Project 3:
Gallergrooverture and the generation of random words

CSCI 1913: Introduction to Algorithms,
Data Structures, and Program Development

1 Essential information

Project 3 is designed as a 3 week assignment. It builds upon our study of data structures, including both material we've already covered at the point of assignment, and some concepts from the final weeks of the course, most notably Tree structured data structures. In this project you will build some highly efficient, customized data structures. In particular, you will build a char bag class which you will design yourself (Bags are like sets, in that they store no order, but unlike sets they do track *how many times* each element is in the Bag). Additionally you will build a Trie data structure. The Trie datastructure is a highly efficient data structure for storing and retrieving values indexed by short (one word) strings (Think of it like a specialized version of a python dictionary).

Project two is not explicitly split into phases, but is instead built of several core classes. These classes can largely be built in any particular order, although I will note that some classes may be easier to understand after a few weeks of lecture. You shouldn't need those lectures to work on these classes, but it may help.

Project 3 is due on Tuesday May 5th at 6:00pm. I do not intend to extend this deadline, as this is already pushing into the finals week. I only feel OK putting the deadline that late as our class has no final.

This is version 2 of the project writeup.

Changelog:

1. (4/18)
   - Updated word list removing a few slurs. My word list source apparently contextualized those as words due to historic use. I don't want those words being generated by my program. So I got rid of them. The Dictionary class is likewise updated so the "example" output works correctly against the new list.
   - fixed a type. Trie should have put, not set.
   - Duplicated section on acceptable code libraries at top of file – you are not allowed to use pre-built datastructures for this. Nothing more advanced than an array. You are expected to build these structures directly.

- Listed an aspirational upper-bound for CharBag performance – it can be done in $O(1)$ for all methods. You are not required to achieve this.
- Further explained `public void train(LetterSample)` method – it needs to get a CharBag from the Trie and add the char to it.
- `TrieNode` method `public TrieNode<T> getChild(char letter)` was explained better.

2. (4/14) initial version

2 Introduction

If there's one thing that slows down quiz writing in the class, it's the generation of nonsense words.[1] Put simply, nonsense terms in quizzes help force you to rely only on your code reading skill to understand the behavior of a piece of code. These nonsense words play an important pedagogical part of "read the code" style problems, but they can be rather loathsome to produce after using all of my favorite nonsense words.

In this project we will be building a `Giberisher` class, a class whose primary goal is to generate gibberish – random words that look like real English language words (I.E. sequences of letters that *should* be pronounceable). Building the Gibberisher will rely on a few problem-specific highly optimized data structures. While the problem of generating gibberish itself is not particularly time-consuming, we will be training our models on 62,915 words, therefore it's important that our code operates efficiently. Before describing the objects we will use to do this, let us first try to understand the algorithm we are using, and why it is more complicated than some "simple" first principals approaches.

2.1 Generating Random Words

Generating random words is actually quite easy:

```
public static String randomWord(int length) {
  Random rng = new Random();
  String word = "";
  for(int i = 0; i < length; i++) {
    word += ('a' + rng.nextInt(26));
  }
  return word;
}
```

The above code will generate a random word, made only of the English letters 'a' through 'z' forced to have a specific pre-specified length. The issue with the above code is that the words it generates are garbage. While `randomWord(6)` can generate such wonders as "docruf", it is just as likely to output "aaaaaa" or "zqzzrz" or any other 6 random letters.

This process can be slightly improved by first noticing that the English language does not use letters every letter equally. Letters like "e" show up much more than letters like "q" or "z". Therefore, we could improve our algorithm by counting how often each letter is used in the English language, and then generating words where letters are used at the correct rate.

[1]This paragraph is laughably not true - most fun part every time

a	7.7%
b	1.8%
c	4.0%
d	3.8%
e	11.5%
...	...
z	0.4%

The class `CharBag` you write will be capable of doing this. Its two primary jobs will be counting how many times it sees each letter, and generating random letters following those probabilities. Using the correct letter frequency does improve things. Now we generate words like "imelesmeldd", "rosle" and "eioirtgnosddmesdynrutkstiellmssore". Unfortunately, these words are still far from pronounceable.

