**CSC 151 Assignment #4**

1. **Honor Code**
2. *For individual assignments: Jane Doe and John Doe will be replaced by your full name(s)*

*I affirm that I have carried out my academic endeavors with full academic honesty.*

*[Signed, Manav Bilakhia]*

1. Resources/References

Geeks for geeks for syntax

1. **Java files and outputs**
2. Java files
3. package assignment;  
   /\*  
    \* I affirm that I have carried out the attached academic endeavors with full academic honesty.  
    \* Manav Bilakhia (MB)  
    \*/  
   */\*\*  
    \* An interface that describes the operations of a bag of objects.  
    \*  
    \** ***@author*** *Frank M. Carrano  
    \** ***@author*** *Timothy M. Henry  
    \** ***@version*** *5.0  
    \*/*public interface BagInterface<T> {  
    */\*\*  
    \* Gets the current number of entries in this bag.  
    \*  
    \** ***@return*** *The integer number of entries currently in the bag.  
    \*/* public int getCurrentSize();  
    */\*\*  
    \* Sees whether this bag is empty.  
    \*  
    \** ***@return*** *True if the bag is empty, or false if not.  
    \*/* public boolean isEmpty();  
    */\*\*  
    \* Adds a new entry to this bag.  
    \*  
    \** ***@param*** *newEntry The object to be added as a new entry.  
    \** ***@return*** *True if the addition is successful, or false if not.  
    \*/* public boolean add(T newEntry);  
    */\*\*  
    \* Removes one unspecified entry from this bag, if possible.  
    \*  
    \** ***@return*** *Either the removed entry, if the removal. was successful, or null.  
    \*/* public T remove();  
    */\*\*  
    \* Removes one occurrence of a given entry from this bag, if possible.  
    \*  
    \** ***@param*** *anEntry The entry to be removed.  
    \** ***@return*** *True if the removal was successful, or false if not.  
    \*/* public boolean remove(T anEntry);  
    */\*\* Removes all entries from this bag. \*/* public void clear();  
    */\*\*  
    \* Counts the number of times a given entry appears in this bag.  
    \*  
    \** ***@param*** *anEntry The entry to be counted.  
    \** ***@return*** *The number of times anEntry appears in the bag.  
    \*/* public int getFrequencyOf(T anEntry);  
    */\*\*  
    \* Tests whether this bag contains a given entry.  
    \*  
    \** ***@param*** *anEntry The entry to find.  
    \** ***@return*** *True if the bag contains anEntry, or false if not.  
    \*/* public boolean contains(T anEntry);  
    */\*\*  
    \* Retrieves all entries that are in this bag.  
    \*  
    \** ***@return*** *A newly allocated array of all the entries in the bag. Note: If the  
    \* bag is empty, the returned array is empty.  
    \*/* public T[] toArray();  
   } // end BagInterface

/\*  
 \* I affirm that I have carried out the attached academic endeavors with full academic honesty.  
 \* Manav Bilakhia (MB)  
 \*/  
package assignment;  
  
*/\*\*  
 \* A class of that tests singly linked lists that have String as the data.  
 \*  
 \** ***@author*** *Zeynep Orhan  
 \*/*public class LinkedBagDriver {  
 */\*\*  
 \* palindrome: Tests whether a string is  
 \*  
 \** ***@param*** *w a String  
 \** ***@return*** *True if the w is a palindrome, or false otherwise.  
 \*/* public static boolean palindrome(String w)  
 {  
 StringBuilder s = new StringBuilder();  
 s.append(w);  
 s.reverse();  
  
 return w.equals(s.toString());  
 }  
 */\*\*  
 \* addAll: adds all the strings in a given String array to this bag.  
 \*  
 \** ***@param*** *w a String array  
 \** ***@return*** *a BagInterface<String> which is a LinkedBag by adding all Strings in w  
 \*/* public static BagInterface<String> addAll(String [] w)  
 {  
 BagInterface<String> bag = new LinkedBag();  
 for (int i = 0; i<w.length; i++)  
 {  
 bag.add(w[i]);  
 }  
  
 return bag;  
 }  
 */\*\*  
 \* allPalindrome: adds all the strings in a given String array to a LinkedBag  
 \* and tests if they are all palindrome  
 \*  
 \** ***@param*** *w a String array  
 \** ***@return*** *True if all strings are palindromes, or false otherwise.  
 \*/* public static boolean allPalindrome(String [] w)  
 {  
 *addAll*(w);  
 for (int i = 0; i<w.length; i++)  
 {  
 if(!*palindrome*(w[i]))  
 return false;  
 }  
 return true;  
 }  
 */\*\*  
 \* allPalindromes: adds all the strings in a given String array to a LinkedBag  
 \* and returns a string of the palindromes  
 \*  
 \** ***@param*** *w a String array  
 \** ***@return*** *a string that has all the palindrome strings in the bag separated by space,  
 \* most recent one should be at the beginning  
 \*/* public static String allPalindromes(String [] w)  
 {  
 BagInterface<String> bag = new LinkedBag();  
 for (int i = 0; i<w.length;i++)  
 {  
 bag.add(w[i]);  
 }  
 Object[] arr = bag.toArray();  
 String s = "";  
 for (int i = 0; i<arr.length; i++)  
 {  
 if(*palindrome*((String)arr[i]))  
 s = s + " " + arr[i];  
 }  
 return s;  
  
