Trees, Stacks, and Queues in R

Tanner Embry, Sebastian Grobelny, Eric Tan

March 22, 2017

1 Introduction

While R facilitates some basic data structures such as vectors and lists, and more speciality data structures commonly used in numerical computing and statistics such as matrices and data frames, there is a lack of common data structures used in computer science. The ones we will implement in R will be the stack, the queue and the binary tree.

One of the issues with R is the lack of pointers. In C and C++, pointers make it convenient to impalement many data strictures since we are allowed to access the data directly. However, in R and many other languages, this is simply not possible. Rather, we can get around this by use an array or a list to hold the data and use class specific operations to access data like we would in a stack, queue or a binary tree. In essence, we can store our data in an array, then write methods which access the array specific to the data structure.

2 Trees

A tree is a data structure with average look up operations which cost $O(\log\, n)$ operations. In languages with pointers available, we could simply have a collection of pointers in parent node point to the children node. However, this will provide a challenge in R since we don't have access to pointers. We will be implementing a binary search tree.

2.1 Theory and Operations

A tree contains nodes linked to each other in some kind of structure. The binary search tree contains nodes with at most 2 nodes connected to it. At the very top of the tree is the root node. This node than can have up to two children nodes, which can in turn also have two parent nodes. This can go on for as many layers as needed. This implementation mimics the average time of O(log(n)) for binary trees by making effective use of the first two columns of our storage matrix.

Algorithm 1 Push operation for a Binary Tree

```
1: Input:  Binary Tree tree, item to insert x
2: tree.nxt = tree.nxt + 1
3: while  true do
4:
5:     if tree[level,3] > x then
6:         dir = 2
7:         if tree[level,3] < x then
8:             dir = 1
9:             if tree[level,dir] == NULL then
10:                tree[level,dir] = tree.nxt
11:                tree[tree.nxt,3] = x
12:                tree[tree.nxt,1] = NULL
13:                tree[tree.nxt,2] = NULL
14:            end if
15:
16:            return tree
```

Next, the pop operation finds the smallest node in the binary tree and deletes it We traverse through the tree, find the value and clear it from the tree. We then reinsert the nodes into the tree so as to maintain the original structure.

Algorithm 2 Pop operation for a Binary Tree

```
1: Input:  Binary Tree tree, item to insert x
2: level = 1
3: while  true do
4:
5:     if tree[level,1] ! = NULL then
6:         level = tree[level,1]
7:     else
8:         tree = tree[-level,]
9:         nodelist = tree[,3]
10:        tree = newbintree()
11:        i = 1
12:
13:        while  length(nodelist) >= i do
14:
15:            tree = push(tree,nodelist[i])
16:            i=i+1
17:        end while
18:    end if
19: end while
20: return tree =0
```

This of course would theoretically give our pop operation an average run time of O(n*log(n)) since we are reinserting every time we delete, but given the nature of how we store our binary tree this method is on the more efficient side.

2.2 Implementation

Since we don't have access to pointers, we decided to impalement a tree with a matrix. Each row in the matrix contains three unique columns which corresponds to a unique node. The first column has the index of the row associated with the node's left child and the second column has the index of the row associated with its right child and the third column has the value associated with the node. In addition, the class contains a variable `nxt` which tracks the size of the binary tree.

2.2.1 push

For the push the user passes the value in along with the associated tree. If our `nxt` variable is greater than the number of rows in the matrix we allocate a new row to our matrix via the built in `rbind`. `newPos` is then initialized based on the `nxt` variable and this will be the row associated with the value to be inserted. A search is then executed in order to place the value in the proper location in the binary tree and ensure that the value has not been stored already.

The search operates based on a variable we call `level`. Initially, we set the variable to 1 and transverse the tree based on whether the input is greater or less than the root. If it is exactly equal to the value at the given level, we decrement the `nxt` variable and exit the push because our value is already in the tree. If it is less than the value stored at a given level, then the index of the first column, our left child, is used as the level we go into in our next iteration. If the value to insert is greater then the second column is used instead. As soon as a null or NA value is found in place of a level to go to, then `newPos` is assigned to the given column since the value to be inserted is now its child.

