Implementing Persistent $O(1)$ Stacks and Queues in R

by Shawn T. O'Neil

Abstract True to their functional roots, most R functions are side-effect-free, and users expect datatypes to be persistent. However, these semantics complicate the creation of efficient and dynamic data structures. Here, we describe the implementation of stack and queue data structures satisfying these conditions in R, available in the CRAN package **rstackdeque**. Guided by important work in purely functional languages, we look at both partially- and fully-persistent versions of queues, comparing their performance characteristics. Finally, we illustrate the usefulness of such dynamic structures with examples of generating and solving mazes.

Introduction

Dynamic data structures such as trees, hash tables, heaps, stacks and queues are fundamental to a wide variety of algorithmic strategies, especially those that are recursive in nature (Cormen et al., 2001). While trees and hash tables are often used for simple name/value lookups—supported well by environments and named lists—stacks (first in, last out) and queues (first in, first out) are used precisely for their efficient insertion and removal operations. While these structures can be implemented via pre-allocated vectors and lists when combined with additional start/end index information, this requires foreknowledge of their maximum size (or a dynamic resizing strategy) as well as careful bookkeeping.

Most implementations of stacks and queues are mutable; they support `push()` and `pop()` operations that have the *side effects* of modifying or mutating the data structure in place for insertion and removal. Although R does support the development of mutating data structures, most common structures do not mutate (without, at least, an explicit assignment) and developers are encouraged to follow this paradigm and create side-effect-free functions. In addition to providing a consistent programming environment, this admonition also facilitates the creation of easily parallelized code (Böhringer, 2013).

More formally, we say that a data structure is "persistent," if, after an insertion or removal the original version is still accessible. To the programmer, it appears as if a modified copy of the structure was returned. Such persistence provides the side-effect-free nature of accessor functions expected by users of functional languages. A structure is "fully persistent" if previous versions support further insertions and removals, and "partially persistent" if previous versions may only be inspected.

Fortunately, the implementation of persistent data structures in purely functional languages is well studied and continues as an active area of research (Chuang and Goldberg, 1993; Kaplan, 1995; Okasaki, 1999). For languages that support certain features there now exist persistent balanced binary trees, queues, stacks, heaps (Okasaki, 1999), union-find structures (Conchon and Filliâtre, 2007), and many others as fast or nearly as fast as their mutating counterparts. The solutions are often deeply recursive and take advantage of tail-call optimization and delayed-evaluation provided by languages such as Haskell and ML.

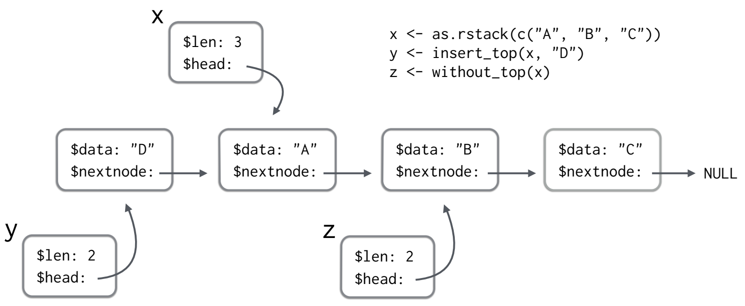
As a quick review, a stack is a first-in, last-out structure: elements are removed in the opposite order they are inserted. Usually, we think of insertions and removals occurring at the "top" of a stack. Stacks are used in a number of programming paradigms, especially depth-first searches of state spaces for optimization and search problems. Queues are first-in, first-out: elements are removed in the same order in which they are inserted, and we usually imagine inserting at the "back" and removing from the "front" of a queue. Deques, or double-ended queues, additionally support insertion and removal from both ends. Queues and deques are used for breadth-first searches of state spaces and also find use in simulation contexts (e.g. Grether et al. 2012). All three structures generally also support "peeking," or inspection of elements that are ready for removal.

