3장. 링크드 리스트

This chapter will cover element and homogeneous data type vectors in more detail. The chapter will move from contiguous memory allocation to a non-contiguous memory allocation data type such as a linked list. The linked list data structure collects data and orders them relative to the other elements that come before and after it. The linear data structure can be thought of as having two ends, and the way an item is added or removed from the linear structure distinguishes one structure from another. The chapter will cover multiple variants of linked lists, such as linear linked lists, doubly linked lists, and circular linked lists. The chapter will introduce below mentioned topics in detail:

* Built-in data types in R, such as vector, and element data types
* Writing object-based programs using R S3, S4, and references classes
* Array-based list implementation
* Linked lists
* Comparison of list implementations
* Element implementations
* Doubly linked lists
* Circular linked lists
* Vector and atomic vector

Data types in R

Before we get into data structure concepts, let's look into data types provided by the R programming language. A basic data structure with a homogenous data type is based on a contiguous sequence of cells to enable fast access to any particular dataset. All homogeneous types support a single data type.

Page 71

﻿

Linked Lists

For example, in Figure 3.1 we have a numeric, logical, and character data type, however, it is stored as character.

Figure 3.1: Example of vector stored as character

Similarly, a matrix with multiple data types, as shown in Figure 3.2 , will be coerced and stored as character data type. The array is an extension of the matrix from 2-D to n-D.

Figure 3.2 : A matrix with numeric and characters are stored as 2D matrix with characters data type

All elements of a homogeneous data structure must be the same type, so R attempts to combine the different data types to the most flexible type in a priority order as shown in Figure 3.3 :

Figure 3.3: Priority order of data types during coercion

[ 60 ]

Page 72

﻿

Chapter 3

Based on Figure 3.3 , you can see that the character data type gets most priority in

homogeneous data type. Logical gets converted into integer, and integer gets converted into numeric if the data type is homogenous. All these are built-in data structures. The following table shows coercion of different data types:

Figure 3.4: The table shows diﬀerent types of vector coercions

Vector and atomic vector

The vector representation stores elements in contiguous memory allocation, and the cells are accessed through indexing such as v[2] denotes second element of vector v . R has six basic atomic vectors.

[ 61 ]

Page 73

﻿

Linked Lists

The following table lists all six basic atomic vectors with their modes and storage modes:

Figure 3.5: The table shows diﬀerent modes and storage modes of atomic vector types

Contiguous memory allocation and access through indexing makes any insertion or deletion quite expensive. For example, say a company wants to manage the current working employees' details such as name, gender, age, department, and so on. For simplicity, let's only consider that we want to use a vector representation for storing the employee name. Let's assume m employees are currently present in the company, and a new employee named Navi joins the company. As employee names are stored in a sorted order, they have to store it after Bob, as shown in the following figure:

Figure 3.6: An example of insertion in vector

To perform this operation, all employee names need to be shifted by one, leading m-k operations to be performed where k is the insertion position in the vector. Similarly, in a deletion operation, all elements needs to be shifted back as shown in figure below:

Figure 3.7: An example of deletion of element from vector

[ 62 ]

Page 74

﻿

Chapter 3

Element data types

R supports various element data types, which have unique properties associated with them. Atomic vectors are the most elemental data types as covered under the preceding section. Other forms of data types are as follows:

Factor

A factor is a vector of integer values labeled to each corresponding set of unique characters in a categorical vector. The content within a factor can be of numeric or character format. The content can be of multiple forms such as character, numeric, logical, and complex. The following example shows how the characters within a categorical vector have been uniquely auto-assigned to an integer (factors). The integer levels are assigned based on the sequence of occurrence of unique character elements within a vector.

> fact1 <- factor(c("a","b","c","c","a","b")) > fact1

[1] a b c c a b Levels: a b c > str(fact1)

Factor w/ 3 levels "a","b","c": 1 2 3 3 1 2

However, it is possible for a user to define levels as per requirements:

> fact2 <-

factor(c("a","b","c","c","a","b"),labels=c(1,2,3),levels=c("c","a","b "))

> fact2

[1] 2 3 1 1 2 3 Levels: 1 2 3 > str(fact2)

Factor w/ 3 levels "1","2","3": 2 3 1 1 2 3

Matrix

A matrix is a two-dimensional array vector defined as rows and columns with homogenous content, which can be of multiple forms such as character, numeric, logical, and complex.

[ 63 ]

Page 75

﻿

Linked Lists

The following example shows a numeric and a character matrix. A mode() is used to check the data type as present in the R environment.