To achieve our goal of random pronounceable words we need to bring in one more idea: a letter can be seen as "determined" by the letters before it. What I mean by this is that we would expect different letters to follow "ont" in a word ('r' and 'e' being most common) than we would expect to follow "ter" ('s' is most common, with 'i' taking a distant second). Therefore it is not enough to have one model for which letter to use for any place in a word, instead we need a different model for which letter to use depending on the preceding letters. That is, we need a different count of letters for "ont" than we use for "ter" or "aft", etc.

Translating into Java, this means we will need many `CharBag` objects, representing the many different word segments we see in our real data. Each `CharBag` needs to be associated with the word segment related to it. We will use a `Trie` data structure to organize these `CharBag`s and give us an efficient way to get the correct `CharBag` for any given situation.

2.2 Our algorithm

Our algorithm has two parts, a "training" part, which needs to happen once (every time we start up the program), and a word generation part which can be done as many times as needed.

For the training part, we will take a list of all English words. We will split each word into "samples" of a given length. For example, the "samples" of length 3 for the word "after" are:

- "" ⇒ 'a' (the word starts with a)
- "a" ⇒ 'f' (after a comes f)
- "af" ⇒ 't' (after af comes t)
- "aft" ⇒ 'e' (after aft comes e)
- "fte" ⇒ 'r' (after fte comes r)
- "ter" ⇒ STOP (the string ends after ter)

Note that at the beginning of the string we have “smaller” samples than 3, but once we get into the middle of the word, all samples are maximum length 3. Likewise, note that we will have a sample at the end indicating that this is where the string stops.

As we generate these samples, we will count them. So we will have one count for each “aft” sample, which will tell us which letters come after “aft” in all the words in our dictionary. We will have a second count for “fte”, another count for “ter” and so-forth. Once we have all words processed this large datastructure will be able to tell us how likely any given letter is to follow any observed 3 letter segment. (as well as having counts for the first letter (“” samples) and second letters such as the “a” sample etc.)

We can then generate words using this pseudocode (where k is the sample size, 3 in the above examples)

```
word = ""
While word doesn't end with the STOP letter:
    sample = get the last k letters of word
    (this should just be word itself, if word is shorter than k)
    get the letter counts for sample
    generate a next letter based on those letter counts
    word = word + nextLetter
return word.
```

For example, we might generate the word “gallergrooverture” with a sample-size 4 model. Our model would start by splitting every word into size 4 samples, and counting each next letter for every 4-letter segment of each string in our dictionary. Then we would start by generating “g” at random. This was based on the distribution of first letters in all words, while not the most common option, “g” is still a relatively likely outcome. Then the algorithm kicks off:

- From “g” we generated “a”
- From “ga” we generated “l”
- From “gal” we generated another “l”
- From “gall” we generated “e”
- From “alle” we generated “g”
- From “lleg” we generated “r”
- From “legr” we generated “o”
- ...
- From “rtur” we generated “e”
- From “ture” we generated “.” (the char, indicating we should stop).

2.3 On understanding this algorithm

This algorithm has a lot of moving pieces, I know that. And I understand it might be hard to understand all these pieces all at once. Don't get discouraged at this point. I've provided a class decomposition which will help you split this large project into a series of self-contained separate problems. Each of these parts can be understood on their own, and can be explained without reference to this broader algorithm until we put them together at the very end. While the classes may seem a bit arbitrary without a sense of the "big picture" they are well defined. You may find it helpful to understand and possibly even implement, many of these core objects on their own before trying to fully piece together this algorithm. Of course, if you are having trouble understanding this algorithm at a high level, I'm always happy to explain it further over email or in office hours.

3 Acceptable code libraries

You should make NO use of Java's built in collection types. A core purpose of this Project is to implement your own data structures. You will lose substantial points for any solution using Java's built-in datastructures. If there's any code you want to use that wasn't explicitly listed in this document you should email the course staff to be sure it's OK before using it.

