ECS 145 Final Project

Joanne Wang, Eric Du, John Nguyen, Jeffrey Tai

March 21 2017

1 No Pointers, No Problem

As frequent programmers of object-oriented languages such as Java and Python, our group is very accustomed to incorporating pointers in our programs. For example, simulating stacks, queues, and binary trees would have been easy with Python's internal pointers by just using a list, and having the push/pop functions update that list that is passed by pointer:

```
def s(stack, op, val):
      if op == 1: # push
            return stack.append(val)
      else: # pop (val is irrelevant)
            return stack.pop()

stack = [1,2,3]
s(stack,1,4) # push 4. stack = [1,2,3,4]
itm = s(stack,2,0) # pop stack. stack = [1,2,3], itm = 4
```

Because of R's functional nature and its lack of pointers, we are faced an additional challenge: updating our data structures keeping in mind that R has no "side effects". In the beginning, it seemed that an easy solution for this was to reassign the copied data structure that the push/pop function was changing.

```
s <- function(stack, op, val=NULL){
      if(op == 1) # push
            return(c(stack,val))
      else # pop
            return(stack[length(stack)])
}
stack <- c(1,2,3)
s(stack,1,4)
print(stack) # global stack remains the same
stack <- s(stack,1,4) # reassign stack with pushed value
print(stack) # now [1,2,3,4]
stack <- s(stack,2) #stack assigned to popped value. NOT what we want.
```

It quickly became apparent that this method would not completely work due to the nature of the push/pop functions, since pop would then be unable to return

the updated contents of the data structure.
Ultimately, our method around this is the use of environments. Environments allow us to allocate variables that are within that environment only. Thus, by invoking a new environment for each new instance of a data structure, we can store and update the contents of that data structure in its own environment. This is the key to our final project, in which we simulate a stack, queue, and binary tree in R.

```
newstack <- function() {
  st <- new.env(parent=globalenv())
  st$data <- c(NA)
  class(st) <- append(class(st), "stack")
  return(st)
}
```

2 Stacks and Queues

The implementation of the stack and queue is nearly identical. With each new instance, a new environment is instantiated, within it a data vector that holds the content is initialized to null, and class attribute is appended with the name of the structure.

Push and Pop methods are generic functions that dispatch to non-generic functions using the `UseMethod()` function. Like R's generic print function, it is more pleasant, from a user standpoint, to call the generic push/pop function and have it internally dispatch to the appropriate function than to call the object specific method.

```
push <- function(obj, val) UseMethod("push")
pop <- function(obj) UseMethod("pop")
```

Both push functions take in the environment object and a value as a parameter, appends the environment's data vector with the value and returns the updated data vector. The only function that differs in stack and queue is the pop function. Both take in the environment as a parameter, but the stack pop function returns the last element in the data vector (LIFO), while the queue pop function returns the first element (FIFO). Printing the stack and queue is done by simply printing out the data vector of the environment. If it empty (only contains NA), nothing will be printed.

```
print.queue <- function(obj){
  if(length(obj$data) == 1) return()
  print(obj$data[-1])
}
```

Exceptions are thrown if the user attempts to push NA value or a value of different type onto the stack or queue, or if the user attempts to pop from the stack or queue when it is empty.

3 Binary Tree

As with stacks and queues, with each new instance of a binary tree, a new environment is instantiated, and the class attribute is appended with the name of the structure. What differs is a binary tree is stored in the form of a matrix, where each row is a node with four columns, each column representing the value, left child row number, right child row number, and count (in the case of duplicates), respectively. Therefore, a data matrix is initialized to `NA` when a binary tree is first instantiated.

```
tree$vals <- matrix(cbind(NA,NA,NA,NA),nrow=1,ncol=4,
                    dimnames = list(c(NULL),
                    c("Value","Left", "Right","Count")))
```

Push and pop methods are also generic functions that dispatch to the appropriate non-generic function with the `UseMethod()` function, for the same reason as the stack and queue implementation.