Ideally, insertion, removal, and peeking on stacks, queues, and deques would be nearly instantaneous no matter their size ($O(1)$ worst-case time). Mutable versions of these structures are easy to implement and provide this speed, but persistence complicates matters. In practice, *amortized* $O(1)$ operations are an excellent compromise; a few individual operations may be slow ($O(n)$ or $O(\log n)$), but n insertions and/or removals are guaranteed to run in $O(n)$ worst-case time overall, for an $O(1)$ time-per-operation on average. Here, we would like to additionally use a new term: "slowly persistent," to describe a structure that is fast when used partially persistently, but may be slower when used in a fully-persistent context.

In the following sections, we describe the implementation of $O(1)$ worst-case, fully persistent stacks in R (which we call 'rstack's), slowly persistent $O(1)$ amortized deques ('rdeque's), and finally $O(1)$ worst-case fully persistent queues ('rpqueue's) as described by Okasaki (1995).

Fully persistent stacks

For languages that disallow mutability entirely, even stack implementations can be moderately complex. In R, however, we can take advantage of mutability via environments and simply provide side-effect-free interface functions. We implement stacks as unordered linked-lists of environments, each with a data element and a reference to the next node. Stacks themselves are S3 objects with interface methods and access to the head stack node as well as length information stored separately.



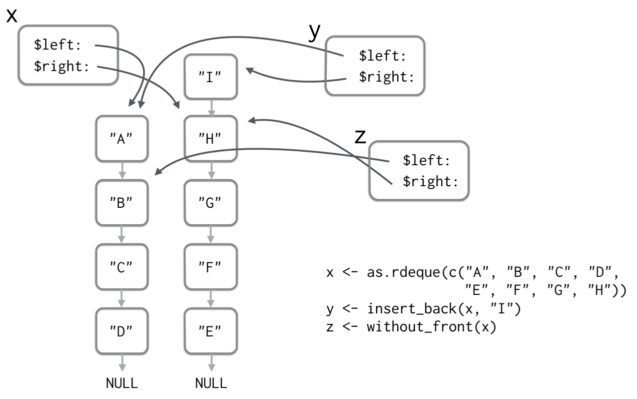
```
x <- as.rstack(c("A", "B", "C"))
y <- insert_top(x, "D")
z <- without_top(x)
```

Figure 1: Illustration of a fully persistent stack as a linked-list.

To emphasize the persistent nature of the structure, rather than implement the traditional `push()` and `pop()` methods, we instead use `insert_top()` (returning a new 'rstack' object with a new top element), `without_top()` (returning a new 'rstack' object without the top element) and `peek_top()` (returning the data element of the top of the stack, or an error if the stack is empty–though we also implement `empty()` for explicit tests of emptyness). As illustrated in Figure 1, insertion and removal are implemented as creation of a new stack with head element referencing the appropriate node, for $O(1)$ worst-case time for all operations as well as full persistence.

Slowly persistent queues and deques

The simplest way to implement persistent queues has long been known (Burton, 1982), and simply keeps two stacks: a "left" that holds the elements at the front of the queue ready for removal, and a "right" holding the elements at the back of the queue in reverse order (where new elements are added).



```
x <- as.rdeque(c("A", "B", "C", "D",
                 "E", "F", "G", "H"))
y <- insert_back(x, "I")
z <- without_front(x)
```

Figure 2: Illustration of a slowly persistent queue as a pair of stacks.

Of course, we need to be able to handle removing from a non-empty queue when the `left` stack is empty. In this case we need to first reverse the `right` stack and put it in the `left` position. This is an $O(n)$ operation, but one whose average expense is inversely proportional to its frequency in any series of insertions/deletions, for $O(1)$ amortized time provided the queue is used only partially persistently (see Okasaki 1999 for a rigorous proof and thorough discussion of amortization).

In actuality, we implement this structure as a double ended queue for R (an 'rdeque') which allows for fast access to both ends. When one side becomes nearly empty, returned structures are first decomposed to a list and recomposed with balanced sides, again providing $O(1)$ amortized time when used partially persistently. They are still available for persistent use, but in this case amortized $O(1)$ time cannot be guaranteed. For example, if `y <- without_front(x)` causes an $O(n)$ rebalance, then so will all similar calls such as `z <- without_front(x)`. In practice, we expect usage patterns that cause significant slowdown to be rare.