2.2.2 pop

For our pop we traverse through our binary tree's left children until there is no left child to be found. This is done by initializing a variable called `level` to 1 and then checking to see if the first column of our current points to `NA`. If it does not we reassign `level` to tree[level,1] as that is the next left child given our storage convention. In the case of an `NA` we are guaranteed that our current value at `level` is the smallest one, so we remove it from our `tree$mat` construct. Then we simply take the collection of all values in our third column and store it into node list. This gives us all the values the user inserted into the binary tree and the correct order they were inserted. We initialize a new binary tree by calling `newbintree()` and then push the contents of node list using `push.bintree()` with our newly created tree. Understandably this does not mimic a binary tree's typical deletion time of O(log(n)) and this is an obvious drawback to not having pointers available to us in R as we would in Python.

We had tried implementing a delete method with a helper that would promote children given that their parent was deleted so as to do everything in place in our construct. This however proved largely complicated and ineffective given that we have a matrix of nodes rather than pointers to the appropriate nodes. Certain edge cases would require us to go back and update the entire matrix when only one node had been deleted, when a similar operation in a pointer implementation would only change the pointer. So we instead decided to take advantage of the fact that we have a matrix at our disposal and an efficient method for inserting values, `push.bintree()`. R has allowed us to make deletion somewhat simpler than it would be in Python as we disregard the nature of pointers using instead the column vector of all the values inserted in the tree, in the exact order they were inserted. This allows us to effectively recreate the tree that was previously in place minus the minimum value that had been deleted in the call to `pop.bintree()`.

3 Stacks

The stack is a first in last out (FILO) data structure. The first item placed onto the stack will be the last to be taken out. The push and pop operations take $O(1)$ time.

3.1 Theory and Operations

The operations for a stack is quite simple. First, the push (or insertion) operation takes the last index, increments it, and then adds the item we are inserting. We keep track of the size of the array by the variable `top`. Since a stack is a FILO data structure, we don't need to keep track of the bottom of the stack.

Algorithm 3 Push operation for a Stack

```
1: Input:  Stack S, item to insert x
2: S.top = S.top + 1
3: S[S.top] = x
```

Next, the pop operation takes the top element and returns it. We then decrement the value of `top`. Note that if we try to decrement an empty stack, we have an underflow, so we need to check for this.

Algorithm 4 Pop operation for a Stack

```
1: Input:  Stack S
2: if S.top == 0 then
3:     Error Stack is empty
4: else
5:     x = S[S.top]
6:     S.top = S.top - 1
7: end if
8: return x
```

3.2 Implementation

The issue regarding the lack of pointers for a stack is less apparent in the implementation. We can simply built up a vector using R's `c()` function and append to it when needed. Since a stack is a FILO data structure, we start building the data structure at index 1 on the vector and iterate up as we keep adding items. Of course, it is possible that our stack is empty, in which case we have a vector with the only element initialized as `NA` (this is how our class is initially constructed).

The stack is initialized with `stack()`. This will initialize the `top` variable representing the top of the stack to 1, `length` representing the length of the stack to 0, and `data` which stores our data to a vector with the first element set to `NA`. Note that `top` will always point 1 above the last item we inserted (so if we have 2 items in the stack, top is set to 3). The `push()`, `pop()`, and `print()` operations will be available.

3.2.1 print.stack()

The print operation takes in a stack as the argument and prints it. The only special case the function has to handle is when the stack is empty. In which case, we simply don't output anything since nothing is in the stack. We check if the size of the stack is not equal to 0 when we are printing, then generate a vector of indices `1:(top - 1)` and print out the stack using the indices generated.

3.2.2 push.stack() and pop.stack()

The push operation puts an item on the stack. First, we insert the item to `data[top]` and then increment `top` and `length`. There is no need for error checking since our vector holding the data can grow to an arbitrary size. The stack makes use of `eval.parent(substitute(...))` [3] so that we can modify values without having to return the stack.

