Immutable Data Structures: AVL, RRB Vectors and Finger Trees

Jordi Bertran de Balanda, Morane Gruenpeter, Guillaume Hivert, Mourtala Issa Saidou, Etienne Mauffret, and Darius Mercadier

Pierre and Marie Curie University, Paris VI
Master's research group

Abstract. Immutable structures are naturally used in functional languages, each presenting a wide range of performances depending on the operation. For this paper we have chosen three data structures to study, compare and implement in C. AVL trees, RRB vectors and Finger trees all show three distinct, interesting approaches to immutable representation of maps and vectors, each with a focus on different operations. In this paper, we will define each structure and the manner in which we implemented its operations. We will show the Finger trees are most adequate for discrete operations such as push and pop, while RRB vectors are best with bulk operation as split and merge.

Keywords: immutable data structures, persistence, AVL, Relaxed-Radix-Balanced trees, Finger Trees

1 Introduction

During our second year of Master's degree we have researched immutable data structures as part of the GRAL research group. First we were sent to inquire about immutable or persistent data structures, with functional languages as an entry point. We discovered that Okasaki's 1998 book was the reference for persistent data structures " Functional programming languages have the curious property that all data structures are automatically persistent." [1] One open question remained, what's new in this domain? As for immutable data structures, it is a structure that preserves all versions of itself when modified. We are using *unref* operation giving the user the possibility to unreference a version and thus delete it.

For the purpose of this research we chose three different data structures, AVL, RRB vectors and Finger Trees which we implemented their immutable versions in an imperative language, C.

Furthermore, the implementation included discrete operations such as lookup, push, pop and update; and bulk operations such as merge and split.

Using immutable structures with bulk operations opens an implicit path to parallelization.

First we will describe the scientific context. Then we will define each structure, the procedure for the operations we implemented and their complexity.

Furthermore we will compare the structure execution time with a benchmark. Finally we will conclude on the advantages and drawbacks of the structures.

1.1 Scientifc context

The AVL tree was invented more than 50 years ago by two Soviet inventors, Georgy Adelson-Velsky and Evgenii Landis. The first description of the structure can be found in their 1962 paper "An algorithm for the organization of information".

Finger trees were originally presented in 1985 by Tsakalidis with various of Btrees, including 2-3 trees. We used the article "Finger Trees: A Simple General-purpose Data Structure" [2] where functional 2-3 Finger trees are an instance of a general design technique introduced by Okasaki (1998), called implicit recursive slowdown. The paper was published in 2006.

The RRB vector is the most recent data structure, first described in "RRB-Trees: Efficient Immutable Vectors" [3], by Phil Bagwell and Tiark Rompf, is the core data structure for RRB-vectors published in the ICFP 2015 Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming [4]. In the 2011 technical report, Bagwell and Rompf first objective was to improve vector concatenation, insert-at and split while using an immutable data structure. They show that RRB-Trees are more efficient than the default immutable vector in Scala and its complexity is in $O(\log N)$ for concatenation, insert-at and split.

2 AVL Vectors and Maps

Both our Vectors and Maps are based on the same implementation of immutables AVL Trees.

AVL Trees are self-balancing binary trees. Thanks to this two properties, accessing any node in an AVL Tree (either for read or write) has a worst case time complexity of $O(log(n))$ where n is the number of nodes in the tree. Extending them with immutability generates an overcost of $O(log(n))$ for each update operation, which doesn't change the big-O complexity of the operation. AVL Trees can then be used to implement immutables vectors or maps.

2.1 Backend implementation : AVL Trees

The AVL structure Our immutable AVL Tree structure is very similar to the mutable one. There are two majors modifications:

- The first one isn't related to immutability, but to genericity. Indeed, we'll use the same AVL implementation to implement both vectors and maps. Therefore, the structure 'avl tree' has a field 'compare' that holds the function that should be used to compare the nodes. AVL Trees are Binary Search Trees (BST), so the properties about the order of the keys that hold for BST also hold for AVL. It means that every data stored in our AVL is seen as a couple key value.

  - In the case of Vectors, the keys are integers. The user has no idea of how this is implemented, and our Vector implementation allows him to manipulate indices while he is actually manipulating those integer keys. The ‘compare‘ function just compares two integers a and b: it returns 1 if $a < b$, 0 if $a == b$ and -1 if $a < b$.
  - In the case of Maps, keys types can be choosen by the user. We provide however some helpers to use easily ‘int‘ or ‘char*‘ (strings). The function ‘compare‘ will be different for each keys. For instance, integer keys will use the same ‘compare‘ as vectors, and string keys will use ‘strcmp‘.
- The second one is directly related to immutability: a reference counter is added to each node of the AVL. We'll detail this later in this presentation. Since our AVL are immutable, some nodes are shared amongst different versions. When a version is freed (‘unref‘), we need to free the nodes that are present in this version only, and left untouched the nodes that are shared. A reference counter solves this problem efficiently.

Additionally, we store the size (number of nodes) of the AVL inside the structure. This information is cheap to maintain : we increment it when adding a node, and decrement it when removing a node. However, since our AVL can't contain two nodes with the same key, and since ‘remove(k)‘ can be called even if no node of the AVL has the key ‘k‘, we need to be careful when updating the size (that is, we check if ‘insert‘ or ‘remove‘ actually did an insertion or a remove before updating the size).

Operations

lookup works exactly the same on immutables and mutables AVL.

Update When updating a node N, every node of the path P from the root to N need to be copied. The sons of a copied node will either be copied (if they belong to P), or will have their reference counter incremented.

On the example bellow, we have an AVL containing 3 nodes, A, B and C. We update the node B. The only nodes in the path from the root A to B are A and B. Both this nodes are copied, creating the nodes A' and B' (and B' is modified accordingly the the update we are doing). C however isn't in the path from A to B. Therefore, it is not copied, and the new AVL points toward it (and thus, its reference counter is incremented).

Insert ‘insert‘ is trickier than ‘update‘, because of the rotations that may be needed to balance the tree. Same as ‘update‘, the path from the root to position where the node we are inserting will go is copied, while the other nodes are shared between the new and the old version. However, since AVL are balanced tree, rotations may be needed to rebalance the tree. Because of the immutability, we want to be sure that the nodes that changes during the rotation aren't shared with the old version. We might think that simply copying those nodes would solve the problem. However, if those nodes were created during the insert, then

when copying them, we'd lose references on them, resulting in a memory leak. This problem could be solved thanks to the reference counter, which allows us to know how much reference there still are on a node. However, a case by case analysis shows that every node that is changed during a rotation following an insert, so we can just do a mutable rotation without worrying.

Delete 'delete' is quite like 'insert', in that we'll need to worry about immutability when performing rotations. Like any function modifying a node in an AVL, every node from the root to the node that we need to delete is copied. Once again, rotations may be needed to adjust the balance of the tree. Same as 'insert', we need to be carefull during the rotations as we don't want to either corrupt the old version, or lose track of some memory. This time, the case-by-case analysis will reveal that when a rotation is needed, exactly one node shared with the old version will be modified. Hence, we just have to copy this node to have an immutable 'delete'.

