HW9 – Neta Tarshish

List -

```
/** A linked list of character data objects.
 *  (Actually, a list of Node objects, each holding a reference to a character data object.
 *  However, users of this class are not aware of the Node objects. As far as they are concerned,
 *  the class represents a list of CharData objects. Likwise, the API of the class does not
 *  mention the existence of the Node objects). */
public class List {


   public boolean tempMark = false;

   // Points to the first node in this list
   private Node first;

   // The number of elements in this list
   private int size;

   /** Constructs an empty list. */
   public List() {
      first = null;
      size = 0;
   }

   /** Returns the number of elements in this list. */
   public int getSize() {
	      return size;
   }

   /** Returns the first element in the list */
   public CharData getFirst() {
      return first.cp;
    }


    /** GIVE Adds a CharData object with the given character to the beginning of this list. */
    public void addFirst(char chr) {
    CharData add = new CharData(chr);
    Node newNode = new Node(add);
    if (first == null) {
      first = newNode;
    } else {
      newNode.next = first;
      first = newNode;
    }
    this.size++;
  }

    /** GIVE Textual representation of this list. */
    public String toString() {
      Node current = first;
      String result = "(";
      for (int i = 0; i < size; i++){
        result += current.cp.toString() + " ";
        current = current.next;
      }
      return result.substring(0,result.length()-1)+")";
    }

    /** Returns the index of the first CharData object in this list
     *  that has the same chr value as the given char,
     *  or -1 if there is no such object in this list. */
```

```
   public int indexOf(char chr) {
   Node current = first;
   int counter = 0;
   while (current != null) {
      if (current.cp.equals(chr)) {
         return counter;
      }
      current = current.next;
      counter++;
   }
   return -1;
}

   /** If the given character exists in one of the CharData objects in this list,
    *  increments its counter. Otherwise, adds a new CharData object with the
    *  given chr to the beginning of this list. */
   public void update(char chr) {
      Node current = first;
      int placeOfchr = indexOf(chr);
      if(placeOfchr==-1){
        this.addFirst(chr);
      }
      else{
         for(int i = 0; i < placeOfchr ; i++){
            current = current.next;
         }
         current.cp.count++;
      }
   }

   /** GIVE If the given character exists in one of the CharData objects
```

```
     *  in this list, removes this CharData object from the list and returns
     *  true. Otherwise, returns false. */
    public boolean remove(char chr) {
    Node current = first;
    Node previousNode = null;
    while (current != null) {
        if (current.cp.equals(chr)) {
            if (previousNode == null) {
                first = first.next;
            } else {
                previousNode.next = current.next;
            }
            size--;
            return true;
        }
        previousNode = current;
        current = current.next;
    }
    return false;
}

    /** Returns the CharData object at the specified index in this list.
     *  If the index is negative or is greater than the size of this list,
     *  throws an IndexOutOfBoundsException. */
    public CharData get(int index) {
        Node current = this.first;
        int counter = 0;
        if(index >= size || index < 0){
            throw new IndexOutOfBoundsException("index cannot be negative or larger than list
size");
        }
```

```
  while(current != null && counter < index){
    current = current.next;
    counter ++;
  }
  return current.cp;
}

/** Returns an array of CharData objects, containing all the CharData objects in this list. */
public CharData[] toArray() {
       CharData[] arr = new CharData[size];
       Node current = first;
       int i = 0;
  while (current != null) {
       arr[i++]  = current.cp;
       current = current.next;
  }
  return arr;
}

/** Returns an iterator over the elements in this list, starting at the given index. */
public ListIterator listIterator(int index) {
       // If the list is empty, there is nothing to iterate
       if (size == 0) return null;
       // Gets the element in position index of this list
       Node current = first;
       int i = 0;
  while (i < index) {
    current = current.next;
    i++;
  }
  // Returns an iterator that starts in that element
```

```
            return new ListIterator(current);
    }
}
```

LanguageModel –

```
import java.util.HashMap;
import java.util.Random;

public class LanguageModel {

    // The map of this model.
    // Maps windows to lists of charachter data objects.