 }  
 public static void main(String[] args) {  
// *TODO Auto-generated method stub* BagInterface<String> words = new LinkedBag<>();  
 String w[] = { "a", "ab", "aba", "abba", "abcba", "abb" };  
 for (String word : w)  
 words.add(word);  
 System.*out*.println("Add all words");  
 System.*out*.println(words);  
 Object wordArray[] = words.toArray();  
 for (Object word : wordArray)  
 if (*palindrome*((String) word))  
 System.*out*.println(word + " is a palindrome");  
 else  
 System.*out*.println(word + " is not a palindrome");  
 System.*out*.println("Is the list empty? " + words.isEmpty());  
 System.*out*.println("Number of aa in the list " + words.getFrequencyOf("aa"));  
 System.*out*.println("Number of abb in the list " + words.getFrequencyOf("abb"));  
 System.*out*.println("Add abb " + words.add("abb"));  
 System.*out*.println("Number of abb in the list after adding one more " + words.getFrequencyOf("abb"));  
 System.*out*.println("Number of words in the list " + words.getCurrentSize());  
 words.clear();  
 System.*out*.println("Number of words in the list after clear " + words.getCurrentSize());  
 LinkedBag<String> words2 = (LinkedBag<String>) *addAll*(w);  
 System.*out*.println("Is the list empty? " + words2.isEmpty());  
 System.*out*.println("Number of aa in the list " + words2.getFrequencyOf("aa"));  
 System.*out*.println("Number of abb in the list " + words2.getFrequencyOf("abb"));  
 System.*out*.println("Add abb " + words2.add("abb"));  
 System.*out*.println("Number of abb in the list after adding one more " + words2.getFrequencyOf("abb"));  
 System.*out*.println("Number of words in the list " + words2.getCurrentSize());  
 words2.clear();  
 System.*out*.println("Number of words in the list after clear " + words2.getCurrentSize());  
 System.*out*.println("Are all palindromes? " + *allPalindrome*(w));  
 System.*out*.println("Show all palindromes " + *allPalindromes*(w));  
 }  
}

package assignment;  
/\*  
 \* I affirm that I have carried out the attached academic endeavors with full academic honesty.  
 \* Manav Bilakhia (MB)  
 \*/  
  
import java.util.StringJoiner;  
  
*/\*\*  
 \* A class of bags whose entries are stored in a chain of linked nodes. The bag  
 \* is never full.  
 \*  
 \** ***@author*** *Frank M. Carrano  
 \** ***@author*** *Timothy M. Henry  
 \** ***@version*** *5.0  
 \*/*public final class LinkedBag<T> implements BagInterface<T> {  
 private Node firstNode; // Reference to first node  
 private int numberOfEntries;  
  
 */\*\*  
 \* add: Adds a new entry to this bag.  
 \*  
 \** ***@param*** *newEntry The object to be added as a new entry  
 \** ***@return*** *True if the addition is successful, or false if not.  
 \*/* public boolean add(T newEntry)  
 {  
 Node newNode = new Node(newEntry);  
 newNode.next = firstNode;  
 firstNode = newNode;  
 numberOfEntries++;  
 return true;  
 }  
  
 */\*\*  
 \* toArray: Retrieves all entries that are in this bag.  
 \*  
 \** ***@return*** *A newly allocated array of all the entries in this bag.  
 \*/* public T[] toArray() {  
 T[] result = (T[]) new Object[numberOfEntries];  
 int index = 0;  
 Node currentNode = firstNode;  
 while ((index < numberOfEntries) && (currentNode != null)) {  
 result[index] = currentNode.data;  
 index++;  
 currentNode = currentNode.next;  
 }  
 return result;  
 }  
  
 */\*\*  
 \* isEmpty: Sees whether this bag is empty.  
 \*  
 \** ***@return*** *True if this bag is empty, or false if not.  
 \*/* public boolean isEmpty() {  
 return numberOfEntries == 0;  
 }  
  
 */\*\*  
 \* getCurrentSize: Gets the number of entries currently in this bag.  
 \*  
 \** ***@return*** *The integer number of entries currently in this bag.  
 \*/* public int getCurrentSize() {  
 return numberOfEntries;  
 }  
  
 */\*\*  
 \* remove: Removes one unspecified entry from this bag, if possible.  
 \*  
 \** ***@return*** *Either the removed entry, if the removal was successful, or null.  
 \*/* public T remove() {  
 T result = null;  
 if (firstNode != null) {  
 result = firstNode.data;  
 firstNode = firstNode.next;  
 numberOfEntries--;  
 }  
 return result;  
 }  
  