Unref Thanks to the reference counter we implemented, an immutable 'unref' is very natural to implement. If the decremented reference counter of the tree to unref is strictly greater than 1 nothing needs to be done. Otherwise, we delete the root, and call recursively 'unref' on the childs.

Invariants Any invariant valid on a binary search tree is valid on an AVL. For instance, every node to the left of the root has a smaller value, and every node to its left has a greater value. Because AVL are balanced, the worst case height of an AVL is $O(log(n))$ where n is the size of the AVL. It can be shown that $height(avl_tree) \leq 1.44\ log(n)$ where n is the number of nodes of avl_tree. This invariant holds before and after any operation.

Immutability is also an invariant: an immutable AVL Tree can't be modified.

Immutables Vectors Once we have our immutables AVL Trees, vectors aren't far away.

An AVL Vector can be represented just by an AVL Tree. However, we'll also store the maximum index of the elements of the vector, which will be usefull to do 'push' and 'pop', and know the size of the vector.

size The size of a vector is equal to the index maximum of the array. Knowing it is just accessing to the field $max - index$ of the vector structure.

update 'update' only needs to call the 'update' function of the AVL Trees, and update the $max - index$ if necessary. 'update' can also insert a node in a Vector if its index isn't already in the Vector. For instance, after,

```
vec = create()
vec2 = update(vec, 10, 1)
                   ^   ^
            the index    the value
```

‘vec2‘ holds a Vector of then elements: the first nine aren’t initialized, and the thenth is ‘1‘.

push Because we keep track of the maximum index of the vector, ‘push‘ becomes quite trivial: we insert the new element at index $max - index + 1$, and increment the $max - index$.

pop Once again, $max - index$ makes things easy: the last element of the array has index $max - index$. ‘pop‘ can then call the AVL Tree ‘remove‘ function on the $max - index$ index.

merge and split This functions are far from efficient on AVL Vectors. Because AVL are balanced and are binary search trees, we can’t insert a tree into antother (for ‘merge‘), or just cut them at a given node (for ‘split‘). Hence, for ‘merge‘, we create a new tree in which we insert, in a mutable way, every nodes of the two vectors we want to merge. And for ‘split‘, we create two vectors in which we insert, in a mutable way, the nodes of the old trees, depending on their index.

Immutable Maps Immutable maps are basically an abstraction layer of the AVL Trees. Every operation on a map is a call to the corresponding function on the AVL, after boxing of the arguments if necessary.

Our maps are very generic: both keys and values can have any type. However, the keys need to be comparable because of the BST nature of the AVL. Therefore, the user needs to supply a comparison function that takes two ‘void*‘ (that he will need to cast), and returns ‘0‘, ‘1‘ or ‘-1‘.

2.2 Complexity

We consider one AVL with n elements. For the merge operation, the second AVL has m elements.
According to the propriety of AVL, the lenght of the first AVL is $log(n)$ and the second is $log(m)$.

Time Complexity

The functions for the immutable AVL use the same algorithm than the classic functions. This way the only difference between those functions is the dupplication of some data to stay immutable.

So we naturally obtain the following time complexity for the operations :

Operation	AVL
Access	$\mathcal{O}(log_2(n))$
Insert	$\mathcal{O}(log_2(n))$
Remove	$\mathcal{O}(log_2(n))$
Merge	$\mathcal{O}(log_2(n))$
Splitting	$\mathcal{O}(log_2(n))$

Table 1: Time Complexity on AVL

Spatial Complexity

In this section we compute the spacial complexity of operations. In deed, the immutability of the data structure forced us to copy some data and it is an important point to consider.
The spatial complexity of an operation *op* is noted : CS_{op}

Insert and remove : Insert operations go through the AVL. For each node, we check which sub-AVL, right or left, we need to recur into. If we need to recur into the right sub-AVL, then copy the value of the node with a pointer on the left sub-AVL and recursively call on the right sub-AVL. This way the algorithm copies only one value for each of the $log(n)$ call. We can also note that only the copied node will be change with the rotation use to re-balance the AVL. So we can deduce that the spatial complexity for this operation is

$$SC_{insert} = \mathcal{O}(log_2(n)) \tag{1}$$

We obtain the same complexity for the remove function.

Merge : The merge operation on AVL forces us to change the whole structure. So this function has to copy every node of both AVLs given, and the complexity is naturally:

$$SC_{merge} = \mathcal{O}(n + m) \tag{2}$$

However, by successively add every value of the second AVL in the first, it is possible to get a complexity in

$$\begin{aligned} SC_{merge} &= \mathcal{O}(\sum_{i=0}^{m} log_2(n+i)) \\ &= \mathcal{O}(log_2(\prod_{i=0}^{m}(n+i))) \\ &= \mathcal{O}(log_2(n^m)) \\ &= \mathcal{O}(m.log_2(n)) \end{aligned} \tag{3}$$

But this technique is slower in time.

Split : The split operation also forced us to re-write the all data structure in two new AVL and we get a complexity in :

$$SC_{Split} = \mathcal{O}(n) \tag{4}$$

AVL : We now want to compute the spatial complexity of an AVL with n elements. We consider that the user did not make any split and the merge has been done by using successively add every value of the second AVL into the first. At least we note k the number of call to remove. The worst case scenario for the spatial complexity is obtained when adding $n + k$ elements then removing the k elements. By removing an elements earlier, the spatial complexity of both the remove and the insert will be smaller.

$$\begin{aligned}
SC_{n+k\ insert} &= \sum_{i=1}^{n+k}(log_2(i)) \\
&= log_2((n+k)!) \\
SC_{k\ remove} &= \sum_{i=0}^{k}(log_2(n+i)) \\
&= log_2(\prod_{i=0}^{k}(n+i))
\end{aligned} \tag{5}$$

$$\begin{aligned}
SC_{AVL} &= SC_{n+k\ insert} + SC_{k\ remove} \\
&= log_2((n+k)!) + log_2(\prod_{i=0}^{k}(n+i)) \\
&= \mathcal{O}(log_2((n+k)^{n+k})) + \mathcal{O}(log_2(n^k)) \\
&= \mathcal{O}((n+k)log_2(n+k)) + \mathcal{O}(k.log_2(n)) \\
SC_{AVL} &= \mathcal{O}((n+k)log_2(n+k))
\end{aligned}$$

We can note that if the number of remove does not depend on the number of insert operations, which mean that the user uses those function from time to time but not regularly, the complexity becomes:

$$\begin{aligned}
SC_{RRB} &= \mathcal{O}((n+k)log_2(n+k)) \\
&= \mathcal{O}(n.log_2(n))
\end{aligned} \tag{6}$$

