CONTENTS

Persistent data structures in functional programming

Scala Functional Programming Data

Krzysztof Atłasik

29 Jun 2022. 6 minutes read

One of the key aspects of the functional programming paradigm is transforming immutable values through pure functions. Whenever a pure function computes a new altered version of the data structure (for example by appending a new node to a list), the original one should remain unmodified and be available via the previous reference.

Holding

Embracing immutable structures has a number of advantages, such as facilitated multi-threading operations. Since transformations don't modify the previous version of the structures, other threads can use them without the risk that they will be inadvertently updated. Hence there is no need for using concurrency synchronization mechanisms like locks or semaphores.

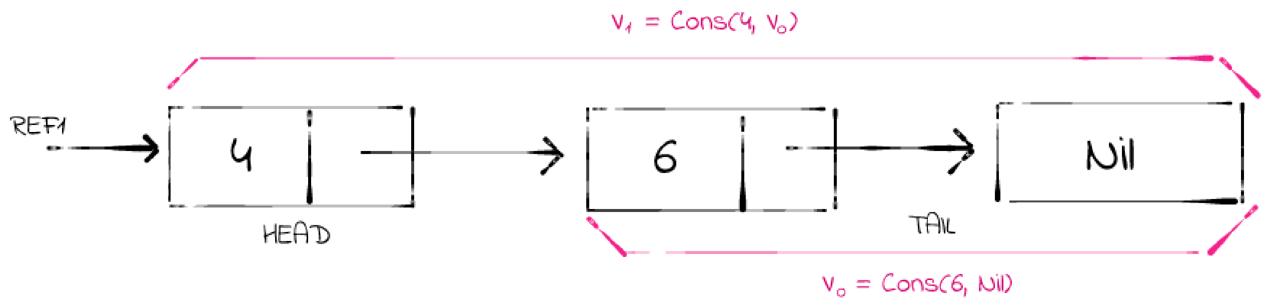
On the other hand, if modifications don't destroy the prior versions of the data structure, doesn't it mean that computations will be quite memory-hungry and inefficient? In fact, it's often possible to create new coexisting versions without copying the whole structure. We'll explore that concept further later in the article. We will take a brief look at persistent implementations of two well-known data structures: a linked list and a binary search tree.

Linked list

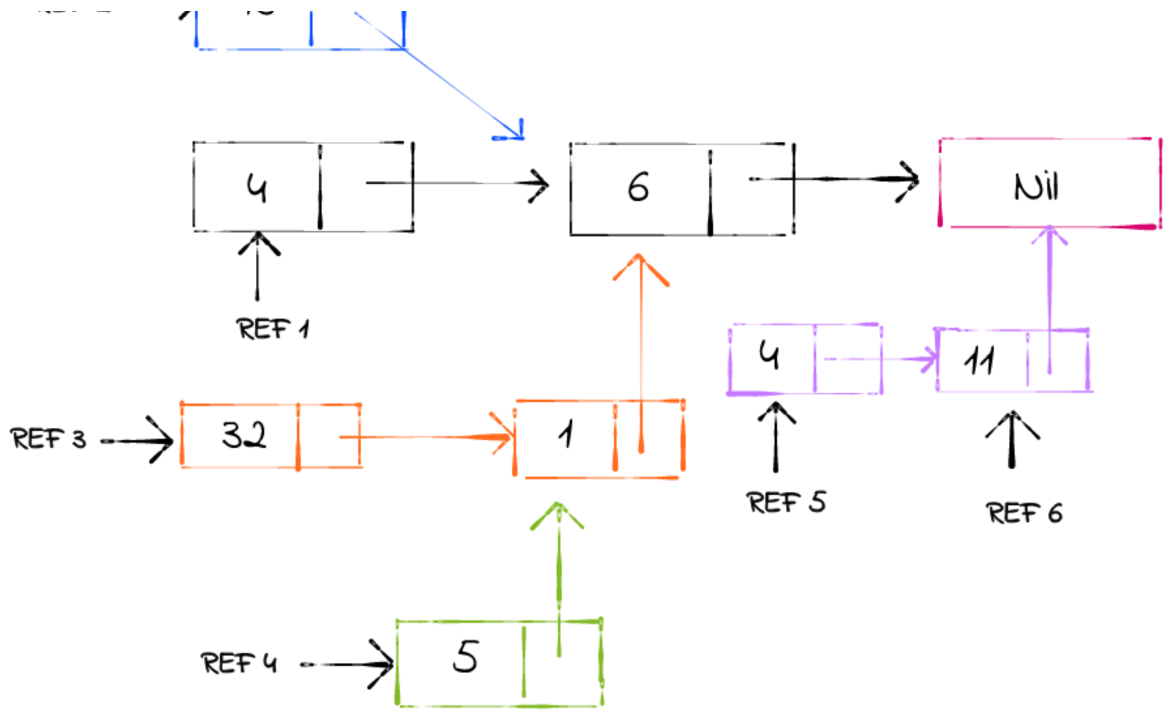
available in Scala or Java’s Vavr.

