Project 3:
Gallergrooverture and the generation of random words

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

1 Change Log

With a common lab assignment we do not typically have time to change the lab in response to student questions, concerns, and common misunderstandings. However, as this project lasts three weeks, it is relatively common for small updated, extra hints, and minor typos to be added. This page will list any such modifications. **I recommend checking occasionally for updates on canvas and listening for updates announced in lecture**

- Version 1.0 2022-11-29

1.1 FAQS

1. The Trie Put method should return the value that was removed from the Trie (I.E. the old data stored on the node associated with the string)

2 Essential Information

2.1 Pacing and Planning

This is a three long assignment. By this point you should have a sense of the depth and complexity of such an assignment. In terms of code-length, expect this project to be longer than project 2 – as it involves about the same number of classes, but those classes are larger on-average. In terms of complexity, this project will be closer to the nature of project 1 – requiring you to understand and implement several specific algorithms. As always, you should **Plan time not only for coding, but for understanding and designing solutions to this problem**.

More than in previous assignments, there are several places where key decisions have been left to you. In particular, one of the classes is entirely yours to design outside of the basic required methods. You will be given a **goal** to meet (in terms of efficiency) – failing to meet that goal will have a small but meaningful effect on your grade.

2.2 Deadline

The deadline for this assignment is Saturday December 17th at 6:00PM. Late work is not generally accepted on projects – there will be a brief grace period on gradescope to help mitigate technical issues – but beyond this grace period late work will not be accepted without an exception. To request an extension, please email course staff, and clearly state both the reason an extension is needed/warranted, and the length of the extension you are requesting. **Unlike labs there is no 1-day-late policy.**

2.3 Files

You will be expected to program the following files:

- `CharBag.java` – a class to count how many times each letter has been added to it's collection – and to generate a random letter from the bag.
- `TrieNode.java` – a class representing a node in a Trie tree (A special tree data structure that is very fast at storing strings)
- `Trie.java` – a class that behaves like a dictionary – specialized for when key-values are short strings.
- `Gibberisher.java` – a class that can be "trained" in how a written language works, and can then generate "gibberish" words for that language.

Additional testing files may be provided over the course of the project, but ultimately you will be responsible for testing, and debugging, the code on your own.

2.4 Other important restrictions

Individual project – Unlike labs, where partner work is allowed, this project is an individual assignment. This means that you are expected to solve this problem independently relying only on course resources (zybook, lecture, office hours) for assistance. Inappropriate online resources, or collaboration at any level with another student will result in **a grade of 0 on this assignment**, even if appropriate attribution is given. Inappropriate online resources, or collaboration at any level with another student without attribution will be treated as an **incident of academic dishonesty**.

To be very clear, you can ask other students questions only about this document itself ("What is Daniel asking for on page 3?"). Questions such as "how would you approach function X", "How are you representing data in part 2?", or even "I'm stuck on part B can you give me a pointer" are considered inappropriate collaboration even if no specific code is exchanged. Coming up with general approaches for data representation, and finding active ways to become unstuck are all parts of the programming process, and therefore part of the work of this assignment that we are asking you to do independently. Likewise, you are free to google search for any information you want *about the java programming language and included java classes* but it is not reasonable to look for information about the problem in this project (I.E. "how to program card games" would be an inappropriate google search)

Allowed java classes There are A LOT of java classes, which solve many interesting problems. In particular, there are one-or-two java classes that can completely solve major parts of this assignment. Our goal here is not to practice USING data structures to represent complicated data (that was project 1) but instead to BUILD data structures to represent complicated data. As such – Our normal rules for java classes apply:

You are not allowed to use any of the pre-built java classes except those we've explicitly discussed in class such as Scanner, Random, String, Character, Arrays etc. If you are unsure if a java class would be allowed here you should ask. Using unapproved java classes can lead to failing this assignment.

Specific examples of BANNED java classes:

- Any subclass of java.util.List (java's list classes like ArrayList and LinkedList)
- Any subclass of java.util.Map (java's dictionary classes such as HashMap and TreeMap)
- Any subclass of java.util.Set (java's set classes such as HashSet and TreeSet)
- java.util.Collections

Every data storage task in this problem can be done by-hand without these pre-built data structures. Seeing how to do that is deliberately part of the assignment.

3 Introduction

In the last few years, computational models of human language have vaulted from the research community into the center-stage of computer science innovation. Modern language models like Chat-GPT can do amazing things, and we're really only starting to understand the implications and utility of these novel language models. You will (unfortunately) NOT be building a so-called "large language model" of human language in this assignment, these models are radically complicated as befits their complicated goals. We will target a more modest goal – random, but pronounceable nonsense words. In doing so we will build a pair of custom-purpose data structures to allow us to store our "small-language-model", and see one way these models can be trained and used.