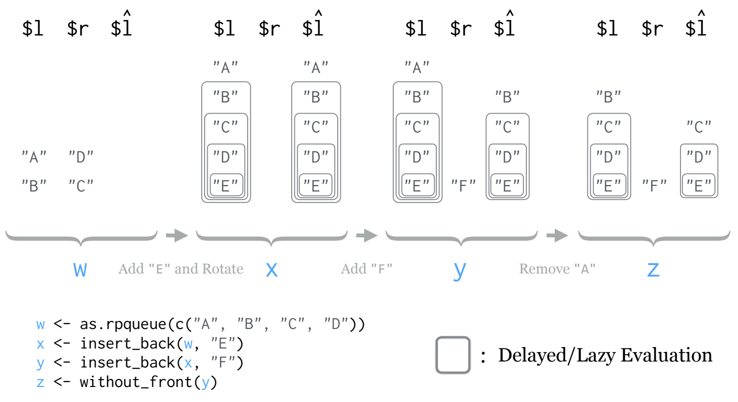
Fast fully persistent queues

True fully persistent, $O(1)$ worst-case queues are also known, but these solutions rely heavily on recursively defined operations and delayed evaluation (Okasaki, 1995). The building block for many functional data structures is the lazy list, a structure whose first element is immediately accessible, but whose "tail" (the rest of the elements) are evaluated in a delayed fashion. Interestingly, this allows for the recursive definition of infinite lists such as the Fibonacci sequence, where successive elements are syntactically defined but only evaluated when accessed (Paulson, 1996).

We implement these persistent queues as 'rpqueue''s, which require three lazy lists: `l`, `r`, and $\hat{\texttt{l}}$. Items are inserted into the back of the queue by addition to the top of the `r` list, and removed from the front by removal from the front of the `l` list. We implement these lazy lists as 'rstack''s (described above) where nodes' `nextnode` elements are assigned with `delayedAssign()` and memoized on first evaluation.

The queue maintains the invariant that the `l` stack is at least as long as the `r` stack–when this invariant would be violated (either by an insertion to `r` or a removal from `l`), the `r` list is rotated and appended to the tail of `l`; this operation is delayed for $O(1)$ insertions and removals. Simultaneously, the $\hat{\texttt{l}}$ list is set to contain the same data as `l` on a rotation. For each subsequent insertion or removal (which don't cause a rotation and are also constant time), an element is also removed from $\hat{\texttt{l}}$. This ensures that future removals are constant time as previously delayed computations are forced even on insertions. (The rotation is incremental, and so all tails of `l` and $\hat{\texttt{l}}$ are delayed.) With these operations, the $\hat{\texttt{l}}$ queue is empty exactly when the invariant is violated causing another delayed rotation. For further details, see Okasaki (1995).

An example operation is shown in Figure 3, where an initial queue of `"A"`, `"B"`, `"C"`, `"D"` receives an insertion of `"E"`, causing a rotation. Next, an insertion of `"F"` does not cause a rotation but does cause a removal from $\hat{\texttt{l}}$, pre-evaluating the answer required for the upcoming removal of `"A"`. When the removal of `"A"` does occur, the operation is $O(1)$ and another element is removed from $\hat{\texttt{l}}$ anticipating future removals from `l`.



```
w <- as.rpqueue(c("A", "B", "C", "D"))
x <- insert_back(w, "E")
y <- insert_back(x, "F")
z <- without_front(y)
```

Figure 3: Example of $O(1)$ worst-case 'rpqueue' operation, based on three stacks (`l`, `r`, and $\hat{\texttt{l}}$) where the "tails" of stacks may be assigned with `delayedAssign()` in the incremental rotation and are memoized upon first evaluation. Items are inserted onto `r` and removed from `l`. The $\hat{\texttt{l}}$ stack is reduced by one on each insertion or removal, ensuring that future versions of `l` have been pre-evaluated for constant time removals.

