CMPT 435 - Fall 2021 - Dr. Labouseur

Assignment One – Finding Palindromes

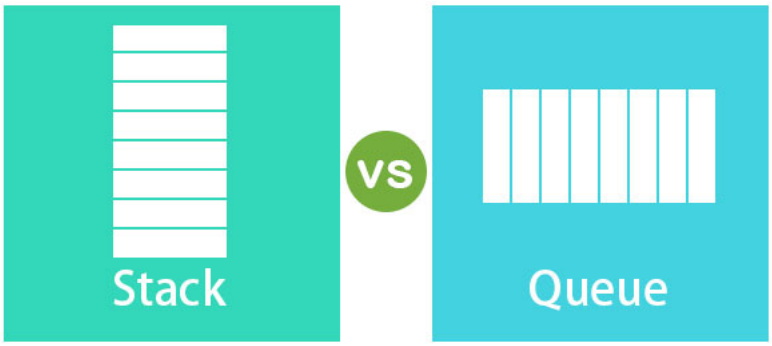
Augusto Gonzalez-Bonorino
augusto.gonzalezbonorino1@marist.edu

September 24, 2021

1 Description of the program

The program described in this document can be, conceptually, divided into two parts: Implementation of Stack, Queue and a singly linked list; utilizing the aforementioned data structures to find palindromes in a given text document.

It consists of four classes: StackGonzalezBonorino, QueueGonzalezBonorino, MainGonzalezBonorino and NodeGonzalezBonorino. The characteristics and methods of each class will be described in more detail throughout this documentation.



2 Stack class

In this section I will describe in detail how I have designed my Stack data structure. In the words of Lewis Carroll "Let's begin at the beginning".
The following extract of code corresponds to the initialization of the necessary instances and default constructor:

```
/**
 *
 * @author Augusto Gonzalez Bonorino
 *
 * This is the class definition for StackGonzalezBonorino
 */
public class StackGonzalezBonorino
{
  /**
   * Instance variable for the top of the stack
   */
  private NodeGonzalezBonorino myTop;

  /**
   * Default constructor for StackGonzalezBonorino
   */
  public StackGonzalezBonorino() {

    myTop = null;
  } // default constructor
```

Next, I defined a function to check if the Stack is empty which comes in handy to avoid hitting a null value while looping through the elements of the Stack.

```
 /**
 * Method to check if Stack is empty
 * @return boolean true if empty false otherwise
 */
public boolean isEmpty() {

  if (myTop == null)
    return true;
  else
    return false;
} // check if empty
```

Lastly, the two most important methods in the Stack are *push()* and *pop()* which are used to add and delete elements from the Stack, respectively.

The method *push()* assigns the element it receives (in this case a character) to the variable *myTop*, which represent the top of the Stack. Then, it assigns the element previously located at the top to a variable I called *oldTop*, thereby adding a new element to the Stack.

```
/**
 * Method to add elements to the Stack
 * @param element
 */
public void push(char element) {
```

```
6
7   NodeGonzalezBonorino oldTop = myTop;
8   myTop = new NodeGonzalezBonorino();
9   myTop.setData(element);
10  myTop.setNext(oldTop);
11
12 } //push
```

The method *pop()* extracts and returns the element located at the top of our Stack. Note that before considering deleting an element we must check that the Stack is not empty, otherwise we would try to access a **null** value and that would raise a ***NullPointerException*** and nobody wants that ☺. Then, if the stack is not empty it assigns the element at *myTop* to the local variable *ans*, updates the top and returns *ans*.

```
1  /**
2   * Method to delete and element from the Stack
3   * @return element extracted
4   */
5  public char pop() {
6
7    char ans = ' ';
8
9    if (!(isEmpty()))
10   {
11     ans = myTop.getData();
12     myTop = myTop.getNext();
13   } //if
14   else {
15     System.out.println("Stack is empty"); } //else
16
17   return ans; } // pop
```

3 Queue class

The queue is a data structure that follows a particular order, referred to as First In First Out (FIFO), in which the operations are performed.
As always, the first thing to do is declare and instantiate the necessary variables for the class. For a Queue we need a variable that refers to the head of the queue and one that refers to the tail, I have called these *myHead* and *myTail* respectively.

```
1  /**
2   *
3   * @author Augusto Gonzalez Bonorino <br><br>
4   *
5   * This is the class definition for QueueGonzalezBonorino
6   */
7  public class QueueGonzalezBonorino {
8
9    /**
10    * Instance variable for the queue's tail
11    */
```

```
12   private NodeGonzalezBonorino myTail;
13
14   /**
15    * Instance variable for the queue's head
16    */
17   private NodeGonzalezBonorino myHead;
18
19   /**
20    * Default constructor for QueueGonzalezBonorino
21    */
22   public QueueGonzalezBonorino() {
23
24     myTail = null;
25   } // constructor
```

Next, just like for our Stack, we need a method to make sure the Queue is not empty.

```
1  /**
2     * Method to check if Queue is empty
3     * @return boolean true if empty false otherwise
4     */
5    public boolean isEmpty() {
6
7      if (myHead == null)
8        return true;
9      else
10       return false;
11   } // check if empty
```

Now, we need a way to add and delete elements. This is achieved by the methods *enqueue()* and *dequeue()*.