Specifically, 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). In theory, you could also use the Gibberisher class to represent other languages (possibly with slight modifications to the lettering system) To build an efficient Gibberisher, we will also need to rely on a few other problem-specific, highly optimized data structures. We will need these data structures to manage our approach – at a high level, the first step of our algorithm will be summarizing an English dictionary of 62,915 words. While this is ultimately a small number for computers – it's certainly enough that our code will need to be efficient, or else it will take forever!

3.1 Learning Goals

This assignment is designed with a few learning goals in mind:

- Build an advanced tree-based data structure.
- Plan and build a task-specific data structure.
- Demonstrate your understanding of a range of data structure representations by choosing the most appropriate representation for this class.
- Implement an advanced algorithm to "model" and generate English text.

This assignment will build upon our ongoing study of data structures. The first few classes should be build able with the material we've already covered at the point of assignment. Later parts of this project should still be build able with the course material currently covered, but may make more sense after we've covered topics in the future – most notably Tree-based data structures. That said – you should be able to at least *start* every part of this assignment at the point it is assigned.

4 Theory: Understanding, and generating, English words

Before describing our algorithm itself, let's first see a **different and simpler** algorithm for the same problem. This will highlight the reasoning behind the parts of the algorithm we will want to generate.

4.1 A Bad algorithm for generating English 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. While this algorithm in theory works – the words it generates are not good. 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. The words generated by this algorithm are often unpronounceable because it does not understand the "rules" that lead to 'pronounceable words.

One improvement upon this algorithm would be to make it understand that letters are not all used equally. Letters like "e" show up much more than letters like "q" and "z". Therefore, we could improve this algorithm slightly if we put all the letters used by every English word into a bag, and then picked letters at random from this bag. If we did this we would find roughly the following distribution of letters in English words.

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

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. This algorithm still does not have a good sense of what kind of words can follow other words.

4.2 Models – and their role in generating text

The core of both *bad* approaches to generating random words is their "model" of English text. A "model" in computer science describes a way of mathematically understanding the world. For our first algorithm, the "model" was simply a list of acceptable letters (a through z). If we wished to use the algorithm on other languages we could simply change the letter list to match that language.

For the second algorithm, the "model" was a relationship between letters and their general rate of use (Captured in the table above). If we wanted to switch to a different language, we would need a different "model" – a different table to capture letters and their rate of use. In this way the "model" represents both our choice of algorithms, and contains everything the algorithm knows about the language it replicates.

To achieve our goal of random pronounceable words we need a more complicated model. In particular, we will want a model that understand that there are certain patterns to what letters follow what other letters. As an example, the letters "ont" are followed by many letters in English ('r' and 'e' being most common) but the letters "ter" are mostly followed by 's', with other options being less common. Therefore – this will become our model: an organized collection of all 3-letter (or in general k-letter) word segments, associated with a count of letters that follow them. With this model, we can generate a word letter-at-a-time while still following the "rules" of the language.

This leaves two problems still unexplored: starting, and stopping words. To handle the starting problem we will allow "partial" word segments. For instance, we will also have a count of words associated with "do" – which represents what letters tend to follow "do" specifically at the start of the word. We can use these "partial segment" word counts to track how English words get started.

The second problem, stopping words, is similarly solvable. We will use a simple approach and add the letter "." at the end of every word. We will treat this like an ordinary letter – so our table of characters following any given word segment will contain a count of 27 letters: a-z and also '.' Whenever our algorithm randomly picks "." as an appropriate next-letter, we will stop our word there (not including "." in the word) This will let us generate words of any length – with realistic "ends" to the word.

An example of generating text with this process may help. Here is an example of how the word "demors" might be generated.

1. Initially we have an empty string. We consult the count of all "first letters" and pick on at random. Let's say we picked "d" (about 1-in-16 chance (6.2%))
2. We now have the word "d". Since this is still too small, we check the count of all next letters after "d" (at the start of the word). We pick "e" (used in 38% of words starting with "d")
3. We now have the word "de". This is still to small, so we check the count of all next letters after "de" (at the start of a word). We pick "m" (used in 8.5% of the time)
4. We now have the word "dem". This is big enough, so we check the count of all next letters after "dem" (anywhere in the string). We pick "o" (used 39% of the time)
5. We now have the word "demo". Since we are using size-3 letter segments we will check the count of all next-letters after "emo". We pick "r" (used 26% of the time)
6. We now have the word "demor". We check the count of all next-letters after "mor". We pick "s" (used 5.3% of the time)
7. We now have the word "demors". We check the count of all next-letters after "ors". We pick "." indicating the end of the word. (70% of the time we saw "ors" it was the end of the word).

Ignoring the "." we generated – we can then return our new word: "demors".

