Stacks and Queues

This chapter will extend the linked list implementation to stacks and queues. Stacks and queues are special cases of linked lists with less flexibility in performing operations than linked lists. However, these data structures are easy to implement and have higher efficiency where such structures are needed. For example, Figure 1.4 , in Chapter 1 , Getting Started shows the implementation of an array with integer data type using stacks and queues. An item can be added (PUSH) or deleted (POP) from a stack from one side only, whereas a queue is an implementation of linear data structure, which allows two sides for insertion (enqueue) and deletion (dequeue). The current chapter will cover array-based and linked list-based implementation of stacks and queues in R. This chapter will cover below topics in detail:

Stacks

Queues

Dictionaries

Array-based stacks Linked stacks

Comparison of array-based and linked stacks Implementing recursion

Array-based queues Linked queues

Comparison of array-based and linked queues

Stacks

Stacks are a special case of linked list structures where data can be added and removed from one end only, that is, the head, also known as the top. A stack is based on the Last In First Out ( LIFO ) principle, as the last element inserted is the first to be removed. The first element in the stack is called the top, and all operations are accessed through the top. The addition and removal of an element from the top of a stack is referred to as Push and Pop respectively, as shown in Figure 4.1 :

Figure 4.1: Example of Push and Pop operation in stacks

A stack is a recursive data structure, as it consists of a top element, and the rest is either empty or another stack. The main ADT required to build a stack is shown in Table 4.1 . This book will cover two approaches – array-based stack and linked stack – to implement the ADT mentioned in Table 4.1 . The implementation is covered using reference classes in R:

Table 4.1 Abstract data type for queue

Array-based stacks

Array-based stack implementation uses the array data structure to store data. Similar to an array-list, an array-based stack uses a fixed size. Thus, the class definition for an array- based stack would be similar to an array-based list, as shown in the following code:

Astack <- setRefClass(Class = "Astack",

fields = list(

Maxsize="integer", topPos="integer", ArrayStack="array"

),

methods = list(

# Initialization function

initialize=function(defaultSize=100L,...){

topPos<<-0L

Maxsize<<-defaultSize # 100L

ArrayStack<<-array(dim = Maxsize)

},

# Check if stack is empty isEmpty=function(){},

# push value to stack

push=function(pushval){},

# Pop value from stack pop=function(){},

# Function to get size of stack stacksize=function(){},

# Function to get top value of stack top=function(){}

) )

In reference classes, fields are modified using <<- (global operator), and all functions are accessible to objects created using the defined class. The preceding class implements a stack with a default array size of 100 cells. The pointer topPos points to the top element of the stack, and Maxsize refers to the maximum size of the array. To check whether the stack is empty or not, we can use the top position in an array stack, as follows:

isEmpty=function(){

if(topPos==0) {

cat("Empty Stack!") return(TRUE)

} else {

return(FALSE)

} }

The push and pop operation of an array stack can be performed by working with the topPos pointer of the class to update the array index:

push=function(pushval){

if((topPos+1L)>Maxsize) stop("Stack is OUT OF MEMORY!") topPos<<-topPos+1L

ArrayStack[topPos]<<-pushval

}

pop=function(){

# Check if stack is empty

if(isEmpty()) return("Empty Stack!") popval<-ArrayStack[topPos] ArrayStack[topPos]<<-NA topPos<<-topPos-1L return(popval)

}

Pushing an element to a stack which is completely full is known as overflow, whereas removing an element from a stack which is empty is referred to as underflow of stack. Thus, both these conditions are added as exceptions in the current class using the isEmpty function Maxsize variable. The size of the array stack can be obtained by returning the topPos variable:

stacksize=function(){ stackIsEmpty<-isEmpty()

ifelse(stackIsEmpty, return(0), return(topPos))

}

Similarly, the top value of a stack can be returned by returning the value pointed to by topPos into ArrayStack :

top=function(){

stackIsEmpty<-isEmpty() if(stackIsEmpty) { cat("Empty Stack")

} else {

return(ArrayStack[topPos])

} }

Due to the simplicity of the structure and implementation of stacks, it is possible to implement multiple stacks within the same initialized array. However, the current implementation is recommended if stacks have an inverse relationship, or if there is a functional relationship which could be used to minimize the array memory. For example, a two-stack system in which the first stack gets data from the pop operation performed on the second stack can be utilized to develop a multi-array stack within the same array, as shown in Figure 4.2 .

