Functional Data Structures

Data structures form an integral part of any algorithm. Heaps, queues, stacks, trees, and hash tables are various forms of data structures widely used across programming languages. Some of them are primarily used for look-ups, such as trees and hash tables, while others, such as heaps, queues, and stacks, are used for update modifications, such as insertions and deletions. The current chapter will build foundation to extend conventional data structure to functional data structure. In this chapter, you will learn the following concepts:

Functional data structures Lazy evaluation

Functional queues Functional stacks

Functional data structure

Functional data structures are special forms of data structure, which are implemented primarily in functional programming languages. R supports functional programming by providing tools for creation and manipulation of functions. For example, R support assigning functions to variables and passing them as an argument within a function. The R support generating the function dynamically and returning them as a result of the function is also known as a closure function. For example, the function which takes a function as an argument is shown as follows:

arg\_function <- function(g) g(seq(1, 100, by=1))

Page 259

﻿

Functional Data Structures

The function arg\_function can take functions as an argument, such as mean or sd as shown in the following code snippet:

> arg\_function(mean) [1] 50.5

> arg\_function(sd) [1] 29.01149

The functional data structure is also known as persistent data structure as they are immutable in the sense that any operation performed on function data structure will create a new copy of data structure with an updated operation. The original functional data structure will always remain intact. The functional data structure possesses different characteristics compared to traditional data structures. These are highly flexible in implementation and support the properties of immutability and persistency. In addition, they are thread-safe; that is, functional data structure ensures safe execution even in multithreaded environment during data structure manipulation. The following are some benefits of immutability:

It supports data hiding and data sharing. The former prevents any possibility of leakage of data (dynamically generated within functions) and the latter supports auto-sharing of data (based on pointers), as per the requirement within the functions.

As functional data structures do not support mutations, it requires no external synchronization. However, some support modifications (additions or deletions) using controlled mutations.

The pointers prevent long-distance coupling. The following are some benefits of persistency:

It augments mutation using constructive updates on the data structure instead of destructive updates. The modifications are incorporated constructively by replicating the whole data structure, keeping the older versions intact. It supports memory-efficient ways of replication, such as sharing of older versions instead of copying.

The following are some benefits of thread safety:

It supports concurrent programming without any need to worry about data races without mutation

[ 248 ]

Page 260

﻿

Chapter 10

During construction of new objects, mutations are performed only within the thread under consideration using pointers, and the original data is kept intact (immutable)

It is memory-efficient as it supports thread-safe reference counting using shared pointers and is optimized as log-free

Lazy evaluation

Functional data structures used in functional programming languages support lazy evaluation. In lazy implementation, the arguments of a function are evaluated if and only if the computational results are used further within the function. Furthermore, once the arguments are evaluated, the computational results are cached and can be reused later, if needed again, using look-up instead of recomputation. This kind of caching (also termed as memoization ) makes it highly difficult to estimate the asymptotic complexity of the algorithm as it is not straightforward to determine when a given argument (or a subexpression) will be evaluated.

Lazy evaluation plays a key role in the implementation of purely functional amortized data structures. It is extremely difficult to analyze the asymptotic performance of algorithms involving lazy evaluation. However, the following framework provides a basic support to calculate the asymptotic performances of algorithms involving lazy evaluation. Firstly, the costs of any given operation are classified, as follows:

Unshared cost : This defines the actual execution time of an operation, provided every deferment pertaining to that operation was forced and memoized prior to executing the operation.

Shared cost : This defines the execution time of all deferred operations, which were earlier excluded, while evaluating the performance of the operation under consideration. The shared cost can be further split into realized and unrealized costs. Realized costs evaluate the runtime of deferred operations, which are executed during overall computation whereas the unrealized costs evaluate the runtime of deferred operations, which are never executed during the overall computation.

Complete cost : This defines the actual execution time of the operation if it is implemented using lazy evaluation. It is the sum of shared and unshared costs, excluding unrealized cost. The minimum value of complete cost is unshared cost, provided no deferred operations are executed during the overall computation. Amortized cost : This defines the shared costs of the overall computation that are accounted using the concept of accumulated debt. Initially, the accumulated debt is set to zero and it starts accumulating for each deferment created. Then, each

[ 249 ]

Page 261

﻿

Functional Data Structures

operation starts paying off the debt. Once the debt is completely paid off, the deferments can then be forced and memoized. Here, the amortized cost of an operation is sum of unshared cost of the operation and debt paid off by each operation.

