Functional Data Structures in R

Advanced Statistical Programming in R

Thomas Mailund

Apress®

Functional Data Structures in R

Advanced Statistical Programming in R

Thomas Mailund

Apress®

Functional Data Structures in R: Advanced Statistical Programming in R

Thomas Mailund
Aarhus N, Denmark

ISBN-13 (pbk): 978-1-4842-3143-2 ISBN-13 (electronic): 978-1-4842-3144-9
https://doi.org/10.1007/978-1-4842-3144-9

Library of Congress Control Number: 2017960831

Cover image by Freepik (www.freepik.com)

Managing Director: Welmoed Spahr
Editorial Director: Todd Green
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Technical Reviewer: Karthik Ramasubramanian
Coordinating Editor: Mark Powers
Copy Editor: Corbin P Collins

Printed on acid-free paper

Table of Contents

About the Author

Thomas Mailund is an associate professor in bioinformatics at Aarhus University, Denmark. He has a background in math and computer science. For the last decade, his main focus has been on genetics and evolutionary studies, particularly comparative genomics, speciation, and gene flow between emerging species. He has published *Beginning Data Science in R*, *Functional Programming in R*, and *Metaprogramming in R* with Apress, as well as other books.

About the Technical Reviewer

Karthik Ramasubramanian works for one of the largest and fastest-growing technology unicorns in India, Hike Messenger, where he brings the best of business analytics and data science experience to his role. In his seven years of research and industry experience, he has worked on cross-industry data science problems in retail, e-commerce, and technology, developing and prototyping data-driven solutions. In his previous role at Snapdeal, one of the largest e-commerce retailers in India, he was leading core statistical modeling initiatives for customer growth and pricing analytics. Prior to Snapdeal, he was part of the central database team, managing the data warehouses for global business applications of Reckitt Benckiser (RB). He has vast experience working with scalable machine learning solutions for industry, including sophisticated graph network and self-learning neural networks. He has a master's degree in theoretical computer science from PSG College of Technology, Anna University, and is a certified big data professional. He is passionate about teaching and mentoring future data scientists through different online and public forums. He enjoys writing poems in his leisure time and is an avid traveler.

Introduction

This book gives an introduction to functional data structures. Many traditional data structures rely on the structures being mutable. We can update search trees, change links in linked lists, and rearrange values in a vector. In functional languages, and as a general rule in the R programming language, data is not mutable. You cannot alter existing data. The techniques used to modify data structures to give us efficient building blocks for algorithmic programming cannot be used.

There are workarounds for this. R is not a pure functional language, and we can change variable-value bindings by modifying environments. We can exploit this to emulate pointers and implement traditional data structures this way; or we can abandon pure R programming and implement data structures in C/C++ with some wrapper code so we can use them in our R programs. Both solutions allow us to use traditional data structures, but the former gives us very untraditional R code, and the latter has no use for those not familiar with other languages than R.

The good news, though, is that we don't have to reject R when implementing data structures if we are willing to abandon the traditional data structures instead. There are data structures that we can manipulate by building new versions of them rather than modifying them. These data structures, so-called *functional data structures*, are different from the traditional data structures you might know, but they are worth knowing if you plan to do serious algorithmic programming in a functional language such as R.

There are not necessarily drop-in replacements for all the data structures you are used to, at least not with the same runtime performance for their operations, but there are likely to be implementations for most abstract data structures you regularly use. In cases where you might have to lose a bit of efficiency by using a functional data structures instead of a traditional one, however, you have to consider whether the extra speed is worth the extra time you have to spend implementing a data structure in exotic R or in an entirely different language.

There is always a trade-off when it comes to speed. How much programming time is a speed-up worth? If you are programming in R, chances are you value programmer-time over computer-time. R is a high-level language and relatively slow compared to most other languages. There is a price to providing higher levels of expressiveness. You accept this when you choose to work with R. You might have to make the same choice when it comes to selecting a functional data structure over a traditional one, or you might conclude that you really do need the extra speed and choose to spend more time programming to save time when doing an analysis. Only you can make the right choice based on your situation. You need to find out the available choices to enable you to work data structures when you cannot modify them.