In Figure 4.2 , Stack1top and Stack2top represent the pointers to the first and second stacks, respectively. As Stack1top moves toward the right, Stack2top moves toward the left, and vice versa. For other scenarios where the memory required is not predefined, a linked stack can be used, as discussed in the next subsection:

Figure 4.2: Example of a multi-array stack

An example of the use of an array-based stack with multiple push and pop operations is as follows:

> array\_stack\_ex<- Astack$new() > array\_stack\_ex$push(1) > array\_stack\_ex$push(2) > array\_stack\_ex$push(3) > array\_stack\_ex$pop() > array\_stack\_ex$push(5) > array\_stack\_ex$pop() > array\_stack\_ex$pop() > array\_stack\_ex$top() [1] 1

> array\_stack\_ex

Reference class object of class "Astack" Field "Maxsize": [1] 100

Field "topPos": [1] 1

Field "ArrayStack":

[1] 1 NA NA NA NA NA NA NA ... NA NA NA

Initially, we push three elements into the stack {1, 2, 3} . We then pop one element out using the LIFO principle, and are, thus, left with the set {1,2} . We then push another element 5 into the stack, updating the set as {1,2,5} . Finally, we pop the top two elements, leaving the stack with only one element, {1} .

Linked stacks

Linked list-based stacks utilize the concept of linked lists with the flexibility to add and remove elements dynamically from the head of a linked list, which is equivalent to the top in an array-based stack. The top points to the first member of the linked list, as shown in Figure 4.3 :

Figure 4.3: A linked stack with the top pointing to the head node of a linked list

The reference class definition of a linked list-based stack is shown as follows:

Linkstack <- setRefClass(Class = "Linkstack",

fields = list(

Lsize="integer",

Lstacktop="environment"),

methods = list(

# Initialization function initialize=function(...) {

Lsize<<-0L

},

# Check if stack is empty isEmpty=function(){},

# Function to create empty R environment create\_emptyenv=function(){},

# Function to create node

Node = function(val, node=NULL) {},

# push value to stack

push=function(pushval){},

# Pop value from stack pop=function(){},

# Function to get top value of stack top=function(){}

) )

As nodes are created dynamically, memory allocation is not required; thus, Lstacktop is defined as an environment variable, and points to the top location of the stack. The Lsize variable stores the stack's size. The fundamental node of a linked list comprises the value and address of the next node. Let's use the same ADT as defined earlier for the array-based stack. We could utilize the same isEmpty function defined for the array stack by replacing topPos with Lsize :

isEmpty=function(){

if(Lsize==0) {

cat("Empty Stack!") return(TRUE)

} else {

return(FALSE)

} }

The node in the linked stack can be defined as an environment object similar to a linked list as defined in Chapter 3 , Linked Lists :

create\_emptyenv = function() {

emptyenv()

}

Node = function(val, node=NULL) {

llist <- new.env(parent=create\_emptyenv()) llist$element <- val llist$nextnode <- node llist

}

The node method consists of element , which stores the value, and nextnode , which points to next node of the linked list. The push method will add a node to the stack, whereas the pop method will remove the top node of the stack:

push=function(val){

stackIsEmpty<-isEmpty() if(stackIsEmpty){

Lstacktop<<-Node(val) Lsize<<-Lsize+1L

} else {

Lstacktop<<-Node(val, Lstacktop) Lsize<<-Lsize+1L

} }

The push function initially checks whether the stack is empty or not. If the stack is empty, it creates a new node, otherwise it adds the newly created node to the top position of the linked list. As accessing the top position or head node of a linked list is quite

straightforward, we do not require to define a separate top variable pointing to the head node in R; thus, top is used as a reference to the head node in linked list stack definitions.

pop=function(){

stackIsEmpty<-isEmpty() if(stackIsEmpty){ cat("Empty Stack")

} else {

Lstacktop<<-Lstacktop$nextnode Lsize<<-Lsize-1L

} }

The pop function also checks for the empty condition, and moves the top position to nextnode using the address pointer if the stack is non-empty. The other functionality of stacks can be built around their basic ADT, such as getting the top value of a stack:

topVal=function(){

stackIsEmpty<-isEmpty() if(stackIsEmpty){ cat("Empty Stack")

} else {

return(Lstacktop$element)

} }

The preceding function returns element from the top node of a stack. This function can be used to set up a list stack. The following is an example of the use of a linked list stack with multiple push and pop operations:

> link\_stack\_ex<-Linkstack$new() > link\_stack\_ex $push(1) > link\_stack\_ex $push(2) > link\_stack\_ex $push(3) > link\_stack\_ex $pop() > link\_stack\_ex $push(5) > link\_stack\_ex $pop() > link\_stack\_ex $pop() > a$topVal() [1] 1

> link\_stack\_ex

The preceding example is similar to the array-based implementation. After all the operations have been performed in this last example, array\_stack\_ex object has one value in the stack, as shown in the output stored in environment 0x00000000405fc248 .

Comparison of array-based and linked stacks

From the perspective of time computation, both array and linked list implementations of stacks are quite comparable. For example, the cost of appending and deleting in both arrays and linked list stacks is O(1) -worst-case. In a linked list-based implementation, appending and deletion is performed by the head pointer, which can be accessed directly. Similarly, in an array-based implementation, each push and pop is performed through the topPos index variable, which keeps moving making access in constant time.

In terms of space, in an array-based stack implementation, preallocation of memory is required during array initialization; thus, (n-m) cells are wasted, where m is the number of elements stored in the array. On the other hand, a linked list stack implementation dynamically allocates and deallocates memory with every push and pop operation respectively, and thus, no memory is wasted. However, a linked list implementation requires an extra nextnode field to store the address of the next node.

Implementing recursion

Recursion is used to implement an iterative process, where each state of a variable in a subroutine is registered by the controller. The primary memory, while executing the code, utilizes four main components, as shown in Figure 4.4 :

Figure 4.4: Memory allocation

The code part of the memory stores instructions and functions from the program. The static variable stores any global and static variable from the program. The stack stores the variables and data of the function. The heap is used for dynamic memory allocation. For example, a subroutine with data and instructions will register into memory, as shown in Figure 4.5 . The subroutine variables x and y are stored in the Stack , whereas instructions are stored in the Code section of memory allocation. When implemented as a recursion, the machine executes a subroutine repeatedly by changing the contents of the registers until the termination condition is reached. At each step, the state of the subroutine is determined based on the register content in the stack, which comprises variable states:

Figure 4.5: Example of memory allocation

The recursive function should always have a termination criterion to be used in real-world applications. Anything which has a self-similar structure with repetition can be

implemented using recursion. For example, the factorial of any non-negative integer for any number n is represented as n! , and can be expressed as follows: factorial(n) = n\*factorial(n-1) where factorial(0)=1 .

The factorial function is self-similar, as it calls itself until it reaches a value factorial(0) . The following represents the R implementation of factorial using recursion:

recursive\_fact<-function(n) {

if(n<0) return(-1) if(n == 0) {

return(1)

} else {

return(n\*recursive\_fact(n-1))

} }

The preceding factorial function uses recursion for evaluation, and uses a stack for computation. For example, for the factorial of 3 , the preceding function will keep pushing values to the stack until the termination condition is achieved (as shown in Figure 4.6 ) before calculating the factorial by popping the stored value from the stack. The function will create a stack of all the values to be multiplied before popping these values from the stack to perform the final multiplication operation to give the factorial value for the given integer:

Figure 4.6: Example of recursion for factorial of 3

The recursion approaches are quite efficient in implementing algorithms that require multiple branching, such as binary trees. The details of these algorithms will be covered in a later part of this book.