The pop operation is a bit more complicated due to the error checking. If the `length` is 0, than the queue is empty and we output a warning that we are popping an empty queue. If not, than we access the `top - 1` element for returning, set that element to `NA`, and then decrement `top` and `length`. We make use of `eval.parent(substitute(...))` once again to update the state of our stack.

4 Queues

A queue is a first in first out (FIFO) data structure. This means that an insertion goes in the end of the queue (tail) and access happens at the beginning (head) of the queue. Both these operations run in $O(1)$ time [1].

4.1 Theory and Operations

The queue supports the basic insertion, deletion, and pop operations. First, the insertion operation, also known as enqueue, inserts the data structure to the tail. The operations is quite simple. We check if the queue is empty and insert to the $i + 1$ position if it isn't empty and to the 1st position if it is.

Algorithm 5 Enqueue (insert or push) operation

```
1: Input:  Queue Q, item to insert x.
2: Q[tail] = x
3: Q.tail = Q.tail + 1
4: Q.length = Q.length + 1
```

The dequeue operation is very similar to the enqueue operation except we are taking items out of the queue. This operation removes the item at the head of the queue and returns it. The head then moves up 1 position. Naturally, if we attempted to pop while the queue is empty, we have an underflow error, so we need to check for this.

Algorithm 6 Dequeue (delete or pop) operation

```
1: Input:  Queue Q.
2: if Q.length == 0 then
3:     Error underflow.
4: else
5:     x = Q[head]
6:     Q.head = Q.head + 1
7:     Q.length = Q.length - 1
8: end if
9: return x
```

4.2 Implementation

The primary issue between theory and application is the lack of pointers. If we had access to pointers, than the algorithms described is 4.1 can be implemented by using a pointer to the head and tail of

the queue and increment the positions of the head and tell when an operation is called. Instead, since R doesn't support pointers, but does support arrays, so we can use an array and add a variable for tracking the head, tail, and size of the array. Note that the tail is always 1 place ahead of the last element of the queue.

The S3 class is initialized with the function `queue()`. This sets the head and tail to 1, the length to 0, and our data array (implemented as a vector) as a vector with the first element initialized to `NA`. When we perform push operations, the vector will grow and the tail and length will reflect it. Pop operations, on the other hand, will not re-size the vector, but moves the position of the head. The other elements will simply be ignored. Of course, the head and length will be updated. The queue has 3 operations; `print()`, `push()`, and `pop()`.

4.2.1 print.queue()

The `print()` operation simply takes the queue and prints it out. If the head and tail are the same, than we print out the element at the index of head. Otherwise, we generate a vector with elements `head:(tail - 1)`. Remember that tail points to one element past the end of the queue, so `tail - 1` is necessary.

4.2.2 push.queue() and pop.queue()

The `push()` operation inserts an element to the end of the queue. The implementation is simple. We insert the element at index `tail` and update the tail and length. Our vector is allowed to grow as needed, so there is no size constraints for overflows. There is no return type so the way the queue gets updated is by using `eval.parent(substitute(...))` on the queue parameters.

The `pop()` operation is very similar, but instead we return the item at the index `head`. We than update the head, length, and set the popped element to `NA`. Note that since R does not support call by reference due to the functional nature of the language, so we can't easily update the head and the length. The way around this is to use `eval.parent(substitute(...))`. This updates the queue even though we can't pass function parameters by reference. One thing to take note of is when the queue is empty. This means that the `head` and `tail` are in the same position, which gives use a queue with no elements. I needed to use an if statement to check if the length was 0 to prevent popping an empty queue.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 2009.

[2] Norman Matloff. *The Art of R Programming*.
`http://heather.cs.ucdavis.edu/ matloff/132/NSPpart.pdf`

[3] TszKin Julian. *Call by reference in R*.
`https://www.r-bloggers.com/call-by-reference-in-r/`

[4] Friedrich Leisch. *Creating R Packages: A Tutorial*.
`https://cran.r-project.org/doc/contrib/Leisch-CreatingPackages.pdf`