`pvector` — pure Scheme persistent vector implementation

Table of Contents

1 Motivation

1.1 Why?

The lack of an efficient Clojure-like purely functional vector in the Guile Scheme standard library has always been a pain point for me. However, the idea of a bit-partitioned vector trie is very beautiful and relatively simple. So once I decided to implement it myself.

1.2 Sources of inspiration

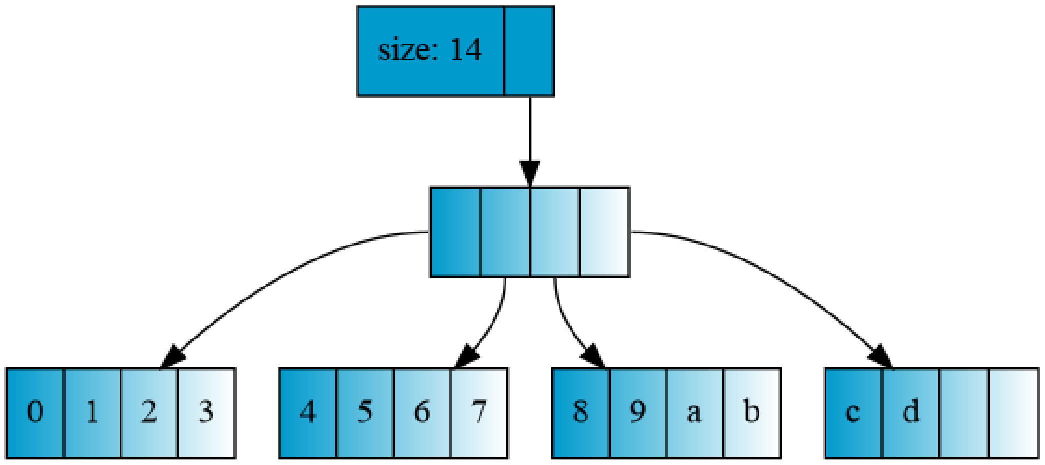
- Persistent vector implementation in Cloure by Rich Hickey: `https://github.com/clojure/clojure/blob/master/test/clojure/test_clojure/vectors.clj`
- Persistent vector implementation in Racket by Alexis King: `https://github.com/lexi-lambda/racket-pvector/tree/master`
- Blog post series «Understanding Clojure's Persistent Vectors» by Jean Niklas L'Orange:
  - `https://hypirion.com/musings/understanding-persistent-vector-pt-1`
  - `https://hypirion.com/musings/understanding-persistent-vector-pt-2`
  - `https://hypirion.com/musings/understanding-persistent-vector-pt-3`
  - `https://hypirion.com/musings/understanding-clojure-transients`
  - `https://hypirion.com/musings/persistent-vector-performance`
  - `https://hypirion.com/musings/persistent-vector-performance-summarised`
- A talk «Postmodern immutable data structures» by Juan Pedro Bolivar Puente at CppCon 2017: `https://www.youtube.com/watch?v=sPhpelUfu8Q`

2 Basic principles

2.1 Representing vector as a wide shallow tree

The simpliest way to store a vector is by using an array (reallocated quantum satis). However, this representation is incompatible with immutability: any write operation (changing an element, adding a new element, removing an element) will result in $O(n)$ steps (where n is the number of elements in the vector). So, this naïve implementation would be unacceptably slow.

The most common approach to inventing persistent data structures (popularized by Chris Okasaki in his PhD thesis[1]) is to store values in a tree and to copy (on write) only the target tree leaf and the path to that leaf. This method can also be applied to persistent vectors.



Vector of 14 elements as a shallow tree.
My reporoduction of a picture by Jean Niklas L'Orange
from `https://hypirion.com/musings/understanding-persistent-vector-pt-1`

We can store the vector as a collection of small arrays, which form the leaves of a tree. So, any single-element operation such as accessing by index, replacement, pushing, or popping will require $O(\log_m n)$ steps, where n is the vector size and m is the branching factor of the tree. Notably, these trees can be very broad and thus very shallow: while a binary tree representing a vector with 1,000,000 elements would have $\lceil \log_2(1000000) \rceil + 1 = 21$ levels, a 32-ary tree would have only $\lceil \log_{32}(1000000) \rceil + 1 = 5$ levels.