 */\*\*  
 \* remove: Removes one occurrence of a given entry from this bag, if possible.  
 \*  
 \** ***@param*** *anEntry The entry to be removed.  
 \** ***@return*** *True if the removal was successful, or false otherwise.  
 \*/* public boolean remove(T anEntry) {  
 boolean result = false;  
 Node nodeN = getReferenceTo(anEntry);  
 if (nodeN != null) {  
 nodeN.data = firstNode.data;  
 firstNode = firstNode.next;  
 numberOfEntries--;  
 result = true;  
 }  
 return result;  
 }  
  
 */\*\*  
 \* clear: Removes all entries from this bag.  
 \*/* public void clear() {  
 while (!isEmpty()) {  
 remove();  
 }  
 }  
  
 */\*\*  
 \* getFrequencyOf: Counts the number of times a given entry appears in this bag.  
 \*  
 \** ***@param*** *anEntry The entry to be counted.  
 \** ***@return*** *The number of times anEntry appears in this bag.  
 \*/* public int getFrequencyOf(T anEntry) {  
 int frequency = 0;  
 int counter = 0;  
 Node currentNode = firstNode;  
 while ((counter < numberOfEntries) && (currentNode != null)) {  
 if (anEntry.equals(currentNode.data)) {  
 frequency++;  
 }  
 counter++;  
 currentNode = currentNode.next;  
 }  
 return frequency;  
 }  
  
 */\*\*  
 \* contains: Tests whether this bag contains a given entry.  
 \*  
 \** ***@param*** *anEntry The entry to locate.  
 \** ***@return*** *True if the bag contains anEntry, or false otherwise.  
 \*/* public boolean contains(T anEntry) {  
 boolean found = false;  
 Node currentNode = firstNode;  
 while (!found && (currentNode != null)) {  
 if (anEntry.equals(currentNode.data)) {  
 found = true;  
 } else {  
 currentNode = currentNode.next;  
 }  
 }  
 return found;  
 }  
  
 */\*\*  
 \* toString: Convert this bag to a String for displaying.  
 \* each item will be comma separated and a space after comma enclosed in [ and ]  
 \* if we have a b c d in the bag a is the most recent one and will be converted as  
 \* [a, b, c, d]. StringJoiner is a good option to use.  
 \*/* @Override  
 public String toString() {  
 StringJoiner joiner = new StringJoiner(", ", "[", "]");  
 Node currentNode = firstNode;  
 for (int index = 0; index < numberOfEntries; index++)  
 joiner.add(currentNode.data.toString());  
 currentNode = currentNode.next;  
 return joiner.toString();  
 }  
 // getReferenceTo: Should be private so not written in Javadoc format  
 // parameter is an entry of type T  
 // Locates a given entry within this bag.  
 // Returns a reference to the node containing the entry, if located,  
 // or null otherwise.  
 private Node getReferenceTo(T anEntry) {  
 boolean found = false;  
 Node currentNode = firstNode;  
 while (!found && (currentNode != null)) {  
 if (anEntry.equals(currentNode.data)) {  
 found = true;  
 } else {  
 currentNode = currentNode.next;  
 }  
 }  
 return currentNode;  
 }  
  
 // Node class: Should be private so not written in Javadoc format  
 // A class that represents a Node with a data of type generic and  
 // a Node type next link  
 private class Node {  
 // Private instance variables  
 // Entry in bag  
 // Link to next node  
 private T data;  
 private Node next;  
 // Constructors.  
 // Constructor with a data of type T  
 public Node(T dataPortion) {  
 this(dataPortion, null);  
 }  
 // Constructor with a data of type T and a next of type Node  
 public Node(T dataPortion, Node nextNode) {  
 data = dataPortion;  
 next = nextNode;  
 }  
 // get/set methods  
  
 public T getData() {  
 return data;  
 }  
  
 public void setData(T data) {  
 this.data = data;  
 }  
  
 public Node getNext() {  
 return next;  
 }  
  
 public void setNext(Node next) {  
 this.next = next;  
 }  
 } // end Node  
} // end LinkedBag

1. Sample output 1
2. Describe your test 1: testing the add function

Text

Description automatically generated

Text

Description automatically generated

1. Text output 1:

Add all words

[abb, abb, abb, abb, abb, abb]

1. Screenshot 1:

****

1. Sample output 2
2. Describe your test 2: checking the palindrome method

Text

Description automatically generated

Text

Description automatically generated

1. Text output 2:

abb is not a palindrome

abcba is a palindrome

abba is a palindrome

aba is a palindrome

ab is not a palindrome

a is a palindrome

1. Screenshot 2:

**Text

Description automatically generated**

1. Sample output 3
2. Describe your test 3: testing the get frequency method

Text

Description automatically generated



1. Text output 3:

Number of aa in the list 0

Number of abb in the list 1

1. Screenshot 3:

****