```
/**
 * 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 {

    // 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 newCharData = new CharData(chr);
    Node newNode = new Node(newCharData, first); // Simplified construction
    first = newNode;
    size++;
}

/** GIVE Textual representation of this list. */
public String toString() {
    StringBuilder str = new StringBuilder("(");
    Node current = first;
    while (current != null) {
        str.append(current.cp.toString());
        if (current.next != null) {
            str.append(" ");
        }
        current = current.next;
    }
    str.append(")");
    return str.toString();
}

/**
 * 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 index = 0;
    while (current != null) {
        if (current.cp.chr == chr) {
            return index;
        }
        current = current.next;
        index++;
        }
        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;
        while (current != null) {
            if (current.cp.chr == chr) {
                current.cp.count++; // Assuming CharData has a 'count' field
                return;
            }
            current = current.next;
        }
        addFirst(chr);
    }

    /**
     * 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) {
        if (first != null && first.cp.chr == chr) {
            first = first.next;
            size--;
            return true;
        }
        Node current = first;
        Node prev = null;
        while (current != null) {
            if (current.cp.chr == chr) {
                if (prev != null) {
```

```
                prev.next = current.next;
                size--;
                return true;
            }
        }
        prev = 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) {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException();
    }
    Node current = first;
    for (int i = 0; i < index; i++) {
        current = current.next;
    }
    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);
    }

}
```

```java
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;
        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;
        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 fileString = "";
    In input = new In(fileName);
    fileString = input.readAll();
    for (int i = 0; i + windowLength < fileString.length(); i++) {
        String key = fileString.substring(i, i + windowLength);
        List value = CharDataMap.get(key);
        if (value != null) {
            if (value.indexOf(fileString.charAt(i + windowLength)) != -1) {
                value.update(fileString.charAt(i + windowLength));

            } else {
                value.addFirst(fileString.charAt(i + windowLength));
            }
        } else {
            CharDataMap.put(key, new List());
            CharDataMap.get(key).addFirst(fileString.charAt(i + windowLength));
        }
        calculateProbabilities(CharDataMap.get(key));
    }
}

// Computes and sets the probabilities (p and cp fields) of all the
// characters in the given list. */
public void calculateProbabilities(List probs) {
        // First, calculate the total number of characters
        int totalChars = 0;
        for (CharData cd : probs.toArray()) {
            totalChars += cd.count;
        }

        // Now calculate and set the probabilities (p and cp)
        double acomulativeProbability = 0.0;
        for (CharData cd : probs.toArray()) {
            cd.p = (double) cd.count / totalChars; // Calculate the probability of each character
            acomulativeProbability += cd.p; // Update
            cd.cp = acomulativeProbability; // Set the cumulative probability for the character
            }
    }

    // Returns a random character from the given probabilities list.
    public char getRandomChar(List probs) {
        double r = randomGenerator.nextDouble(); // random number in [0,1)
        CharData[] charDataArray = probs.toArray(); // Assuming List has a toArray() method returning CharData[]

        // Iterate through the list until finding the character whose cumulative probability is greater than r
        for (CharData cd : charDataArray) {
            if (cd.cp > r) {
                return cd.chr; // Return the character of the current element
            }
        }

        return charDataArray[charDataArray.length - 1].chr;
    }

    /**
     * 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 (initialText.length() >= windowLength) {
            StringBuilder generatedText = new StringBuilder(initialText);
            for (int i = 0; i < textLength; i++) {
                String currentWindow = generatedText.substring(generatedText.length() - windowLength);
                List probs = CharDataMap.get(currentWindow);
                if (probs == null) {
                    break; // If the current window is not found, stop the generation process
                }
                char nextChar = getRandomChar(probs); // Get a random character based on the current window's
probabilities
                generatedText.append(nextChar);
```

```java
            }
            return generatedText.toString();
        } else {
            return initialText; // Return the initial text if its length is less than the window length
        }
    }


    /** 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 + "\n");
        }
        return str.toString();
    }

    public static void main(String[] args) {
        // Your code goes here
    }
}
```