Because lazy lists are potentially deeply nested, we can't fully evaluate them recursively due to interpreter call-stack limitations. In a similar vein, an important property of these purely-functional queues from an implementation perspective is that they are indeed $O(1)$ worst-case. Thus, the evaluation cascade that might result from delayed assignment is never deeper than $O(1)$ for any insertion or removal, and we needn't worry about call-stack overflows.

Features

The three data structures we've described, 'rstack', 'rdeque', and 'rpqueue' are all available in the R package, **rstackdeque**, on CRAN. All provide an `as.list` convenience function that collates all elements into a pre-allocated list of the correct size with the top/front elements being placed in the first index of the list. Similarly, `as.rstack()`, `as.rdeque()`, and `as.rpqueue()` convert their inputs into a stack, deque, or fully persistent queue, after converting to an intermediary list representation with `as.list()`. These bulk operations are somewhat slower for 'rpqueue's because of their delayed nature and the need to process modifications incrementally, in addition to the supporting structures needed for their implementation. All three structures also support modification by direct assignment to their tops or ends, as in `peek_top(x) <- "A"` for stacks, `peek_front(x) <- "A"` for queues, and additionally `peek_back(x) <- "A"` for deques.

While these examples have illustrated storing simple character vectors in stacks and queues, any R data type may be stored efficiently. One of the more common errors for new R programmers with experience in non-functional languages is the attempt to use a loop to dynamically grow a data structure like a dataframe. As experienced R programmers know, the persistent nature of dataframes but lack of efficient growth capabilities may result in excessive copying and $O(n^2)$ behavior. By contrast, with $O(1)$ insertion provided by stacks and queues, the programmer can easily and quickly insert single-row dataframes (or lists representing rows) and later extract and collate them into a full dataframe in $O(n)$ total time. The structures described here thus also come with a convenience function `as.data.frame()` that efficiently collates elements, so long as they all have the same `length()` and if any are named the names do not conflict. (This is accomplished with a call to `rbind` via `do.call` on a list-representation of the structure.) We emphasize that this feature is not meant as a replacement for the functional solutions provided in base R like `lapply()` or excellent split-apply-combine strategies such as in the **dplyr** package (Wickham and Francois, 2015). Rather, we hope that stacks and queues will provide R programmers with highly dynamic data structures that are useful for certain applications while fitting well with the existing R ecosystem.

Performance comparison

We briefly wish to compare the performance characteristics of these three data structure implementations. Although certainly much slower than less dynamic pre-allocated-list solutions, persistent stacks, deques, and queues should be efficient enough to be used naturally when they are most appropriate. In addition, performance comparisons between the two implementations of queues described above—partially persistent and fully persistent based on lazy evaluation—would be the first to our knowledge.

Using the **microbenchmark** package (Mersmann, 2014), we tested the time taken to 1) insert n items, 2) remove n items, and 3) perform a mix of n insertions and deletions, where a deletion followed every two insertions. Results can be found in Figure 4.

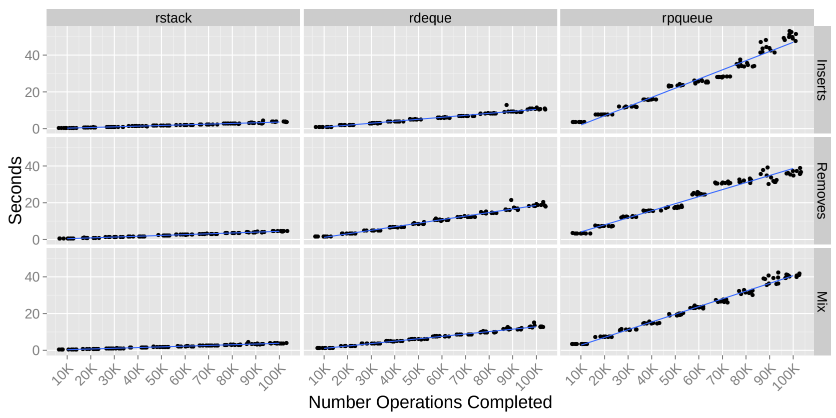


Figure 4: Benchmark timings for n insertions, n removals, and a mix of n insertions and removals for 'rstack's, 'rdeque's, and 'rpqueue's. Each point represents one trial, with jitter and linear model fits added for clarity. The close fit of the linear models to the points reinforces the $O(1)$ amortized time for all three structures in these tests.