Thus, a deferred operation can only be forced and memoized once the accumulated debt is completely paid off. As the total amount of accumulated cost is capped at realized shared costs, the amortized cost cannot increase beyond the total actual cost.

Functional stacks

A stack is a First In Last Out ( FILO ) form of a data structure, wherein both insertions and deletions usually happen at the beginning (or top) of the stack. These forms are widely used in Depth-first search ( DFS ) algorithms. In ideal scenarios, the insertion, removal, and peeking operations require very little time generally with an asymptote of O(1) for the worst case scenario. Thus, during a worst case scenario, n insertions or removals would have an asymptote of O(n) , provided average asymptote of each operation of O(1) . Now, let's understand the implementation of fully-persistent stacks in R. The

implementation of a fully-persistent stack data structure is available in an rstackdeque cran package contributed by Shawn T. O'Neil. In R, mutability can be ensured using environment variables using side-effect-free interface functions.

In this package, stacks are implemented using unordered linked lists, wherein each node (list) consists of both data elements and reference to the next node. These stacks are S3 objects, accessible using the head stack node.

Let's consider a stack with five character elements:

a <- as.rstack(c("p", "q", "r","s","t"))

Analogous to a traditional push function, insert\_top is used to return a stack with new elements at the top. Here, instead of creating a new stack, the head element of b is pointed towards the o element, which is in-turn pointed toward stack a , as shown in Figure 10.1 :

b <- insert\_top(a, c("o"))

In case of withdrawal of top element from stack ( pop ), the without\_top function is used. Similar to insert\_top , the primary a stack is not destructively updated, but the pointer of stack c is shifted towards right by one element, as shown in Figure 10.1 :

c <- without\_top(a)

[ 250 ]

Page 262

﻿

Chapter 10

The peek\_top function is used to return the data element present at the top of the stack:

d <- peek\_top(a)

The following Figure 10.1 illustrates the implementation of fully-persistent stacks in R:

Figure 10.1: Working of fully-persistent stacks based on diﬀerent types of insertion or deletion operations. (a) represents a stack of ﬁve characters, each character linked to another character as in case of linked lists. (b) represents a stack after inserting a new element o on top of the stack (a). (c) represents a stack after removing the top element of the stack

(a). (d) represents a character vector with top element of stack (a).

Functional queues

A queue is a First In First Out ( FIFO ) form of a data structure, wherein deletions happen in the same order of insertion. One way of implementing queue is to insert elements from end (top of the queue) and delete elements from opposite end (bottom of the queue). Some queues support insertions and deletions from both the ends. These are termed as deques or double-ended queues. Queues and deques are used in Breadth-first search ( BFS )

algorithms. Similar to stacks, ideally the insertion, removal, and peeking operations require very little time with an asymptote of O(1) for the worst case scenario. Thus, during a worst case scenario, n insertions or removals would have an asymptote of O(n) , provided average asymptote of each operation of O(1) .

Now, let's understand the implementation of fast, fully-persistent and slowly-persistent queues and deques in R using the rstackdeque cran package contributed by Shawn T. O'Neil.

[ 251 ]

Page 263

﻿

Functional Data Structures

Fast fully-persistent queues

Fast and fully-persistent queues are governed primarily by recursively-defined operations and delayed evaluations. In other words, these queues are implemented using lazy lists wherein the first elements are immediately accessible and rest of the elements are evaluated with a delay. This is helpful, in the case of recursive large lists where elements are only evaluated whenever accessed.

The R function to implement fast persistent queues is rpqueue . It comprises of three last lists: l , r , lhat . Here, the elements are inserted at the back of the queue and removed or deleted from the front of the queue. In other words, the insertion of new elements occurs at the top of the r list and the deletion of existing elements occurs from front of the l list. These lazy lists are implemented as rstacks and every node's nextnode elements are assigned using the delayedAssign function. These are subsequently memoized on first evaluation.