## Numeric Matrix

> mat1 <- matrix(1:10,nrow=5) > mat1

[,1] [,2]

[1,] 1 6 [2,] 2 7 [3,] 3 8 [4,] 4 9 [5,] 5 10 > mode(mat1) [1] "numeric"

## Categorical Matrix

> mat2 <- matrix(c("ID","Total",1,10,2,45,3,26,4,8),ncol=2,byrow=T) > mat2

[,1] [,2]

[1,] "ID" "Total" [2,] "1" "10" [3,] "2" "45" [4,] "3" "26" [5,] "4" "8" > mode(mat2)

[1] "character"

Array

An array is an n dimensional vector with homogenous content, which can be of multiple forms such as character, numeric, logical, and complex.

The following example shows how to generate a three-dimensional array:

> arr1 <- array(1:18,c(3,2,3)) > arr1 , , 1

[,1] [,2]

[1,] 1 4 [2,] 2 5 [3,] 3 6 , , 2

[,1] [,2]

[1,] 7 10 [2,] 8 11 [3,] 9 12 , , 3

[,1] [,2]

[ 64 ]

Page 76

﻿

Chapter 3

[1,] 13 16 [2,] 14 17 [3,] 15 18

The c(3, 2, 3) column vector defines the dimension of array in such a way that length of the column vector defines the dimension of the array and the values of the column vector define grid size. In this case, X has 3 units, Y has 2 units and Z dimension has 3 units.

Dataframes

A dataframe is a two-dimensional table with combinations of multiple forms of vectors (heterogeneous content) of equal length. It possesses properties of both list and matrix. The content can be of multiple forms such as character, numeric, logical, and complex. The following is a dataframe with five observations and four attributes:

> Int <- c(1:5); Char <- letters[1:5];

Log <- c(T,F,F,T,F); Comp <- c(1i,1+2i,5,8i,4) > data.frame(Int,Char,Log,Comp)

Int Char Log Comp 1 1 a TRUE 0+1i 2 2 b FALSE 1+2i 3 3 c FALSE 5+0i 4 4 d TRUE 0+8i 5 5 e FALSE 4+0i

List

A list is a way of grouping all possible objects (including lists themselves) and assigning them to a single object. It has a one-dimensional property, which can take in heterogeneous objects. It is also called recursive, as it can contain multiple lists within one. The content can be of multiple forms such as character, numeric, logical, and complex.

The following code explains how to create a list, and what it it looks like in the R environment:

> list1 <- list(age = c(1:5), #numeric vector

+ name = c("John","Neil","Lisa","Jane"), #character vector + mat = matrix(1:9,nrow = 3), #numeric matrix

+ df = data.frame(name = c("John","Neil","Lisa","Jane"), gender = c("M","M","F","F")), #data frame

+ small\_list = list(city = c("Texas","New Delhi","London"), country = c("USA","INDIA","UK"))) #list > list1

[ 65 ]

Page 77

﻿

Linked Lists

$age

[1] 1 2 3 4 5 $name

[1] "John" "Neil" "Lisa" "Jane" $mat

[,1] [,2] [,3]

[1,] 1 4 7 [2,] 2 5 8 [3,] 3 6 9 $df

name gender 1 John M 2 Neil M 3 Lisa F 4 Jane F $small\_list

$small\_list$city

[1] "Texas" "New Delhi" "London" $small\_list$country

[1] "USA" "INDIA" "UK"

R also supports multiple ways to implement data types using object-oriented

programming ( OOP ) such as S3, S4, and R5 classes. The next section provides the basics on OOP, which will later be used for implementing different data structures.

Object-oriented programming using R

As you already know, R is primarily a functional language; it also supports OOP. OOP in R is an archetype wherein objects and their interactions are used to design various generic functions. It defines the process of constructing modular bits of code, which can be integrated to form a large function. Some key concepts related to OOP are as follows:

Object : An instance of a class or an output of a function in R Class : Used to define type and attributes of objects in R

Method : An implementation of a generic function for an object of a particular class

Generic function : A generalized function which calls multiple methods without performing any computation itself

[ 66 ]

Page 78

﻿

Chapter 3

R supports three forms of OOP systems based on different objects, classes, and methods:

S3 : An informal, simple, interactive, and widely used OOP system in R. Basic packages such as base and stats are primarily built using the S3 system. The following are some generic functions built for multiple objects such as

dataframes, vectors, or the output of lm() function for its corresponding method:

Figure 3.8: Generic S3 functions for diﬀerent methods

S4 : Unlike S3, S4 is much more formal, robust, and provides a uniform mode to create objects. Also, the generic function can be dispatched multiple times to pick methods based on the class of any number of arguments. In S4, new objects are created using the new() function, and class components are defined using the setClass() function. A class has three main properties:

a name : This is an alphanumeric string used to identify the class. a representation : This is used to define a list of attributes (or slots) along with their data types. For example, an employee class of shop will be represented by a name represented as character , age as numeric , and gender represented as character as shown below:

representation(name="character", age="numeric", gender="character")

contains or character vector : a vector of classes used for multiple inheritance. Caution should be taken while using contains in S4 as it makes method lookup intricate.

[ 67 ]

Page 79

﻿

Linked Lists

R5 (Reference classes) : Unlike S3 and S4 which implement generic functions, R5 implements message passing object-oriented programs similar to other object- oriented programs such as Java, C++, and C#, where methods belong to classes rather than functions. R5 objects are also mutable, as they are not dependent on R's modify semantics.

The following table compares the S3, S4, and R5 systems:

Figure 3.9: A comparison between S3, S4 and R5 OOs

Linked list

A list can be defined as a collection of a finite number, or as sequence of data items known as elements. Each element of a list will have a specific data type-in the simplest scenario, all elements of a list have the same data type. In R, list implementation is essentially arrays of R objects (SEXP). The array-based implementation of lists will be discussed in the next section. To implement the list data structure, we will use environments, also known as objects in R:

# Example list with array, data.frame, matrix, and character

> elist <- list(vec=1:4,df=data.frame(a=1:3, b=4:6),mat=matrix(1:4, nrow=2), name="pks") > elist[["vec"]] [1] 1 2 3 4

[ 68 ]

Page 80

﻿

Chapter 3

In a linked list, each item holds a relative position with respect to the others. In a list, there is no requirement for contiguous memory, thus, data can have non-contiguous allocation. For example, Figure 3.6 shows the implementation of contiguous and non-contiguous memory allocation:

Figure 3.10: Memory allocation

In non-contiguous memory allocation, data is stored at random locations. To effectively use non-contiguous memory allocation, the data structure needs to be embedded with the file system, as shown in Figure 3.7 . Linked lists store this collection by linking each cell in an ordered format. The start and end of a linked list are also referred to as head and tail respectively.

Figure 3.11: An example of a linked list

[ 69 ]

Page 81

﻿

Linked Lists

Linked lists can be of different types such as:

Linear linked list Doubly linked list Circular linked list

Also, a linked list can be defined based on how the elements are arranged. For example, a linked list positioning the elements in a sorted order is known as a sorted list, whereas a linked list with no pattern between the element value and its position is referred to as an unsorted linked list.

Linear linked list

A linear linked list is also known as one-way list or singly linked list. A singly linked list is a sequence of nodes, where each node stores an element and a link to the next node, as shown in Figure 3.12 The elements in a singly linked list may or may not be stored in consecutive memory locations, so pointers are used to maintain a linear order.

Figure 3.12: Singly link list building block and example

Each node in a linked list consists of an element field and a next field. The Element field of a linked list stores the item value and Next points to the next node. In the last node, Next points to NULL value. Before getting into implementation of singly linked lists, we should focus on defining the ADT requirements for linked lists.

[ 70 ]

Page 82

﻿

Chapter 3

Figure 3.13: An example of ADT for linked list

The ADT may depend on the problem requirement. The first item in ADT is to set up an environment:

create\_emptyenv <- function() {

emptyenv()

}

The linked list can also be represented as an ordered tuple where e n is the n th term in the linked list. The empty link is represented by the tuple notation <> . The

create\_emptyenv() function creates an empty environment, which can hold a collection of named objects and a pointer to an enclosing environment. Before creating a new list, the isEmpty() function checks if the list is empty or not, using an identical function from R.

isEmpty <- function(llist) {

if(class(llist)!= "linkList") warning("Not linkList class") identical(llist, create\_emptyenv())

}

The next step is to define a linked list node as shown in Figure 3.6(a):

linkListNode <- function(val, node=NULL) { llist <- new.env(parent=create\_emptyenv()) llist$element <- val llist$nextnode <- node

class(llist) <- "linkList" llist

}

[ 71 ]

Page 83

﻿

Linked Lists

In the linkListNode() function, an element contains element and nextnode . The element field stores the item value, and nextnode points to the next linked list node. An example of a linked list can be created using the linkListNode function as follows:

LList <-linkListNode(5,linkListNode(2,create\_emptyenv()))

The constructed list can be dynamically expanded by adding and deleting nodes. The elements and nodes in a linked list can be accessed using functions, as follows:

setNextNode<-function(llist){

llist$nextnode

}

setNextElement<-function(llist){

llist$element

}

The next part of ADT is to get the size of the linked list. The size of a linked list requires a pointer to scan through the linked list. The scanning is implemented using recursion in R.

sizeLinkList<-function(llist, size=0){

if (isEmpty(llist)) {

return(size)

} else {

size<-size+1L

sizeLinkList(llist$nextnode, size)

} }

The sizeLinkList function starts from the first position, and keeps scanning the list nodes till it finds an empty environment. Similarly, the addition of an item can be performed at the start, end, or at any position in the linked list. To add a linked list node at the start, just connect the pointer to the existing linked list as shown in Figure 3.14 Similarly, add a linked list node at the end by updating the empty pointer to the newly created node. To add an element in between, the node needs to be updated as shown in Figure 3.15 .

