Assignment One – Palindrome Reader

Christopher Ravosa
christopher.ravosa1@marist.edu

February 13, 2019

1 Code Listing - StackRavosa

```
1 public class StackRavosa {
2     //Declare variable for StackRavosa objects
3     private NodeRavosa myTop;
4
5     public StackRavosa() {
6         myTop = null;
7     }//null
8
9     public boolean isEmpty() {
10        boolean ans = false;
11        if(myTop == null)
12            ans = true;
13        return ans;
14     }//isEmpty
15
16     public void push(char newLetter) {
17        NodeRavosa push = new NodeRavosa();
18        push.setNext(myTop);
19        push.setData(newLetter);
20        myTop = push;
21     }//push
22
23     public char pop() {
24        char ans = '\u0000';
25        if(!isEmpty()) {
```

```
26              ans = myTop.getData();
27              myTop = myTop.getNext();
28          }//if
29          return ans;
30      }//pop
31
32      public void printDetails() {
33          NodeRavosa current = myTop;
34          while (current != null) {
35              System.out.print(current.getData());
36              current = current.getNext();
37          }//while
38      }//printDetails
39  }//StackRavosa
```

2 Explanation - StackRavosa

This class impements a stack data structure using linked nodes. The only variable necessary for a singly linked stack data structure is a private node that is declared in line 3 of the class. The node is called "myTop" and is a reference to the last item in the stack, which will be the first item to be popped. This is the only node in the stack we need to keep track of because a stack is a LIFO data structure. This means that the top of the stack, the last object in, will be the first object to leave or be "popped".

The StackRavosa method that begins on line 5 is known as a "null constructor". It initializes the myTop variable of a new instance of the class "StackRavosa" to null. This value will change as nodes with stored values are added to the stack. Often, there is also a "full constructor", but that is not necessary here since the NodeRavosa class will determine what the value stored in the first node should be.

The next method that is implemented in StackRavosa is isEmpty(). This method returns a boolean value to determine whether the stack has any nodes stored in it. If the value of myTop is equal to null, then the stack has no stored values. This is an important method that will come in handy later.

On line 16, we see the push() method. This method returns nothing and is the first one we see that takes in a parameter. The parameter is a character here. It does not accept a NodeRavosa object because, as you can see on line 17, it instantiates one here. It then sets the "next" value of the node to the current myTop. This is a pointer that connects to the "next" node on the stack, specifically the node beneath it, which is our current top. Next, the stored character in the node object is set to the value of the character that was passed in the "newLetter" parameter. Lastly, the value of myTop is then switched to the newly added node. To summarize in a simple way, the method accepts a letter, creates a node, makes the node store that letter, and then adds

it to the top of the stack.

After push() is our pop() method. Pop() will remove the top item from the list and begins by telling us that it is going to return a character. The character it will return is the stored value of the top node on the stack. On line 24, a char variable "ans" is initialized to "\u0000". This value is the char equivalent of null to a string. "ans" is initialized as null because it is possible that the stack might be empty when we try to pop the top value. Instead of receiving an error, this will allow the method to return a null value. If the stack is not empty, the value of "ans" will be set to the stored character of the node on the top of the stack. This is where the isEmpty() method is useful. Next, the value of myTop will then be set to the current top's .next. This will relocate the top to the next node that will be popped.

Lastly, we have the printDetails() method. This method creates an instance of NodeRavosa called "current". This then becomes a reference to the top of the stack. If the list is not empty, and without underflowing the stack, this reference will go through each node in the stack and print its stored value. This will print out each letter in the stack one at a time.

That's all there is to StackRavosa. It builds a simple data structure that can easily be implemented as a linked list of node objects. Most comments were excluded in the above code listing, but are densely present in the final Java program.

3 Code Listing - QueueRavosa

```
1 public class QueueRavosa {
2     //Declare variables for QueueRavosa objects
3     private NodeRavosa myFront;
4     private NodeRavosa myRear;
5
6     public QueueRavosa() {
7         myFront = null;
8         myRear = null;
9     }//null constructor
10
11     public boolean isEmpty() {
12         boolean ans = false;
13         if (myFront == null)
14             ans = true;
15         return ans;
16     }//isEmpty
17
18     public void enqueue(char newLetter) {
19         NodeRavosa enqueue = new NodeRavosa();
20         enqueue.setData(newLetter);
21         if (isEmpty()) {
```

```
22              myFront = enqueue;
23              myRear = enqueue;
24          }//if
25          else {
26              myRear.setNext(enqueue);
27              myRear = enqueue;
28          }//else
29      }//enqueue
30
31      public char dequeue() {
32      char ans = myFront.getData();
33      myFront = myFront.getNext();
34      return ans;
35      }//dequeue
36
37      public void printDetails() {
38          NodeRavosa current = myFront;
39          while (current != null) {
40              System.out.print(current.getData());
41              current = current.getNext();
42          }//while
43      }//printDetails
44 }//QueueRavosa
```

4 Explanation - QueueRavosa

While in a singly linked stack we only need to keep track of the top or head, in a singly linked queue we have two variables to keep track of. This is because a queue is a FIFO data structure. This means that the front of the queue, or the "head", must be stored as a node so it can leave. Similarly, the back of the queue still needs to be maintained so that each object points to the one behind it.

These variables are declared on lines 3 and 4 as myFront and myRear. They are then initialized in the null constructor. Similar to the stack, we don't need a full constructor because the NodeRavosa class will determine the value stored in the first node of the list.

This class also kicks off with an isEmpty() method. This method is similar to the stack class' isEmpty() in that it returns a boolean. The boolean value is stored in another variable called "ans". A queue is empty if the head of the queue is null, meaning no node holds the position of myFront. If this is the case, the method will return true, otherwise it will return false to show the queue is not empty.

Enqueue() is the queue equivalent of a push. Here, we are adding a value to the queue which will come in the form of a node whose value is the character we accept as a parameter. If the queue is empty, the new node will become both

the head and the tail of the queue, otherwise, the tail's pointer will be rerouted to the new node and the new node will become the tail. This is shown in the if/else statement between lines 21 and 28.

Removing an item from the queue seems more complex than for a stack because we are taking from the front, but it's quite simple. When we dequeue, we save the head's stored value and then eliminate it by making the new head the next item in the list. We then return the value of the previous head's data to be compared back in PalindromeReaderRavosa.

Lastly, we again have a printDetails() method. This method simply shuffles through the list until the reference to the current node falls off the tail. It prints the stored value of each node as it goes through. This was simply used to assist in debugging the program and was not called on in the final program.

The linked list implementation of a queue is just as simple as a stack. The key difference is the order in which items are removed from the list. For both stacks and queues, they enter from the back, but in a queue they leave from the front. Once again, most comments were removed in this code listing and they are excessively present in the actual program.