Collections in R: Review and Proposal

by Timothy Barry

Abstract R is a powerful tool for data processing, visualization, and modeling. However, R is slower than other languages used for similar purposes, such as Python. One reason for this is that R lacks base support for *collections*, abstract data types that store, manipulate, and return data (e.g., sets, maps, stacks). An exciting recent trend in the R extension ecosystem is the development of *collection packages*, packages that provide classes that implement common collections. At least 12 collection packages are available across the two major R extension repositories, the Comprehensive R Archive Network (CRAN) and Bioconductor. In this article, we compare collection packages in terms of their features, design philosophy, ease of use, and performance on benchmark tests. We demonstrate that, when used well, the data structures provided by collection packages are in many cases significantly faster than the data structures provided by base R. We also highlight current deficiencies among R collection packages and propose avenues of possible improvement. This article provides useful recommendations to R programmers seeking to speed up their programs and aims to inform the development of future collection-oriented software for R.

Introduction

R is one of the most popular languages used for data analysis due to its rich package ecosystem, robust user support community, and powerful modeling and graphical capabilities (Muenchen, 2012). However, R is not a fast language. R has an unusual and costly combination of language features, including lazy function evaluation, extreme dynamism, and pass-by-value to functions (Morandat et al., 2012). Additionally, R lacks base support for common computer science data structures (Wickham, 2014), impeding users from selecting efficient data structures for specific tasks (Cormen, 2009). Some of these limitations are necessary trade-offs of a programming environment that facilitates sophisticated modeling for practitioners across many fields. However, the general trend toward larger datasets makes the performance cost of these limitations more acute.

A promising avenue for addressing these difficulties is the development of R packages that implement *collections*. In computer science, a collection is an interface for storing, manipulating, and retrieving a group of like data items (Dale and Walker, 1996). Common examples of collections include sets, maps, and stacks. Collections are used frequently in programming; the languages Python, Java, and C++ ship with comprehensive collection frameworks (van Rossum and Drake, 2011; Stroustrup, 2013; Schildt, 2014). The addition of collections to R is an exciting prospect. Collections are (in general) backed by mathematically optimal algorithms and thus help accelerate program execution times. In the case of R, collections can be implemented in C++ via **Rcpp** for increased speed (Eddelbuettel et al., 2011). An added benefit of collections is that they make programs more abstract and thus easier to write, read, debug, and revise (Liskov and Zilles, 1974).

In recent years, researchers have developed R packages designed specifically implement collections. For example, Peng (2006), Brown (2013), and Russell (2017) implemented maps; Meyer and Hornik (2009) implemented sets; O'Neil (2015), Csárdi (2016), Collier (2016), Schmidt (2016), and Poncet (2017) implemented stacks and queues; and Bengtsson (2015), Giuliano (2017), and Pagés et al. (2017) implemented sequences (also known as *lists*). These packages vary in their features, performance, memory footprint, design philosophy, and ease of use. Consequently, selecting the appropriate package for a particular task is challenging. Moreover, these packages have inconsistent application programming interfaces (APIs), and so switching between them or using several in unison is awkward or in some cases impossible.

The purpose of this paper is twofold: first, to describe and compare R packages that implement collections (henceforth called *collection packages*); second, to propose an avenue for improving the collection capabilities of R. The former will aid R users in navigating the existing collection package ecosystem, while the latter will help guide the development of new R collection software. We begin with a brief review of common computer science data structures and clarify the distinction between abstract data types and data structures. We then evaluate R collection packages and compare them in terms of features and performance. We conclude with a discussion of future directions and areas of potential improvement. In particular, we propose the development of a unified, simple, and efficient R collection framework.

Background

We briefly review data structures and abstract data types.

Data structures

A data structure is a way of organizing information in a computer's memory (Black, 1998). There exist myriad data structures, each of which has strengths and weaknesses. In this section we review what we consider to be the three data structures most relevant to R users: arrays, linked lists, and hash tables. We assume the reader is familiar with big O notation [1]. Readers interested in learning more should refer to, for example, Cormen (2009).

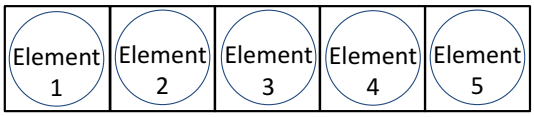

Figure 1: An example of an array. The circles represent data elements, and the squares represent locations of storage in memory.

An array is a linear sequence of elements stored contiguously in memory (Figure 1). Retrieving or modifying an element at a particular index is $\mathcal{O}(1)$. Removing an element from the middle or beginning of an array is $\mathcal{O}(n)$ because elements that follow the element that has been removed must be copied and shifted one position to the left. Checking whether an element is present in an array (called *search*) typically requires iterating over each element and in this case is $\mathcal{O}(n)$. Finally, assuming sufficient space has been allocated so as to not require array resizing, appending an element to the end of an array is $\mathcal{O}(1)$. (If resizing is necessary, appending an element is $\mathcal{O}(n)$.)

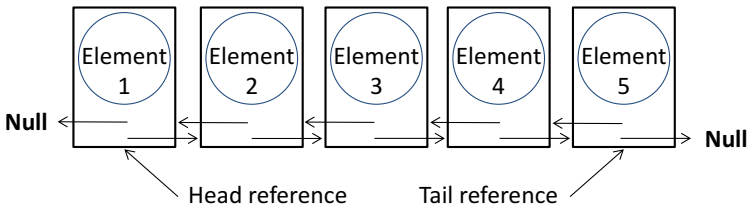

Figure 2: An example of a doubly-linked list. The circles represent data elements, and the rectangles represent nodes. The front and rear nodes have references to `NULL`. The list has both head and tail references.

A linked list is a linear sequence of objects called *nodes* (Figure 2). Each node contains a single data element and a reference to the following node (and, in the case of *doubly-linked lists*, a reference to the previous node as well). We focus here on doubly-linked lists because they are most relevant to the current paper. Insertion or deletion at a node for which a reference exists is $\mathcal{O}(1)$. Because the program must store a reference to the front (or *head*) node to access the list, insertion or removal at the front of a doubly-linked list is $\mathcal{O}(1)$. The program can optionally store a reference to the rear (or *tail*) node, in which case insertion or removal at the rear of the list is $\mathcal{O}(1)$.

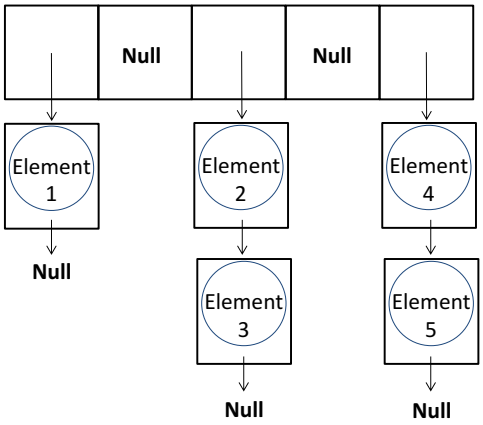

Figure 3: An example of a hash table. The circles represent data elements, the connected squares represent storage locations of an array, and the disconnected rectangles represent linked lists of nodes.

[1] Briefly, for functions $f, g : \mathbb{Z}^{\geq 0} \to \mathbb{R}$, we say f is $\mathcal{O}(g)$ if there exist $c, n_0 > 0$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.

A hash table is a data structure built to support fast insertion, removal, and search (Figure 3). Hash tables can be implemented in a number different ways, but one common implementation is roughly as follows: Allocate an array. Define a function that associates data elements to be stored in the hash table with an index of the array. When adding an element, compute the index with which it is associated and store it in the appropriate array position. If multiple data elements are mapped to the same array position, store the elements in a linked list. Hash tables support amortized $\mathcal{O}(1)$ insertion, removal, and search.

