Term Project Write Up
ECS145 Winter 2017

WAI YING LI (998092926)
SALLY LY (999882177)
HALEY SANDERS-TURNER (912296300)
UC Davis
March 18, 2017

I. Introduction

R's policy of no pointers and no side effects proved to make the term project more difficult than we expected. First, setting up the constructors for the data structures was relatively easy. We set up S3 classes for all three. In more detail: for the binary tree, we created a class named **bintree**, and used a three column matrix named **data**. In this matrix, each node was represented by a row in the matrix, and the value of the node was stored in the first column of a row. The left hand child's row number was stored in the second column of the row, and the right hand child's row number was stored in the third column. For both the **queue** and **stack** classes, we used a list to hold two things, **Items** and **numItems**. **Items** is a vector to hold the actual queue or stack, and **numItems** is a counter. This list is assigned the class **queue** or **stack** in the constructors.

II. Implementation

Binary Tree

For the binary tree, the **push** function was implemented relatively simply. First we determined whether or not an item had been assigned to the root. Next we found the first two nodes that were missing either a left or right child, and made a separate row for the item being pushed into the tree. Using this information, we assigned the new row as the child of the appropriate parent, with the parent missing a right hand child taking precedence. The **pop** function was implemented by using the **min**

and **which** functions to determine which row needed to be deleted. These functions were also used while determining which rows would need to have their child "links" updated. The **print** function for **bintree** was implemented using a recursive helper function that would print the value of the current node before calling itself on the current node's children.

Stack and Queue

As for both **stack** and **queue**, the implementations are quite similar and straightforward. The **push** functions for both check if the data structure is empty in order to assign the first element. Otherwise, the element to be inserted will just be appended to the existing vector and the modified data structure will be saved to a global environment variable. (More on global environment variables in `section III`) The **pop** function is the main difference between the two classes. Both **pop** functions first checks if the vector is empty to prevent going out of the vector's boundary when deleting the popped element. For the **stack**, the **pop** function will save the last element of the vector to a variable that will be returned and modify the **Items** vector to exclude the last element. This modified stack will then be saved to a global environment variable. The same goes for **queue**, but **queue** will just **pop** the first element rather than the last element.

Generic Functions

We created an R file called **genfunc.r** that contained the generic functions for **push** and **pop** so the separate files of **bintree**, **stack**, and **queue** could use them.[1] Both **push** and **pop** have **dataStruct** as the first argument so the generic functions know which data structure method to dispatch to. **push** has an extra argument item which is simply the item that will be added to the data structure. Finally, the argument **name** is the name of the data structure (the variable name that you assigned to the data structure) which is used to update the data structure. This will be explained in more detail in `Section III` where we discuss the issues we faced with R's policy of no pointers.

III. Challenges

In languages like C, C++, or Python, updating a data structure class using methods was simple; we just simply passed the object by reference

[1] **print** already exists as a generic function in R so we don't have to declare it. We simply just create **bintree**, **stack**, and **queue** versions of print.

into the method. We could update a class variable inside a function. However, this is difficult to do in R because of its hard policy of "no side effects." For S3 classes, this policy means that we can't actually change our original objects in our class methods. We would have to return the copied object and re-assign it our variable. In order to avoid this, we tried different methods.

First, we tried using the super assignment operator "**«-**" thinking we could turn our class "global" and then update it globally in our function methods. However, this proved to not work as we thought it would. As an example:

```
newqueue <- function() {
   que <<- list()
   class(que) <<- "queue"
   ...
   return(que)
}
```

On the line `que «- list()`, the super assignment operator looks for a variable called **que** on the global environment and if there isn't one, it creates a variable named **que** to assign to. Now assume that we want to create a new queue by writing:

```
q <- newqueue()
```

What our **newqueue** function would do is not only create a global variable **que** but also return a copy of this object and assign it to **q**! To emphasize, **q** holds a copy of **que** and NOT a pointer because of R's "no pointers" rule. Thus, if we were to look at one our earlier versions of a class method (with some code omitted to conserve space):

```
push.queue <- function(que, item) {
   ...
   que$numItems <<- que$numItems + 1
   ...
   return(que)
} #appends item to end of queue, returns the new object
```

The line incrementing **numItems** would update the variable **que** and NOT **q** because the super assignment operator searches the global environment for only the name **que**. There are two problems to this: (1) the user would be forced to reassign **q** instead of simply writing `push(q, item)` and **push** updating the data structure itself. This poses an even bigger problem with

pop as the function returns the popped value and we can't return the new updated data structure. (2) What if the user wants to create multiple queues? If they created a second queue (for the example, we call it **q2**), **que** would be overwritten and we would lose the first queue. And if we called for a **push** on **q**, we would essentially be pushing onto the new queue **q2** (via **que**)!

The essential problem is that we are updating the **que** variable and not the variables we actually want to hold the data structure (from the previous example, we want to update **q**). We wanted to find a way to tell our program to specifically update **q**. This is where the **assign** function comes in. With this function, we can name any variable **name** and **assign** something to it. We can even explicitly tell **assign** which environment it should look in. To grab our variable name, we added an extra argument in **push** and **pop** called **name** which is basically a string of the variable name. From our queue example, **name** would hold âĂIJqâĂİ since we want to update the object in **q**. We added in:

```
assign(name, que, envir=.GlobalEnv)
```

in queue.r's **pop** and **push** functions after we finished updating our local variable **que**. The **assign** function then searches the global environment for **q** and assign **que** to it. And viola! **q** updates successfully without reassigning the returned value! [2]

IV. Conclusion

The primary function of data structures is to provide efficient data storage and even more efficient data retrieval and updating. For this reason, data structures are generally implemented using pointers, which minimize the amount of memory necessary to store a structure. Functional languages like R, however, have a focus on the reduction of side effects, and so forego pointer systems. This means that the only method of passing data around in R is via pass by value, which complicates the development of data structures. This forced us to further explore the nuances of R in order to develop other methods to update the data stored in the structure, preferably without needing to copy and reassign the entire structure to a variable each time. We did this chiefly by utilizing R's **assign()** function. It's important to note that

[2]We didn't have to change **push** as we could just reassign the returned value unlike for **pop** where we must return the popped value and had no other way to return the updated data structure. However, we added the **name** argument to **push** anyway as we thought it would be easier for the user to simply push items onto the data structure and not have to constantly reassign (unless they wanted to assign it to a different variable, etc). Our code is also more uniform since **pop** and **push** are used in almost the same manner.

R was not completely unhelpful.

R does have pseudo-structures called **S3 classes** that are useful in data retrieval. Further, in R it is easy to reduce the time complexity of some functions by using its vector subsetting operations and the **which()** function, which removed the need to use loops in both the bintree **pop** and **push** functions. In the end, one cannot underestimate the need to choose one's tools wisely – programming languages are not one-size fits all, but in some rare cases, ingenuity and resourcefulness can temporarily patch the worst holes .