CHAPTER 1

Introduction

This book gives an introduction to functional data structures. Many traditional data structures rely on the structures being mutable. We can update search trees, change links in linked lists, and rearrange values in a vector. In functional languages, and as a general rule in the R programming language, data is *not* mutable. You cannot alter existing data. The techniques used to modify data structures to give us efficient building blocks for algorithmic programming cannot be used.

There are workarounds for this. R is not a *pure* functional language, and we can change variable-value bindings by modifying environments. We can exploit this to emulate pointers and implement traditional data structures this way; or we can abandon pure R programming and implement data structures in C/C++ with some wrapper code so we can use them in our R programs. Both solutions allow us to use traditional data structures, but the former gives us very untraditional R code, and the latter has no use for those not familiar with other languages than R.

The good news, however, is that we don't have to reject R when implementing data structures if we are willing to abandon the traditional data structures instead. There are data structures we can manipulate by building new versions of them rather than modifying them. These data structures, so-called *functional data structures*, are different from the traditional data structures you might know, but they are worth knowing if you plan to do serious algorithmic programming in a functional language such as R.

There are not necessarily drop-in replacements for all the data structures you are used to, at least not with the same runtime performance for their operations—but there are likely to be implementations for most abstract data structures you regularly use. In cases where you might have to lose a bit of efficiency by using a functional data structure instead of a traditional one, you have to consider whether the extra speed is worth the extra time you have to spend implementing a data structure in exotic R or in an entirely different language.

There is always a trade-off when it comes to speed. How much programming time is a speed-up worth? If you are programming in R, the chances are that you value programmer time over computer time. R is a high-level language that is relatively slow compared to most other languages. There is a price to providing higher levels of expressiveness. You accept this when you choose to work with R. You might have to make the same choice when it comes to selecting a functional data structure over a traditional one, or you might conclude that you really *do* need the extra speed and choose to spend more time programming to save time when doing an analysis. Only you can make the right choice based on your situation. You need to find out the available choices to enable you to work data structures when you cannot modify them.

CHAPTER 2

Abstract Data Structures

Before we get started with the actual data structures, we need to get some terminologies and notations in place. We need to agree on what an *abstract* data structure is—in contrast to a *concrete* one—and we need to agree on how to reason with runtime complexity in an abstract way.

If you are at all familiar with algorithms and data structures, you can skim quickly through this chapter. There won't be any theory you are not already familiar with. Do at least skim through it, though, just to make sure we agree on the notation I will use in the remainder of the book.

If you are not familiar with the material in this chapter, I urge you to find a text book on algorithms and read it. The material I cover in this chapter should suffice for the theory we will need in this book, but there is a lot more to data structures and complexity than I can possibly cover in a single chapter. Most good textbooks on algorithms will teach you a lot more, so if this book is of interest, you should not find any difficulties in continuing your studies.

Structure on Data

As the name implies, data structures have something to do with structured data. By *data*, we can just think of elements from some arbitrary set. There might be some more structure to the data than the individual data points, and when there is we keep that in mind and will probably want to exploit that somehow. However, in the most general terms, we just have some large set of data points.

So, a simple example of working with data would be imagining we have this set of possible values—say, all possible names of students at a university—and I am interested in a subset—for example, the students that are taking one of my classes. A *class* would be a subset of students, and I could represent it as the subset of student names. When I get an email from a student, I might be interested in figuring out if it is from one of *my* students, and in that case, in which class. So, already we have some structure on the data. Different classes are different subsets of student names. We also have an operation we would like to be able to perform on these classes: checking membership.

There might be some inherent structure to the data we work with, which could be properties such as lexicographical orders on names—it enables us to sort student names, for example. Other structure we add on top of this. We add structure by defining classes as subsets of student names. There is even a third level of structure: how we represent the classes on our computer.

The first level of structure—inherent in the data we work with—is not something we have much control over. We might be able to exploit it in various ways, but otherwise, it is just there. When it comes to designing algorithms and data structures, this structure is often simple information; if there is order in our data, we can sort it, for example. Different algorithms and different data structures make various assumptions about the underlying data, but most general algorithms and data structures make few assumptions. When I make assumptions in this book, I will make those assumptions explicit.

The second level of structure—the structure we add on top of the universe of possible data points—is information in addition to what just exists out there in the wild; this can be something as simple as defining classes as subsets of student names. It is structure we add to data for a purpose, of course. We want to manipulate this structure and use it to answer questions while we evaluate our programs. When it comes to algorithmic theory, what we are mainly interested in at this level is which operations are possible on the data. If we represent classes as sets of student names, we are interested in testing membership to a set. To construct the classes, we might also want to be able to add elements to an existing set. That might be all we are interested in, or we might also want to be able to remove elements from a set, get the intersection or union of two sets, or do any other operation on sets.

What we can do with data in a program is largely defined by the operations we can do on structured data; how we implement the operations is less important. That might affect the efficiency of the operations and thus the program, but when it comes to what is possible to program and what is not—or what is easy to program and what is hard, at least—it is the possible operations that are important.

Because it is the operations we can do on data, and now how we represent the data—the third level of structure we have—that is most important, we distinguish between the possible operations and how they are implemented. We define *abstract data structures* by the operations we can do and call different implementations of them *concrete data structures*. Abstract data structures are defined by which operations we can do on data; concrete data structures, by how we represent the data and implement these operations.

Abstract Data Structures in R

If we define abstract data structures by the operations they provide, it is natural to represent them in R by a set of generic functions. In this book, I will use the S3 object system for this.[1]

Let's say we want a data structure that represents sets, and we need two operations on it: we want to be able to insert elements into the set, and we want to be able to check if an element is found in the set. The generic interface for such a data structure could look like this:

```
insert <- function(set, elem) UseMethod("insert")
member <- function(set, elem) UseMethod("member")
```

Using generic functions, we can replace one implementation with another with little hassle. We just need one place to specify which concrete implementation we will use for an object we will otherwise only access through the abstract interface. Each implementation we write will have one function for constructing an empty data structure. This empty structure sets the class for the concrete implementation, and from here on we can access the data structure through generic functions. We can write a simple list-based implementation of the set data structure like this:

```
empty_list_set <- function() {
  structure(c(), class = "list_set")
}

insert.list_set <- function(set, elem) {
  structure(c(elem, set), class = "list_set")
}
```

[1]If you are unfamiliar with generic functions and the S3 system, you can check out my book *Advanced Object-Oriented Programming in R* book (Apress, 2017), where I explain all this.

```r
member.list_set <- function(set, elem) {
  elem %in% set
}
```

The `empty_list_set` function is how we create our first set of the concrete type. When we insert elements into a set, we also get the right type back, but we shouldn't call `insert.list_set` directly. We should just use `insert` and let the generic function mechanism pick the right implementation. If we make sure to make the only point where we refer to the concrete implementation be the creation of the empty set, then we make it easier to replace one implementation with another:

```r
s <- empty_list_set()
member(s, 1)
## [1] FALSE
s <- insert(s, 1)
member(s, 1)
## [1] TRUE
```

When we implement data structures in R, there are a few rules of thumb we should follow, and some are more important than others. Using a single "empty data structure" constructor and otherwise generic interfaces is one such rule. It isn't essential, but it does make it easier to work with abstract interfaces.

More important is this rule: keep modifying and querying a data structure as separate functions. Take an operation such as popping the top element of a stack. You might think of this as a function that removes the first element of a stack and then returns the element to you. There is nothing wrong with accessing a stack this way in most languages, but in functional languages, it is much better to split this into two different operations: one for getting the top element and another for removing it from the stack.

The reason for this is simple: our functions can't have side effects. If a "pop" function takes a stack as an argument, it cannot modify this stack. It can give you the top element of the stack, and it can give you a new stack where the top element is removed, but it cannot give you the top element and then modify the stack as a side effect. Whenever we want to modify a data structure, what we have to do in a functional language, is to create a new structure instead. And we need to return this new structure to the caller. Instead of wrapping query answers *and* new (or "modified") data structures in lists so we can return multiple values, it is much easier to keep the two operations separate.

Another rule of thumb for interfaces that I will stick to in this book, with one exception, is that I will always have my functions take the data structure as the first argument. This isn't something absolutely necessary, but it fits the convention for generic functions, so it makes it easier to work with abstract interfaces, and even when a function is not abstract—when I need some helper functions—remembering that the first argument is always the data structure is easier. The one exception to this rule is the construction of linked lists, where tradition is to have a construction function, `cons`, that takes an element as its first argument and a list as its second argument and construct a new list where the element is put at the head of the list. This construction is too much of a tradition for me to mess with, and I won't write a generic function of it, so it doesn't come into conflict with how we handle polymorphism.

Other than that, there isn't much more language mechanics to creating abstract data structures. All operations we define on an abstract data structure have some intended semantics to them, but we cannot enforce this through the language; we just have to make sure that the operations we implement actually do what they are supposed to do.

Implementing Concrete Data Structures in R

When it comes to concrete implementations of data structures, there are a few techniques we need in order to translate the data structure designs into R code. In particular, we need to be able to represent what are essentially pointers, and we need to be able to represent empty data structures. Different programming languages will have different approaches to these two issues. Some allow the definition of recursive data types that naturally handle empty data structures and pointers, others have unique values that always represent “empty,” and some have static type systems to help. We are programming in R, though, so we have to make it work here.

For efficient data structures in functional programming, we need recursive data types, which essentially boils down to representing pointers. R doesn’t have pointers, so we need a workaround. That workaround is using lists to define data structures and using named elements in lists as our pointers.

Consider one of the simplest data structures known to man: the linked list. If you are not familiar with linked lists, you can read about them in the next chapter, where I consider them in some detail. In short, linked lists consist of a *head*—an element we store in the list—and a *tail*—another list, one item shorter. It is a recursive definition that we can write like this:

```
LIST = EMPTY | CONS(HEAD, LIST)
```

Here `EMPTY` is a special symbol representing the empty list, and `CONS`—a traditional name for this, from the Lisp programming language—a symbol that constructs a list from a `HEAD` element and a tail that is another `LIST`. The definition is recursive—it defines `LIST` in terms of a tail that is also a `LIST`—and this in principle allows lists to be infinitely long. In practice, a list will eventually end up at `EMPTY`.

We can construct linked lists in R using R's built-in list data structure. That structure is *not* a linked list; it is a fixed-size collection of elements that are possibly named. We exploit named elements to build pointers. We can implement the CONS construction like this:

```
linked_list_cons <- function(head, tail) {
  structure(list(head = head, tail = tail),
            class = "linked_list_set")
}
```

We just construct a list with two elements, head and tail. These will be references to other objects—head to the element we store in the list, and tail to the rest of the list—so we are in effect using them as pointers. We then add a class to the list to make linked lists work as an implementation of an abstract data structure.

Using classes and generic functions to implement polymorphic abstract data structures leads us to the second issue we need to deal with in R. We need to be able to represent empty lists. The natural choice for an empty list would be NULL, which represents "nothing" for the built-in list objects, but we can't get polymorphism to work with NULL. We can't give NULL a class. We could, of course, still work with NULL as the empty list and just have classes for non-empty lists, but this clashes with our desire to have the empty data structures being the one point where we decide concrete data structures instead of just accessing them through an abstract interface. If we didn't give empty data structures a type, we would need to use concrete update functions instead. That could make switching between different implementations cumbersome. We really *do* want to have empty data structures with classes.

The trick is to use a sentinel object to represent empty structures. *Sentinel* objects have the same structure as non-empty data structure objects—which has the added benefit of making some implementations easier to write—but they are recognized as representing "empty." We construct a sentinel as we would any other object, but we remember it for future reference. When we create an empty data structure, we always return the same sentinel object, and we have a function for checking emptiness that examines whether its input is identical to the sentinel object. For linked lists, this sentinel trick would look like this:

```
linked_list_nil <- linked_list_cons(NA, NULL)
empty_linked_list_set <- function() linked_list_nil
is_empty.linked_list_set <- function(x)
  identical(x, linked_list_nil)
```

The `is_empty` function is a generic function that we will use for all data structures.

The `identical` test isn't perfect. It will consider any list element containing `NA` as the last item in a list as the sentinel. Because we don't expect anyone to store `NA` in a linked list—it makes sense to have missing data in a lot of analysis, but rarely does it make sense to store it in data structures—it will have to do.