[ 72 ]

Page 84

﻿

Chapter 3

Figure 3.14: Addition of an element to an existing link list

The implementation of insertion at start is done as follows:

addElement<-function(new, llist) {

if (isEmpty(llist)) { llist<-linkedlist(new)

} else {

llist<-linkListNode(llist, new)

}

llist

}

Figure 3.15: Addition of an element in between items in the list

[ 73 ]

Page 85

﻿

Linked Lists

The deletion implementation follows a similar principle as addition (shown in Figure 3.15 ) by skipping the node to be deleted, and updating links accordingly:

delElement<-function(llist, pos=NULL){

if(is.null(pos)) warning("Nothing to delete") listsize<-sizeLinkList(llist)

if(pos>listsize) stop("Position greater than size of list") if (isEmpty(llist)) { warning("Empty List")

} else if(pos==1){

PreviousNode<-llist$nextnode

} else {

PreviousNode<-linkListNode(llist$element) for(i in 1:(listsize-1)){

if(pos==(i+1)){

PreviousNode$nextnode<-setNextNode(llist$nextnode)

} else {

PreviousNode$nextnode<-llist$nextnode llist<-llist$nextnode

} } }

return(PreviousNode)

}

Searching for an item can be implemented by recursively scanning through the linked list from the starting position till the end:

findItem<-function(llist, item, pos=0, itemFound=FALSE){

if (itemFound==TRUE) {

return(itemFound)

} else if(isEmpty(llist)){

return(FALSE)

} else {

pos<-pos+1L

if(llist$element==item) itemFound<-TRUE

findItem(llist$nextnode, item, size, itemFound)

} }

The function will return TRUE if it finds an item, otherwise it will return FALSE .

[ 74 ]

Page 86

﻿

Chapter 3

Doubly linked list

A doubly linked list extends a linear linked list by including pointers to the previous and the next node, as shown in Figure 3.16 :

Figure 3.16: Doubly-linked list node

The pointers on both sides allow moving in both directions. For a one-node linked list, the previous and next pointers are set to NULL . The two pointers make this data structure more memory intensive as compared to a single linked list. Similar to singly linked list, a doubly linked list's start and end locations are referred to as the head and tail respectively. The dlinkListNode function provides the definition to create a doubly linked list node.

dlinkListNode <- function(val, prevnode=NULL, node=NULL) {

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

class(llist) <- "dlinkList" llist

}

An example of a doubly linked list created using the preceding node structure will look like what is shown in Figure 3.17 :

Figure 3.17: An example of doubly link list

[ 75 ]

Page 87

﻿

Linked Lists

Circular linked list

A circular linked list extends both singly and doubly linked lists by connecting the null connection with the tail and head accordingly. The circular linked list extension from a singly linked list and doubly linked list is shown in Figure 3.18 :

Figure 3.18: An example of circular linked lists

This can be obtained by passing the list to the null node of the linked list. For example, a singly linked list can be converted into a circular linked list by passing the head of the linked list to the tail node:

cicularLinkList<-function(llist, val){

if(isEmpty(llist)){

llist<-linkListNode(val) head<-llist

} else {

llistNew<-linkListNode(val) llistNew$nextnode<-head

llist<-linkListNode(llist, llistNew)

}

llist

}

[ 76 ]

Page 88

﻿

Chapter 3

The circular linked list has usage in multiplayer games such as the bridge card game, where the pointer keeps moving from one player to another player in a circular fashion till the game ends.

Array-based list

The array-based list, also known as array list, is a resizable array implementation. Thus, as more elements are added to the linked list, its size increases dynamically. The array-based list assigns an element to the assigned array; however, if a new element is assigned some data, and there is no space in the array, then it allocates a new array, and moves all the data to the newly allocated array. For example, as shown in Figure 3.19 , since the array is full, all of the data is reassigned to a bigger array by increasing the size by a default value.