Operation	Time Complexity	Spatial Complexity
Access	$\mathcal{O}(log_2(n))$	-
Insert	$\mathcal{O}(log_2(n))$	$\mathcal{O}(log_2(n))$
Remove	$\mathcal{O}(log_2(n))$	$\mathcal{O}(log_2(n))$
Merge	$\mathcal{O}(log_2(n))$	$\mathcal{O}(n+m)$ or $\mathcal{O}(m.log_2(n))$
Splitting	$\mathcal{O}(log_2(n))$	$\mathcal{O}(n)$
AVL	-	$\mathcal{O}((n+k)log_2(n+k))$

Table 2: Time and spatial Complexity on AVL

3 Relaxed Radix Balanced Vectors

3.1 Definition

Relaxed Radix Balanced trees are used as an extension of immutable Vectors in functional languages. The immutable vector is based on balanced wide trees, called RB-trees (Radix Balanced trees). Bagwell and Rompf allow the structure to be partially filled. The RB tree has m-branches that are fixed and filled, while the RRB tree uses relaxing technique on the fixed branching factor m. At a given level there are expected to be exactly m^{k-1} items where $m = 32$. Thus the index i will be found in the sub-tree $[i/(m^{k-1})]$

At first, the structure has only one level of pointers, when the first level is filled with 32 elements, a new level is created. The new level has the possibility to point to 32 different vectors, making the structure max size 32^n, where n is the number of levels. When elements are inserted the structure is filled without leaving any space, but when we use split or merge, we can keep the elements at the same location of elements. However changing there location can be more efficient for future operations.

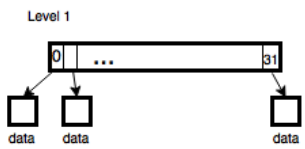

Fig. 1. RRB vector with 32 leaves and only one level

The structure implementation in C

```
typedef struct _rrb {

    int level;    // Depth of Node.

    int ref;      // Number of elements pointing to it.
```

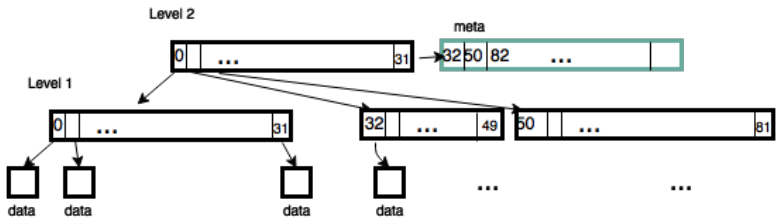

Fig. 2. RRB vector with two levels and metadata

```
    int elements; // Number of elements contained.

    int *meta;    // Indicates if relaxed tree.

    bool full;    // Indicates if the node is full.

    union {

        struct _rrb** child;

        imc_data_t** leaf;

    } nodes;   // Contains the lefs or the nodes.

} rrb_t;
```

Sizes metadata is a parameter kept for unbalanced nodes as a cumulative array of all sub-tree sizes. The most left element represent the size of the most left sub-RRB tree, after that each slot adds the size of the sub-RRB tree to sum of all the trees to the left. The metadata array is illustrated in figure 2 where the second sub-tree isn't full.

3.2 Operations

Push is trivial when adding an element to a RRB vector that isn't full, similarly to a normal vector we add the leaf to an empty slot. For immutability we clone the path to the slot. When the RRB vector is full we need to add a new level, as we can see in figure 3 when adding the

Pop in RRB vectors is taking out the element on the right path of the tree structure. To keep immutability of the structure we will clone the path before deleting the element

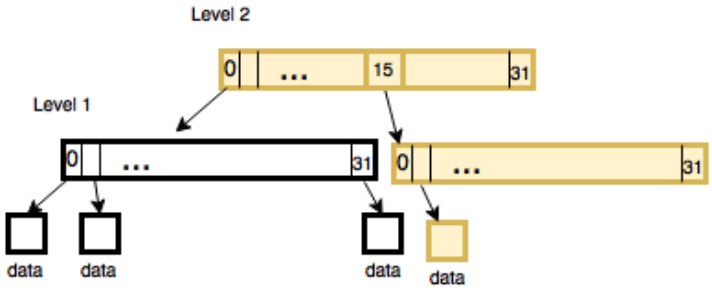

Fig. 3. Push operation on RRB vectors for 449 element

Lookup is an operation that neither changes the structure nor doing any side effect. Therefore it is an immutable operation by nature. First we will check if there is a metadata array, when the structure isn't relaxed and is filled from left to right without spaces there is no need for the metadata array. After using the metadata, a calculation of the position is possible with the function bellow:

```
calc_position -> (index >> (5 * (level - 1))) & 31$
```

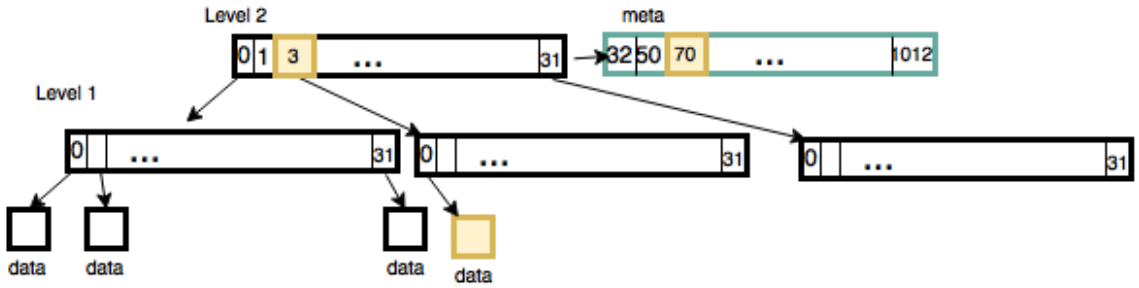

Fig. 4. Lookup operation on RRB vector

Update in RRB vector can be used only on indexes that already exists in the structure. The path to the updated element is cloned and the new version is returned.

Split divides the RRB-Tree into two trees and stores both parts into left and right using the index to indicates the cut. The Element pointed by the index will be contained in left resulting tree. If the tree is not relaxed, left will not be relaxed. Right will always be relaxed, due to the nature of the split. Than we will use a balancing function to rearrange the tree to make it balanced as possible,

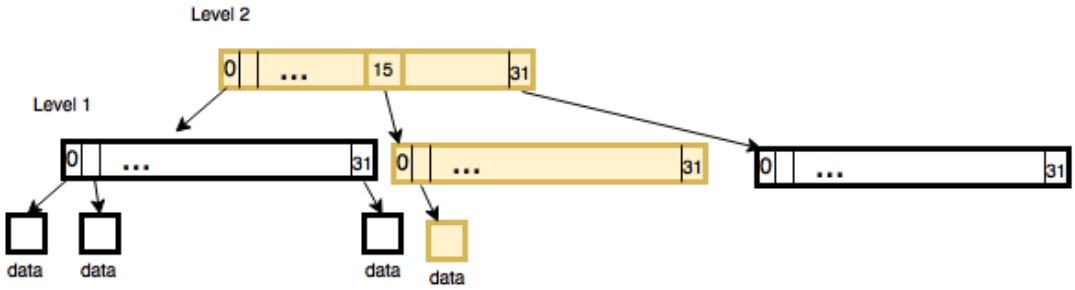

Fig. 5. update operation on RRB vector

but really costly, as it involves a lot of memory moves from different nodes and leaf in the tree, and as many copies. There will be always a compromise between performance and memory use.