Generally, 'rdeque' structures are about twice as slow as 'rstack's on insert and four times slower on remove, reflecting the additional environment references on insert and occasional rebalancing needed for removals. The 'rpqueue's are slower still (by a constant factor per element) owing to their complex insertion and removal routines.

Because of the supporting structures needed for 'rdeque's and 'rpqueue's, we also looked at how many environments were created by n insertions as reported by `memory.profile()` immediately after n insertions. (Before each set of insertions, the structures were removed with `rm()` and the garbage collector was run with `gc()`.) Results are shown in Figure 5. As expected, the partially persistent stacks and deques use (asymptotically) one environment per data element. The number of environments utilized by the 'rpqueue's varies depending on the internal evaluation state of the structure, with at some points several environments supporting each data element. Note that because the insertion time is $O(1)$ worst-case, the total number of environments must also be $O(n)$ after n insertions.

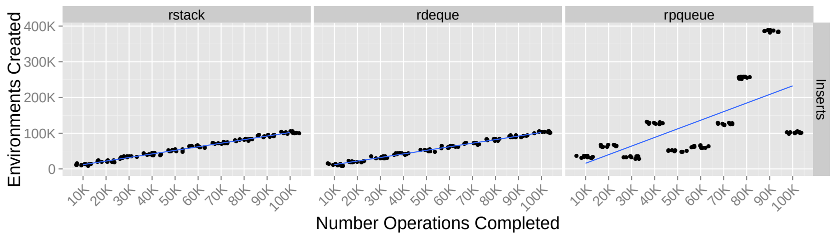


Figure 5: Benchmark environment counts after n insertions for 'rstack's, 'rdeque's, and 'rpqueue's. Each point represents one trial, with jitter and linear model fits added for clarity.

Example usage

To highlight the usefulness of stacks and queues as well as give more in-depth examples, we consider a whimsical application that makes heavy use of these structures: designing and solving mazes.

There are a variety of ways to design a maze; one of the more elegant is known as "recursive backtracking." This strategy builds corridors by extending them from the start in random directions, inserting the cells as they are visited onto the top of a stack. When the corridor would intersect itself, the path is traced backward (removing from the stack) and another corridor is opened up. A small ASCII maze built with this method is shown in Figure 6.

```
#######################################################
#######################################################
##E#           #         #                       #   ##
## # ######### # ##### # ##### ################# # # ##
## #   # #     # #     #       #   #         # # # # ##
## ### # # ####### ############# ### ##### # # # # # ##
##     # #       # #     #   #       #     #   # # # ##
######## ####### # # ### # # ### ##### ####### # ### ##
##       #       #     #   #   #   #   #   #   #   # ##
## ####### ################### ##### ### ### ##### # ##
##         #   #       #       #     #   #   #     # ##
## ######### # # # ### # ####### ##### ### ### ##### ##
##       #   # # # #   # #   #   #     #     # #   # ##
######## ### # # # # ### # ### # ##### # ##### # # # ##
## #   #     # # # #     # #   # #   #   # #   # #   ##
## # # ####### ### ####### # ### # # # ### # ### ### ##
##   #       #   # #         # #   # #     # #   # # ##
## ####### ##### # # ######### ##### ##### # # ### # ##
## #     # #   # # #         #   #   #   # # #   # # ##
## # ### # # # # # ######### ### # ### # ### ### # # ##
##   #   #   #     #             #     #         #  S##
#######################################################
#######################################################
```

Figure 6: Example ASCII maze generated via recursive backtracking supported by a stack.