Abstract data types

An abstract data type is a collection of objects whose behavior is defined by a set of operations (Dale and Walker, 1996). Abstract data types are mathematical formalisms rather than concrete objects stored on a computer's hardware. They are useful because they allow a programmer to focus on an object's functionality rather than its implementation details. A *collection* is an abstract data type that stores data elements. Common examples of collections include sets, maps, stacks, queues, and sequences. We briefly describe each of these collections here:

- A *set* is a collection that does not contain duplicate elements. Sets support the operations '`insert`', '`delete`', '`search`', and occasionally '`intersect`' and '`union`'. '`insert`' adds a given element to the collection; '`delete`' removes a given element from the collection; '`search`' checks for the presence of a given element in the collection; and '`intersect`' and '`union`' implement standard set intersection and union procedures.
- A *map* is a set that consists of ordered pairs, called *key-value* pairs. A key x is said to be *associated* with a value y if the ordered pair (x, y) is an element of the collection. Maps impose the condition that each key be associated with a single value. Maps support the operations '`put`', '`get`', and '`remove`'. '`put`' adds a given key-value pair to the collection; '`get`' queries the value associated with a given key; and '`remove`' deletes a given a key-value pair from the collection.
- A *stack* is a collection that supports the following two operations: '`push`' and '`pop`'. '`push`' adds a given element to the collection; '`pop`' removes the most recently-added element from the collection.
- A *queue* is a collection that supports the following two operations: '`enqueue`' and '`dequeue`'. '`enqueue`' adds a given element to the collection; '`dequeue`' removes the least-recently added element from the collection.
- A *sequence* is an ordered collection of elements. The exact functionality of a sequence depends on its implementation, but typically sequences support the operations '`append`', '`replace`', '`getObjectAtIndex`', '`getIndexOfObject`', '`remove`', and '`search`'. '`append`' adds a given element to the end of the collection; '`replace`' replaces a given element or the element located at a given index with another element; '`getObjectAtIndex`' returns the element located at a given index; '`getIndexOfObject`' returns the index (or indices) at which a given element is located; '`remove`' deletes a given element or the element located at a given index from the collection; and '`search`' checks for the presence of a given element in the collection. Familiar examples of sequences include R vectors and lists. Vectors in R are homogeneous sequences (i.e., can store a single type of object), while lists in R are heterogeneous sequences (i.e., can store different types of objects).

The names we use to refer to the different collection types are standard (Dale and Walker, 1996), with the possible exception of *sequence*. Many programmers prefer the term *list* to *sequence*, but we choose to use the term *sequence* to avoid confusion with R lists.

Abstract data types and data structures are related in that data structures are used to implement abstract data types. To illustrate, one could implement a sequence abstract data type using an array data structure (Figure 1) or a linked list data structure (Figure 2). If using an array, '`append`', for instance, involves inserting an element into the memory location following the memory location of the rightmost element; if using a linked list, on the other hand, '`append`' involves inserting a new node at the tail of the linked list. At the level of hardware, these operations are clearly distinct. However, at a more abstract, theoretical level, these operations both implement '`append`'.

Methods

We searched the Comprehensive R Archive Network (CRAN) and Bioconductor for collection packages, evaluated the API and design of the packages returned by this search, and then ran benchmark tests to assess the performance of these packages.

Searching CRAN and Bioconductor

We searched CRAN and Bioconductor for packages that implement one or more of the following collection types: map, set, stack, queue, sequence. Table 1 lists the terms used in the search, the number of packages returned by the search, the number of packages excluded from review after the search, and the names of the packages ultimately selected for review. We searched CRAN and Bioconductor because these platforms are the two most popular repositories for R packages, and because packages must pass a review before publication on these platforms. CRAN packages published through July 3, 2017, and Bioconductor packages published through March 24, 2018, were included in the search.

The vast majority of packages we excluded from review were unrelated to either collections or data structures. For example, the search term "array" returned many packages related to the analysis of microarray data. Some excluded packages, however, did relate to collections or data structures. We note those packages here.

- The packages **hashr** (van der Loo, 2015) and **hashFunction** (Zhan, 2013) implement functions that generate hash codes for R objects. While hashing plays an important role in the construction of hash tables, **hashr** and **hashFunction** do not implement complete map classes.
- The package **filehashSQLite** (Peng, 2012) implements a key-value map using SQLite as the backend. For the sake of simplicity, we neglected **filehashSQLite** to focus exclusively on its more popular companion package, **filehash** (Peng, 2006).
- The package **tictoc** (Izrailev, 2014) implements a complete stack class. The primary purpose of **tictoc** is to benchmark R code; the stack class is tangential. Moreover, the implementation of the **tictoc** stack class is virtually identical to that of a dedicated collection package already selected for review (namely, **liqueueR**).
- The package **DSL** (Bengtsson, 2015) implements a sequence class that can be stored in a distributed manner. While important, distributed computing is outside the scope of the current work.
- The packages **bit64** (Oehlschlägel, 2017) and **bit** (Oehlschlägel, 2014) implement sequence classes that can efficiently store 64-bit integers and 1-bit booleans, respectively. While useful in certain contexts, **bit64** and **bit** are too specialized for inclusion this article.
- The package **Oarray** (Rougier, 2013) implements a sequence class that supports 0-based indexing. This class is too similar to the standard base R vector for inclusion in this review.

Collection	Search terms	N. packages returned	N. packages excluded	Packages kept
Set	'set,' 'hash'	448	447	**sets** (Meyer and Hornik, 2009)
Map	'map,' 'associative array,' 'dictionary,' 'dictionaries,' 'hash,' 'symbol table'	257	254	**filehash** (Peng, 2006), **hash** (Brown, 2013), **hashmap** (Russell, 2017)
Stack	'stack,' 'last-in-first-out,' 'LIFO'	14	9	**rstackdeque** (O'Neil, 2015), **rstack** (Csárdi, 2016) **liqueueR** (Collier, 2016), **dequer** (Schmidt, 2016), **flifo** (Poncet, 2017)
Queue	'queue,' 'first-in-first-out,' 'FIFO'	7	4	**rstackdeque** (O'Neil, 2015), **liqueueR** (Collier, 2016), **dequer** (Schmidt, 2016)
Sequence	'sequence,' 'list,' 'vector,' 'array'	449	446	**listenv** (Bengtsson, 2015), **S4Vectors** (Pagés et al., 2017), **stdvectors** (Giuliano, 2017)

Table 1: Details of the search for collection packages on CRAN and Bioconductor.

We selected a total of 11 packages from CRAN (**filehash**, **hash**, **hashmap**, **sets**, **rstackdeque**, **rstack**, **liqueueR**, **dequer**, **flifo**, **listenv**, and **stdvectors**) and one package from Bioconductor (**S4Vectors**) for review.

Analysis of packages

We analyzed the design and features of the packages selected for review. For a given package (e.g., **rstackdeque**) and collection class implemented by that package (e.g., "stack"), we determined the following:

1. API of the class
2. types of R objects that can be stored by the class
3. language in which the class was implemented (e.g., pure R, C++, C, etc.)
4. data structure(s) used to implement the class
5. algorithmic complexity of the operations provided by the class
6. whether the operations provided by the class have side effects or are side effect-free
7. whether the class was implemented using the S3, S4, or R6 object-oriented system
8. any unusual features or limitations of the class

We conducted this analysis by reading package vignettes and manuals, corresponding with package authors, and examining package source code.

Benchmark tests

We wrote and ran scripts to measure the execution times of key functions implemented by the packages selected for review. We refer to these trials of execution times as *benchmark tests*. For a given collection type, abstract operation defined on that collection type, and package implementing that collection type, we measured the execution time of the given function by the given package. We repeated this test 25 times each for collection objects containing different numbers of elements, and then plotted mean execution time against object size for that collection type, package, and operation. The commands used to generate the benchmark test results are given in the appendix.

As an example of this procedure, consider the map collection type. For a given package p among those that implement the map collection type (**filehash**, **hash**, and **hashmap**), for a given operation o among those defined on the map collection type ('`put`', '`get`', and '`remove`'), and for a positive integer n, we measured the mean execution time t of 25 calls to method o of a map object from package p containing n elements. We then plotted the point (n, t) on the graph of *mean function execution time vs. number of elements in collection* corresponding to package p and operation o. For a given function, the plot of *mean function execution time vs. number of elements in collection* reflects function scalability (or lack thereof): plots with shallow growth suggest the function scales well, while plots with steep growth suggest the function does not scale as well. The values of n we tested ranged from 100 to 100,000, and are similar to the values of n used by other authors for benchmark tests (e.g., O'Neil (2015), Russell (2017)). Benchmark test result plots are available in the appendix.

To help readers interpret benchmark test results, we developed our own custom implementation of each collection type using a base R list. For example, we implemented the map abstract data type using a named list, wherein names served as keys and entries served as values. The '`put`', '`get`', and '`remove`' operations were implemented by appending data to, indexing, and subsetting the list, respectively. Given the centrality and importance of the list to R programming, our base R list implementation of a given collection type can be seen as the "control" or "default" implementation of that collection type.

We measured function execution times using the package **microbenchmark** (Mersmann, 2015). The **microbenchmark** package provides a function that measures the execution time of code with sub-millisecond accuracy. **microbenchmark** uses the operating system-level routine `` `mach_absolute_time' `` on MacOSX and is more precise than R's native `'System.time'` function. Scripts were run in R version 3.4.1 (2017-06-30) using an Apple computer with an Intel Core i5 CPU (2.5 GHz, 2 cores, 4 logical processors), 4 GB of RAM, and 500 GB of storage. Additional details on the benchmark tests are available in the appendix.

Results

Sets

The only package we located on CRAN that implements the set abstract data type is **sets**. The **sets** package is rich in features and provides broad support for basic sets, generalized sets (i.e., sets in which elements are mapped to a real number), and customizable sets (i.e., sets in which the user can specify iteration order and the function used to check for element equality). The basic set class is

called "set". The "set" class is equipped with a wide variety of methods, including 'search', 'union', 'intersection', 'complement', 'power set', 'Cartesian product', and more. The "set" class does not have 'insert' and 'delete' methods per se, but its 'union' and 'complement' methods implement these operations in effect. The package is written predominantly in R and partially in C. "set" objects are S3 objects and operations on "set" objects are side-effect free.

Under the hood, "set" objects are R lists that are ordered to allow for the comparison of nested sets. The function 'search' essentially creates a hash table, inserts the elements of the calling set into the hash table, and then checks for the existence of the queried element. This procedure is $\mathcal{O}(n)$. The methods 'insert' (as implemented by calling 'union' on the set and the object to be added) and 'remove' (as implemented by calling 'complement' on the set and the object to be removed) are more involved. Ultimately, both methods perform a comparison-based sort and are therefore $\mathcal{O}(n\ log(n))$ at best. The **sets** package performed worse than a base R List implementation of the set abstract data type on all operations and for all set sizes (Figure 5).

Maps

Among the packages selected for review, **hash**, **hashmap**, and **filehash** implement the map abstract data type.

The map class of the package **hash** is called "hash". The "hash" class implements the standard map API, as well as methods to invert a "hash" object and return the number of key-value pairs in a "hash" object. Keys stored in "hash" objects must be atomic character vectors, but values can be any R object. The "hash" class is essentially a convenience wrapper around R environments. Because environments are implemented internally as hash tables (Wickham, 2014), the 'put', 'get', and 'remove' methods of the "hash" class are $\mathcal{O}(1)$ amortized. The "hash" class is implemented in S4, and its methods (in general) have side-effects.

The **hashmap** package provides a map class called "Hashmap". The "Hashmap" class implements a number of useful methods in addition to 'put', 'get', and 'remove', such as methods to return a named vector representation of a "Hashmap" object and merge "Hashmap" objects. "Hashmap" objects can store a variety of atomic vector types as keys and values, but not general R objects. The **hashmap** package is implemented in C++, and the "Hashmap" class is essentially a wrapper around a hash table class in the C++ Standard Template Library. Accordingly, 'put', 'get', and 'remove' operations are $\mathcal{O}(1)$ amortized. The "Hashmap" class is implemented in S4 and has methods that (in general) operate via side effects. An interesting feature of **hashmap** is that the 'put', 'get', and 'remove' methods are efficiently vectorized.

The map class of the package **filehash**, called "filehash", is rather different from "hash" and "Hashmap". While "hash" and "Hashmap" store keys and values in working memory, "filehash" stores keys and values on disk. This allows the user to work with datasets that take up more space than is available in RAM. The API of **filehash** includes the standard map operations as well as methods to test for the existence of a key and return the number of items in a "filehash" object. Keys must be atomic character vectors, while values can be arbitrary R objects. The "filehash" class is implemented predominantly in R, but C code powers file input and output. "filehash" objects are S4 objects, and operations on "filehash" objects (in general) have side effects.

The data structure underlying a "filehash" object is somewhat complex. Essentially, a "filehash" object is a list containing a reference to a file on disk and a reference to an R environment. The file on disk stores keys and values in serialized format, and the environment maps each key to the byte location of its associated value in the file on disk. The 'put' and 'remove' operations are $\mathcal{O}(1)$. The running time of 'get' is proportional to the number of elements inserted into the map object since 'get' was last called. Thus, if a user calls 'get' twice after inserting a large number of elements, the first call will be slow while the second will run in constant time. We expect 'get' to be a fast operation in most cases.

The **hash**, **hashmap**, and **filehash** packages all performed well on benchmark tests compared to a base R list implementation of the map abstract data type. The base list was fastest for small datasets. However, **hash**, **hashmap**, and **filehash** began outperforming the base list at around 750 - 4,000 elements, 2,000 - 15,000 elements, and 10,000 - 70,000 elements, respectively (Figure 6).

Stacks

Our search for packages that implement the stack abstract data type returned **rstackdeque**, **dequer**, **liqueueR**, **flifo**, and **rstack**.

The package **rstackdeque** provides a stack class called "rstack". The "rstack" class implements standard 'push' and 'pop' operations, as well as functions to obtain the length of a stack, reverse the

order of elements in a stack, and inspect the first element of a stack. Internally, an "rstack" object is a singly linked list of R environments. Accordingly, an "rstack" object can store any R object and supports $\mathcal{O}(1)$ 'push' and 'pop'. "rstack" objects are implemented in S3, and operations on "rstack" objects are side-effect free. Thus, **rstackdeque** allows for functional programming, which in fact is one of its major strengths.

The stack class provided by the package **dequer** is called "stack". The "stack" class implements the standard stack API, as well as a method to print the top element of a stack. 'stacks' are S3 objects and can store any type of R object. Internally, 'stacks' are implemented as doubly-linked lists in C. Thus, 'push' and 'pop' are $\mathcal{O}(1)$ operations. The **dequer** package has two limitations that should be noted. First, the only way to extract elements from a "stack" is to coerce the entire "stack" into an R list; this procedure is $\mathcal{O}(n)$. Second, due to implementation details of the R protection stack, it is really only possible to instantiate and use a single "stack" object per R session (Schmidt, personal communication).