    HashMap<String, List> CharDataMap;

    // The window length used in this model.
    int windowLength;

    // The random number generator used by this model.
            private Random randomGenerator;




    /** Constructs a language model with the given window length and a given
     *  seed value. Generating texts from this model multiple times with the
     *  same seed value will produce the same random texts. Good for debugging. */
    public LanguageModel(int windowLength, int seed) {
        this.windowLength = windowLength;
        this.randomGenerator = new Random(seed);
        CharDataMap = new HashMap<String, List>();
    }
```

```
/** Constructs a language model with the given window length.
 * Generating texts from this model multiple times will produce
 * different random texts. Good for production. */
public LanguageModel(int windowLength) {
    this.windowLength = windowLength;
    this.randomGenerator = new Random();
    CharDataMap = new HashMap<String, List>();
}

/** Builds a language model from the text in the given file (the corpus). */
          public void train(String fileName) {
String window = "";
char c;
int counter = 0;
In in = new In(fileName);
while (counter < this.windowLength && !in.isEmpty()) {
    window += in.readChar();
    counter++;
}
//to be removed
//int printable = 0;

while (!in.isEmpty()){
    c = in.readChar();
    if (CharDataMap.get(window) == null) {
        List probs = new List();
        CharDataMap.put(window, probs);

      }
    List probsToUpdate = CharDataMap.get(window) ;
    probsToUpdate.update(c);
    window = window.substring(1) + c;


  }


  for (List probs : CharDataMap.values()) {
    calculateProbabilities(probs);
  }
}


  // Computes and sets the probabilities (p and cp fields) of all the
        // characters in the given list. */
        public void calculateProbabilities(List probs) {
  int size = probs.getSize();
  int numberOfChars = 0;
  double cp = 0;
  for(int j = 0; j < size; j++){
        numberOfChars += probs.get(j).count;
  }


        for(int i = 0; i < size; i++) {
                CharData item = probs.get(i);
                double p = (double) (item.count) / (double) (numberOfChars);
                cp += p;
                item.p = p;
                item.cp = cp;
  }
```

```
}

   // Returns a random character from the given probabilities list.
            public char getRandomChar(List probs) {
                        double random = randomGenerator.nextDouble();
                        for(int i = 0; i<probs.getSize(); i++){
                                    if(probs.get(i).cp > random){
                                                return probs.get(i).chr;
                                    }
                        }
                        return '1';
            }

   /**
             * Generates a random text, based on the probabilities that were learned during training.
             * @param initialText - text to start with. If initialText's last substring of size numberOfLetters
             * doesn't appear as a key in Map, we generate no text and return only the initial text.
             * @param numberOfLetters - the size of text to generate
             * @return the generated text
             */
            public String generate(String initialText, int textLength) {
                        if (textLength < windowLength) {
      return initialText;
   }

   StringBuilder generatedText = new StringBuilder(initialText);

   String window = initialText.substring(initialText.length() - windowLength);
```

```
      while (generatedText.length()-windowLength < textLength) {
        List charList = CharDataMap.get(window);

        if (charList == null) {
          System.out.println("Break");
          break;
        }

        char nextChar = getRandomChar(charList);
        generatedText.append(nextChar);
        window = window.substring(1) + nextChar;

      }

      return generatedText.toString();
    }



      /** Returns a string representing the map of this language model. */
            public String toString() {
                    StringBuilder str = new StringBuilder();
                    for (String key : CharDataMap.keySet()) {
                            List keyProbs = CharDataMap.get(key);
                            str.append(key + " : " + keyProbs.toString() + "\n");
                    }
                    return str.toString();
            }

      public static void main(String[] args) {
            int windowLength = Integer.parseInt(args[0]);
        String initialText = args[1];
        int generatedTextLength = Integer.parseInt(args[2]);
        Boolean randomGeneration = args[3].equals("random");
        String fileName = args[4];
        // Create the LanguageModel object
        LanguageModel lm;
        if (randomGeneration)
                lm = new LanguageModel(windowLength);
        else
                lm = new LanguageModel(windowLength, 20);
                // Trains the model, creating the map.

        lm.train(fileName);

        // Generates text, and prints it.
        System.out.println(lm.generate(initialText, generatedTextLength));
    }

    }
```