Merge depends on the number of levels of each tree, if the number of level isn't the same, the lower tree will be absorbed in the higher tree. When the number of levels is equal, the merge will start from the bottom of the merging branches. The right branch of the left tree and the left branch of the right tree. First the an operation on merging leaves will take place and then for each level a new subtree will be created. Then a balanced operation will rebalance the subtree.

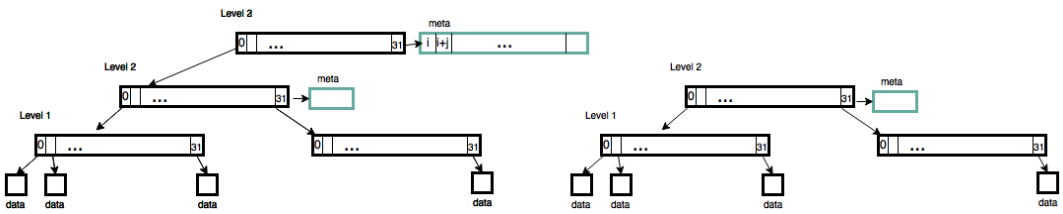

Fig. 6. Two RRB vectors ready for merge

3.3 Complexity

We consider one RRB Vector with n elements. For the merge operation, the second RRB Vector has n_2 elements.
According to the propriety of RRB Vector, the height of the first RRB Vector is $log_m(n)$ and the second is $log_m(n_2)$ with m the size of a vector.

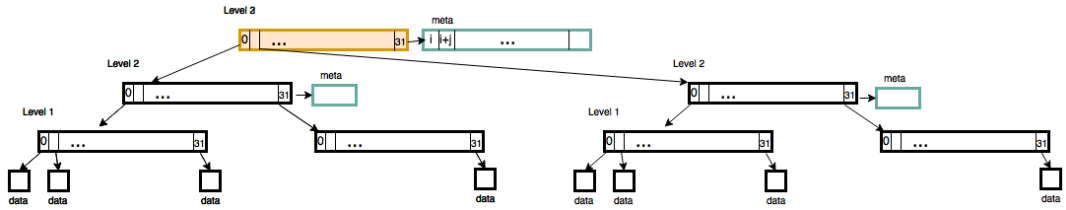

Fig. 7. Simple merge when one RRB has more levels than the other

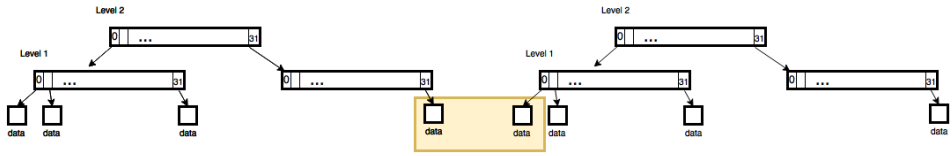

Fig. 8. Merge on RRB- bottom up- step 1

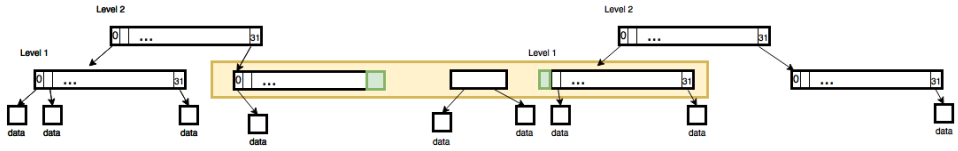

Fig. 9. Merge on RRB- bottom up- step 2

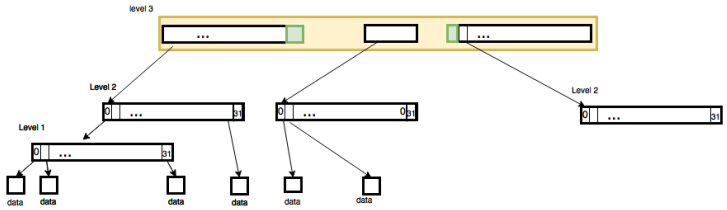

Fig. 10. Merge on RRB- bottom up- step 3

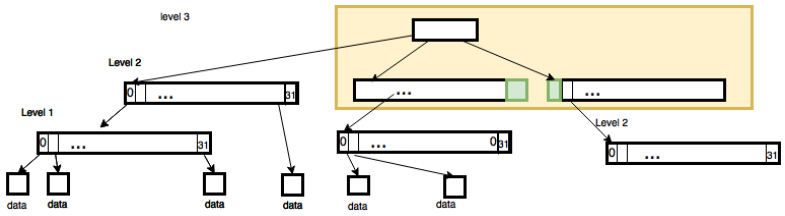

Fig. 11. Merge on RRB- bottom up- step 4

Time Complexity

The functions for the immutable RRB Vector use the same algorithm as the classic functions. This way, the only differences between mutable and immutable are the reference counting and cloned paths. These features are required to ensure the duplication of data with the goal of remaining immutable.

So we naturally obtain the following time complexity for the operations :

Operation	RRB Vector
Access	$\mathcal{O}(log_m(n))$
Push	$\mathcal{O}(log_m(n))$
Pop	$\mathcal{O}(log_m(n))$
Update	$\mathcal{O}(log_m(n))$
Merge	$\mathcal{O}(log_m(n))$
Splitting	$\mathcal{O}(log_m(n))$

Table 3: Time Complexity on RRB

Spatial Complexity

Push, pop and update : Push operations compute where to push the value and only copy the path to access the vector the change should occur in. We can note that if the RRB Vector is full, the push function will not copy anything but will simply add another level.

$$SC_{Push} = \mathcal{O}(log_m(n)) \tag{7}$$

We obtain the same complexity for the pop and the update function.