In the enqueue method I first declare a new *NodeGonzalezBonorino* variable to hold the element previously at the tail of the queue. Next, *myTail* is updated by creating a new *NodeGonzalezBonorino*, setting its data to the element the user inputted and setting the next node to **null**. Then, we have to set the head of the queue if the Queue is empty (meaning that if the element being added is, currently, the only one on the Queue both *myHead* and *myTail* are equal) or update the Node to which *oldTail* is pointing at.

```
1  /**
2     * Method to add elements to the Queue
3     * @param element
4     */
5    public void enqueue(char element) {
6
7      NodeGonzalezBonorino oldTail = myTail;
8      myTail = new NodeGonzalezBonorino();
9      myTail.setData(element);
10     myTail.setNext(null);
11
12     if (isEmpty())
13       myHead = myTail;
14     else
15       oldTail.setNext(myTail);
16   } //enqueue
```

The dequeue method is implemented in the following fashion: A local variable to hold the character to be extracted is initialized which I creatively called *ans*, we make sure the Queue is not empty (otherwise Java would raise a ***NullPointerException***), assign the data extracted to *ans* and update the value of *myHead* by getting the value of the node it's pointing to. It is worth mentioning that if we are extracting the last element of the queue the data structure will be empty afterwards. Thus, we need to make sure that if this is the case we set *myTail* to null. This is implemented in lines 14-15.

```
1  /**
2     * Method to delete and element from the Queue
3     * @return element extracted
4     */
5    public char dequeue() {
6
7      char ans = ' ';
8
9      if (!isEmpty()) {
10
11       ans = myHead.getData();
12       myHead = myHead.getNext();
13
14       if (isEmpty())
15         myTail = null;
16
17     } //if
18
19     else
20       System.out.println("Queue is empty");
21
22     return ans;
23
24   } //dequeue
25
26 } // QueueGonzalezBonorino
```

4 Node class (Singly linked list)

Linked list is a special data structure that allows us to go beyond regular arrays by linking elements to pointers instead of storing them in a contiguous location. Thus, the linked list is not bounded by a predetermined length like regular arrays. Also, it is "singly" linked because it only points to the element the comes next, unlike "doubly" linked lists which can move forwards and backwards. I leverage this data structure to implement both the Stack and the Queue.

Our Node class contains two variables on is used to hold the data, which I conveniently called *myData*, while the second is an instance of itself used to point (link) to the next node in the list, which I also conveniently called *myNext*. In addition, it contains the common default constructor and a not-so-common semi-constructor that gives us the flexibility to assign *myData* a new value.

```
1  /**
2   *
```

```
 * @author Augusto Gonzalez Bonorino
 * This is the class definition for NodeGonzalezBonorino
 *
 */
public class NodeGonzalezBonorino {
  /**
   * Instance variable to hold the data
   */
  private char myData;

  /**
   * Instance variable for the data of the next node
   */
  private NodeGonzalezBonorino myNext;

  /**
   * Default constructor for NodeGonzalezBonorino
   */
  public NodeGonzalezBonorino() {

    myData = ' ';
    myNext = null;

  } // Default Constructor

  /**
   * Semi-constructor for NodeGonzalezBonorino
   * @param newData value to assign the data to myData
   */
  public NodeGonzalezBonorino(char newData){

    myData = newData;
    myNext = null;

  } // Semi-constructor
```

Besides the unusual semi-constructor there is nothing really interesting to explore in detail in the remaining code. The following lines of code correspond to the getters and setter of the class.

```
/**
   * Method to get myData
   * @return myData
   */
  public char getData() {

    return myData;
  } // getData

  /**
   * Method to get the value of the next node
   * @return myNext
   */
  public NodeGonzalezBonorino getNext() {

    return myNext;
  } //getNext

```

```
  /**
   * Method to set the value of myData
   * @param newData
   */
  public void setData(char newData) {

    myData = newData;
  } //setData

  /**
   * Method to set the value of myNext
   * @param newNext
   */
  public void setNext(NodeGonzalezBonorino newNext) {

    myNext = newNext;
  } // setNext

} // Node
```

5 Main class

Great, but how do I actually find cool palindromes? Well, all the fun stuff takes place in the MainGonzalezBonorino class. First, like with any class, I import the packages needed to process user input and read external files (util and io). Then, I instantiate all the variables I will use throughout the program and read the file with the following code:

```
import java.util.*;
import java.io.*;
/**
 *
 * @author Augusto Gonzalez Bonorino <br>
 *
 * assignment1GonzalezBonorino <br>
 * Due Date and Time: 09/24/21 before 9:00 a.m. <br><br>
 *
 * Purpose: Develop program that leverages a singly linked list,
    stacks and queues to find palindromes. <br><br>
 *
 * Input: A text file containing the words or sentences to check
    for palindromes.
 *
 * Output: The program prints out those words or sentences that
    were indeed palindromes.<br><br>
 *
 *
 * Certification of Authenticity: <br>
 *
 * I certify that this assignment is entirely my own work. <br>
 */
public class MainGonzalezBonorino {

  static Scanner keyboard = new Scanner(System.in);