Figure 3.19: Example array-based link list

Let us set up a reference class ALinkList for an array list in R. To set up an array linked list, the class fields required are as follows:

Alist : To store the dataset

listsize : Pointer to the current location in the array; this can also be used to get the current list size

arraysize : Default expansion size maxSize : Maximum array size

The defined class initializes an arraysize of 100 elements, thus Alist is initialized for 100 elements with listsize initialized to 0 and maxSize to 100 .

ALinkList<-setRefClass(Class = "ALinkList",

fields = list( Alist="array",

listsize="integer", arraySize="integer",

[ 77 ]

Page 89

﻿

Linked Lists

maxSize="integer"

),

methods = list(

initialize=function(...){

listsize<<-0L

arraySize<<-100L

Alist<<-array(dim = arraySize) maxSize<<-arraySize

} ) )

Methods to ALinkList class can be added based on the defined ADT. Let's define the basic ADT for the ALinkList class as shown in Figure 3.20 :

Figure 3.20: An example of ADT for array list

The length of an array can be obtained by returning listsize into the method:

listlen = function() {

return(listsize)

}

Adding an item into the array list requires an additional check on size of list. If listsize is greater than maxSize, then the array needs to be expanded based on arraySize .

updateArrayList=function(){

Alist<<-c(Alist, array(dim=arraySize)) maxSize<<-maxSize+arraySize

},

addItem=function(item){ if(maxSize<=listsize){

updateArrayList()

[ 78 ]

Page 90

﻿

Chapter 3

}

listsize<<-listsize+1L Alist[listsize]<-item return(listsize)

}

An item can be removed from the list based on the array index:

removeItem = function(i) {

Alist[i] <<- NULL

listsize <<- listsize - 1L

}

Deletion or searching of an item in array list can be performed by scanning through it. For example, searching for a position based on item is shown in the following code snippet:

searchItem = function(val){

pointer<-1L

while(pointer!=listsize){ if(Alist[pointer]==val){

break

}

pointer<-pointer+1L

}

return(pointer)

}

The searchItem function scans the array list, and returns the position once it finds the value.

Analysis of list operations

The complexity of list operations depends on traversal. For a linked list of n nodes, the isEmpty() method is O(1), as it only compares the first node to see if it is an empty environment. Similarly, the sizeLinkList() method requires O(n) operations to determine the length of a linked list, as the linked list has to traverse through all the nodes for length determination.

[ 79 ]

Page 91

﻿

Linked Lists

The deletion and searching for an item in the linked list in worst case will take O(n) operations , as the pointer may have to scan through all the nodes before it finds the item for deletion. On the other hand, the addElement() method will take O(1) time as it is directly adding a new element to the head of the linked list. Insertion based on position will take O(p) time, as the linked list has to traverse through p nodes before performing an insertion. For example, say we want to insert 11 at the third position in the list <1, 2, 5, 4> . The current insertion operation will require the pointer to move from the head to the third position. In the worst case, where insertion needs to be done after the last node, it would require O(n) computational effort.

In an array list, moving to any position requires O(1) operations, as the elements can be accessed directly. The insert and delete operations are quite straightforward in array list implementation, and require O(1) computational effort if performed at the tail of the list, as no data needs to be moved in the array list. However, if an element is deleted from or inserted in between items in a list, then all the elements need to be moved one position towards the head or tail respectively. For example, insertion of an element in the example shown in the next image requires all the other elements to move towards the tail. So, if an element is inserted at the p th location, it will require n-p elements to be moved to the tail as shown in Figure 3.14 , thus requiring O(n) computational effort.

Figure 3.21: Insertion in an array list

To summarize, array lists are very efficient in accessing a dataset with O(1) computational effort, whereas linked lists are just average in accessing a dataset with O(n) computational effort. However, insertion and deletion require O(n) computations in array lists and O(1) computations in linked lists, making linked lists more efficient in handling insertion and deletion if the pointer is at the location of insertion or deletion.

[ 80 ]

Page 92

﻿

Chapter 3

Exercises

1. Modify the addElement function of a singly linked list to add an item at any

position.

2. Write a function for reversing a singly linked list. 3. Write the ADT for circular linked lists.

4. Write an R function for creating, inserting, and searching for a circular linked list. 5. Write a function which will return the index of an item for a linear linked list.

Summary

The current chapter covered the fundamental data structures built-in in R, and also covered the concepts of lists and their implementation in R. The chapter introduced built-in data types in R such as vector, and element data types. Also, object-based programming, including S3, S4 and reference classes was introduced. The chapter also introduced one of the most fundamental data structures, link list, and its different variation, such as circular link list and array-based list. The next chapter will introduce stacks and queues data structure.

[ 81 ]