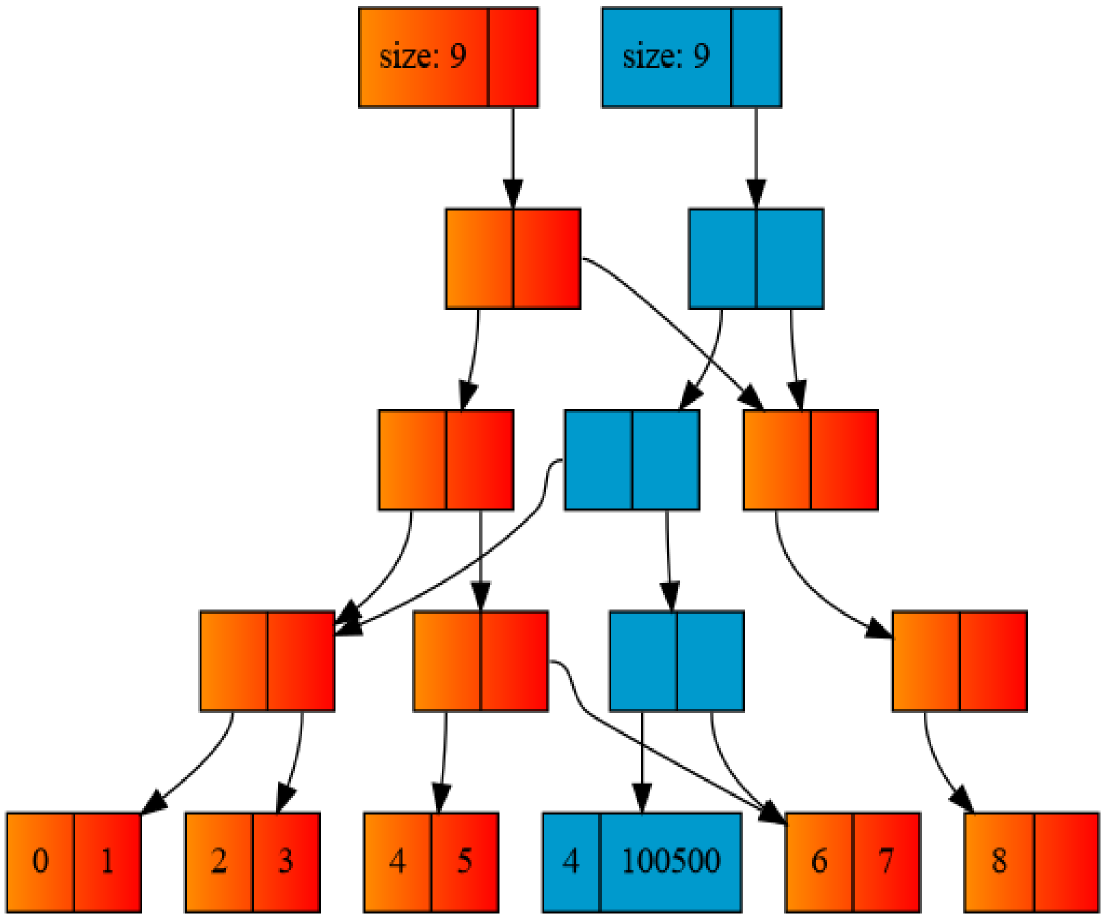
The `pvector` library uses 32-ary trees.

2.2 Implicit sharing

A purely functional structure never changes. Instead of modifying existing values, we create new versions by copying old values. This approach allows parts that are common between

[1] Purely Functional Data Structures by Chris Okasaki, Carnegie Mellon University, Pittsburgh, 1996; see also his book with the same name — Purely Functional Data Structures, Cambridge University Press, 1999

the old and new versions of the structure to be safely shared, ensuring efficient use of memory.



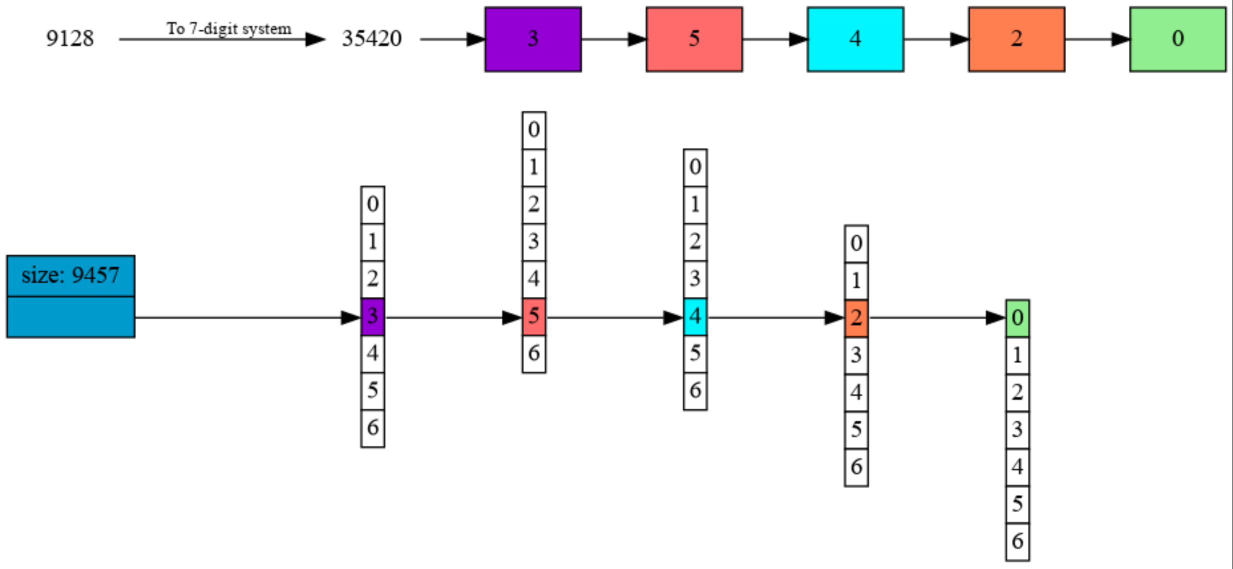
My reporoduction of a picture by Jean Niklas L'Orange
from `https://hypirion.com/musings/understanding-persistent-vector-pt-1`