The class "Stack" from the package **liqueueR** is a simple implementation of the stack abstract data type. "Stack" implements the operations 'push' and 'pop', as well as methods to query the size of a "Stack" object and combine "Stack" objects. Internally, "Stack" objects are R lists. The 'push' operation is implemented by appending an element to the end of the list, and the 'pop' operation is implemented by removing the last element of the list. Because R lists are copied when modified, 'push' and 'pop' are $\mathcal{O}(n)$ operations. 'Stacks' can store any type of R object. "Stack" is implemented using R6, and so 'push' and 'pop' have side-effects.

The package **flifo** is similar to **liqueueR** in that **flifo** implements the stack abstract data type using an R list. Moreover, the stack class provided by **flifo**, called "lifo", operates via side-effects. There are, however, several unusual design features of **flifo** that warrant special attention. First, when pushing a variable onto a "lifo" object, the variable is deparsed and then deleted from the calling environment. For example, the following code, if executed in the global environment, will result in the deletion of the variable 'myVariable': `myStack <-lifo(); myVariable <-1:10; push(myStack, myVariable)`. Second, **flifo** implements "lifo" objects using the S3-object oriented system. Because operations on "lifo" objects are designed to have side-effects, the 'push' and 'pop' functions must record the calling "lifo" object's name and then reassign this object (by name) to the calling environment. This behavior could be more naturally achieved if "lifo" objects were instead implemented using R6. Due to **flifo**'s unusual design features and similarity to **liqueueR**, we choose not to benchmark **flifo**.

Finally, the packages **rstack** and **filehash** implement stack classes, but we choose not to benchmark these classes. The stack class from **rstack** is implemented using the R6 reference system and is designed to support $\mathcal{O}(1)$ 'push' and 'pop'. However, due to a bug in the R6 reference system, the 'push' and 'pop' methods of this class are currently $\mathcal{O}(n)$ (Csàrdi, personal communication). The package **filehash**, which implements the map abstract data type (as described in the *Maps* section), incidentally implements a stack class as well. We discovered this only after downloading and exploring the package, because **filehash** was not returned in our search stack packages on CRAN and Bioconductor (see Table 1). We choose not to benchmark the stack class provided by **filehash** because it is essentially a variation on the map class.

The packages **dequer** and **Rstackdequeue** performed favorably compared to the base R list implementation of a stack, while the package **LiqueuR** performed less favorably. **dequer** and **Rstackdequeue** began outperforming the base R list at about 700-1,000 elements and 9,000-10,000 elements, respectively. **LiqueuR** exhibited strictly worse performance than the base R list (Figure 7).

Queues

Of the packages that implement stacks (**rstackdeque**, **dequer**, **liqueueR**, **flifo**, and **rstack**), all but **rstack** also implement queues. Overall, the queue classes provided by these packages are quite similar to the corresponding stack classes. The queue class of **rstackdeque** is implemented using linked lists of R environments, supports amortized $\mathcal{O}(1)$ 'enqueue' and 'dequeue', and is side-effect free. **liqueueR**'s queue class is implemented as a simple R list, supports $\mathcal{O}(n)$ 'enqueue' and 'dequeue', and has side-effects. The queue class of **flifo** is virtually the same as that of **liqueueR** in terms of implementation and algorithmic complexity. We do not run benchmark tests on the **flifo** queue class for the same reasons we do not run benchmark tests on the **flifo** stack class. It turns out that **filehash** also implements a queue class, which we likewise do not benchmark for reasons given in the previous section.

The queue class of **dequer**, much like the stack class of **dequer**, is implemented as a doubly-linked list of nodes, is written in C, and has methods that operate via side-effects. The queue class of **dequer**, however, appears to have some bugs that the stack class of **dequer** does not. First, **dequer**'s queue class works only when it has fewer than approximately 10^6 elements. Second, benchmark tests suggest the 'dequeue' method of **dequer**'s queue class runs in $\mathcal{O}(n)$ time (Figure 8). In theory, this method

should run in $\mathcal{O}(1)$ time. The cause of this discrepancy is unclear despite an extensive examination of source code.

The results of the benchmark tests of the queue packages were broadly similar to those of stack packages. The **liqueueR** package performed strictly worse than the base R list implementation of a queue. **Rstackdeque** began outperforming the base R list at about 20,000 - 60,000 elements. Finally, **dequer** was strictly slower than the base R list with respect to the 'dequeue' operation, but was faster with respect to the 'enqueue' operation on queues containing more than 1,000-2,000 elements (Figure 8).

Sequences

Among the packages under review, **stdvectors**, **listenv**, and **S4Vectors** implement the sequence abstract data type.

The package **stvectors** implements a sequence class called "stdvector". The "stdvector" class has a small API: "stdvector" objects support the operations 'append', 'replace', 'getObjectAtIndex', and 'remove' (by index), but not 'search', 'getIndexOfObject', or 'remove' (by element). Internally, a "stdvector" object is a C++ array that grows dynamically. Accordingly, the functions 'getIndexOfObject' and 'replace' are $\mathcal{O}(1)$, the function 'append' is $\mathcal{O}(1)$ amortized, and the function 'remove' (when called on an element in the middle of the sequence) is $\mathcal{O}(n)$. "stdvector" objects are implemented in S3 and can store any type of R object. Operations on "stdvector" objects (in general) have side-effects.

The sequence class provided by the **listenv** package is called "listenv". The "listenv" class has the same API of the "stdvector" class (described above), is written in pure R, and can store any type of R object. Internally, a "listenv" object is a list containing a character vector and a reference to an R environment. Elements are stored in the environment and are indexed by names that are stored in the character vector. The methods 'append' and 'remove' involve modifying both the R environment and the character vector. Because character vectors are copied upon modification, 'append' and 'remove' are $\mathcal{O}(n)$ operations. The methods 'getObjectAtIndex' and 'replace' involve first copying the character vector and then performing a lookup in the environment. 'getObjectAtIndex' and 'replace' are $\mathcal{O}(n)$, but grow very slowly in execution time as n increases. Operations on "listenv" objects have side-effects, and the "listenv" class is implemented using the S3 system.

S4Vectors is a large and richly-featured package that provides broad support for the sequence abstract data type. At the core of **S4Vectors** is "Vector", an abstract class (i.e., class that cannot be instantiated) that supports basic sequence functionality. Most of the classes implemented by **S4Vectors** extend "Vector" in some useful way. For example, the "Rle" class efficiently represents sequences that contain long subsequences of repeating elements; the "Pairs" class stores sequences consisting of ordered pairs; and the "LLint" class stores sequences consisting of very large integers. Users can easily write their own, custom extension of "Vector" class as well. **S4Vectors** is written in both R and C, and the many classes provided by **S4Vectors** are implemented using the S4 object-oriented system. The package **IRanges** (Lawrence et al., 2013), a companion package of **S4Vectors**, implements many additional helpful extensions of the "Vector" class.

We decided to run benchmark tests on the 'SimpleList' class of the **S4Vectors** package. The 'SimpleList' class is directly analogous to a base R list and provides essentially the same functionality as a base R list. In fact, under the hood, a 'SimpleList' is an S4 object containing a base R list. The operations 'append', 'replace', and 'remove' run in $\mathcal{O}(n)$ time on objects of this class, and the 'getObjectAtIndex' operation runs in $\mathcal{O}(1)$ time. It should be noted that the 'SimpleList' class is fairly generic; more specialized extensions of the "Vector" class (e.g., "Rle") will likely perform better under certain conditions.