4 Required Classes

There are five required classes for this project. While these can be implemented in any order, I recommend the following order.

- LetterSample - representing a part of a word, and the letter that follows it.
- CharBag - a datastructure for counting characters.
- TrieNode and Trie - a linked datastructure organized as a tree for quickly storing and retrieving data based on a series of letters. This class can be programmed before we cover Trees in class, but may be easier to understand after we've discussed the basics.
- Gibberisher - this class implements the primary algorithm as described above. If you make use of the other classes well, this class doesn't actually end up having much code.

4.1 LetterSample

The `LetterSample` class has relatively simple non-static properties, and a somewhat tricky static method. We will list these requirements separately.

Each instance of Letter Sample represents one of the "letter samples" from above (I.E. "aft" ⇒ 'e') and stores a segment (a short string) and a nextLetter (the char following the segment) Each instance is required to have the following public properties.

- A constructor `public LetterSample(String, char)` which takes a value for the segment string and the nextLetter char.

- `public String getSegment()` which gets the segment string for this object.

- `public char getNextLetter()` which gets the next letter for this object

- `toString` which generates output in the format seen in the test file (roughly speaking: it should return `"segment" -> nextLetter` (with quotes added around segment))

That's it.

LetterSample additionally has two static properties. First, a public static final char:

```
public static final char STOP = '.';
```

This marks the specific letter which we will use to represent "end of string". This will allow us to latter generate letters until we hit a naturally generated end of string.

Secondly:

```
    public static LetterSample[] toSamples(String input, int segmentSize) {
```

This function is in charge of taking a string and generating letter samples from it.

The function should have the following major steps:

- "clean up" the string. This should involve making sure the string is lowercase, and removing any non-alphabetic characters from the string. We want the string to only have lowercase alphabetical letters before the next step.

- add the STOP character to the end of the string.

- Split the string into letter segments and fill an array with them.
  - The array should have the same length as the string (after being cleaned, and including the STOP character)
  - Each segment should represent one letter (again including STOP) of the string, and include the preceding `segmentSize` letters in the String before that letter.
  - If the letter is towards the beginning of the string, and therefore doesn't have `segmentSize` letters before it, then it should simply return all letters in the string before it.
  - Examples of this are given both in the test file for this class and in the discussion above.

You may find the following built-in java methods useful:

- `String.charAt`

- `Character.isAlphabetic`

- `String.length`
- `Math.min`
- `String.replace` (I didn't end up needing this one, but I could see some solutions that would use them...
- `String substring` (possibly both versions, make sure you understand the inputs to the two versions of this method)

4.2 CharBag

The CharBag class is a custom purpose version of the Bag Abstract Data Type (ADT) The Bag ADT is most similar to the Set ADT you worked with in Lab. Like a Set, the Bag stores elements with no concept of order, a letter is simply in the Bag, or it is not. Unlike a Set, however, a letter can be in a Bag many times.

While a generic purpose Bag class would be a fun project, making these maximally efficient is difficult. For our purposes we only need a bag to count occurrences of chars, in particular the 26 lowercase English letters 'a' through 'z', and then one additional count for any other letter (we'll use '.' the STOP character again here) Since we only have 27 chars we care about, you can also think of this class as having the job of counting the letters.

This object will also have the responsibility of randomly generating letters, following the frequencies it stores. So if this object stores 5 letters ('a', 'a', 'a', 'b', and 'b') then it should generate 'a' with chance $3/5 = 60\%$ and 'b' with chance $2/5 = 40\%$.

I will not mandate a specific approach to implementing this datastrucutre. You will have to figure out a fast enough way to do this based on what we've done so far. (There is no "right" way, but there certainly are some wrong ways that lead to excessive memory use or inefficient algorithms) You should know enough to achieve an efficient implementation as of the date this project is assigned.

Some ideas to think on here. (Note, these don't all push towards the same solution...)