In the illustration, the new vector (blue) uses orange (old) nodes for all values, not visited during the update. This approach significantly optimizes memory usage. See `https://www.youtube.com/watch?v=sPhpelUfu8Q` for an example of editing a file larger than the available memory.

2.3 Index bits as a route in a tree

We can view our tree of arrays as a *trie* — a key-value data structure where the key is represented as a path to the given node. In a tree with a branching factor of m, the index in a positional numeral system with radix m describes a path to an element. Specifically, the leftmost digit indicates the zero-based index of the root's children, the next digit determines which child node to choose at level 2, and so on, with the rightmost digit representing the target element's index within the leaf.

Imagine we store a vector with 9,457 elements in a 7-ary tree and we should get an element with the index 9,128. Our tree will have $\lceil \log_7 9457 \rceil = 5$ intermediate levels (we will choose the path in the tree 5 times). To navigate through this structure, we need to transorm our index to the 7-digit system: $9457 = 35420_7$.



My reporoduction of a picture by Jean Niklas L'Orange
from `https://hypirion.com/musings/understanding-persistent-vector-pt-2`

The base-7 representation can now guide our path through the tree. At any level i (enumerated from the *leaves* level, zero-based) we can obtain the i-th digit by performing integer division by 7^i to remove the least significant digits, then taking the reminder from the division by 7 to get current digit. This digit will indicate the target branch at the current level.

$\lfloor 35420_7/(7_{10})^4 \rfloor = \lfloor 35420_7/10000_7 \rfloor = 3_7$, no need to get a reminder,

$\lfloor 35420_7/(7_{10})^3 \rfloor = \lfloor 35420_7/1000_7 \rfloor = 35_7,\ 35_7 \,\%\, 7_{10} = 5_7$

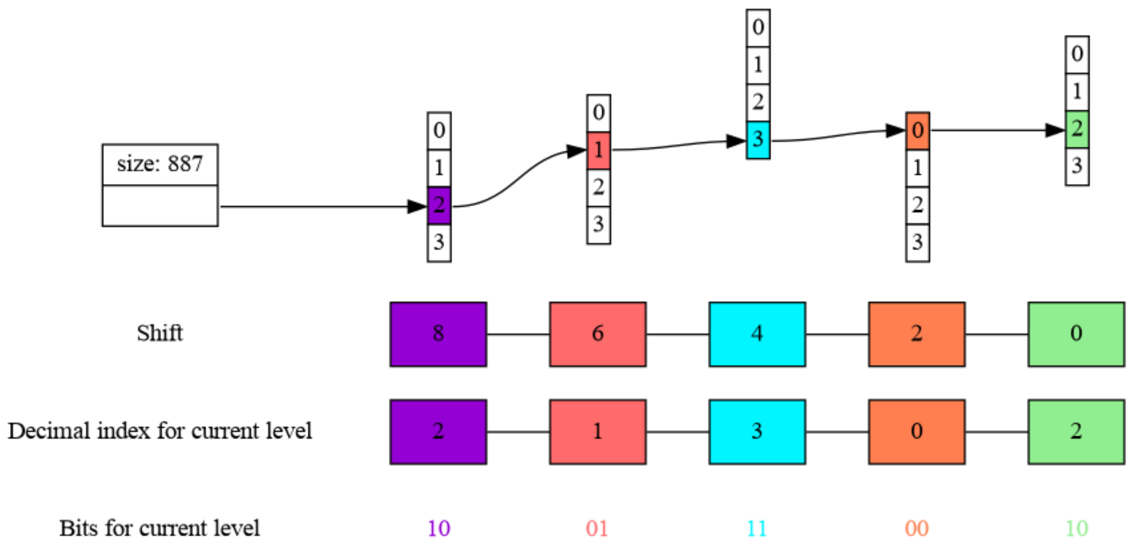
$\lfloor 35420_7/(7_{10})^2 \rfloor = \lfloor 35420_7/100_7 \rfloor = 354_7,\ 354_7 \,\%\, 7_{10} = 4_7$