The benchmark test results for packages implementing the sequence abstract data type were mixed. The **stdvectors** package was faster than the base R list with respect to the 'append' and 'remove' operations on objects containing more than 1,000 - 2,000 elements, but was strictly slower with respect to the other operations. The **listenv** package outperformed the base R list with respect to the 'append' operation on objects containing more than 4,000 elements, but performed strictly worse with respect to the other operations. Finally, the **S4Vectors** package was strictly slower than the R list with respect to all operations (Figure 8).

Discussion

The state of R collection packages

CRAN and Bioconductor have many collection packages available for download. At present, deciding which package to use is challenging given the variety of options. In this section we provide practical

recommendations to users exploring the collection package ecosystem.

Among the packages we reviewed that implement the map collection (**filehash**, **hash**, and **hashmap**), the best package for most situations is probably **hash**. **hash** is fast, and **hash** map objects can store any type of R object. The package **hashmap** has efficiently vectorized functions, but **hashmap** appears to be slower than **hash** and the map class it implements cannot store general R objects. The package **filehash** is useful when working with large datasets (i.e., datasets that consume more memory than is available in RAM), but should be avoided in favor of **hash** when working with smaller datasets.

The only package we located on CRAN that implements the set abstract data type is **sets**. The **sets** package is versatile and provides a rich collection of functions and classes. However, **sets** is slow; '`search`' runs in linear time, and '`insert`' and '`remove`' run in superlinear time. These functions would run in constant time if "set" objects were implemented as hash tables rather than R lists. Users interested in working with a solid implementation of the set abstract data type, and less interested in speed, should consider the **sets** package.

We consider **rstackdeque** to be the best package for general-purpose use among the packages we reviewed that implement the stack collection (**rstackdeque**, **dequer**, **liqueueR**, **flifo**, and **rstack**). **rstackdeque** is reasonably fast, fully-featured, and supports functional programming. The package **dequer** is faster than **rstackdeque**, but does not allow for functional programming and suffers from several bugs and drawbacks (see *Results*). The packages **liqueueR**, **flifo**, and **rstack** do not allow for functional programming and implement $\mathcal{O}(n)$ '`push`' and '`pop`', which we consider to be prohibitively slow. We also recommend **rstackdeque** among the packages we reviewed that implement the queue abstract data type for similar reasons.

A standard R list is the probably the best implementation of the sequence abstract data type in most situations. R lists are fast and support the full sequence API (as defined in the *Background* section). The **S4Vectors** package provides a wide range of classes that implement sequences. The class that we benchmarked, '`SimpleList`', is slower than a base R list. However, many of the classes implemented by **S4Vectors** (and its companion package **IRanges**) are excellent choices for storing more specialized, structured sequence data. The sequence class implemented by **stdvectors** outperforms the base R list with respect to the operations '`append`' and '`remove`' for large datasets. However, **stdvectors** does not support the full sequence API and is slower than a base R list with respect to the other sequence operations. The **listenv** sequence class is, for the most part, slower than a base R list. Also, it implements fewer methods.

Many R users only use the data structures provided by base. While base data structures are reasonably fast, there are cases in which base data structures are much slower than data structures implemented by collection packages. For example, the "rstack" class from **rstackdeque** is about an order of magnitude faster than a base R list when operating on large datasets. R programmers should consider using the packages recommended above to speed up program execution times when efficiency is important.

Most of the collection packages reviewed in this paper are used by other packages as imports or dependencies. **hash**, for instance, is a dependency of the package **neuroim** (Buchsbaum, 2016). **neuroim** allows users to store and manipulate brain imaging data, and the map class from **hash** serves to associate character identifiers with different brain regions. **sets** is used by the **FindMinIC** (Lange et al., 2013) package. **FindMinIC** creates models from all subsets of a given list of predictor variables and then rank-orders the models by information criterion (e.g., AIC). The '`power set`' operator from **sets** is used generate the space of possible models. **S4Vectors**, being among the top 5% of most downloaded packages on Bioconductor, is imported by a large number of other packages. One such package is **GenomicRanges** (Lawrence et al., 2013), which uses **S4Vectors** to represent genomic annotations and alignments. These examples, and many others, illustrate the usefulness of collection packages in R software development.

Looking forward

The collection package ecosystem in R is young and could be improved in several ways. First, set objects could be implemented as hash tables rather than R lists (as is the case with **sets**) to maximize the efficiency of set operations. Second, stack and queue objects could be stored internally as linked lists of *nodes* rather *environments* (as is the case with **rstackdeque**). Environments in R are by default hash tables, and therefore are more expensive in space than nodes, which are structures that only store data and a reference to another node. Third, every collection object could be equipped with an iterator to facilitate iteration through the elements of the collection. Minor improvements related to the implementation of the map and sequence abstract data types are also possible.

One compelling way to implement these changes would be to create a single, unified collection package. Such a package might have "Map," "Set," "Stack", "Queue," and "Sequence" classes that

derive from a generic "Collection" abstract class. The "Collection" class might have methods to check for the presence of an element ('`contains`' or '`%in%`'), return the number of elements stored ('`size`'), return an iterator ('`iterator`'), and print to the console ('`print`'). The derived classes would implement these methods as well as their own, specialized methods (Figure 4). Each class might be implemented in C or in C++ via the package **Rcpp**. In addition to offering probable performance improvements over current collection packages, such a package would provide a consistent API between different collection classes and aggregate important abstract data types into a single location. We anticipate that developers will disagree on the specific implementation details of the proposed package. For instance, some may want to implement purely functional data structures, while others will prefer non-functional data structures; some may want to implement the classes using the S3 object oriented system, while others will prefer S4, and still others R6; some may want to implement additional abstract data types (e.g., priority queues, graphs, etc.), while others will prefer to keep the package lean and implement only those abstract data types mentioned above. Designing and writing the proposed package will be a major project. The R community should take inspiration from the latest in functional (e.g., OCaml, Haskell) and scripting (e.g., Python, Ruby) languages as it considers this possibility.

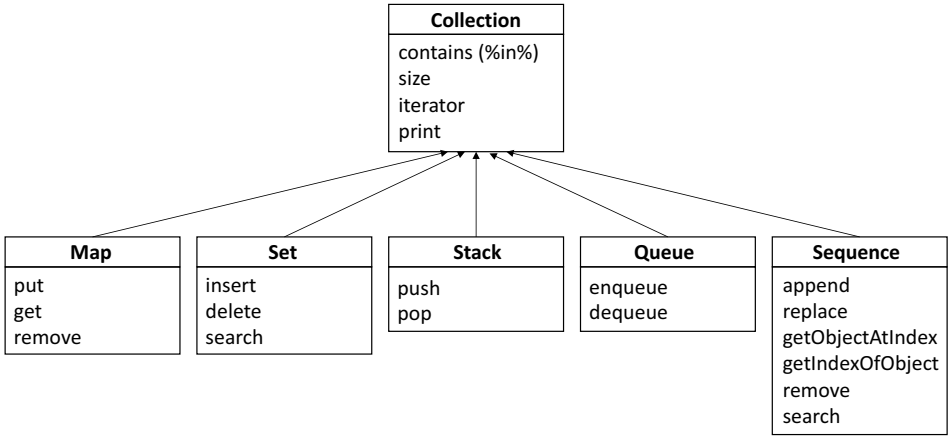

Figure 4: Inheritance diagram of a package that implements the map, set, stack, queue, and sequence collections.