- Bag is similar to Set, you've implemented a Set that worked pretty fast.
- This is strictly fewer features than a List type datastructure. Assuming you can get efficiency down, any of our list approaches could work here.
- Were going to have to store A LOT of chars, but mostly copies of the same char. One object might be storing 20,000 'a'. Instead of storing many copies, perhaps modify your design to store a single letter, and an associated "count".
- We know the exactly list of 27 chars in advance, you could theoretically do this without any data structure design and just 27 int variables. (This solution would be simple, but long to code, as it means 27 if statements, but maybe you could use an array and some char math to make this work... with less code)

- I'm giving you control here, but I'm willing to make some pretty strong recommendations if you really can't think of a good way to do this.
- A great implementation can achieve $O(1)$ performance for all methods (O.K. some will be $O(27)$, but that still counts as $O(1)$). You will not lose points if you don't find this solution. But I wanted to give a sense of the what's possible.

Your `CharBag` class has must have the following public properties:

- A default constructor which creates an empty `CharBag`
- `public void add(char)` This function should add a char to the charBag. If the char is an uppercase letter, it should be converted to a lowercase letter before adding (uppercase letters and lowercase letters should be treated as equal) If the char is not an English alphabet letter ('a' through 'z') it should be converted to the STOP character '.'
- `public void remove(char c)` This function should remove a char from the charBag. If the input letter is not in the charBag no change should happen. If the letter is in the charBag multiple times, only one copy should be removed, not all copies. The input char should be converted as noted on add, uppercase letters should be treated as lowercase letters, and non-letters should all be treated like '.'
- `public int getCount(char)` gets how many times a given char is in the `CharBag`. Follow the character conversion/equivalence rules from add/remove. If a letter is not in the `CharBag` this function should return 0.
- `public int getSize()` returns the total size of the `charBag` (I.E. the number of adds minus the number of successful removes.
- `public String toString()` should return a string noting the count of each letter. The format for this can be found in the test file.
- `public char getRandomChar()` This function should return a randomly chosen char from the chars in the char bag. This should be chosen in proportion to the number of times a given char is in the datastructre.

  This method can be somewhat tricky to implement depending on how you store your characters. If you store them in one big 100,000 long array it's quite easy, simply pick a random element from the array. But if you are using any sort of storage where you only store each char once, with an associated count, you need to be a little more clever. One approach goes something like this:

  ```
  Let count =  integer between 0 (inclusive) and getSize(exclusive)
  for letters a through z:
      count -= getCount(letter)
      if count < 0 return letter
  return '.'
  ```

This algorithm may be pretty inefficient if you implement it exactly as written (kinda depends on getCount and getSize), but this idea can probably be adopted to your data structure easily.

If the `CharBag` is empty, this function should return the stop character '.'

For this class you will find the Random and Character classes and methods useful.