So how does it work? Let’s start by introducing the concept of the **Cons pair**. It’s an immutable structure that represents a single node of the list. It contains two fields: a stored value and a pointer to the previous node.

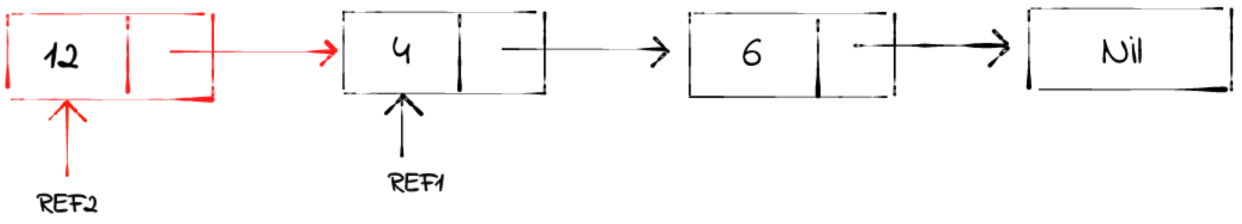
Every time a new value is prepended to the list, a fresh Cons is created. Its pointer is set to reference the previous node. The first node of the list is called the **head**, and the pointer to the previous node is the **tail**. The tail list contains all the nodes except the head. The pointer of the last value node references the **Nil** node, which is a representation of the empty list.



Each of the nodes is immutable. That gives an ability to reference a single node by multiple lists. We can create two completely separate lists by appending new heads to the original list and they still will share the tail. This way of reusing nodes between lists is called **structural sharing**. A single node has only information about its previous node, it doesn’t know or care if any other node points to it. It can be simultaneously the first element of the list for some references used in the application or in the middle of the list for another.

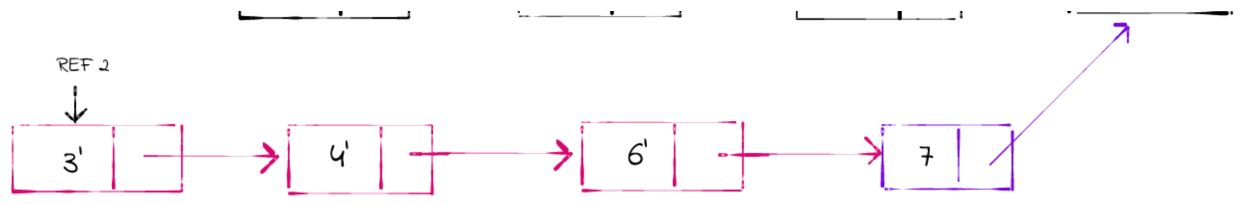


The operation to prepend the value requires only the creation of a new node. It’s pretty efficient: it is constant on memory and time. Similarly, if we want to remove the head value, we just need to return the reference of the previous node.



The original (available via REF1) and updated list (available via REF2).

The operation to append a new value to the other end of the list is more complicated. Since the nodes are immutable, we can’t just mutate the former last cell and update its pointer to the newly created last node. Instead, we copy the node and set its tail pointer to reference the new tail. Then we copy the second-to-last node and also modify its pointer to reference the just created copy of the former last cell. We repeat the operation until we have copied all the nodes.



original list and list with appended value

The same formula applies if we need to insert a new node in the middle of the list or update any of the values.

Since we need to iterate through **n** nodes, these operations have linear time complexity. They also at least partially break structural sharing: some of the nodes need to be copied.

As you might have noticed, the persistent list is singly-linked. The reason why we can't have an immutable doubly-linked list is mundane: we can't modify the previous node while appending.

And when are values actually removed from the memory? In languages that support garbage collection (most functional languages have GC), the memory of nodes that are no longer referenced can be automatically reclaimed. It's still possible to implement persistent data structures without GC, but it increases the complexity of the solution since dead nodes need to be manually deleted.

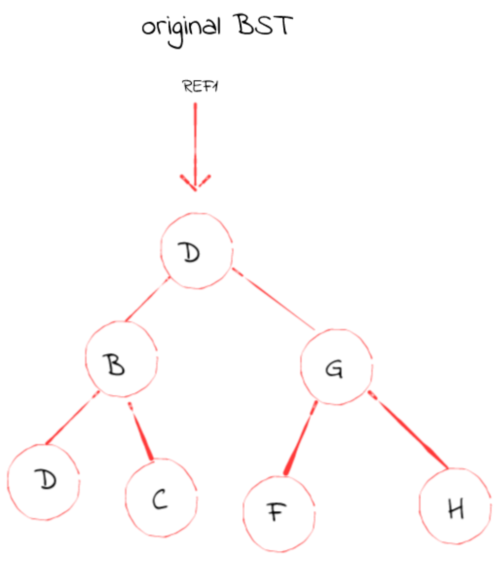
For some operations, a persistent linked list can be as efficient as its mutable variant. For another, like modifications in the middle of the list, it lags behind. Fortunately, there are also other functional data structures with better performance for such operations. For example, Scala offers Vector, which allows for efficient random access. Internally, it's backed by another persistent data structure, the immutable base-32 trie.