Merge : The merge operation on two RRB Vector is made by taking the rightmost value of the first and merge it with the leftmost value in the second RRB Vector. The algorithm need to copy the upper level to note this change and do it recursively up to obtain a correct RRB Vector. The complexity is in :

$$SC_{Merge} = \mathcal{O}(log_m(n)) \tag{8}$$

Split : The split operation first looks up the location of the split. Then the first RRB Vector is a pointer on the left size and we copy the right part. We obtain a complexity in :

$$SC_{Split} = \mathcal{O}(log_m(n)) \tag{9}$$

RRB Vector : We now want to compute the spatial complexity of a RRB Vector with n elements. We note k the number of call to pop and r the number of calls to update. The worst case scenario for the spatial complexity is obtained when going to $n+k$ elements without removing any elements, then doing the r update, at least removing the k elements. By removing an element earlier, the spatial complexity of the remove, update and insert operations will be smaller. Note that we can add value and/or merge two RRB Vector to get to $n+k$ elements. We can also pop or split to go to the n elements. In the worst case scenario, the split on a RRB Vector with nb value will create a RRB Vector with 1 element and the other RRB Vector will have $nb-1$ elements.

$$\begin{aligned}
SC_{to\ n+k} &= \sum_{i=1}^{n+k}(log_m(i)) \\
&= log_m((n+k)!) \\
SC_{k\ remove} &= \sum_{i=0}^{k}(log_m(n+i)) \\
&= log_m(\prod_{i=0}^{k}(n+i)) \\
SC_{r\ update} &= r.log_m(n+k)
\end{aligned}$$

$$\begin{aligned}
SC_{RRB} &= SC_{to\ n+k} + SC_{k\ remove} + SC_{rupdate} \\
&= log_m((n+k)!) + log_m(\prod_{i=0}^{k}(n+i)) + r.log_m(n+k) \\
&= \mathcal{O}((n+k)log_m(n+k)) + \mathcal{O}(k.log_m(n)) + \mathcal{O}(r.log_m(n+k)) \\
&= \mathcal{O}((n+k+r)log_m(n+k)) \\
SC_{RRB} &= \mathcal{O}((n+k+r)log_m(n+k))
\end{aligned} \tag{10}$$

We can note that if the number of pop and update does not depend on the number of push, which mean that the user use those function from time to time but not regularly, the complexity become :

$$\begin{aligned}
SC_{RRB} &= \mathcal{O}((n+k+r)log_m(n+k)) \\
&= \mathcal{O}(n.log_m(n))
\end{aligned} \tag{11}$$

Operation	Time Complexity	Spatial Complexity
Access	$\mathcal{O}(log_m(n))$	-
Push	$\mathcal{O}(log_m(n))$	$\mathcal{O}(log_m(n))$
Pop	$\mathcal{O}(log_m(n))$	$\mathcal{O}(log_m(n))$
Update	$\mathcal{O}(log_m(n))$	$\mathcal{O}(log_m(n))$
Merge	$\mathcal{O}(log_m(n))$	$\mathcal{O}(log_m(n))$
Splitting	$\mathcal{O}(log_m(n))$	$\mathcal{O}(log_m(n))$
RRB Vector	-	$\mathcal{O}((n+k+r)log_m(n+k))$

Table 4: Time and spatial Complexity on RRB

4 Finger Trees

4.1 Description

Finger tree data structure is introduced by Hinze and Patterson (ref. bib) as a representation of persistent sequences supporting access to both its ends in amortized constant time. Usual Operations s.t split and merge are in time logarithmic. Finger Trees (also called 2-3 Finger Trees) are based on 2-3 trees data structures and thus, every internal node has two or three children. The data values are on the external nodes (leaf). A complete illustration of a finger tree and its elements is given in figure 12.
The structure of finger tree data structure is given below:

$$\begin{aligned} \text{FingerTree a} = \mathit{Empty} & \\ \mid \text{Single a} & \\ \mid \text{Deep (Digit a) (FingerTree (Node a)) (Digit a)} & \end{aligned} \tag{12}$$

$$\begin{aligned} \text{Digit a} = \text{One a} & \\ \mid \text{Two a a} & \\ \mid \text{Three a a a} & \\ \mid \text{Four a a a a} & \end{aligned} \tag{13}$$

$$\begin{aligned} \text{Node a} \ = \text{Node2 a a} & \\ \mid \text{Node3 a a a} & \end{aligned} \tag{14}$$

Hinze and Patterson Finger trees are inherently persistents [2](immutables) as they are designed and implemented in Haskell. To complete our task, that is re-designing and re-implementing finger tree in C language, we use references on the finger tree nodes. Operations (update, push, pop...) on a finger tree do not recreate the entire tree, but modify only the necessary nodes and refer to the rest of the tree using reference. Because a reference (typically only the size of a pointer) is usually much smaller than the object itself, this results in memory savings and a potential boost in execution speed.

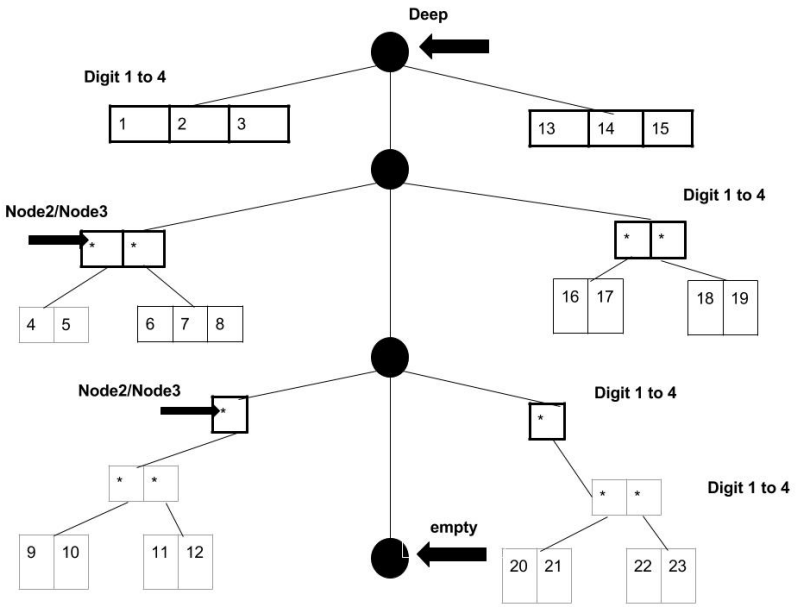

Fig. 12. An example finger tree containing values 1 to 23

4.2 Operations

Push Finger trees are mainly created to facilitates operations at the ends. We always append data values at the top suffix. The interesting case in when our finger tree is of type: `Deep prefix T suffix`

If the prefix contains less than four data values, we simply add the new data in the prefix. Note that we obtain a new version of our finger tree by recreating only the prefix and referencing the nodes of the original tree.

If the prefix contains four data values, we split the prefix in two nodes of one and three data values; we append the data value to the first node and the resulting node will be our new prefix. We then recursively try to append the second node at a deeper level.An illustration of this case is in figure13.
Similarly, only affected nodes are recreated and we use references on the rest.

Pop In contrast with Push, Pop can only remove an element from the end of the structure. Simply put, it is the dual of the Pop operation, where we remove a node from the end of the structure, and seek to promote deeper nodes to the current level if the removal means that the current level would have an empty lateral node.