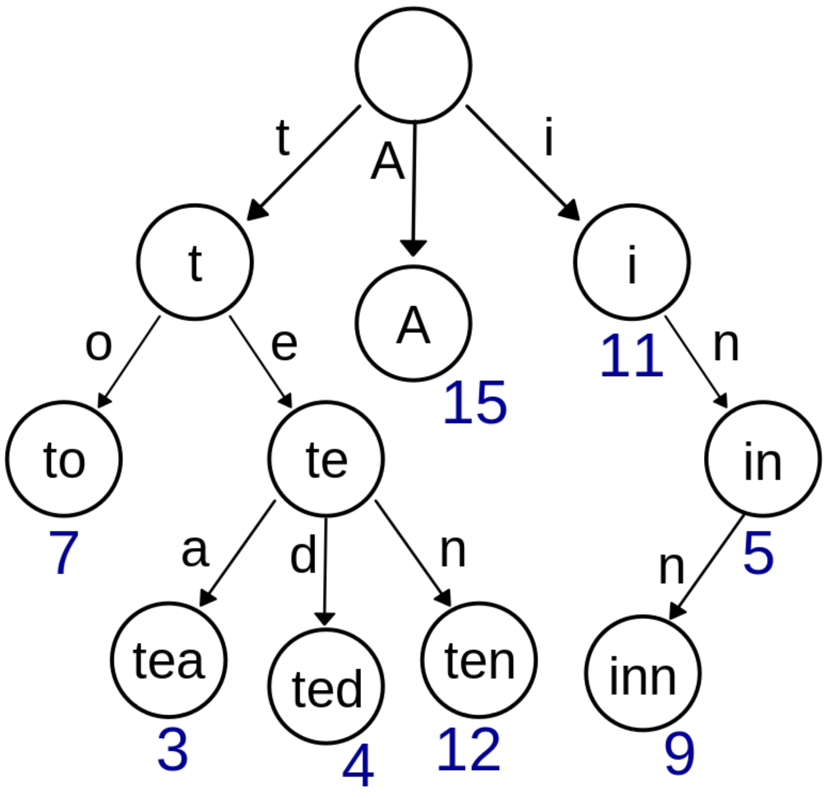
4.3 TrieNode and Trie

The Trie is a specialized, Tree-type datastructre for implementing the Map/Dictionary ADT. In essence, it's goal is to serve like python's dictionaries, but only when the keys are short (one word) strings.

While we haven't covered Tree data structures yet in lecture, this one will actually be a fair deal *simpler* than the one we will study. Tree data structures are kind of like Linked chain data structures, in that they are formed of nodes, and links between nodes. Unlike the LinearNode class we've seen so-far in class, the Trie Node class has *26* next nodes (children), one for each letter in the alphabet.

Each word can be mapped to one specific node in the tree based on the letters in the tree. This is mapped *structurally* meaning that the nodes won't store the string they go with, instead by choosing which of the 26 next Nodes to follow we can always arrive at the right node. For example the TrieNode for the word ted would be found by starting at the node for the empty string (the root node at the top of the picture below) and going to the child for "t", then from the "t" node, go to the child for "e". Then from the "te" node, go to the child for "d", bringing you to the "ted node". In this way instead of storing the key-strings in the nodes themselves, you can identify keys based on where a given node is in the structure of the Trie.

An example Trie can be seen in the following image (Source, wikipedia) Note, the text labels in each node do not represent strings actually stored, but rather the logical/abstract label for those nodes based on the structure. Also note that of the 26 possible children each node can have, in this example, most only have one or two.



While this might sound rather complicated, it's not as hard to program. You will need two classes:

4.3.1 TrieNode

The TrieNode class will have a type parameter. You can name this type parameter whatever you want, but I'll use T to represent this type parameter.

TrieNode should have:

- A private T variable data - storing the current value associated with this node.
- A private array of `TrieNode<T>`'s - storing the children (next links) of this node. Initially this array will be full of 26 nulls as we will only "fill in" child nodes as needed.
- A constructor – this should take no parameters, initialize data to null, and initialize the children to an array with 26 spaces (all null).
- A `public T getData()` method that gets the data.
- A `public TrieNode<T> getChild(char letter)` method which returns the `TrieNode<T>` associated with the given letter.
  - if not given a lowercase English letter ('a' through 'z') it should return null.

  - Otherwise you will be returning a TrieNode from the children array. The letters 'a' through 'z' map to `children[0]` to `children[25]` so you will need to convert the char to an index into this array. This process is quite similar to something you needed to do in CaesarCipher.
  - If the correct array element is null, a new TrieNode should be put in the array and returned. Otherwise the current TrieNode in the array should be returned.

- A `public int getTreeSize()` which returns the number of nodes in the tree (or rather the number of nodes in the part of the tree visible to this node). This can be computed as 1 plus the sum of treeSizes of all non-null children nodes.

4.3.2 Trie

The Trie Class is the class that your program will actually interact with. The purpose of this class is to keep TrieNodes organized, and provide an easy-to-use interface on this data storage. Like TrieNode, this class will have a type parameter, which I will call T, but you can name whatever you want. This type parameter represents the type of data that we are looking up by string.