$\lfloor 35420_7/(7_{10})^1 \rfloor = \lfloor 35420_7/10_7 \rfloor = 3542_7,\ 3542_7 \,\%\, 7_{10} = 2_7$

$\lfloor 35420_7/(7_{10})^0 \rfloor = \lfloor 35420_7/1 \rfloor = 35420_7,\ 35420_7 \,\%\, 7_{10} = 0_7$

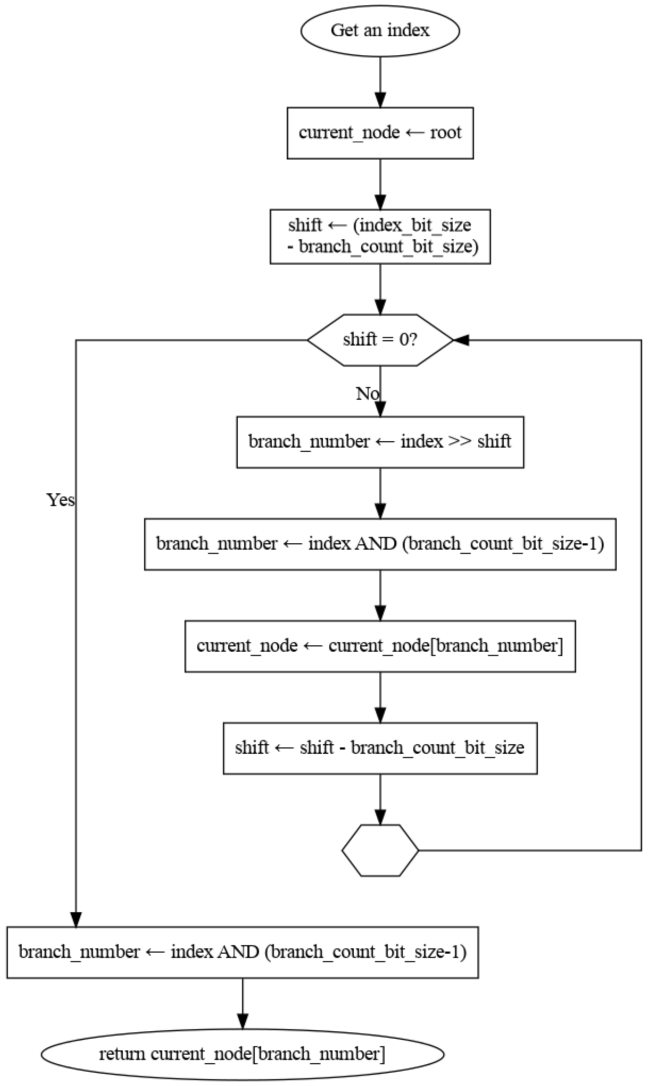
The only drawback is that integer division and modulo operations are relatively slow.

However, if the branching factor is a power of two, these operations can be optimized. We can perform the integer division using a bit shift by the bit size of the branching factor, and the reminder can be calculated using a bitwise AND with a number that is one less than the branching factor (essentially, a number composed of all bit 1-s in all significant bits). These bitwise operations are very fast on modern computers.



My reporoduction of a picture by Jean Niklas L'Orange
from `https://hypirion.com/musings/understanding-persistent-vector-pt-2`

Thus, the search within the tree can be implemented using the following algorithm (for simplicity's sake, let's consider an imperative version):


Get an index
current_node ← root
shift ← (index_bit_size - branch_count_bit_size)
shift = 0?
No
Yes
branch_number ← index >> shift
branch_number ← index AND (branch_count_bit_size-1)
current_node ← current_node[branch_number]
shift ← shift - branch_count_bit_size
branch_number ← index AND (branch_count_bit_size-1)
return current_node[branch_number]