Lookup The lookup operation leverages what the functional implementation calls a monoid: a function that, applied to the individual nodes, provides data that enables us to obtain properties of the tree as a whole and its subtrees. In our

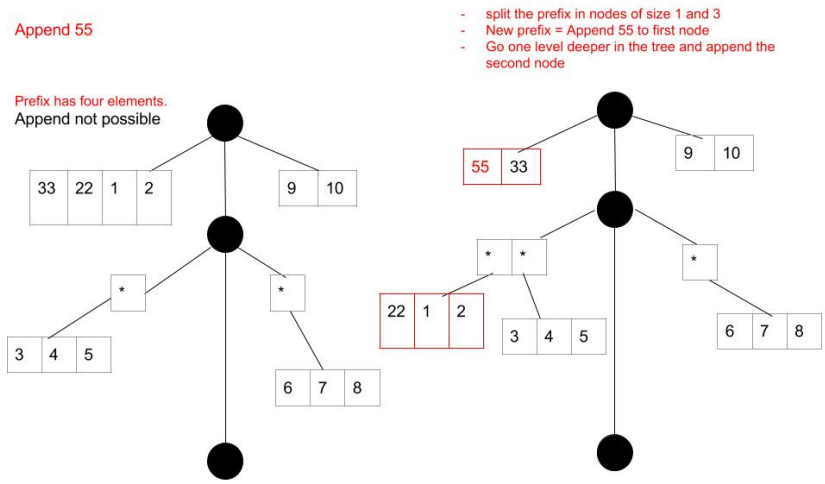

Fig. 13. An example finger tree containing values 1 to 23

case, we simply use a field in the lateral node data structure to count the quantity of individual data points available below the node. Lookup then becomes a simple case of traversing the spine while accumulating the quantity of data nodes already skipped, and doing the same within the chosen lateral node once it has been located.

Update The update operation leverages the fact that we can use side-effects to optimize the tree traversal, as opposed to a functional context where we would be obliged to traverse the whole spine in the case where the target node is located in the right side of the tree. Using a modified lookup, we determine the side the target data is located on, as well as the depth we have to go into the spine to reach the root lateral node corresponding to the target. We then simply recursively rebuild the tree, traversing up to the obtained depth and into the lateral node up to the target. In the following figure, we thus only rebuild the green node, instead of needing to rebuild the whole spine.

4.3 Complexity

We consider one Finger Tree Vector with n elements. For the merge operation, the second Finger Tree has m elements.
According to the propriety of Finger Tree, the height of the first Finger Tree is $log_4(n)$ and the second is $log_4(m)$.

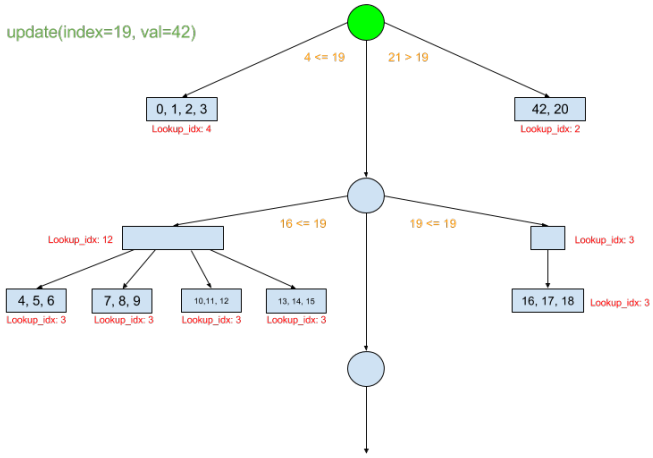

Fig. 14. Finger Tree- update operation- done

Time Complexity

The functions for the immutable Finger tree use the same algorithm than the classic functions. This way the only difference between those functions is the duplication of some data to stay immutable.

So we naturally obtain the following time complexity for the operations :

Operation	Finger Tree	Amortized cost
Access	$\mathcal{O}(log_4(n))$	-
Push	$\mathcal{O}(log_4(n))$	$\mathcal{O}(1)$
Pop	$\mathcal{O}(log_4(n))$	$\mathcal{O}(1)$
Update	$\mathcal{O}(log_4(n))$	-
Merge	$\mathcal{O}(log_4(n))$	-
Splitting	$\mathcal{O}(log_4(n))$	-

Table 5: Time Complexity on Finger Tree

Spatial Complexity

Push, pop : Push add the element in the array in the top of the finger tree. Then if this vector has more than 4 elements, it keep one of them and put the

4 others in the sub-level, and so on recursively. In the worst case scenario, the function need to recursively modify the whole side being pushed/popped on, making the complexity:

$$SC_{Push} = \mathcal{O}(log_4(n)) \tag{15}$$

How ever the worst case appear only when every level is full, which occur only every n push. So we can compute the amortized cost for the spatial complexity (ASC):

$$\begin{aligned} ASC_{Push} &= \frac{SC_{push}}{n} \\ &= \mathcal{O}(1) \end{aligned} \tag{16}$$

We obtain the same complexity for the pop function, for the opposite reason: when a node is popped, we promote a node from the deeper tree on the same side, and in the worst case need to rotate nodes from the opposite side to our side, in order to maintain a valid finger tree structure.

Update : To update a value, we first determine the depth of the value to update in the spine. We then copy every node in the path from the root to the node in which the value is located, and simply alter the new copy of that last node. In the worst case scenario the value is in the last level of the spine, and the algorithm needs to copy the spine and the path up to the value which has the same number of array as the level it is on.

$$SC_{update} = \mathcal{O}(log_4(n)) \tag{17}$$

Merge : The merge algorithm builds the resulting tree out of simultaneous deconstruction of both input trees until one or both are empty. In the worst case scenario, one of the trees is fully loaded, and the merge operation causes a cascade reallocation of all nodes on both sides, while the recreation of the spine will *always* be necessary.

$$\begin{aligned} SC_{Merge} &= \mathcal{O}(log_4(n) + log_4(m)) \\ &= \mathcal{O}(log_4(n.m)) \end{aligned} \tag{18}$$

Split : this operation was not implemented, but its complexity should be

$$SC_{Split} = \mathcal{O}(log_4(n)) \tag{19}$$

Finger Tree : We now want to compute the spatial complexity of a Finger Tree with n elements.
We note k the number of call to pop and r the number of call to update. The worst case scenario for the spatial complexity is obtain when going to $n + k$ elements without removing any elements then doing the r update, at least removing the k elements. By removing an elements earlier, the spatial complexity

of both the remove, update and the insert will be smaller.
Note that we can add value and/or merge two Finger Tree to get to $n + k$ elements. We can also pop or split to go to the n elements. In the worst case scenario, the split on a Finger Tree with nb value will create a Finger Tree with 1 element and the other Finger Tree will have $nb - 1$ elements.

