Assignment One

Harrison Zheng
harrison.zheng1@marist.edu

September 25, 2021

1 Assignment Description

For assignment 1, students were asked to develop a program that reads from a text file and determines whether each line is a palindrome. A palindrome is a word or phrase that reads the same forward and backward (ex: radar). Students were required to write and use a linked list, stack, and queue in their solution without built-in language features, like Java.Collections.Stack. Although I was not able to get my program to print out the palindromes, I was able to successfully create and manipulate these three data structures.

2 Main Program

Execution of my program begins in the main method, which is held in a class named PalindromeChecker. The method first reads the content from a given text file, MagicItems.txt, into an array. It then creates and populates a linked list, stack, and queue for each magic item (more details on these processes are provided in sections 3 - 5). The program uses class functions to create 2 arrays that hold the characters of magic items in the stack and queue. Finally, the main method calls on another method to compare the characters in the arrays, and if they all match, then the magic item is a palindrome and the program prints it out (Listing 2.1 Line 56-64).

2.1 Main program source code listings

```
1  public class PalindromeChecker {
2      public static void main(String[] args) {
3          System.out.println("The following magic items are palindromes: ");
4  
5          //Read file items into an array
6          String[] arr = new String[666];
7          try {
8              File fileObj = new File("magicitems.txt");
9              Scanner myReader = new Scanner(fileObj);
10             for (int i = 0; i < arr.length; i++) {
```

```java
                arr[i] = myReader.nextLine();
            }
            myReader.close();
        }
        catch (FileNotFoundException ex) {
            System.out.println("See error below.");
            ex.printStackTrace();
        }

        //Create a linked list, stack, and queue, and
        //determine if magic item is a palindrome
        for (int i = 0; i < arr.length; i++) {
            String magicItem = arr[i];

            //Sanitize the string
            magicItem = magicItem.replaceAll("\\s", "");
            magicItem = magicItem.replaceAll("[0-9+-,._()/\']", "");
            magicItem = magicItem.toLowerCase();

            //Create a LinkedList
            LinkedList listOfChars = LinkedList.createLinkedList(magicItem);

            //Initialize a stack and queue
            Stack stackOfChars = new Stack();
            Queue queueOfChars = new Queue();

            //For every node in linked list, push and enqueue it
            Node n = listOfChars.head;
            while (n != null) {
                stackOfChars = Stack.Push(stackOfChars, n);
                queueOfChars = Queue.Enqueue(queueOfChars, n);
                n = n.next;
            }

            //Remove nodes from the stack and queue and store their character values
            char[] stackArr = Stack.Pop(stackOfChars, magicItem.length());
            char[] queueArr = Queue.Dequeue(queueOfChars, magicItem.length());

            if (isPalindrome(stackArr, queueArr, magicItem.length())) {
                //Print the magic item if it passes the palindrome check
                System.out.println(magicItem);
            }
        }
    }

    //Checks if the magic item is a palindrome
    public static boolean isPalindrome(char[] stack, char[] queue, int strLength) {
        for (int i = 0; i < strLength; i++) {
            if (stack[i] != queue[i]) {
                return false;
            }
        }
        return true;
    }
}
```

3 Node & Linked List

The Node class is used to create nodes, which are then used to create linked lists, queues, and stacks by other functions in the program. A Node object has attributes to store a particular character from a magic item and to point to the memory address of another node (Listing 3.1 Line 3-4). This makes nodes useful for all three data structures required by this assignment. The LinkedList class uses nodes by assigning characters to each one and then appending them to one another to represent a magic item. When appending nodes after the first one, the program iterates through the current linked list until it finds the last one and it adds a pointer to the new node there (Listing 3.2 Line 23-27).

3.1 Node class source code listings

```
//Node class is used for creating elements in linked lists, stacks, and queues
public class Node {
        char character;
        Node next;

    //Constructor initializes properties of new Node instances
    public Node(char character) {
        this.character = character;
        this.next = null;
    }

}
```

3.2 LinkedList class source code listings

```
 1 //LinkedList class is used to create linked lists
 2 public class LinkedList {
 3     Node head;
 4 
 5     //Constructor
 6     public LinkedList() {
 7         this.head = null;
 8     }
 9 
10     //Makes linked list out of string passed in
11     public static LinkedList createLinkedList(String magicItem) {
12         LinkedList listOfChars = new LinkedList();
13         for (int j = 0; j < magicItem.length(); j++) {
14             Node character = new Node(magicItem.charAt(j));
15 
16             //Set the first char node as the list's head
17             if (j == 0) {
18                 listOfChars.head = character;
19             }
20             //For rest of chars, go to last node in list
21             //and add new char node to its next pointer
22             else {
23                 Node tail = listOfChars.head;
24                 while (tail.next != null) {
25                     tail = tail.next;
26                 }
27                 tail.next = character;
28             }
29         }
30         return listOfChars;
31     }
32 
33 }
```