Queues

A queue is an ordered collection of elements as shown in Figure 1.4(b) in Chapter 1 , Getting Started . In queues, addition is restricted to one end, referred to as rear , and deletion is restricted to another end, which is known as the front . Queues follow the First In First Out ( FIFO ) principle, also known as the first-come-first-served approach. Thus, an element pushed into a queue will wait until all the elements in front are removed. The queue data structure can be applied to any shared resources scenario. For example, in a network printer case where multiple users are sending printing jobs to the same printer, the jobs are arranged in a queue, and are processed in order of arrival. Another example of a queue from our day-to-day life is a shop counter serving multiple people – they use a queue for serving and, thus, follow the FIFO principle in serving the people in the queue. Also, databases accessed by multiple departments/users also use queues to process their queries on data in the order of their arrival. Thus, queues have a lot of application in different domains.

The major operations required by a queue are adding an element (enqueue), deleting an element (dequeue), and size of the queue as defined as ADT requirement in Table 4.2 :

Table 4.2 Abstract data type for queue

Array-based queues

The array-based implementation of queues is not an efficient implementation, as we select one side of a queue to add an element and other side to remove. The task can be accomplished by using two pointers – front and rear. An element is added to the front and removed from the rear of the queue. This leads to a drifting issue, as shown in Figure 4.7 :

Figure 4.7: Drifting issue with queue implementation using array

In Figure 4.7 , it can be seen that there could be a situation when the queue is full, yet there is free space available in the array:

Figure 4.8: Approach 1 to address drifting issue in queue

The problem can be resolved by keeping the rear at the first position, and moving the rest of the array towards the rear by one unit, as shown in Figure 4.8 . However, this makes the removal operation O(n) , which is computationally inefficient. Another way to tackle this problem is by using a circular implementation of queue, as shown in Figure 4.9 . Circular implementation allows reusing of empty cells once the array length ends. This implementation makes addition and removal operations O(1) , which is quite efficient computationally. However, this introduces another challenge related to determining whether the queue is full or empty, as in both situations, empty and full queue, the rear will hold a position less then the front pointer. The current problem can be addressed by keeping track of the number of elements in the queue, or creating an array with n+1 to store n elements:

Figure 4.9: Circular array implementation of queue

Let's implement a queue using reference classes in R. The ADT implementation of a queue in R is shown as follows:

aqueue<-setRefClass(Class = "aqueue",

fields = list( Alist="array",

queuesize="integer", maxSize="integer", rear = "integer", top = "integer"

),

methods = list(

initialize=function(qSize, ...){

queuesize<<-0L rear<<-1L top<<-0L

maxSize<<-as.integer(qSize) Alist<<-array(dim = maxSize)

}

# Queue is empty

isEmpty = function() {}, # Add element to the queue enqueue = function(val){}, # remove element from queue dequeue = function() {},

# size of queue

size = function() {}

) )

The new reference class can be generated using setRefClass() , and the method can be created using a method list within setRefClass . The new queue can be created using the new() function:

> q<-aqueue$new() > q

Reference class object of class "aqueue" Field "Alist":

[1] NA NA NA NA NA NA NA NA NA NA NA NA NA ... NA NA NA NA NA Field "queuesize": [1] 0

Field "arraySize": [1] 100

Field "maxSize": [1] 100

Field "rear": [1] 0

Field "top": [1] 0

The other implementation from ADT can be added to the queue class by adding methods. That the queue is empty can be checked using the queuesize variable, as follows:

isEmpty = function() { return(queuesize==0L)

}

For adding and deleting an element in a queue, the methods enqueue() and dequeue() can be used respectively in the method list of setRefClass :

enqueue = function(val){ if(queuesize<maxSize){

if(top==maxSize) top<<-0L top<<-top + 1L

Alist[top]<<-val

queuesize<<-queuesize+1L

} else {

cat("Queue Full!")

}

},

dequeue = function() {

if(queuesize>0L){ Alist[rear]<<-NA

ifelse(rear==maxSize, rear<<-1L, rear<<-rear+1L) queuesize<<-queuesize-1L

} else {

cat("Empty Queue!")

} }

The preceding function is a circular implementation; thus, the top and rear position is reset to the start of the array once the top and rear pointer hits the maxSize index.