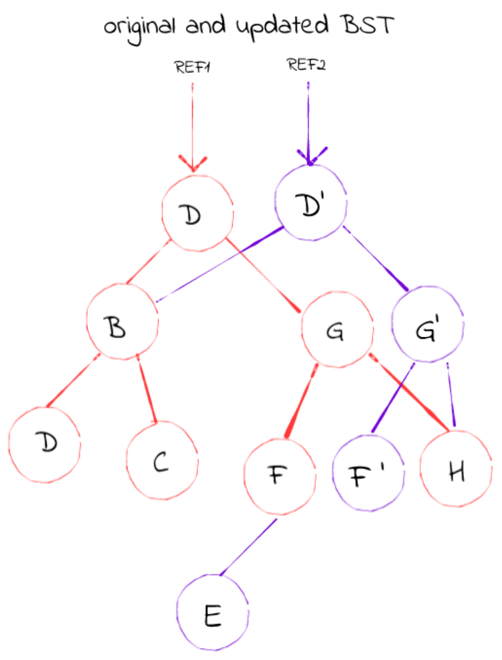
Binary Search Tree

A binary search tree is a binary tree variant that stores values in symmetric order. That means that any value at any given node is greater than each value in its left subtree and lesser than each value in its right subtree. This characteristic applies both to mutable and immutable BST.

This property allows for efficient checks of whether a value exists in the tree. The function for testing membership traverses the tree starting from the root element. If the queried value is smaller than the root element, then the function recursively searches the left subtree. Otherwise, when the query element is larger, then the right subtree is checked. In the end, the function either encounters a searched value or an empty node. Reaching an empty node means that the value is not a member of BST.

The insert function traverses the tree using the same strategy as a member function. The crucial difference is that it creates a copy of each element along the way. When it finally reaches an empty node, it inserts a fresh node there with the new value.





Each copied node and the original node will share one subtree that was not on the search path. For bigger trees, the search path contains only a small fraction of the

Mutable data structures

In many applications, immutable data structures can provide sufficient performance. It’s not always the case, though. Sometimes restrictions imposed by purely functional design (like not allowing for any in-place updates) might hamper the efficiency of processing.

Dr Chris Okasaki in his book *Purely Functional Data Structures* writes:

> … from the point of view of designing and implementing efficient data structures, functional programming's stricture against destructive updates (i.e., assignments) is a staggering handicap, tantamount to confiscating a master chef's knives. Like knives, destructive updates can be dangerous when misused, but tremendously effective when used properly. Imperative data structures often rely on assignments in crucial ways, and so different solutions must be found for functional programs.

So in certain applications using mutable data structures is inevitable, especially if performance requirements are strict.
Still, functional languages do not limit us from employing mutability. We just have to remember that updating a non-persistent structure is a destructive action. Sometimes it’s the language or framework that indicates which operations are doing side-effecting mutations. For instance, Haskell’s MArray interface implementations allow for in-place changes, but those must be wrapped in a monad like IO.

Finally, it’s generally a good idea to limit the mutability only to certain places in the code when efficiency is vital (“hot paths”) and further expose the results of the computations only with immutable values.

Conclusions

interested you in this fascinating topic. Perhaps in the future, I will be able to write another article focusing on more advanced data structures, like persistent heaps, queues or tries. Till next time!

M **morebf2pls**
8 months ago

Many thanks for your effort. What did you use while drawing diagrams?

1 0 Reply • Share ›

K **Krzysztof Atłasik** ↪ morebf2pls
9 months ago

Thanks, I used https://sketch.io/sketchpad/

1 0 Reply • Share ›

Let's do things together!

+48 22 188 11 33 (PL)
+44 56 0156 3406 (UK)

hello@softwaremill.com

Name

E-mail

Message

Your personal data collected in this form will be used only to contact you and talk about your project. For more information, see our Privacy Policy.

Send message

Company logo

Latest from our Tech Blog

Read more

webp image

7 quick steps to improve your Dockerfile

Adam Pietrzykowski, 15 Mar 2023

SoftwareMill S.A.

ul. Na Uboczu 8/87
02-791 Warszawa
Poland

Services

Overview
Backend
Frontend
Platform Engineering
Machine Learning
Big Data

Technologies

Scala
Java
Apache Kafka
Open Source

At a glance

Portfolio
How we work
Why choose us
Remote
About us

Services

Blog

Technology Trends

Contact us