Trie should have:

- a single private `TrieNode<T>` root.
- a constructor that initializes the root to a new node
- a private getNode function that takes a string and returns the appropriate trieNode. This is relatively easy to do using the getChild method on Trie Node repeatedly. For example, if the string is "dog" you would return `root.getChild('d').getChild('o').getChild('g')` You will likely need a loop to make this work for any String.
- a public `T get(String)` that gets the data currently stored on the TrieNode associated with the input string
- a public `T put(String, T)` that sets the data currently stored on the TrieNode associated with the input string to the T value provided.
- a `public TrieNode<T> getRoot()` method, which is used for testing and returns the root node.

4.4 Gibberisher

The Gibberisher class implements the primary random word generation algorithm as described at the beginning of this writeup.

A Gibberisher should have the following properties

- A private `Trie<CharBag>` which stores the assorted letter counts for each possible word segment. I call this the model, as it contains a model of how letters follow each other in the English language.
- A private integer tracking the segment length used for this gibberisher. Different gibberisher models can be trained with different sample lengths. Once complete, I recommend playing around with different segment lengths to see how it effects the words generated.
- A private int tracking how many LetterSamples it's processed.
- A constructor that takes the value of the segment length, and initializes the Trie and sample count variables.
- A method `public void train(LetterSample)` which adds one sample into the model. This will mean using the string from the LetterSample to get the appropriate CharBag, and then adding the char from the LetterSample to that CharBag. This function will need to deal with the possibility that it is the first sample for a given string segment (and therefore you need to make a new CharBag and add it.).
- A method `public void train(String)` which uses the LetterSample class to generate LetterSamples, and then uses the preceding function to train the model for each LetterSample
- A method `public void train(String[])` which calls the preceding method for each word in an array.
- A method `public getSampleCount()` that gets the number of samples used so far to train the model.
- A method `public String generate()` that actually generates a string. The algorithm for this is provided in the beginning of this document.

Once this is complete you should be able to run the GibberisherMain method to generate words. Try it out and see if you can find a new favorite word. My new favorite word is gallegrooverture. It can be pretty fun to try to define these words.

The gibberisher main method should be able to run in one or two seconds maximum. Any longer than that and you should revise your design and look for inefficiency, ideally we're talking about entire loops that could be removed. My code, on my computer, can (according to the time measurements in GibberisherMain) train a model with sampleSize 4, in about a quarter of a second (250ms) and can generate 2000 words in less than 20 ms.

5 Acceptable code libraries

You should make NO use of Java's built in collection types. A core purpose of this Project is to implement your own data structures. You will lose substantial points for any solution using Java's built-in datastructures. If there's any code you want to use that wasn't explicitly listed in this document you should email the course staff to be sure it's OK before using it.

6 Files on Canvas

The following files will be posted on canvas for you to use when starting:

- `CharBagTest.java` - tests for the CharBag class
- `Dictionary.java` - a file for loading the words list into a String array.
- `words.txt` - a list of words, this is intended to be read by Dictionary.java. **IMPORTANT:** This file should be placed in your Intellij project, but not in the src folder. Instead it should be *outside* if the src folder, under the main project folder. Placing it anywhere else will probably cause the file to not be found by Dictionary.
- `EmergencyDictionary.java` - This is just an array of words, its much smaller than the words file, but can be used for testing when you're having trouble getting the dictionary file to run. You will not get good words out of this though. You are still expected to get Dictionary working as part of testing your code.
- `GibberisherMain.java` - a file to run the Gibberisher class.
- `LetterSampleTest.java` - a test for the LetterSample class
- `TrieNodeTest.java` - a test file for TrieNode
- `TrieTest.java` - a test file for Trie

7 Deliverables

The deadline for this project is Tuesday May 5th at 6:00pm. Once you've completed all three phases you should submit a single zip file containing the following files to the submission link on canvas.

- `CharBag.java`
- `Gibberisher.java`
- `LetterSample.java`
- `Trie.java`
- `TrieNode.java`