: https://docs.oracle.com/javase/specs/jvms/se14/html/index.html.

Madsen, M., Yee, M.-H., and Lhoták, O. (2016). "From Datalog to Flix: A Declarative Language for Fixed Points on Lattices". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery, pp. 194–208. DOI: 10.1145/2908080.2908096.

Madsen, M., Zarifi, R., and Lhoták, O. (2018). "Tail Call Elimination and Data Representation for Functional Languages on the Java Virtual Machine". In: *Proceedings of the 27th International Conference on Compiler Construction*. Association for Computing Machinery, pp. 139–150. DOI: 10.1145/3178372.3179499.

Okasaki, C. (1998). *Purely Functional Data Structures*. Cambridge University Press. DOI: 10.1017/CBO9780511530104.