Package authors can take away several general strategies for improving package performance based on the analysis of the packages reviewed for this article. First, authors should avoid unnecessarily copying objects inside functions (as **listenv** does, for instance). Copying objects, especially those that are large, is time-intensive. Authors should also keep in mind that most arguments in R are passed by deep copy to functions. Passing arguments by reference instead (as **rstackdeque** does by using environments) can accelerate function execution times. Second, authors should avoid adding gratuitous side-effects to functions that could be written without side-effects. At its core, R is a functional language. Users expect the data they pass into functions to remain unchanged. Functions with seemingly unnecessary side-effects (like those from **liqueueR** and **flifo**) may catch users off-guard. That said, some functions (like '`insert`' on map data structures) require side-effects to run in constant time. Finally, authors should remember that certain R functions and data structures have $\mathcal{O}(n)$ complexity "hidden" in them. The '`dequeue`' function from **dequer**, and the '`push`' and '`pop`' functions from **rstack**, are supposed to run in $\mathcal{O}(1)$ time. Unfortunately, these functions appear to run in $\mathcal{O}(n)$ time for reasons that are not immediately obvious. Authors can detect surprises like these by benchmarking code before package release.

The utility of reviewing R packages

R's diverse selection of packages is one of its greatest strengths, but also a source of frustration for package users and developers. Users often must sift through many packages to find one that suits their needs, and developers sometimes inadvertently write packages that essentially replicate the functionality of packages that already exist on CRAN or Bioconductor. A good way for the R community to address this annoyance is to write more package reviews. Reviews can help users quickly find the best package for their purposes, avert authors from developing packages that in effect already exist, and highlight deficiencies in the R package ecosystem. There exist very few review articles in the archives of *The R Journal* or *The Journal of Statistical Software*, the two academic journals most directly related to R. Given the growth in popularity of R and the proliferation of R packages, additional review articles would be helpful.

Summary

For many statisticians, programmers, and scientists, R is the go-to tool for data processing, modeling, and visualization. However, R's lack of base support for collection data types can be a hindrance, especially as data sets continue to grow in size. In this article, we reviewed *collection packages* – packages that implement collection data types like sets, maps, stacks, queues, and sequences. We demonstrated that data types implemented by collection packages are in many cases faster than corresponding base R data types, and provided specific package recommendations to users navigating the collection package ecosystem. An interesting line of future work would be to investigate collection packages that implement more advanced abstract data types, such as graphs and priority queues. Such data types, though less common than the ones explored in this review, are quite useful in a variety of applications. Data structures and algorithms lie at the core of computer science. As R users develop new collection packages, and effectively use those that already exist, they will wield the language more powerfully.

Acknowledgments

We thank the Howard Hughes Medical Institute for supporting this study. We also thank Jason Brunson, Eliezer Gurarie, and two anonymous reviewers for reading and providing helpful comments on the manuscript.

Bibliography

H. Bengtsson. *Listenv: Environments Behaving (Almost) as Lists*, 2015. URL http://CRAN.R-project.org/package=listenv. R package version 0.6.0. [p1, 4]

P. E. Black. Dictionary of algorithms and data structures. *NISTIR*, 1998. [p2]

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. [p1, 4]

B. R. Buchsbaum. *Neuroim: Data Structures and Handling for Neuroimaging Data*, 2016. URL https://CRAN.R-project.org/package=neuroim. R package version 0.0.6. [p9]

A. Collier. *liqueueR: Implements Queue, PriorityQueue and Stack Classes*, 2016. URL http://CRAN.R-project.org/package=liqueueR. R package version 0.0.1. [p1, 4]

T. H. Cormen. *Introduction to Algorithms*. MIT press, 2009. [p1, 2]

G. Csárdi. *Rstack: Stack Data Type as an 'R6' Class*, 2016. URL http://CRAN.R-project.org/package=rstack. R package version 1.0.0. [p1, 4]

N. Dale and H. M. Walker. *Abstract Data Types: Specifications, Implementations, and Applications*. Jones & Bartlett Learning, 1996. [p1, 3]

D. Eddelbuettel, R. François, J. Allaire, J. Chambers, D. Bates, and K. Ushey. Rcpp: Seamless R and C++ integration. *Journal of Statistical Software*, 40(8):1–18, 2011. [p1]

M. Giuliano. *Stdvectors: C++ Standard Library Vectors in R*, 2017. URL http://CRAN.R-project.org/package=stdvectors. R package version 0.0.5. [p1, 4]

S. Izrailev. *Tictoc: Functions for Timing R Scripts, as Well as Implementations of Stack and List Structures*, 2014. URL http://CRAN.R-project.org/package=tictoc. R package version 1.0. [p4]

N. Lange, T. Fletcher, and K. Zygmunt. *FindMinIC: Find Models with Minimum IC*, 2013. URL https://CRAN.R-project.org/package=FindMinIC. R package version 1.6. [p9]

M. Lawrence, W. Huber, H. Pagès, P. Aboyoun, M. Carlson, R. Gentleman, M. Morgan, and V. Carey. Software for computing and annotating genomic ranges. *PLoS Computational Biology*, 9, 2013. URL https://doi.org/10.1371/journal.pcbi.1003118. [p8, 9]

B. Liskov and S. Zilles. Programming with abstract data types. In *ACM Sigplan Notices*, volume 9, pages 50–59. ACM, 1974. [p1]

O. Mersmann. *Microbenchmark: Accurate Timing Functions*, 2015. URL http://CRAN.R-project.org/package=microbenchmark. R package version 1.4-2.1. [p5]

D. Meyer and K. Hornik. Generalized and customizable sets in R. *Journal of Statistical Software*, 31(2): 1–27, 2009. [p1, 4]

F. Morandat, B. Hill, L. Osvald, and J. Vitek. Evaluating the design of the R language. In *European Conference on Object-Oriented Programming*, pages 104–131. Springer, 2012. [p1]

R. A. Muenchen. The popularity of data analysis software, 2012. URL http://r4stats.com/popularity. [p1]

J. Oehlschlägel. *Bit: A Class for Vectors of 1-Bit Booleans*, 2014. URL http://CRAN.R-project.org/package=bit. R package version 1.1-12. [p4]

J. Oehlschlägel. *Bit64: A S3 Class for Vectors of 64bit Integers*, 2017. URL http://CRAN.R-project.org/package=bit64. R package version 0.9-7. [p4]

S. T. O'Neil. Implementing persistent O(1) stacks and queues in R. *R Journal*, 7(1), 2015. [p1, 4, 5]

H. Pagés, M. Lawrence, and P. Aboyoun. *S4Vectors: S4 Implementation of Vector-like and List-like Objects*, 2017. R package version 0.16.0. [p1, 4]

R. D. Peng. Interacting with data using the filehash package. *R News*, 6(4):19–24, 2006. URL https://cran.r-project.org/doc/Rnews/. [p1, 4]

R. D. Peng. *filehashSQLite: Simple Key-Value Database Using SQLite*, 2012. URL http://CRAN.R-project.org/package=filehashSQLite. R package version 0.2-4. [p4]

P. Poncet. *Flifo: Don't Get Stuck with Stacks in R*, 2017. URL http://CRAN.R-project.org/package=flifo. R package version 0.1.4. [p1, 4]

J. Rougier. *Oarray: Arrays with Arbitrary Offsets*, 2013. URL http://CRAN.R-project.org/package=Oarray. R package version 1.4-5. [p4]

N. Russell. *Hashmap: The Faster Hash Map*, 2017. URL http://CRAN.R-project.org/package=hashmap. R package version 0.2.0. [p1, 4, 5]