Using a sentinel for empty data structures can also occasionally be useful for more than dispatching on generic functions. Sometimes, we actually want to use sentinels as proper objects, because it simplifies certain functions. In those cases, we can end up with associating meta-data with "empty" sentinel objects. We will see examples of this when we implement red-black search trees. If we do this, then checking for emptiness using `identical` will not work. If we modify a sentinel to change meta-information, it will no longer be identical to the reference empty object. In those cases, we will use other approaches to testing for emptiness.

Asymptotic Running Time

Although the operations we define in the interface of an abstract data type determine how we can use these in our programs, the *efficiency* of our programs depends on how efficient the data structure operations are.

Because of this, we often consider the time efficiency part of the interface of a data structure—if not part of the *abstract* data structure, we very much care about it when we have to pick concrete implementations of data structures for our algorithms.

When it comes to algorithmic performance, the end goal is always to reduce *wall time*—the actual time we have to wait for a program to finish. But this depends on many factors that cannot necessarily know about when we design our algorithms. The computer the code will run on might not be available to us when we develop our software, and both its memory and CPU capabilities are likely to affect the running time significantly. The running time is also likely to depend intimately on the data we will run the algorithm on. If we want to know exactly how long it will take to analyze a particular set of data, we have to run the algorithm on this data. Once we have done this, we know exactly how long it took to analyze the data, but by then it is too late to explore different solutions to do the analysis faster.

Because we cannot practically evaluate the efficiency of our algorithms and data structures by measuring the running time on the actual data we want to analyze, we use different techniques to judge the quality of various possible solutions to our problems.

One such technique is the use of *asymptotic complexity*, also known as *big-O notation*. Simply put, we abstract away some details of the running time of different algorithms or data structure operations and classify their runtime complexity according to upper bounds known up to a constant.

First, we reduce our data to its size. We might have a set with n elements, or a string of length n. Although our data structures and algorithms might use very different actual wall time to work on different data of the same size, we care only about the number n and not the details of the data. Of course, data of the same size is not all equal, so when we reduce all our information about it to a single size, we have to be a little careful about what we mean when we talk about the algorithmic complexity of a problem. Here, we usually use one of two approaches: we speak of the *worst-case* or the *average/expected* complexity. The worst-case runtime complexity of an algorithm is the longest running time we can expect from it on any data of size n. The expected runtime complexity of an algorithm is the mean running time for data of size n, assuming some distribution over the possible data.

Second, we do not consider the *actual* running time for data of size n—where we would need to know exactly how many operations of different kinds would be executed by an algorithm, and how long each kind of operation takes to execute. We just count the number of operations and consider them equal. This gives us some function of n that tells us how many operations an algorithm or operation will execute, but not how long each operation takes. We don't care about the details when comparing most algorithms because we only care about asymptotic behavior when doing most of our algorithmic analysis.

By *asymptotic behavior*, I mean the behavior of functions when the input numbers grow large. A function $f(n)$ is an asymptotic upper bound for another function $g(n)$ if there exists some number N such that $g(n) \leq f(n)$ whenever $n > N$. We write this in big-O notation as $g(n) \in O(f(n))$ or $g(n) = O(f(n))$ (the choice of notation is a little arbitrary and depends on which textbook or reference you use).

The rationale behind using asymptotic complexity is that we can use it to reason about how algorithms will perform when we give them larger data sets. If we need to process data with millions of data points, we might be about to get a feeling for their running time through experiments with tens or hundreds of data points, and we might conclude that one algorithm outperforms another in this range. But that does not necessarily reflect how the two algorithms will compare for much larger data. If one algorithm is asymptotically faster than another, it *will* eventually outperform the other—we just have to get to the point where n gets large enough.

A third abstraction we often use is to not be too concerned with getting the exact number of operations as a function of n correct. We just want an upper bound. The big-O notation allows us to say that an algorithm runs in any big-O complexity that is an upper bound for the actual runtime complexity. We want to get this upper bound as exact as we can, to properly evaluate different choices of algorithms, but if we have upper and lower bounds for various algorithms, we can still compare them. Even if the bounds are not tight, if we can see that the upper bound of one algorithm is better than the lower bound of another, we can reason about the asymptotic running time of solutions based on the two.