The R code to generate an ASCII maze like this is somewhat tedious so we won't reproduce it here except in pseudocode (though see Section Availability):

```
## Maze generation pseudocode
V <- rstack()
set the current location L to the start of the maze at S
while(any cell is unvisited) {
  find a random unvisited neighbor N of L
  if(N is not NULL) {
    V <- insert_top(V, L)
    build a corridor between N and L
    L <- N
    mark L visited
  else if(the stack is not empty)
    # pull the next location to try from stack
    L <- peek_top(V)
```

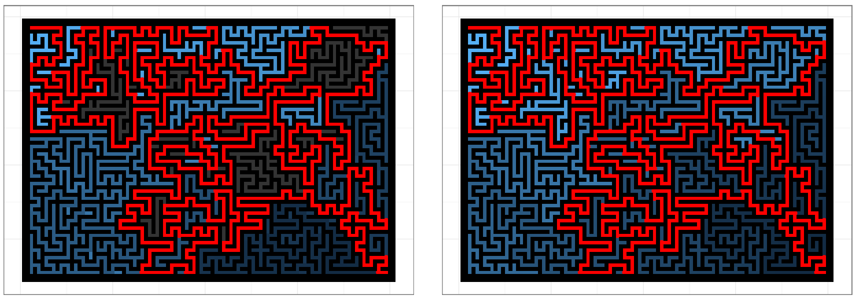


Figure 7: Maze solved with depth-first search supported by a stack (left) and breadth-first search supported by a queue (right); lighter colored cells were visited later by the search and the solution (from the start in the lower right to the end in the upper left) is shown in red.

```
    V <- without_top(V)
  else
    set L to a random unvisited cell in the maze
```

Solving mazes of this type using depth-first search is relatively simple with the help of a stack. The start location is inserted into the top of the stack, and so long as peeking at the top of the stack doesn't reveal the end location, neighboring unvisited locations are inserted. If there are no unvisited locations nearby, indicating a dead end, the top of the stack is removed instead until the search can continue. (This function calls a helper function, `random_unvisited_neighbor`, see Section Availability for details.)

```
solve_ascii_maze_dfs <- function(maze) {
  ## find the start and end of the maze
  end <- data.frame(which(maze == "E", arr.ind = TRUE))
  start <- data.frame(which(maze == "S", arr.ind = TRUE))

  maze[maze == " " | maze == "E" | maze == "S"] <- "." ## mark corridors unvisited

  loc <- start ## initialize the solution stack and start location
  path <- rstack()
  path_history <- rstack()
  path <- insert_top(path, loc)

  step <- 1
  while(any(peek_top(path) != end)) { ## while we're not out of the maze
    loc <- peek_top(path)

    ## if the current location is unvisited, mark it visited with the current
    ## timestep
    if(maze[loc$row, loc$col] == ".") {maze[loc$row, loc$col] <- step}

    ## grab a random unvisited neighbor; if there is one push it on the current
    ## path
    nextloc <- random_unvisited_neighbor(maze, peek_top(path), dist = 1)
    if(!is.null(nextloc)) {
      path <- insert_top(path, nextloc)
    } else {
      # otherwise backtrack and try again
      path <- without_top(path)
    }
    step <- step + 1
    path_history <- insert_top(path_history, path)
  }

  end <- peek_top(path) ## mark solution by negating visit numbers
  maze[end$row, end$col] <- step
  while(!empty(path)) {
    loc <- peek_top(path)
    path <- without_top(path)
    maze[loc$row, loc$col] <- -1 * as.numeric(maze[loc$row, loc$col])
  }
  return(list(maze, path_history))
}
```

In this implementation we mark each cell as visited with the timestep in which it is visited, and in the end we reverse those numbers for the solution by tracing back through the solution path stored in the stack. An auxiliary function plots the solved ASCII maze, including this solution path, resulting in Figure 7 (left) for a larger maze.