H. Schildt. *Java: The Complete Reference*. McGraw-Hill Education Group, 2014. [p1]

D. Schmidt. dequer: Stacks, queues, and deques for R, 2016. URL https://cran.r-project.org/package=dequer. R package version 2.0-0. [p1, 4]

B. Stroustrup. *The C++ Programming Language*. Pearson Education, 2013. [p1]

M. van der Loo. *Hashr: Hash R Objects to Integers Fast*, 2015. URL http://CRAN.R-project.org/package=hashr. R package version 0.1.0. [p4]

G. van Rossum and F. L. Drake. *The Python Language Reference Manual*. Network Theory Ltd., 2011. [p1]

H. Wickham. *Advanced R*. CRC Press, 2014. [p1, 6]

X. Zhan. *hashFunction: A Collection of Non-Cryptographic Hash Functions*, 2013. URL http://CRAN.R-project.org/package=hashFunction. R package version 1.0. [p4]

Timothy Barry
University of Maryland Department of Biology
Biology-Psychology Building
USA
ORCiD: 0000-0002-4356-627X
timbarry@umd.edu

Appendix

Tables 3 – 6 display the code we used to benchmark the operations of the packages under review. In Table 3, the variables 'key' and 'value' represent a key-value pair. In Tables 2 - 6, the variable 'elem' represents a data element. In all tables, the variable 'y' represents the collection object itself. Benchmark test results are displayed in Figures 5 - 9. In all figures, results for different packages are plotted on different y-scales due to high between-package variance in execution time. The points represent median execution time. The colored bands represent the 25th and 75th percentiles of execution time. The code used to produce these plots is available at github.com/Timothy-Barry/Collections-in-R.

Sets

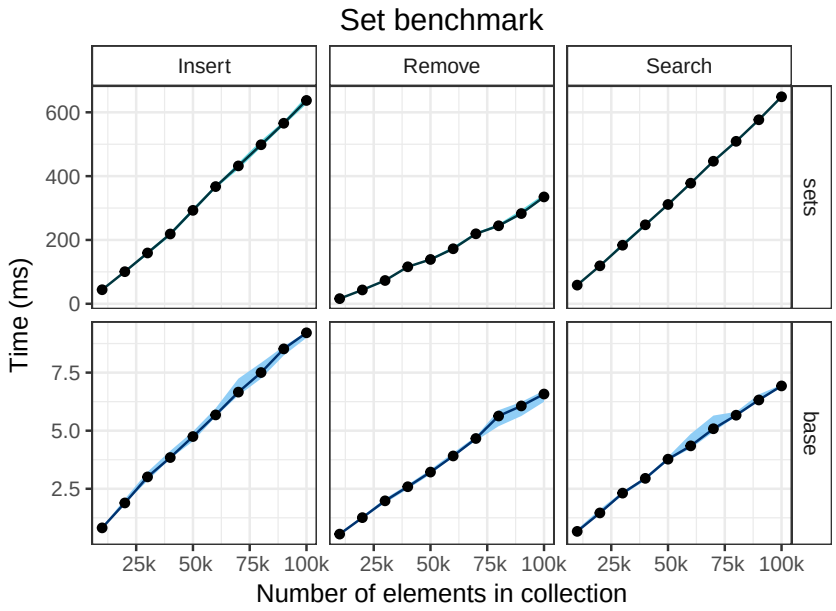

Figure 5: Benchmark test results the **sets** package.

	'Insert'	'Remove'
sets	`y <-set_union(y,elem)`	`y <-set_complement(elem,y)`
base	`if !(elem %in% y) c(y,elem) else y`	`y <-y[!(y == elem)]`
	'Search'	
sets	`set_contains_element(elem,y)`	
base	`elem %in% y`	

Table 2: Commands used in the benchmark for the set collection type.

Maps

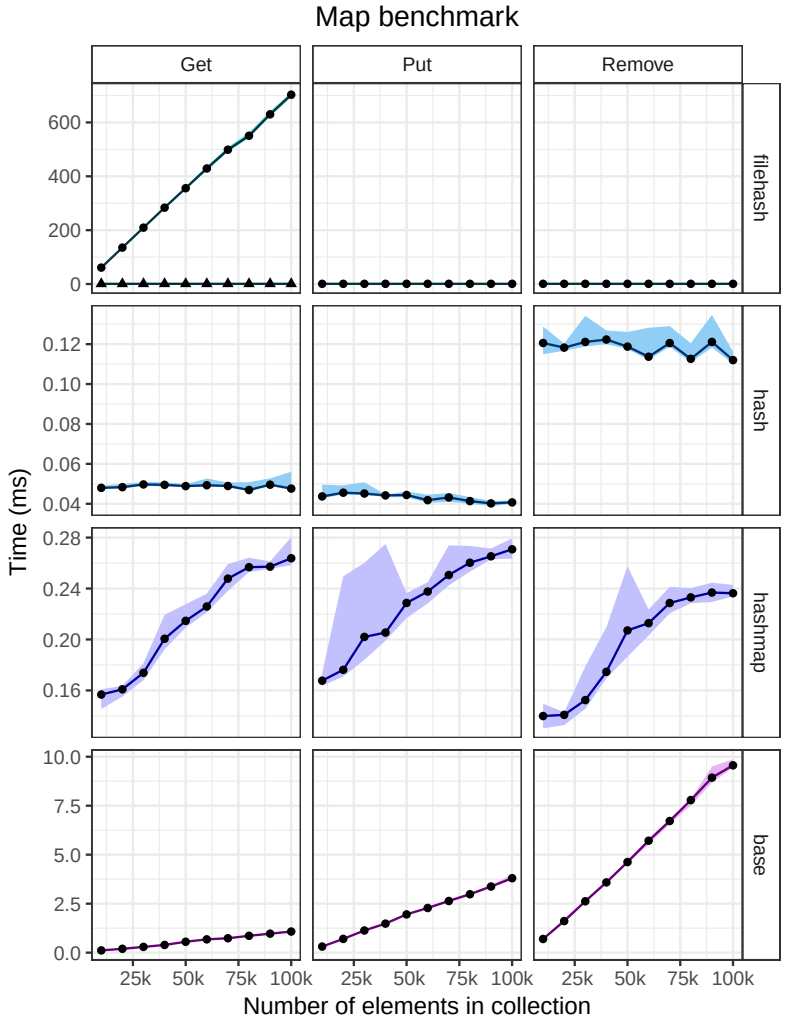

Figure 6: Benchmark test results for packages that implement the map abstract data type. The plot that shows the results of the 'get' benchmark for the package **filehash** has two lines. The line with circular points gives the running time of a single call to 'get'; the line with triangular points gives the running time of the second of two consecutive calls to 'get'.

	'Get'	'Put'	'Remove'
filehash	`dbFetch(y,key)`	`dbInsert(y,key,value)`	`dbDelete(y,key)`
hash	`y[[key]] <-value`	`y[[key]]`	`delete(key,y)`
hashmap	`y[[key]]`	`y[[key]] <-value`	`y$erase(key)`
base	`y[key]`	`y[key] <-value`	`y[!(names(y) %in% key]`

Table 3: Commands used in the benchmark for the map collection type.