```

```
25  public static void main(String[] args) {
26
27    File theMagicFile = null;
28
29    String fileName = "magicitems.txt";
30    String tempString = null;
31
32    String [] myMagicList = new String[666];
33    int numItems = 0;
34
35    String ans = "\nSome suggestions: \n"
36        + "\n* Check that the name of the file was typed correctly"
37        + "\n* Make sure that you are not missing any information
    in your item description in the file"
38        + "\n* Make sure you are not entering more or less items
    than specified";
39
40    try
41    {
42      System.out.print("Enter the name of the file: ");
43      fileName = keyboard.next();
44
45      theMagicFile = new File(fileName);
46
47      Scanner input = new Scanner(theMagicFile);
48
49      while(input.hasNextLine()) {
50
51        tempString = input.nextLine();
52        myMagicList[numItems] = tempString;
53
54        numItems++;
55
56      } //while
57
58      input.close();
59
60    } //try
61
62    catch(IndexOutOfBoundsException ex)
63      {
64      System.out.println("Oops, something went wrong!");
65      System.out.println("It seems that the program has reached an
    index out of bounds.");
```

Note the usage of *try* and *catch* when reading the file. There are more exceptions being taken into account but I excluded them from this snippet of code to avoid crowding the documentation with irrelevant code. Basically, after the user inputs the name of the file, the program reads each line *while* the file still has a line to read and appends it to an array. The usage of the while loop provides greater flexibility and cleaner code, which is why I prefer it over the for loop.

Once the file has been completely read, and the array fully populated, we must loop through every element in the array and push/enqueue it to a Stack and a Queue. For this, I use three loops. The *outer for loop* goes through

every element in the array, the *inner for loop* adds every character in that respective element to our Stack and Queue, and the *while* loop which then extracts and compares each character. In order to find our *coolWord* it is necessary to "normalize" every String in the array. I achieve this by eliminating all white spaces and capitalizing every character in the String (note that this is done in the outer loop, every time a new element is read).

```
1
2     for (int i = 0; i < myMagicList.length; i++) {
3
4       StackGonzalezBonorino stack = new StackGonzalezBonorino();
5       QueueGonzalezBonorino queue = new QueueGonzalezBonorino();
6       String coolWord = myMagicList[i].replaceAll(" ", "").
    toUpperCase();
7
8
9       for (int j = 0; j < coolWord.length(); j++) {
10
11        char c = coolWord.charAt(j);
12
13        stack.push(c);
14        queue.enqueue(c);
15
16
17      } //inner for loop
18
19      boolean isPalindrome = true;
20
21      while ( !stack.isEmpty() && isPalindrome) {
22
23        char elemS = stack.pop();
24        char elemQ = queue.dequeue();
25        if (Character.compare(elemS, elemQ) != 0)
26          isPalindrome = false;
27
28      } //while
29
30      if(isPalindrome)
31        System.out.println(myMagicList[i]);
32
33    } //Outer for loop
34
35
36  } // main
37
38 } // MainGonzalezBonorino
```

Inside the outer for loop is really where the magic happens. So, to ensure the reader comprehends it I will expand on its design a little further. Here are the inner loops extracted from the previous listing of code:

```
1 for (int j = 0; j < coolWord.length(); j++) {
2
3         char c = coolWord.charAt(j);
4
5         stack.push(c);
6         queue.enqueue(c);
```

```
7
8
9       } //inner for loop
10
11      boolean isPalindrome = true;
12
13      while ( !stack.isEmpty() && isPalindrome) {
14
15        char elemS = stack.pop();
16        char elemQ = queue.dequeue();
17        if (Character.compare(elemS, elemQ) != 0)
18          isPalindrome = false;
19
20      } //while
21
22      if(isPalindrome)
23        System.out.println(myMagicList[i]);
```

First, I take each character on our *coolWord* one by one and add it to both the Stack and the Queue. Then, the *while* loop pops/dequeues an element from both the Stack and the Queue, check if they are the same, and proceeds in the same fashion until the characters are different. If the *while* loop goes over the entire word, then *isPalindrome* remains true and the program prints our *coolWord*. Do this for every word in the text file and Voila! We have found all the palindromes in the given document.

6 Final thoughts

Documentation is about explaining how the program works and how it was designed. But, it does not have to be just that, here are a few funny palindromes to sweeten this report:

First we have a great bargain:
A nut for a jar of tuna.
But there's always someone unwilling to give up a nut, so he hired a hitman with the following pitch line:
Murder for a jar of red rum.

And a funny meme to conclude this document: