10장. 함수적 데이터 구조

데이터 구조는 알고리즘의 필수 요소이다. 힙, 큐, 스택, 트리, 그리고 해시 테이블은 프로그래밍 언어에서 널리 사용되는 데이터 구조의 여러 형태이다. 그 중 일부는 트리나 해시 테이블과 같이 주로 검색에 사용되고, 힙, 큐, 스택과 같은 것은 삽입 또는 삭제처럼 업데이트를 위해 사용된다. 10장에서는 평범한 데이터 구조를 함수적 데이터 구조로 확장하기 위한 기초를 수립할 것이다. 10장에서 배울 개념은 다음과 같다.

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:

* 함수적 데이터 구조
* 지연 평가(Lazy evaluation)
* 함수적 큐
* 함수적 스택

<대> 함수적 데이터 구조

함수적 데이터 구조는 주로 함수형 프로그래밍 언어에서 구현되는 데이터 구조의 특별한 형태이다. R은 함수의 생성 및 처리를 위한 도구를 제공함으로써 함수형 프로그래밍을 지원한다. 예를 들어, R은 함수를 변수에 할당하고 그것을 함수 내에서 인수로 전달하는 것을 지원한다. R은 동적으로 함수를 생성하고 함수의 결과로서 그것을 반환하는 클로저 함수(closure function)를 지원한다. 예를 들어, 함수를 인수로 취하는 함수는 다음과 같다.

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))

arg\_function 함수는 mean 이나 sd 같은 함수를 인수로 취하며, 그 코드 예제는 다음과 같다.

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

함수적 데이터 구조는 영구적(persistent) 데이터 구조라고도 한다. 함수적 데이터 구조에서 수행된 모든 작업은 업데이트 된 작업과 함께 데이터 구조의 새로운 복사본을 생성한다는 의미에서 불변이기 때문이다. 함수적 데이터 구조의 원본은 항상 그대로 유지된다. 함수적 데이터 구조는 전통적인 데이터 구조와는 다른 특성을 가지고 있다. 함수적 데이터 구조는 구현시에 매우 유연하며 불변성과 지속성을 지원한다. 또한 멀티쓰레드 환경에서 데이터 구조를 처리할 때도 안정적인 실행을 보장한다(thread-safe).

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.

다음은 불변성(immutability)이 주는 이점들이다.

The following are some benefits of immutability:

* 데이터 은닉 및 데이터 공유를 지원한다. 데이터 은닉은 데이터의 유출 가능성을 방지하고 (함수 내에서 동적으로 생성됨), 데이터 공유는 함수 내에서 필요에 따른 데이터의 자동 공유를 지원한다(포인터 기반).
* 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.
* 함수적 데이터 구조는 변형을 지원하지 않기 때문에 외부 동기화를 할 필요가 없다. 그러나 통제된 변형을 통한 수정(추가 또는 삭제)을 일부 지원한다.
* As functional data structures do not support mutations, it requires no external synchronization. However, some support modifications (additions or deletions) using controlled mutations.
* 포인터가 장거리 연결되는 것을 방지한다.
* The pointers prevent long-distance coupling.

다음은 지속성(persistency)이 주는 이점들이다.

The following are some benefits of persistency:

* 파괴적인 업데이트 대신에 데이터 구조에 대한 건설적인 업데이트를 통해 변형이 일어나도록 한다.
* 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.
* 단순한 복사가 아닌 이전 버전의 공유와 같은 메모리 효율적인 방식의 복제를 지원한다.
* It supports memory-efficient ways of replication, such as sharing of older versions instead of copying.

다음은 쓰레드 안정성이 주는 이점들이다.