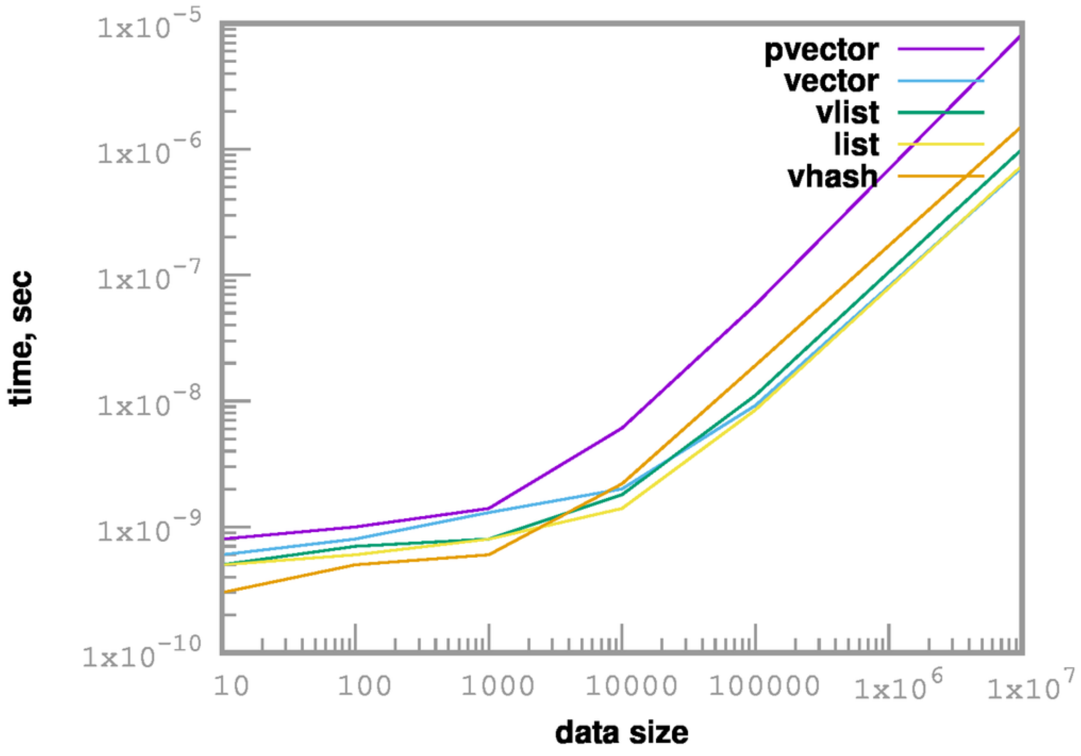
3 Optimizations

3.1 Elimination of n-times element lookup in full-vector operations (`fold`, `foldi`, `map`)

Each index lookup takes $O(\log_m n)$ steps, where n is the size of the vector and m is the branching factor.

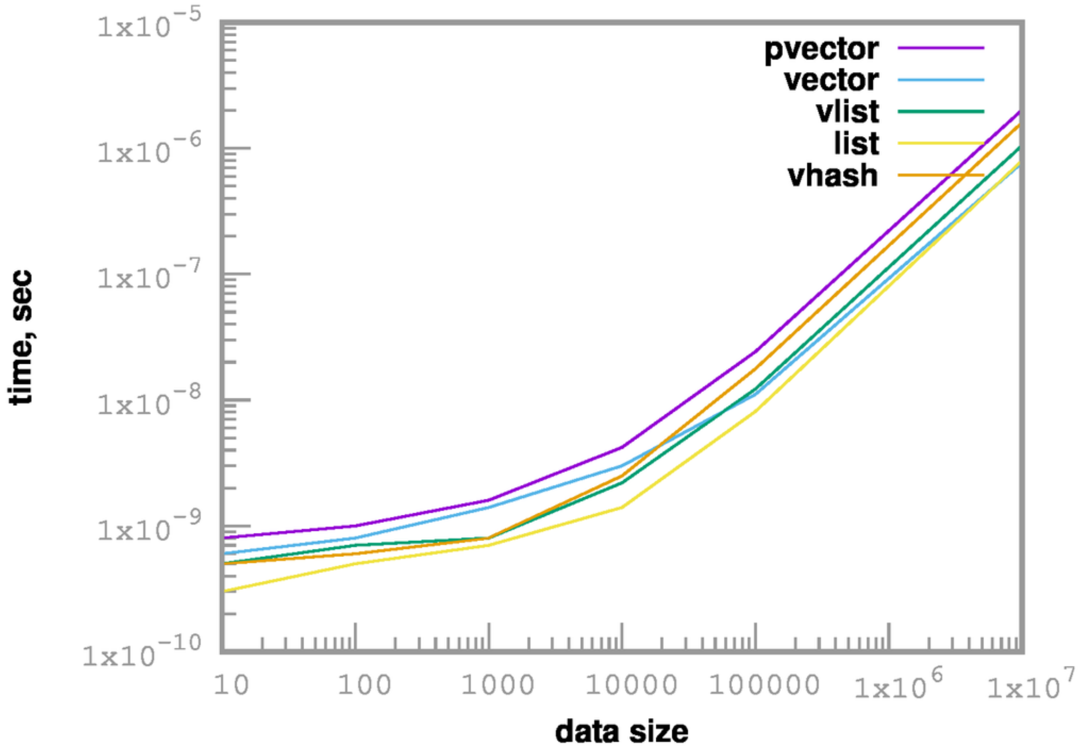
However, for operations involving the entire vector, such as folds, mapping, and searches, we don't need to access each element by index. Instead, we can process entire leaves at once.

According to Jean Niklas L'Orange's master thesis[1] the lookup of each leaf (32-element array in our case) in the tree has an amortised complexity of $O(1)$. Consequently, the entire operation has a complexity of $O(n)$, rather than the $O(n \log n)$ complexity of a naïve solution.