Traditionally, the queue tries to ensure that the length of l stack is at least equal to the length of the r stack. However, during any insertion or deletion operation, the lengths are disturbed. The length of r stack increases upon insertion and the length of l stack decreases upon deletion. Post these operations, the l and r stacks are readjusted by appending the last elements of r stack with l stack. This readjustment requires an asymptote of O(1) for any sort of insertion or deletion happening within l and r stacks. Upon readjustment, lhat is assigned with data of stack l . Then, for each subsequent insertion or deletion, the elements of lhat are removed one by one until the lhat stack becomes empty. Once lhat becomes empty, the lengths of l and r stacks are evaluated and the elements are readjusted, again delaying the time of iteration.

Let's understand the working of fully-persistent queues using an example. Consider a persistent queue of length four:

a <- as.rpqueue(c("p","q","r","s"))

The "p" , "q" , and "r" elements are assigned to left stack l , element "s" is assigned to right stack "r" , and, elements "q" and "r" are assigned to stack lhat . The following Figure 10.2 illustrates working of persistent queues based on insertion and deletion operations. The insertion operation is performed using the insert\_back function and deletion operation is performed using the without\_front function.

[ 252 ]

Page 264

﻿

Chapter 10

Figure 10.2: Working of fully-persistent queues based on diﬀerent types of insertion or deletion operations. (a) represents a new persistent queue. (b) represents a queue after inserting a new element “ t ” in the back of queue (a). (c) represents a queue after deleting the front element from queue (b). (d) represents queue after inserting a new element “ v ” at the back of queue (c). (e) represents queue after inserting a new element “ w ” at the back of queue (d). (f) represents a queue after deleting an element from front of queue (e).

[ 253 ]

Page 265

﻿

Functional Data Structures

The following are R codes, which are used in the preceding illustration:

b <- insert\_back(a, "t") # insert a new element "t" c <- without\_front(b) # remove front element "b" d <- insert\_back(c,"v") # insert a new element "v" e <- insert\_back(d,"w") # insert a new element "w" f <- without\_front(e) # remove front element "e"

Slowly-persistent queues and deques

The queues are implemented as two stacks, that is, the left stack and the right stack. The left stack holds the first set of elements of the queue, which is used for deletion operations, and the right stack holds the last set of elements of the queue, which is used for insertion operations. On the other hand, the left stack can also be used for insertion and the right stack for deletion. The right stack holds elements in the reverse order as shown in Figure 10.3 .

Let's consider an a queue with seven character elements:

a <- as.rdeque(c("p", "q", "r","s","t","u","v"))

The a queue is split into left and right queues, as illustrated in Figure 10.3 and Figure 10.4 . Using the insert\_front or insert\_back functions, the elements are inserted at the front or back of the queue respectively:

b <- insert\_front(a, c("o")) c <- insert\_back(a, c("w"))

Similarly, elements can also be deleted from the front or back of the queue using the without\_front or without\_back functions:

d <- without\_front(a) e <- without\_back(a)

The following Figure 10.3 and Figure 10.4 describes implementation of slowly-persistent queues in R:

[ 254 ]

Page 266

﻿

Chapter 10

Figure 10.3: Working of slowly-persistent queues and dequeus based on diﬀerent types of insertion or deletion operations. (a) represents a queue with left and right stacks. (b)

represents queue after inserting a new element “ o ” at the front of queue (a).

The Figure 10.3 presents how slowly-persistent queues stores initial seven character element passed to a . The left side of the queue can be extracted using a$l and similarly right side of queue can be extracted using a$r . The b part of image demonstrate how functional queue a will be updated when an element o is added using function insert\_front(a, c("o")) .

[ 255 ]

Page 267

﻿

Functional Data Structures

Figure 10.4: (c) represents queue after inserting a new element w in the back of queue (a). (d) represents queue after deleting the front element from queue (a). (e) represents

queue after deleting the last element from queue (a).

[ 256 ]

Page 268

﻿

Chapter 10

In the current implementation, double-ended queues, which rebalance after every insertion or deletion, are used as shown in Figure 10.4 . If both left and right stacks become highly unbalanced, then they are both first decomposed into a list and then recomposed back into two nearly balanced stacks.

Summary

Apart from academic purposes, functional programming has found its way into a wide range of industrial applications such as aerospace, telecommunications and robotics. Pure functional languages such as Haskell, Scala, F# are becoming popular across these

industries. The properties of immutability and persistency make these data structures more robust in terms of transparency, simulations and efficiency. The current chapter primarily covers functional data structures, concepts of lazy evaluation and detailed analysis of functional stacks and functional queues.

[ 257 ]

Page 269