A breadth-first search replaces the stack with a queue, so that new cells are visited in order of their distance from the start—cells further from the start are added to the back of the queue for later visiting and cells are marked visited from the front of the queue. One small complication is the need to keep track of where cells are sampled from relative to their neighbors so that we can trace back the eventual solution. We do this with a simple stack that keeps track of this to/from information and use a while-loop for the traceback.

```
solve_ascii_maze_bfs <- function(maze) {
  ## find the start and end of the maze
  end <- data.frame(which(maze == "E", arr.ind = TRUE))
  start <- data.frame(which(maze == "S", arr.ind = TRUE))

  maze[maze == " " | maze == "E" | maze == "S"] <- "." ## mark corridors unvisited

  loc <- start ## initialize the solution stack and start location
  visits <- rdeque()
  visits_history <- rstack()
  visits <- insert_back(visits, loc)

  ## keep a stack to remember where each visit came from in the BFS
  camefrom <- rstack()

  step <- 1
  while(!empty(visits)) { ## while there are still cells we can visit
    loc <- peek_front(visits)
    visits <- without_front(visits)

    neighbors <- random_unvisited_neighbor(maze, loc, dist = 1, all = TRUE)
    for(neighbor in neighbors) {
      camefrom <- insert_top(camefrom, list(from = loc, to = neighbor))

      ## push neighbors on the queue and mark them visited with the timestep
      visits <- insert_back(visits, neighbor)
      maze[neighbor$row, neighbor$col] <- step
      step <- step + 1
      visits_history <- insert_top(visits_history, visits)
    }
  }

  loc <- end ## set loc to the end and track the camefrom path back to find the solution
  while(any(loc != start)) {
    maze[loc$row, loc$col] <- as.numeric(maze[loc$row, loc$col]) * -1
    pathpart <- peek_top(camefrom)
    while(any(pathpart$to != loc)) {
      camefrom <- without_top(camefrom)
      pathpart <- peek_top(camefrom)
    }
    loc <- pathpart$from
  }
  maze[loc$row, loc$col] <- as.numeric(maze[loc$row, loc$col]) * -1
  return(list(maze, visits_history))
}
```

This solution will run longer (as it will visit all of the maze rather than stop the first time the end is

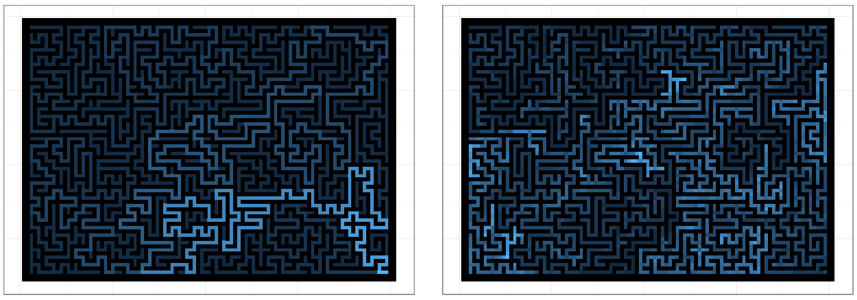


Figure 8: Mazes illustrating the length of time each cell spent waiting in the search structure, with lighter colors indicating longer periods in the 'rstack' for the depth-first search (left) or 'rdeque' for the breadth-first search (right).

found), but is guaranteed to find the shortest path from the start to the end. In this case both solutions are the same, as shown in Figure 7 (right). (In fact, mazes generated by recursive backtracking are trees, so there is only one simple solution.)

These algorithmic examples don't make explicit use of the persistence provided by 'rstack's and 'rdeque's. Some persistent data structures are an integral part of the methods they are used in; for example persistent binary trees are used in the planar point location problem (Sarnak and Tarjan, 1986), and many other applications of persistence are found in the area of computational geometry in general (Kaplan, 1995). Persistence also supports multi-threaded parallelization, since different threads needn't worry about the mutability of their parameters (Chuang and Goldberg, 1993).

More simply for our examples, because the structures are persistent we can save the state of the search over time (in `path_history` and `visits_history` in the functions above). These histories can later be analyzed, for example to illustrate how many steps each cell spent waiting in the stack or deque (Figure 8).