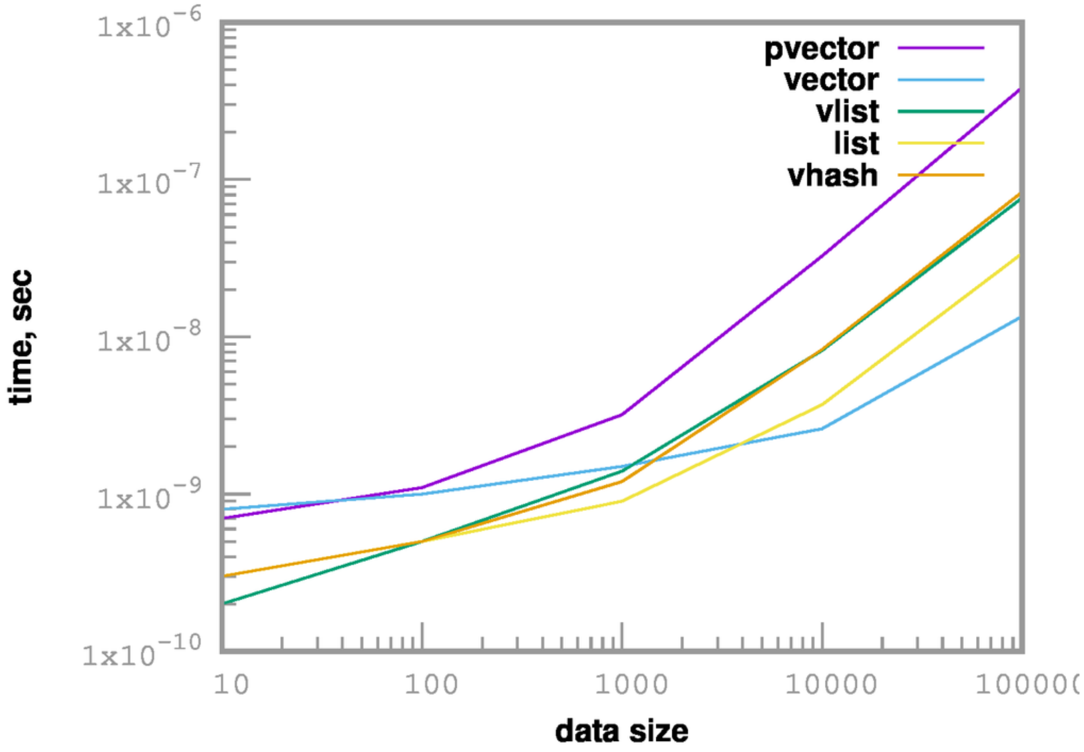


Performance of `fold` function before optimization.

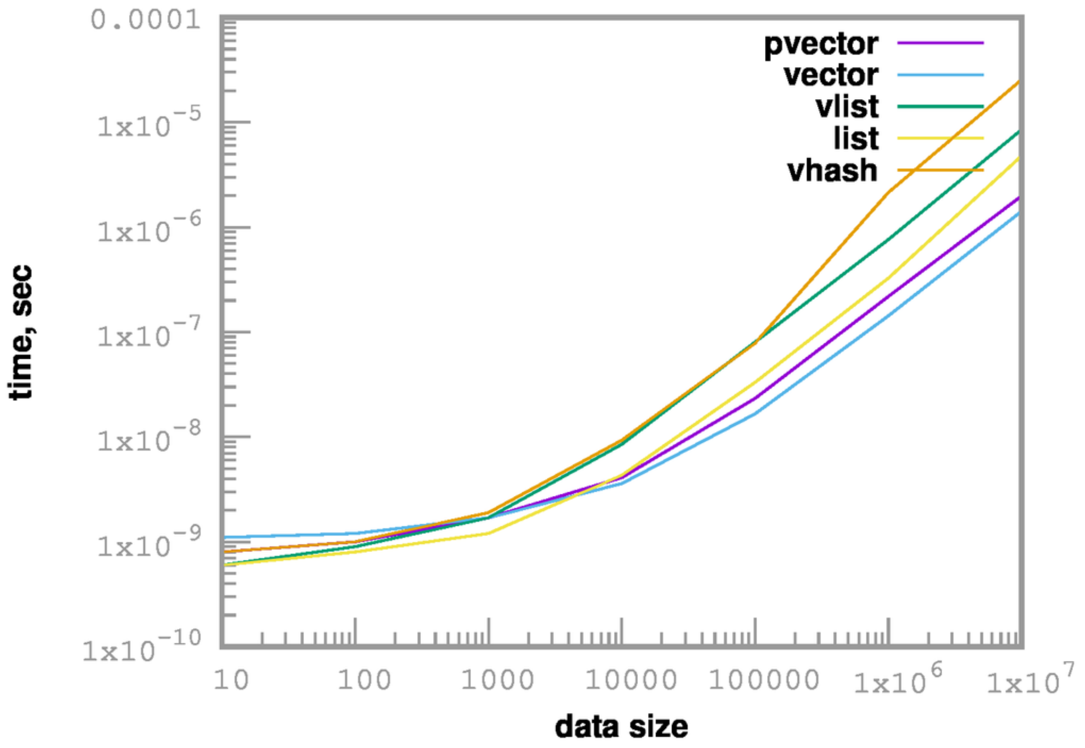
[1] Improving RRB-Tree Performance through Transience by Jean Niklas L'Orange, Norwegian University of Science and Technology, 2014, pp.21–22, Theorem 2.5 (`https://hypirion.com/thesis.pdf`)