4 Stack

My program uses the Stack class to create stacks and perform operations on them. It has a method named Push, which adds the old node currently at the top of the stack to a new node's next pointer and makes the new node the top (Listing 4.1 Line 10-15). It has a Pop method, which moves the stack's top pointer to the node below the one currently on top so that the old top node is effectively removed from the stack. It then stores the popped node in an array that is used by the main program, and it repeats this process until every node has been popped from the stack (Listing 4.1 Line 17-36). The class also has a method named isEmpty, which checks if the stack is empty and it is used by the Pop method (Listing 4.1 Line 38-46).

4.1 Stack class source code listings

```
 1 //Stack class is used to create and perform operations on stacks
 2 public class Stack {
 3     Node top;
 4 
 5     //Constructor
 6     public Stack() {
 7         this.top = null;
 8     }
 9 
10     //Adds a node at the top of a stack
11     public static Stack Push(Stack stackOfChars, Node newNode) {
12         newNode.next = stackOfChars.top;
13         stackOfChars.top = newNode;
14         return stackOfChars;
15     }
16 
17     //Pops every node in a stack and puts them into an array
18     public static char[] Pop(Stack stackOfChars, int strLength) {
19         char[] stackArr = new char[strLength];
20         int stackCounter = 0;
21         Node s = stackOfChars.top;
22         while (s != null) {
23             Node topNode = s;
24             if (!Stack.isEmpty(stackOfChars)) {
25                 //Change top pointer to point to the node below the one currently on top
26                 stackOfChars.top = stackOfChars.top.next;
27             } else {
28                 //Handle an underflow error
29                 System.out.println("Warning! Cannot pop an empty stack!");
30             }
31             stackArr[stackCounter] = topNode.character;
32             stackCounter += 1;
33             s = s.next;
34         }
35         return stackArr;
36     }
37 
38     //Checks whether a stack is empty
39     public static boolean isEmpty(Stack stackOfChars) {
40         if (stackOfChars.top == null) {
41             return true;
42         }
43         else {
44             return false;
45         }
46     }
47 
48 }
```

5 Queue

My program uses the Queue class to create queues and perform operations on them. It is important to note that unlike linked lists and stack, queues have 2 pointers; one at the head and one at the tail. The class has a method named Enqueue, which adds the pointer to a new node to the node currently at the end of the queue so that the new node becomes the tail (Listing 5.1 Line 12-23). It has a Dequeue method, which moves the queue's head pointer to the next node down. Like the Stack class's Pop method, Dequeue stores the dequeued node in an array that is used by the main program, and it repeats this process until every node has been dequeued from the queue (Listing 5.1 Line 25-48). The class also has a method named isEmpty to check if the queue is empty, and it is used by the Enqueue and Dequeue methods (Listing 5.1 Line 50-59).

5.1 Queue class source code listings

```
1  //Queue class is used to create and perform operations on queues
2  public class Queue {
3      Node head;
4      Node tail;
5
6      //Constructor
7      public Queue() {
8          this.head = null;
9          this.tail = null;
10     }
11
12     //Adds a node to the end of a queue
13     public static Queue Enqueue(Queue queueOfChars, Node newNode) {
14         if (isEmpty(queueOfChars)) {
15             queueOfChars.head = newNode;
16             queueOfChars.tail = newNode;
17         }
18         else {
19             queueOfChars.tail.next = newNode;
20             queueOfChars.tail = newNode;
21         }
22         return queueOfChars;
23     }
24
25     //Dequeues every node in the queue and puts them into an array
26     public static char[] Dequeue(Queue queueOfChars, int strLength) {
27         char[] queueArr = new char[strLength];
28         int queueCounter = 0;
29         Node q = queueOfChars.head;
30         while (q != null) {
31             Node headNode = q;
32             if (!Queue.isEmpty(queueOfChars)) {
33                 //Change head pointer to point to the next node
34                 queueOfChars.head = queueOfChars.head.next;
35                 if (Queue.isEmpty(queueOfChars)) {
36                     queueOfChars.tail = null;
37                 }
38             }
39             else {
40                 //Handle an underflow error
41                 System.out.println("Warning! Cannot dequeue an empty queue!");
42             }
43             queueArr[queueCounter] = headNode.character;
44             queueCounter += 1;
```

```
            q = q.next;
        }
        return queueArr;
    }

    //Checks whether a queue is empty
    public static boolean isEmpty(Queue queueOfChars) {
        if (queueOfChars.head == null) {
            return true;
        }
        else {
            return false;
        }
    }
}
```