$$\begin{aligned} SC_{to\ n+k} &= (n+k) \times ASC_{to\ nk} \\ &= \mathcal{O}(n+k) \\ SC_{k\ remove} &= k \times ASC_{k\ remove} \\ &= \mathcal{O}(k) \\ SC_{r\ update} &= r.log_4(n+k) \end{aligned} \tag{20}$$

$$\begin{aligned} SC_{Finger} &= SC_{to\ n+k} + SC_{k\ remove} + SC_{rupdate} \\ &= \mathcal{O}(n+k) + \mathcal{O}(k) + \mathcal{O}(r.log_4(n+k)) \\ &= \mathcal{O}(n+k+r.log_4(n+k)) \end{aligned}$$

We can note that if the number of pop and update does not depend on the number of elements (i.e the user use those functions time to time but not regularly) then the spatial complexity is close to the spatial complexity for the mutable finger tree:

$$\begin{aligned} SC_{Finger} &= \mathcal{O}(n+k+r.log_4(n+k)) \\ &= \mathcal{O}(n+log_4(n)) = \mathcal{O}(n) \end{aligned} \tag{21}$$

Operation	Time Complexity	Spatial Complexity
Access	$\mathcal{O}(log_4(n))$	-
Push	$\mathcal{O}(n)(AC\ in\ \mathcal{O}(1))$	$\mathcal{O}(n)(ASC\ in\ \mathcal{O}(1))$
Pop	$\mathcal{O}(n)(AC\ in\ \mathcal{O}(1))$	$\mathcal{O}(n)(ASC\ in\ \mathcal{O}(1))$
Update	$\mathcal{O}(log_4(n))$	$\mathcal{O}(log_4(n))$
Merge	$\mathcal{O}(log_4(n))$	$\mathcal{O}(log_4(n))$
Splitting	$\mathcal{O}(log_4(n))$	$\mathcal{O}(log_4(n))$
Finger Tree	-	$\mathcal{O}(n+k+rlog_4(n+k))$

Table 6: Time and spatial Complexity on Finger Tree

5 Benchmark

The papers presenting the various data structures often add benchmarks to their presentation. However, it's difficult to trust their impartiality since their goal is to show how great the structure that they are presenting is. Furthermore, these structures are usually not used is the same languages, and it's difficult or

impossible to find a comparative benchmarks of the structures implemented in the same language.

Therefore, to be able to have a concrete idea of how this three data structures behave in various situation, we realized a complete set of benchmarks for each operation.

Moreover, this project was realized in competition against another team. Hence, we needed a way to compare our algorithm, and benchmarking seemed like the best way to do so.

5.1 Methodology

We created a small language to write programs that use this structures, in order to be able to run exactly the same tests on every structure. The language is very basic, it doesn't have any control structure or loops, but allows the manipulation of immutable arrays or map. A benchmark program will be a list of command. For instance, a benchmark for vectors could be:

```
    vec = create()
vec2 = push(vec, 0)
unref(vec)
vec3 = push(vec2, 1)
...
```

The bench files contain 5 sections:

- '[struct]' describe if the bench concerns Vectors or Maps.
- '[implem]' describe which datastructure (AVL, RRB, Finger) should be able to run the file (for instance, RRB won't be able to run a map benchmark).
- '[type]' describe the type of the data.
- '[init]' contains a list of command that should be executed before the actual benchmark: these are the preparation command, that aren't timed. For instance, if we write a file to benchmark the 'pop' function, we'll 'push' our data in the vector in the 'init' section.
- '[bench]' is the list of command that is actually being benchmarked.

Since we only have one implementation of Map (AVL), we didn't benchmarked them.

To execute the benchmarks, we wrote a parser that creates an Abstract Syntax Tree (AST) from a benchmark file. Then, we loop over the AST we a standard *switch/case* and execute each instruction. To obtain more precise results, we executed every benchmark 100 times, and looked at the execution time on average.

Our code was compiled and executed on macOS 10.12.3, Core i5 3,1 GHz, 16 Go RAM, MBP 2016 using 'gcc 4.4.5', with '-O3' optimizations.

5.2 Results

Every major operation on the vectors was benchmarks: ‘push‘, ‘pop‘, ‘update‘, ‘merge‘, ‘split‘. For each of them, we generated random instances of 4 different sizes (the size is the number of instructions): 10, 100, 1000 and 10000. The results follows. Note the exponential abscissa scale on the charts.

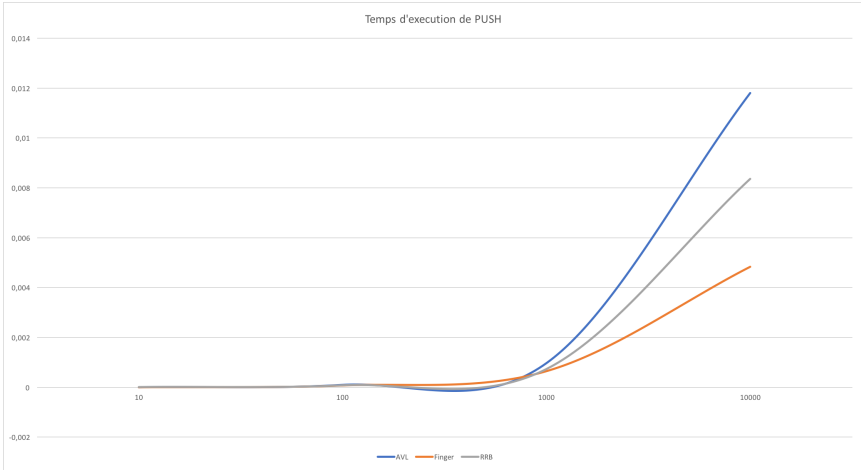

Fig. 15. Execution time for push operations on different structure sizes

For instances of a rather small size (ie. bellow 1000 instructions), the structures takes roughly the same amount of time. However, beyond the point, disparities emerged. Finger Trees are more than twice faster than AVL Vectors, and RRB Vectors are somewhere in between. However, we note that the three structure seem to have a linear complexity on this test.

For pop and update, the differences in performance are even more blatant. The Finger Trees are almost five time faster than AVL to do a ‘pop‘ or an ‘update‘. Once again, the RRB are better than the AVL: around 25% faster. The complexity of the AVL and RRB looks linear, while the complexity of the Finger Trees seems somewhat sub-linear.

‘merge‘ and ‘split‘ were only benchmarked on RRB vectors and AVL vectors. The latter are more than twice slower than the former on ‘merge‘, and about twice slower on ‘split‘. This confirms what we were expecting, since RRB vectors are optimized for ‘merge‘ and ‘pop‘. Once again, the complexity appear linear for both structures.