Performance of `fold` *function after optimization.*



Performance of `map` *function before optimization.*

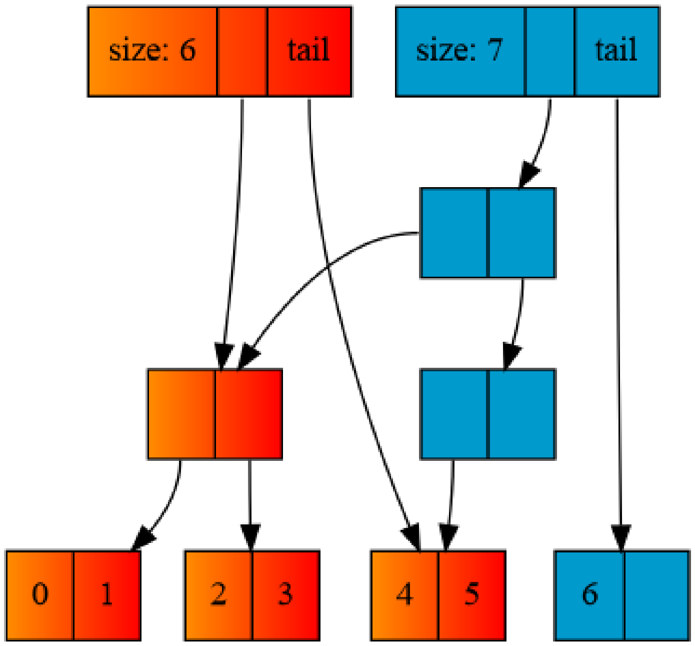


Performance of `map` *function after optimization.*

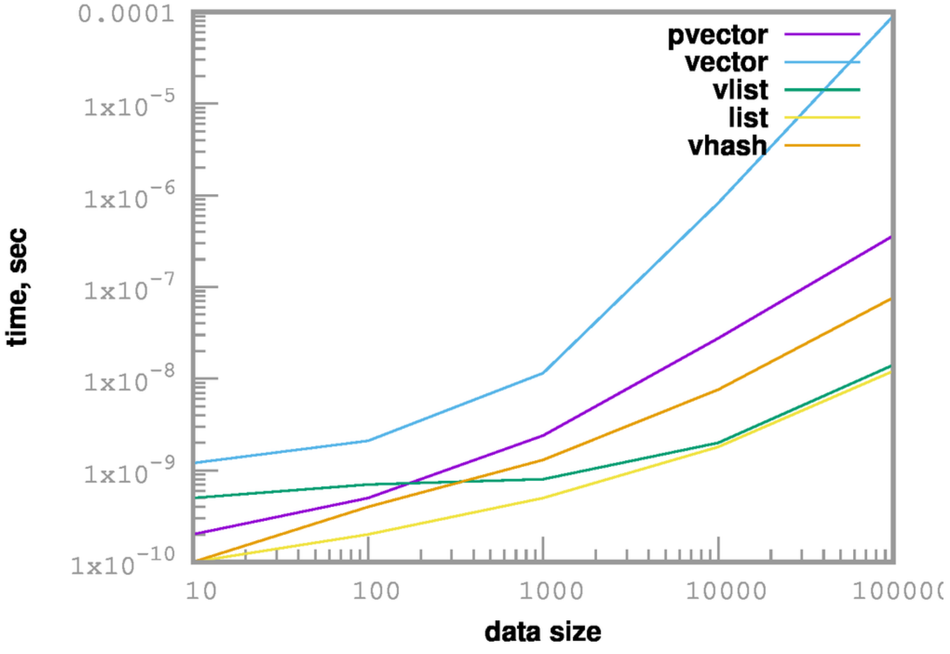
3.2 Shortcut to tail

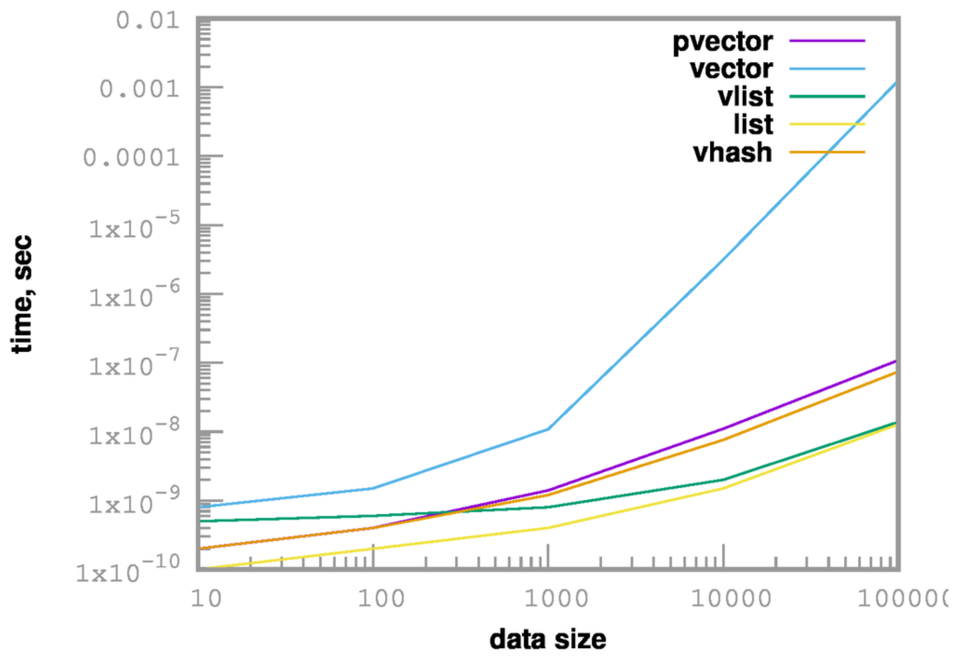
When a program intencivelly pushes elements into the vector, it repeatedly needs to look up the last (rightmost) leaf in the tree. This process can be optimized by storing a reference to the last segement of the vector at the head of the data structure.

If the tail is full, we insert it into a tree as a last (rightmost) leaf. Conversely, if the tail becomes empty after several deletions, we remove the last leaf from the tree and make it a new tail.



Below, you can see the performance of new element insertions before and after the optimization.




pvector
vector
vlist
list
vhash
time, sec
0.01
0.001
0.0001
1x10^{-5}
1x10^{-6}
1x10^{-7}
1x10^{-8}
1x10^{-9}
1x10^{-10}
10
100
1000
10000
10000
data size

4 API

The following is the list of modules provided by this library.

4.1 (pvector)

pvector is a persistent vector library for Guile Scheme

This module provides the following interface.

4.1.1 Macros

`pvector-length` [Macro]

`#<syntax-transformer pvector-length>`

`pvector?` [Macro]

`#<syntax-transformer pvector?>`

4.1.2 Procedures

`list->pvector` *l* [Procedure]

(list->pvector l) -> pvector

Converts elements of list to pvector

`make-pvector` [Procedure]

(make-pvector) -> pvector

Creates empty persistent vector

`pvector` *. l* [Procedure]

(pvector l) -> pvector

Creates persistent vector from argument list `l`

`pvector->list` *pv* [Procedure]

(pvector->list pv) -> list

Converts values of pvector to list

`pvector-append` *pv other* [Procedure]

(pvector-append pv other) -> vector