To see the asymptotic reasoning in action, consider the set implementation we wrote earlier:

```
empty_list_set <- function() {
  structure(c(), class = "list_set")
}

insert.list_set <- function(set, elem) {
  structure(c(elem, set), class = "list_set")
}

member.list_set <- function(set, elem) {
  elem %in% set
}
```

It represents the set as a vector, and when we add elements to the set, we simply concatenate the new element to the front of the existing set. *Vectors*, in R, are represented as contiguous memory, so when we construct new vectors this way, we need to allocate a block of memory to contain the new vector, copy the first element into the first position, and then copy the entire old vector into the remaining positions of the new vector. Inserting an element into a set of size n, with this implementation, will take time $O(n)$—we need to insert n+1 set elements into newly allocated blocks of memory. Growing a set from size 0 to size n by repeatedly inserting elements will take time $O(n^2)$.

The membership test, `elem %in% set`, runs through the vector until it either sees the value `elem` or reaches the end of the vector. The best case would be to see elem at the beginning of the vector, but if we consider worst-case complexity, this is another $O(n)$ runtime operation.

As an alternative implementation, consider linked lists. We insert elements in the list using the cons operation, and we check membership by comparing elem with the head of the list. If the two are equal, the set contains the element. If not, we check whether elem is found in the rest of the list. In a pure functional language, we would use recursion for this search, but here I have just implemented it using a while loop:

```
insert.linked_list_set <- function(set, elem) {
  linked_list_cons(elem, set)
}

member.linked_list_set <- function(set, elem) {
  while (!is_empty(set)) {
    if (set$head == elem) return(TRUE)
    set <- set$tail
  }
  return(FALSE)
}
```

The insert operation in this implementation takes constant time. We create a new list node and set the head and tail in it, but unlike the vector implementation, we do not copy anything. For the linked list, inserting elements is an $O(1)$ operation. The membership check, though, still runs in $O(n)$ because we still do a linear search.

Experimental Evaluation of Algorithms

Analyzing the asymptotic performance of algorithms and data structures is the only practical approach to designing programs that work on very large data, but it cannot stand alone when it comes to writing efficient code. Some experimental validation is also needed. We should always perform experiments with implementations to 1) be informed about the performance constants hidden beneath the big-O notation, and 2) to validate that the performance is as we expect it to be.

For the first point, remember that just because two algorithms are in the same big-O category—say, both are in $O(n^2)$—that doesn't mean they have the same wall-time performance. It means that both algorithms are asymptotically bounded by some function $c \cdot n^2$ where c is a constant. Even if both are running in quadratic time, so that the upper bound is actually tight, they could be bounded by functions with very different constants. They may have the same asymptotic complexity, but in practice, one could be much faster than the other. By experimenting with the algorithms, we can get a feeling, at least, for how the algorithms perform in practice.

Experimentation also helps us when we have analyzed the *worst case* asymptotic performance of algorithms, but where the data we actually want to process is different from the worst possible data. If we can create samples of data that resemble the actual data we want to analyze, we can get a feeling for how close it is to the worst case, and perhaps find that an algorithm with worse *worst case* performance actually has better *average case* performance.

As for point number two for why we want to experiment with algorithms, it is very easy to write code with a different runtime complexity than we expected, either because of simple bugs or because we are programming in R, a very high-level language, where language constructions potentially hide complex operations. Assigning to a vector, for example, is not a simple constant time operation if more than one variable refers to the vector. Assignment to vector elements potentially involves copying the entire vector. Sometimes it is a constant time operation; sometimes it is a linear time operation. We can deduce what it will be by carefully reading the code, but it is human to err, so it makes sense always to validate that we have the expected complexity by running experiments.

In this book, I will use the `microbenchmark` package to run performance experiments. This package lets us run a number of executions of the same operation and get the time it takes back in nanoseconds. I don't need that fine a resolution, but it is nice to be able to get a list of time measurements. I collect the results in a `tibble` data frame from which I can summarize the results and plot them later. The code I use for my experiments is as follows:

```
library(tibble)
library(microbenchmark)

get_performance_n <- function(
  algo
  , n
  , setup
  , evaluate
  , times
  , ...) {

  config <- setup(n)
  benchmarks <- microbenchmark(evaluate(n, config),
                               times = times)
  tibble(algo = algo, n = n,
         time = benchmarks$time / 1e9) # time in sec
}

get_performance <- function(
  algo
  , ns
  , setup
  , evaluate
  , times = 10
  , ...) {
```