Stacks

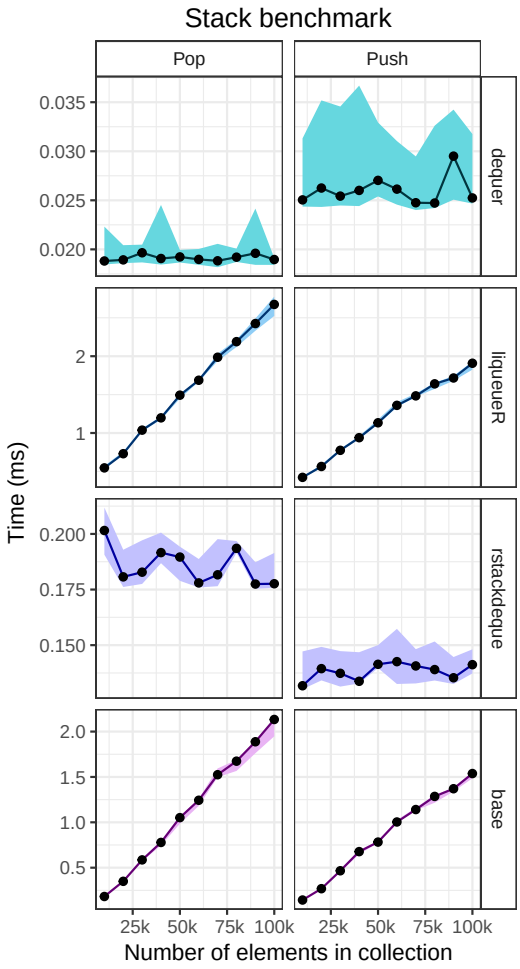

Figure 7: Benchmark test results packages that implement the stack abstract data type.

	'Pop'	'Push'
dequer	`pop(y)`	`push(y,elem)`
liqueR	`y$pop()`	`y$push(elem)`
rstackdeque	`without_top(y)`	`insert_top(y,elem)`
base	`y <-y[1:(length(y)-1)]`	`y <-c(y,elem)`

Table 4: Commands used in the benchmark for the stack collection type.

Queues

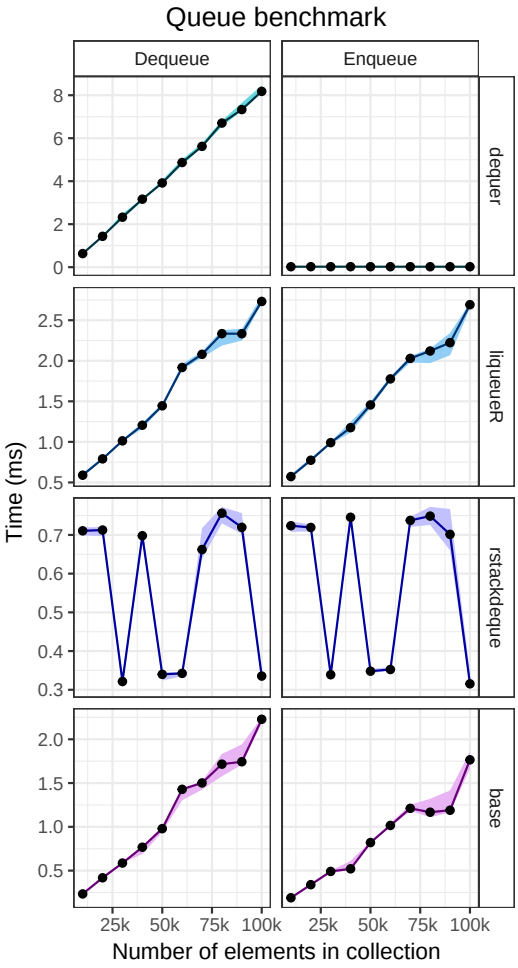

Figure 8: Benchmark test results of packages that implement the queue abstract data type.

	'Dequeue'	'Enqueue'
dequer	`pop(y)`	`pushback(y,elem)`
liqueueR	`y$pop()`	`y$push(elem)`
rstackdeque	`y <-(without_front(y))`	`y <-insert_back(y,elem)`
base	`y <-c(elem,y)`	`y <-y[1:(length(y)-1)]`

Table 5: Commands used in the benchmark for the queue collection type.

Sequences

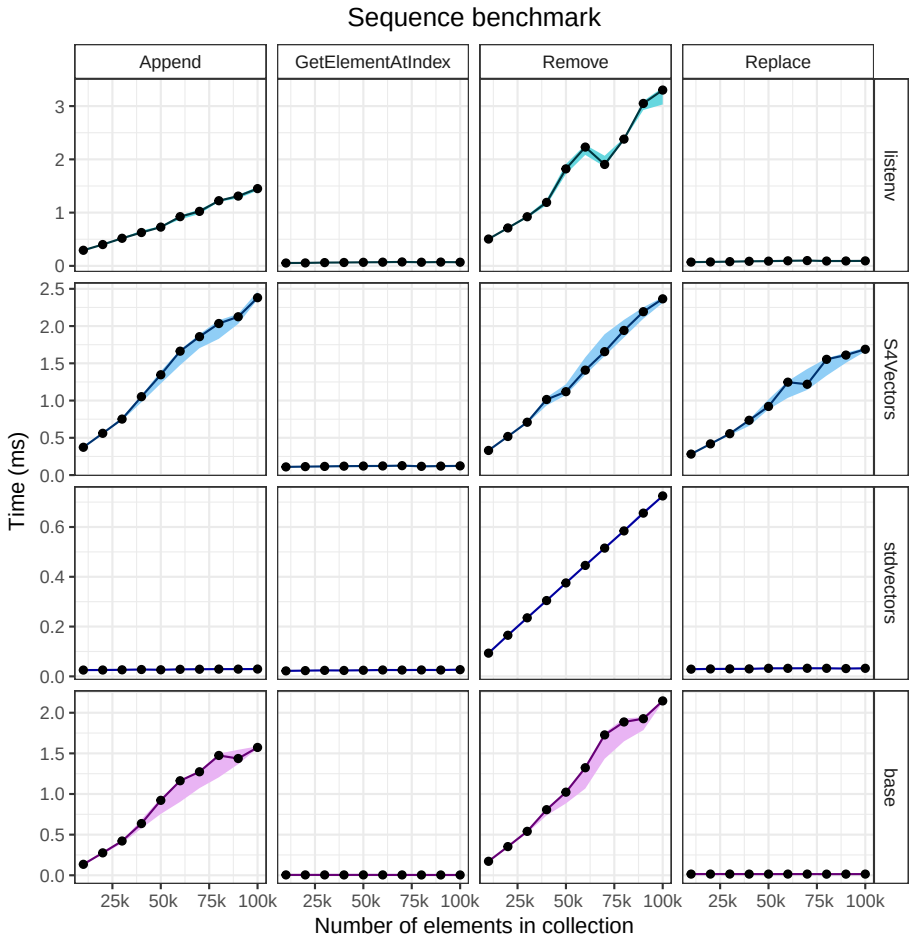

Figure 9: Benchmark test results for packages that implement the sequence abstract data type.

	'Append'	'GetElementAtIndex'
listenv	`y[[length(y) + 1]] <- elem`	`y[[index]]`
S4Vectors	`y[[length(y) + 1]] <-elem`	`y[[index]]`
stdvectors	`stdvectorPushBack(y,elem)`	`stdvectorSubset(y,index)`
base	`y <-c(y,elem)`	`y[index]`
	'Remove'	**'Replace'**
listenv	`y[[index]] <-NULL`	`y[[index]] <-elem`
S4Vectors	`y[[index]] <-NULL`	`y[[index]] <-elem`
stdvectors	`stdvectorErase(y,index,index)`	`stdvectorReplace(y,index,elem)`
base	`y <-y[-index]`	`y[index] <-elem`

Table 6: Commands used in the benchmark for the sequence collection type.