Adds of element of persistent vector *other* to the end of the persistent vector *pv*

`pvector-cons` *v pv* [Procedure]

(pvector-cons v pv) -> pvector

Adds *v* to the end of persistent vector *pv*

`pvector-drop-last` *pv* [Procedure]

Returns a copy of persistent vector *pv*, but without a last element

`pvector-empty?` *pv* [Procedure]

(pvector-empty? pv) -> bool

Checks if `pv` is empty

`pvector-fold` *f acc pv* [Procedure]

(pvector-fold f acc pv) `->` pvector

Accepts function `f`, accumulator `acc` and pvector `pv`. Function `f` accepts the element of `pvector` and an accumulator value and returns a new accumulator value. `pvector-fold` returns a result of a sequential application of `f` to all the values of `pvector`, with accumulating intermediate results in `acc`

`pvector-foldi` *f acc pv* [Procedure]

(pvector-fold f acc pv) `->` pvector

Accepts function `f`, accumulator `acc` and pvector `pv`. Function `f` accepts an index of the current element, the current element of `pvector` and an accumulator value and returns a new accumulator value. `pvector-fold` returns a result of a sequential application of `f` to all the values of `pvector`, with accumulating intermediate results in `acc`

`pvector-map` *f pv* [Procedure]

(pvector-map f pv) `->` pvector

Apply `f` to all values of `pv`. Accepts function `f` and pvector `pv`. Function `f` accepts the element of `pv` and returns the respective value for the new `pvector`.

`pvector-push` *pv v* [Procedure]

(pvector-push pv v) `->` pvector

Adds *v* to the end of persistent vector *pv*

`pvector-ref` *pv index* [Procedure]

(pvector-ref pv index) `->` value

Returns element of persistent vector *pv* with an index *index*

`pvector-set` *pv index v* [Procedure]

(pvector-set pv index v) `->` pvector

Returns a new pvector, like *pv*, but with a *v* at index *index*