Linked queues

Linked queues are a much simpler implementation, as nodes are dynamically created and destroyed. In linked list queues, an element is inserted at the rear and removed from the front, as shown in Figure 4.10 :

Figure 4.10: Example of link list queue

The class implementation of the queue ADT in R using reference classes for a linked list- based queue is shown as follows:

ListQueue <- setRefClass(Class = "ListQueue",

fields = list(

Lsize="integer",

front="environment", rear = "environment", Lqueue="environment"),

methods = list(

initialize=function(...) {

Lsize<<-0L

},

# Check if list is empty isEmpty=function(){},

# create empty environment

create\_emptyenv = function() {},

# Create node

Node = function(val, node=NULL) {},

# Function to add value to link list enqueue=function(val){},

# Function to remove node from link list dequeue=function(){}

# Function to get link list size

size=function(){}

) )

The function isEmpty checks whether the linked list is empty using the Lsize variable, as shown in the following code snippet. For an empty linked list, Lsize has zero value:

isEmpty=function(){

if(Lsize==0) {

cat("Empty Stack!") return(TRUE)

} else {

return(FALSE)

} }

The link list node in R is represented using environment; thus, the create\_emptyenv function creates an empty environment:

create\_emptyenv = function() {

emptyenv()

}

The node representation is similar to the linked list node, and consists of element and nextnode .

Node = function(val, node=NULL) {

llist <-new.env(parent=create\_emptyenv()) llist$element <- val llist$nextnode <- node llist

}

As the elements in a queue are added to the rear of the queue, the rear pointer is used to capture the environment location for the last node as follows:

enqueue=function(val){ ListIsEmpty<-isEmpty() if(ListIsEmpty){

Lqueue<<-Node(val) Lsize<<-Lsize+1L rear<<-Lqueue } else {

newNode<-Node(val)

assign("nextnode", newNode, envir = rear) rear<<-newNode

Lsize<<-Lsize+1L

} }

The assign statement is used to attach the reference of a new node using the rear pointer reference. As elements are deleted from the front node, the front pointer is not necessary, and the first element can be accessed and removed directly, as shown in the following code snippet:

dequeue=function(){

stackIsEmpty<-isEmpty() if(stackIsEmpty){ cat("Empty Queue")

} else {

Lqueue<<-Lqueue$nextnode Lsize<<-Lsize-1L

} }

The size of the linked list is contained in the Lsize variable of the class function.

Comparison of array-based and linked queues

The linked list implementation of queues require O(1) (worst-case) computation effort, where enqueuing is performed by appending to the rear, and dequeuing is implemented at the head of the linked list. However, new allocation is required with every operation, which may make it slow.

The enqueuing operation in array-based queues is implemented by using a circular buffer, which works as inserting element at the next free position. The implementation can be a dynamic array implementation, where a new array is created with a bigger size when the max memory is reached in the current array. The enqueuing and dequeuing operations are performed using front and rear references, thus requiring O(1) computational effort. In terms of memory allocation, queues behave similar to stack implementations – a linked implementation needs an extra nextnode field to store the address of the next node, thus increasing the memory overhead.

Dictionaries

A dictionary can be defined as an ordered or unordered list of key-element pairs, where keys (usually unique) are used to locate elements (not necessary unique) in the data structure. For example, a data structure that stores customer information in a retail shop can be considered as a dictionary, where the consumer ID serves as the key for identification of different customers. Dictionaries are also known as associative arrays or maps, as they map keys to values to perform retrieval operations such as addition, removal, and search. Every element of a dictionary consists of a key and an associated element, also known as a key- value pair. This is shown in Figure 4.11 :

Figure 4.11: Dictionary key-value pair structure

The key in a dictionary is used to differentiate between each key-value pair. It can be any randomly chosen set of values, such as real numbers or strings, with the only restriction that each key is unique, and can be differentiated from the others. The values in the dictionary are also known as vocabulary.