Availability

The stack, deque, and queue structures described here are available as an R package called **rstackdeque** available on CRAN. The maze examples and benchmarking code may be found on GitHub at `https://github.com/oneilsh/rstackdeque_examples`. Persistent structures such as these also support illustrating the operation of dynamic algorithms with the powerful plotting capabilities of **ggplot2** (Wickham, 2009) and similar packages. As an example, the `mazes.mp4` video in the `rstackdeque_examples` GitHub repository illustrates the creation and solution of a maze through time.

Summary

We've described the implementation of dynamic stack and queue data structures, guided by research in purely-functional languages so that they fit well into the R ecosystem via persistence. Rather than serve as a replacement for standard containers like lists and dataframes, stacks and queues (along with hashes as provided by the **hash** package (Brown, 2013)) enable a wide range of staple algorithmic techniques that until now have been more naturally implemented in languages like Python, Java, and C.

Further work might consider implementing additional persistent data structures, such as red-black trees or heaps. Because persistent versions of these are often $O(\log n)$ amortized insertion and removal, it may be beneficial to implement them via **Rcpp** and functional approaches for C++ (McNamara and Smaragdakis, 2000; Eddelbuettel and François, 2011; Eddelbuettel, 2013).

Acknowledgements

Thanks to Zhian Kamvar and Charlotte Wickham for comments, and the Center for Genome Research and Biocomputing at Oregon State University for support. Special thanks to reviewers, whose suggestions improved clarity and motivation for this article as well as contributed to the usability of the **rstackdeque** package.

Bibliography

S. Böhringer. Dynamic parallelization of R functions. *The R Journal*, 5(2):88–96, 2013. [p118]

C. Brown. *hash: Full Feature Implementation of Hash/Associated Arrays/Dictionaries*, 2013. URL http://CRAN.R-project.org/package=hash. R package version 2.2.6. [p125]

F. W. Burton. An efficient functional implementation of FIFO queues. *Information Processing Letters*, 14 (5):205–206, 1982. [p119]

T.-R. Chuang and B. Goldberg. Real-time deques, multihead Turing machines, and purely functional programming. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 289–298, 1993. [p118, 125]

S. Conchon and J.-C. Filliâtre. A persistent union-find data structure. In *Proceedings of the 2007 Workshop on ML*, pages 37–46. ACM, 2007. [p118]

T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001. [p118]

D. Eddelbuettel. *Seamless R and C++ Integration with Rcpp*. Springer, New York, 2013. [p125]

D. Eddelbuettel and R. François. Rcpp: Seamless R and C++ integration. *Journal of Statistical Software*, 40(8):1–18, 2011. [p125]

D. Grether, A. Neumann, and K. Nagel. Simulation of urban traffic control: A queue model approach. *Procedia Computer Science*, 10:808–814, 2012. [p118]

H. Kaplan. Persistent data structures. In *Handbook on Data Structures and Applications*, pages 241–246. CRC Press, 1995. [p118, 125]

B. McNamara and Y. Smaragdakis. Functional programming in C++. *ACM SIGPLAN Notices*, 35(9): 118–129, 2000. [p125]

O. Mersmann. *microbenchmark: Accurate Timing Functions*, 2014. URL http://CRAN.R-project.org/package=microbenchmark. R package version 1.4-2. [p121]

C. Okasaki. Simple and efficient purely functional queues and deques. *Journal of Functional Programming*, 5(04):583–592, 1995. [p119, 120]

C. Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999. [p118, 119]

L. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1996. [p120]

N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679, 1986. [p125]

H. Wickham. *ggplot2: Elegant Graphics for Data Analysis*. Springer, New York, 2009. URL http://had.co.nz/ggplot2/book. [p104, 125]

H. Wickham and R. Francois. *dplyr: A Grammar of Data Manipulation*, 2015. URL http://CRAN.R-project.org/package=dplyr. R package version 0.4.2. [p121]

Shawn T. O'Neil
Center for Genome Research and Biocomputing
Oregon State University
3021 ALS, Corvallis, OR 97333
United States of America
shawn.oneil@cgrb.oregonstate.edu