The following are some benefits of thread safety:

* 데이터 변형에 대한 걱정 없이 동시성 프로그래밍을 지원한다.
* It supports concurrent programming without any need to worry about data races without mutation
* 새로운 객체를 생성하는 동안, 변형은 포인터가 가리키는 쓰레드 내에서만 수행되며, 원본 데이터는 그대로 유지된다(불변).
* During construction of new objects, mutations are performed only within the thread under consideration using pointers, and the original data is kept intact (immutable)
* 공유 포인터를 사용하여 쓰레드 안전 참조 카운팅을 지원하고, 로그 프리 (log-free)로 최적화 되기 때문에 메모리 효율적이다.
* It is memory-efficient as it supports thread-safe reference counting using shared pointers and is optimized as log-free

<중> 지연 평가

함수형 프로그래밍 언어에서 사용되는 함수적 데이터 구조는 지연 평가를 지원한다. 함수의 인수는 계산 결과가 함수 내에서 더 사용될 경우에만 평가되는 것을 지연 평가라 한다. 더 나아가 인수가 평가되고 나면 계산 결과는 캐시(cache)로 저장되고, 다시 필요한 경우 재계산을 하는 대신에 저장된 결과를 재사용한다. 이런 유형의 캐싱(메모이제이션 memoization 이라고도 함)은 주어진 인수(또는 부분 표현식)가 언제 평가되는지 결정하기 쉽지 않기 때문에 알고리즘의 점근적 복잡도를 추정하기 아주 어렵게 만든다.

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.

지연 평가는 함수적 데이터 구조의 구현에서 핵심적인 역할을 한다. 지연 평가가 포함된 알고리즘의 점근적 성능을 분석하는 것은 매우 어렵다. 하지만 다음 프레임워크는 지연 평가가 포함된 알고리즘의 점근적 성능을 계산하기 위한 기초를 제공한다. 우선, 연산의 비용은 다음과 같이 분류된다.

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:

* 비공유 비용 : 작업의 실제 실행 시간으로 정의된다. 작업을 실행하기 전에 이 작업과 관련되어 연기된 모든 작업이 강제되고 메모된다.
* 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.
* 공유 비용 : 이전에 제외되었던 모든 연기된 작업의 실행 시간으로 정의되며, 현재 실행중인 작업의 성능을 평가한다. 공유 비용은 실현 비용과 미실현 비용으로 더 나눌 수 있다. 실현 비용은 전체 계산 중에 실행된 지연된 작업의 런타임을 평가하는 반면, 미실현 비용은 전체 계산 중에 전혀 실행되지 않는 지연된 작업의 런타임을 평가한다.
* 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.
* 전체 비용 : 지연 평가를 사용하여 구현된 작업의 실제 실행 시간으로 정의된다. 이것은 미실현 비용을 제외한 공유 비용과 비공유 비용의 합이다. 전체 계산 중에 지연된 작업들이 실행되지 않으면 전체 비용의 최소값은 비공유 비용과 같다.
* 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.
* 상각 비용(Amortized cost) : 누적 채무의 개념을 사용하여 계산된 전체 계산의 공유 비용으로 정의된다. 처음에 누적 채무는 0으로 설정되고, 지연이 발생할 때마다 누적된다. 그 다음에 각 작업은 시작되면서 채무 상환을 한다. 채무가 완전히 상환되면 작업에 대한 연기가 강제되고 메모될 수 있다. 여기서 작업의 상각 비용은 작업의 비공유 비용과 각 작업에 의해 지불된 채무의 합이다.
* 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 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.

따라서 누적된 채무가 완전히 상환되어야만 작업 연기가 강제되고 메모될 수 있다. 누적 비용의 총계가 실현된 공유 비용으로 제한되기 때문에 상각 비용은 실제 총 비용을 넘어설 수 없다.

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.

<중> 함수적 스택

스택은 선입후출(First In Last Out, FILO) 형식의 데이터 구조로, 삽입과 삭제는 일반적으로 스택의 최상단에서 발생한다. 이 형식은 깊이 우선 탐색(DFS)에서 널리 사용된다. 이상적인 시나리오에서 삽입, 삭제, 그리고 검색 작업은 매우 적은 시간이 요구되며, 일반적으로 최악의 경우 시나리오에 대해서 O(1)의 점근선을 갖는다. 따라서 최악의 경우 시나리오에서 n 번의 삽입 또는 삭제는 점근선 O(n)을 가지며, 각 작업의 평균 점근선은 O(1)이다.

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)

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).

<중> 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.

<소> 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.

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).

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"

<소> 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:

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")) .

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).

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.

<대> 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.