When a value is to be pushed, if the tree is empty, the node is simply added to the matrix at row 2, which is the fixed root, with left and right children set to `NA`. Otherwise, a helper function is called to recursively traverse down the tree matrix to the appropriate "node" to connect the value to. A row is then 'allocated' for the value and its columns are filled in with the value left and right child row number set to `NA`, and count to 1 or incremented if it is a duplicate. Its parent's left or right child row number updated concurrently, and the updated tree matrix is returned.

```
push_helper <- function(root, inVal, graph){
  root <- as.numeric(root)
  # base case, insert as left child
  if(inVal < graph[root,1] & is.na(graph[root,2])){
    rowToInsert <- push_firstNArow(graph) # find avail row
    if(rowToInsert == -1) { # create new row
      graph <- rbind(graph,c(inVal,NA,NA,1))
      graph[root,2] <- nrow(graph) # update parent node's left child
           value
    } else {
      graph[rowToInsert,] <- c(inVal,NA,NA,1) # put val in found NA row
      graph[root,2] <- rowToInsert
    }
    return(graph)
  }
  # base case, insert as right child
  if(inVal > graph[root,1] & is.na(graph[root,3])){ # base case, insert
      to right
    rowToInsert <- push_firstNArow(graph) # find avail row
    if(rowToInsert == -1) { # create new row
      graph <- rbind(graph,c(inVal,NA,NA,1))
      graph[root,3] <- nrow(graph) # update parent node's right child
          value
    } else {
      graph[rowToInsert,] <- c(inVal,NA,NA,1) # put val in found NA row
      graph[root,3] <- rowToInsert
    }
    return(graph)
  }
  # base case, duplicate item
  if(inVal == graph[root,1]) {
    graph[root,4] <- as.numeric(graph[root,4]) + 1 # increment count
    return(graph)
  }
  # search left tree
  if(inVal < graph[root,1]) push_helper(graph[root,2],inVal,graph)
  # search right tree
  else push_helper(graph[root,3],inVal,graph)
}
```

Note we say that a row is 'allocated' because when a new value is to be pushed, we look for the first row of `NA`'s that is in the matrix. The occurrence of a row of `NA`'s will be due to an item being popped previously.

When we want to pop a value, the tree will traverse to its leftmost (smallest) node, starting from the root, which is row 2, and following its left child row number. It keeps track of the parent node. When the smallest value is found, it is returned after the parent node has its child row number updated to the deleted row's right subtree if applicable. The row's count column is either decremented, or if there are no more duplicates, the row is set to `NA`.
`pop.bintree()` takes care of the case where the root is the smallest element. Other than that, a helper function does the recursive work to traverse the left subtrees of the tree.

```
pop_helper <- function(tree, row, parent) {
  row <- as.numeric(row)
  parent <- as.numeric(parent)
  # base case, found smallest element
  if(is.na(tree$vals[row,2])) {
    top <- tree$vals[row,1] # value to return
    # next we check for duplicates and whether to move right subtree up
    if(tree$vals[row, 4] == 1) { # no duplicates remaining
      rTree <- tree$vals[row,3]
      if(!is.na(rTree)) { # right subtree exists
        tree$vals[parent, 2] <- rTree # update parent to new subtree
        tree$vals[row,] <- NA
      } else {
        tree$vals[parent,2] <- NA
        tree$vals[row,] <- NA
      }
      return(top)
    } else { # dupliate remain, decrement count
      tree$vals[row, 4] <- as.numeric(tree$vals[row, 4]) - 1
    }
    return(top)
  }
  pop_helper(tree, tree$vals[as.numeric(row),2], row)
}
```

The binary tree is printed out in in-order format. Therefore, the binary tree print function traverses down through the tree matrix as a normal binary tree would traverse depth-first, to obtain the left child, root, then right child. This method is done recursively with a printhelper function that calls itself, moving continuously down the tree matrix until it reaches the leftmost child, then moves onto the root, and last the right child.

```
printhelper <- function(obj, row) {
    if(!is.na(obj[as.numeric(row),2])) {
      printhelper(obj,obj[as.numeric(row),2])
    }
    print(obj[as.numeric(row),1])
    if(!is.na(obj[as.numeric(row),3])) {
      printhelper(obj,obj[as.numeric(row),3])
    }
  }
```

Keeping in mind that the insertion of a character string will convert all values of the matrix to characters also, our tree allows that and will account for that when indexing the tree matrix. When the user tries to insert a character string to an integer valued tree, it will warn the user. From then on, tree insertion will be based on ASCII values.

Aside from R's built in exception handler that stops the program when a user attempts to add to the tree a different data type than what is already in it, we add a couple more exceptions. Additional exceptions that stop the program is when a user attempts to add an NA value to the tree, and when the user attempts to pop from the tree when it is empty.