The standard ADT for dictionaries are as follows:

Table 4.3 Abstract data type for dictionaries

The class for the preceding ADT can be implemented using array-based data structures, as shown in the following code:

Adict<-setRefClass(fields = list(

Alist="list",

listsize="integer", key="integer"

),

methods = list(

# Re-initialize dictionary initialize=function(...){

listsize<<-0L Alist<<-list()

},

# Check length of value size = function(){},

# Add following key value pair in Array addElement = function(key, val){},

# remove value with defined

removeElement = function(key){},

# remove value with following findElement = function(key){},

}

The size function determines the current size of an array. The addElement and removeElement functions add and remove elements respectively. The findElement function determines the value of the provided key. The dictionary can be implemented in one of the following two ways:

Ordered

Unordered

The unordered implementation adds items as they arrive; thus, the addElement function takes O(1) computational effort. However, the findElement and removeElement methods take O(n) computational effort. Therefore, this implementation is recommended if the number of additions of elements is much larger than that of removal. In the ordered implementation of dictionaries, items are added to the initially empty dictionary in a non- decreasing order of their keys. Thus, the addElement function will take O(n) computational effort in the worst-case scenario. In the ordered implementation, the removeElement function will also take O(n) computational effort, as any removal requires gaps created by the operation to be filled. Thus, this implementation is inferior to an unordered dictionary in terms of adding and removing elements. However, the efficiency of the search operation, findElement , is considerably improved by using search strategies such as binary search. Thus, in scenarios where the database is static with very little addition or removal of data points and where mostly search operations are required, then ordered implementation is a better choice.

Let's implement an unordered implementation using the list data type from R. The size function can be implemented by monitoring the listsize variable, as shown in the following code snippet:

size = function(){

return(listsize)

}

The addElement function can be implemented by passing the key and value to the list data type, and the removeElement function utilizes the key to retrieve the element before deleting it from the list:

addElement = function(key, val){

Alist[[key]]<<-val

listsize<<-listsize+1L

}

removeElement = function(key){

Alist[[key]]<<-NULL

listsize<<-listsize-1L

}

Finding a value in a dictionary can be performed using a key search. The keys in the list data type in R are stored as names of the list:

findElement = function(key){ return(key%in%names(Alist))

}

Finding elements based on a key can be a very simple query just based on ID comparison, and using basic operators such as == , >= , and <= if the key is an integer. However, if the key is a character, %in% in R could be used to check whether the key is present in the set of keys available in the data structure. Based on the previous functions, let's create an example with characters as keys:

> dictvar<-Adict$new()

> dictvar$addElement("key1", 1) > dictvar$addElement("key2", 1) > dictvar

Reference class object of class "Adict" Field "Alist": $key1 [1] 1 $key2 [1] 1

Field "listsize": [1] 2

Field "key": integer(0)

> dictvar$Size() [1] 2

> dictvar$findElement("key1") [1] TRUE

> dictvar$removeElement("key1")

The preceding implementation can be obtained using other data structures, such as linked list, with minor updates.

Exercises

1. What will be the top value for the following stack operations?

PUSH(1) PUSH(2) PUSH(6) PUSH(3)

POP() POP()

2. Assume a stack with the elements {1, 2, 3, 5, 6, 7} in order, with 7 on top. Write the

operations required to insert 4 after 3 in the current stack.

3. Explain the difference in the outputs of the following recursion functions:

Output 1 Output 2

problemfun1<-function(n){ problemfun1<-function(n){

if(n<1) return(1) problemfun1(n-1) print(n)

if(n<1) return(1) print(n)

problemfun1(n-1)

}

}

4. Write a recursive algorithm to evaluate the Fibonacci sequence. (In a Fibonacci

sequence, each item is the sum of the previous two.) 5. Write a function to invert the values in a stack.

6. Write a function to implement two stacks using only one array. The routines

should not indicate an overflow unless every cell in the array is filled.

7. Create a data structure which supports push, and pop, and finds the maximum

values, all in O(1) worst-case scenario.

8. Write a class for an array queue implementation using stacks, assuming no other

data type is available.

Summary

The current chapter covered the fundamentals of stacks and queues and also introduced implementation using R reference classes. We covered the fundamentals of stacks as a data structure which is based on the LIFO principle. The two types of stack implementation were introduced – array-based and linked list-based stacks along with a comparison of their computational and memory efficiencies. Recursion-based functions utilize stacks inherently, and are covered within the stack functionality. Queues are another very useful data structure that we covered. They follow the FIFO principle in addition and deletion of elements from the data structure. We discussed two types of queues – array-based queue and linked list-based queue implementation. In addition, we learned about dictionaries, an interface ADT for retrieving data from a data structure. The chapter also covered the array- based implementation of dictionaries.

[ 105 ]