6 conclusion

The Finger Trees won the benchmarks on **push**, **pop** and **update** by far. Followed by the RRB Vectors, and not very far behind, the AVL Vectors. It was

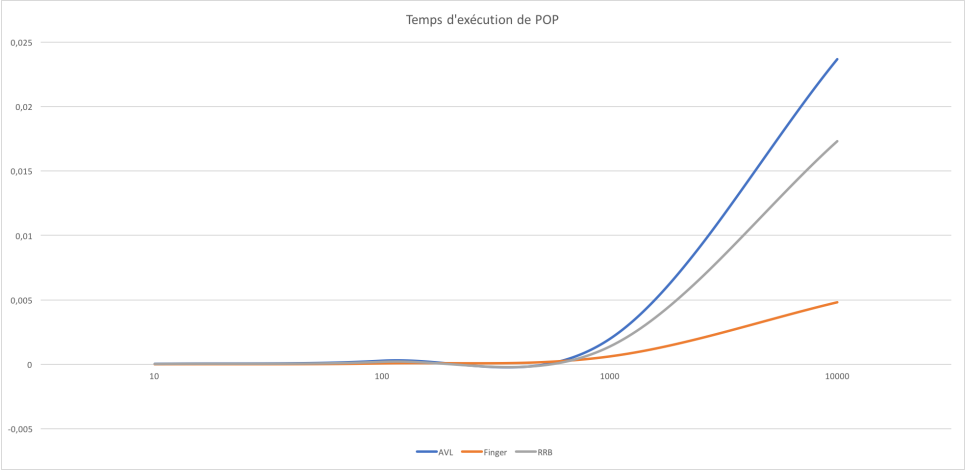

Fig. 16. Execution time for push operation on different sizes

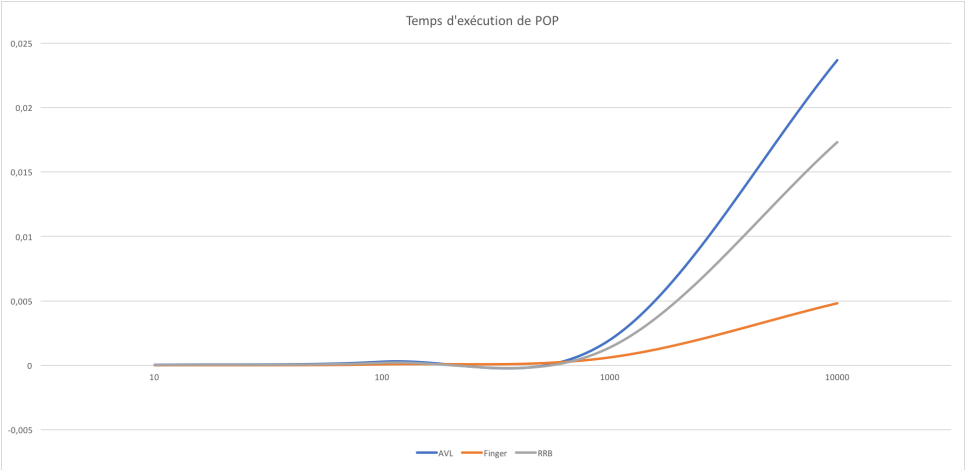

Fig. 17. Execution time for update operation on different sizes

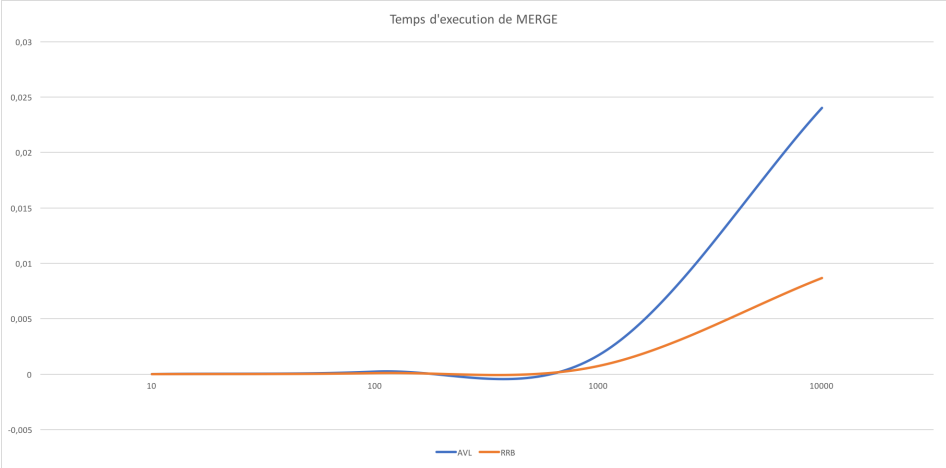

Fig. 18. Execution time for merge operation on different sizes

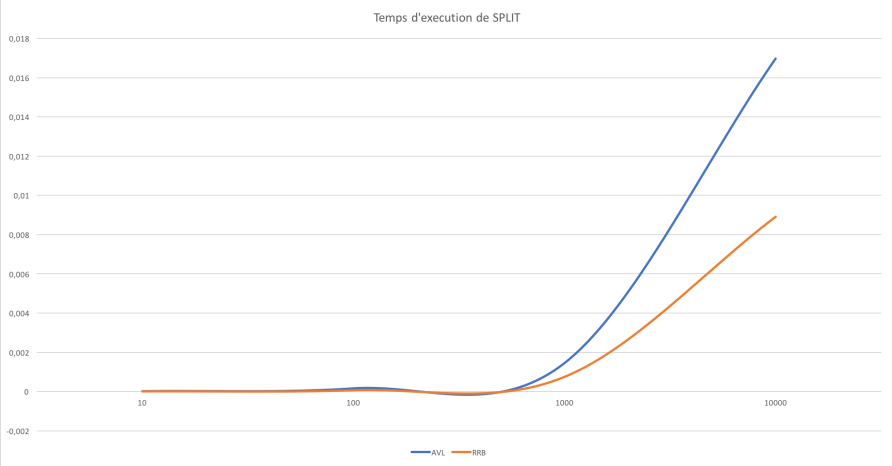

Fig. 19. Execution time for split operation on different sizes

expected that the AVL would be the slowest. Indeed, these structure wasn't designed to be used as an array, while both others were. The Finger Trees being designed to implement fast queues, their **push** and **pop** is more efficient than the RRB Vectors that are a more general structure. However, faster Finger Trees than RRB Vectors might not have been what one would have expected, since Finger Trees are mostly known to have optimized head and tail access, while RRB Vectors have a very good logarithmic access time to any element. The 'split' / 'merge' part of the benchmarks was won by the RRB Vectors. The gain of RRB over RB being an efficient split and merge, this was to be expected.

Acknowledgments. We would like to thank Frédéric Peschanski for providing feedback and specifications on Immutable data structures.

References

1. Chris Okasaki. *Purely Functional Data Structures.* Cambridge University Press, New York, NY, USA, 1998.
2. Ralf Hinze and Ross Paterson. Finger trees: A simple general-purpose data structure. *J. Funct. Program.*, 16(2):197–217, March 2006.
3. Tiark Rompf and Phil Bagwell. Rrb-trees: Efficient immutable vectors.
4. Nicolas Stucki, Tiark Rompf, Vlad Ureche, and Phil Bagwell. Rrb vector: A practical general purpose immutable sequence. *SIGPLAN Not.*, 50(9):342–354, August 2015.