Putting that process into pseudocode (where k is the sample size, 3 in the above) we have the following:

```
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 nextLetter based on those letter counts
    word = word + nextLetter
return word.
```

As you can see – at each step we will consult a potentially different count of letters to decide what the appropriate "next" letter is. This large collection of next-letter-counts is the "model" of this problem – and represents everything the computer needs to know about pronounceable English text.

4.3 Representing the letter-counts model

The "model" for this algorithm, therefore, is quite important – as it contains everything the algorithm knows about how the English language "works". We will represent this "model" with two custom purpose data structures:

CharBag The CharBag data structure will be used to represent letter-counts. For example, the count of all "next letters" after the segment "cti" in our data is:

```
CharBag{a:1, b:9, c:55, d:0, e:0, f:18, g:0, h:0, i:0, j:0, k:0, l:10, m:9, n:106,
        o:303, p:0, q:0, r:0, s:4, t:9, u:0, v:147, w:0, x:0, y:0, z:0, .:0}
```

Which tells us that "ctio" is the most common, but "ctiv", "ctin", and "ctic" are all respectable ways to continue a string. We may have thousands of these CharBag objects, each associated with a different series of preceding letters.

Trie The Trie class will be our organizational tool to hold the thousands of CharBags and associate them with their preceding letters. The Trie data structure is a very specialized data structure for organizing things (like CharBags) when they are specifically associated with short strings. This will give us VERY FAST access to the correct CharBag for any series of letters our algorithm can generate.

4.4 Training the letter-counts model

The final part to explain of this algorithm is how we generate the model itself. This process is normally called "training" the model. This training process only needs to happen once, usually at the start of the program. Once the model is "trained" we can re-use it to generate many different words.

Broadly – our approach is simple: show the "model" every English word in the dictionary, and have it count each letter transition. The first step of this process will be splitting each word into a

series of "LetterSamples" – one for each letter in the word (and one extra for "STOP" – the end of the word) These LetterSamples tell us which letters preceded each letter in the word. 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 count 'a' as a potential first-letter (associated with sequence ""), and we will count 'e' as a potential next letter after the sequence "aft" and so-forth. While any one word will not make a big difference to the counts we store – after loading tens of thousands of words we our model will have a pretty good understanding of English language.

4.5 On understanding this algorithm

This algorithm has a lot of moving pieces, I know that. It can be difficult to see all of these parts on their own at the beginning – and it can be hard to visualize (since there are probably more than 100,000 distinct numbers we will need). Don't get discouraged at this point. We've decomposed this problem into a series of classes which you can treat as self-contained problems. Only at the end, once you know how each piece of this puzzle works, will you build the "Gibberisher" class that represents this entire process. Before you get to that part, try to have answer to the following questions:

- How does this algorithm represent English text?
- How can we generate a word with this algorithm?
- How do we "train" this representation of English text?

These are all questions you can discuss (not-in-code) with other students – as the non-code parts are all listed above. Of course, if you are having trouble understanding this algorithm at a high level, we are always happy to explain it further over email or in office hours. Likewise, as said, you can also move forward and build most of this project without these pieces clicking just yet.

5 Formal Requirements

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

- 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.

5.1 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. 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 (more like a map than a set)

This object will also have the responsibility of randomly generating letters, following the frequencies it stores. Think of this function as reaching into the class and picking a random letter out of the bag. 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 $2/5 = 40\%$.

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 '.' (Use the static variable from the LetterSample class)
- `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. If the `CharBag` is empty, this function should return the stop character '.'

Some notes / hints:

- We will not recommend or require any particular approach to storing the data in this class. There is no one "right" way to do this class, but there are some clearly BAD ways to do this.

  - There will be a small number of points allocated to your approach to this class. The exact point-allocation is not set yet, but tentatively assume around 6 points – roughly one per method.
  - Your goal here is $O(1)$ runtime for all methods.
  - An approach that runs in time $O(n)$ for most methods (where n is size as defined in getSize) may be easy to implement – but will get not credit for efficiency.
  - Partial credit will be rewarded for speed-ups above the $O(n)$ family of approaches (even if they don't reach the $O(1)$ target for all methods)
- Some hints on how to think about structuring this – note these are not all designed to lead to the same answer
  - To get $O(n)$ efficiency – a direct array-based solution, similar to an array list, should be sufficient.
  - However, to get $O(1)$ efficiency you're going to need to look for a *trickier* solution.
  - We're 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 each letter once associated with a "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 (although the number of if statements involved in such a solution would make me very sad.)
  - If your algorithm runs a loop over all *letters* a through z – this is not $O(n)$ it's $O(1)$ since the number of letters (27) does not scale with size.
- The random function is easy to implement with a slow $O(N)$ design for other functions – but hard to implement efficiently when all other functions run $O(1)$. (due to how the data must be stored internally) 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 '.'
```

For this class you will find the Random and Character classes and methods useful. You may also find various "char" manipulations useful:

- chars can be compared with $<$ and $>$. With this you can write code like `'a' <= c && c <= 'z'` to check if a letter is in a range of letters.
- operators like `++` can be used to move between "adjacent" chars in the alphabet ('a'++ == 'b' for instance)
- char math (such as `(int)(c - 'a')`) can convert a char into an integer. (Experiment with this more to find out how it works)
- You can write a for loop over chars by combining the above:

```
for (char c = 'a'; c <= 'z'; c++)
```

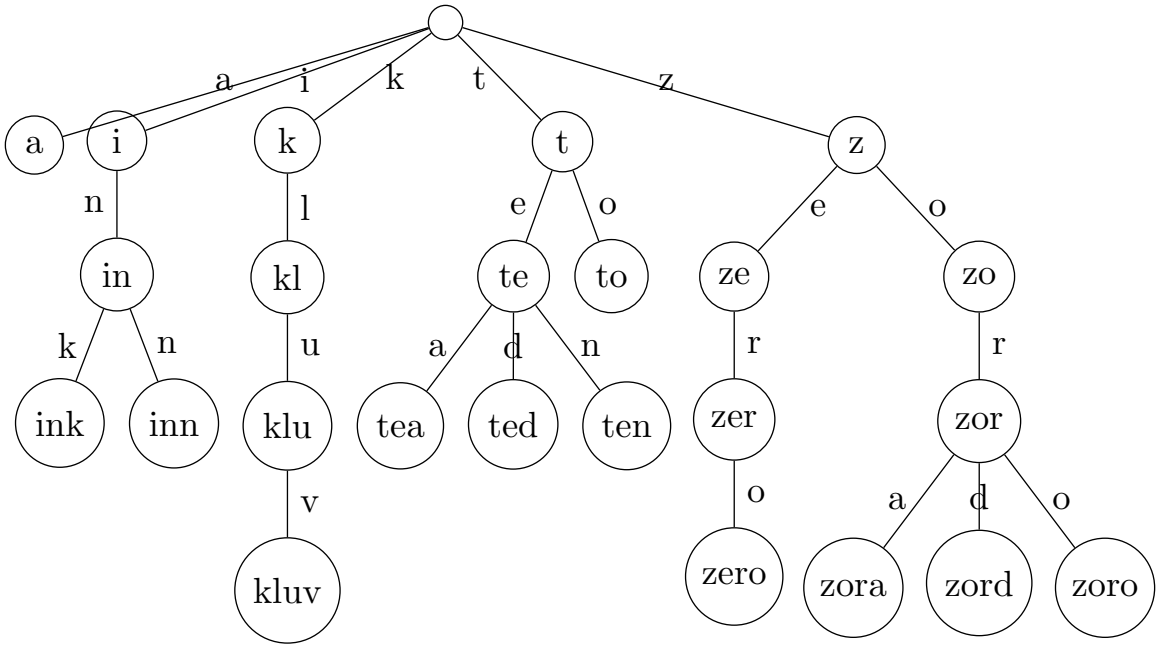
5.2 TrieNode and Trie

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 Node class we will soon see 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, starting at the root and choosing the correct child (of 26) at each node – we can find a Node for each word. 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" (the node labeled "te" below). 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. 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 (if we showed 26 child nodes the tree would be incomprehensible...)



You may also find it useful to read more about Tries in Zybooks chapter 27.11. Our design might not exactly mirror the way zybooks approaches this data structure – but the general outline (and animated visualizations) there can make this make a lot more sense if you're struggling to understand the Trie.

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

5.2.1 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 void setData(T data)` method that sets 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 can be done with code like: `(int)(c-'a')` Experiment with this to learn more.
  - 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 function is primarily useful for debugging as you build the language model.
  - This can be computed as 1 plus the sum of treeSizes of all non-null children nodes.
  - You are **REQUIRED** to solve this problem recursively.
  - (Also – while this problem can be solved without recursion it's way more complicated – the recursive solution can be quite easy to read/understand.)

Note – Unlike the charBag – the TrieNode and Trie classes do not represent a stop character – they only represent the 26 alphabetical letters, with all other letters being ignored.

5.2.2 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.

5.3 LetterSample

The `LetterSample` class has been provided for you. Each instance of LetterSample will represent a sample of text which will be used by the Gibberisher class. The `LetterSample` class is pretty simple – None the less, it is recommended you review it carefully so you can use it competently before starting Gibberisher.

Each instance of Letter Sample represents one of the "letter samples" from word generation algorithm at the beginning of this writeup and stores a segment (a short string) and a nextLetter (the char following the segment). For example, the LetterSample "aft" ⇒ 'e' indicates that in a word we saw the letter 'e' following the letters "aft" (and therefore 'e' is generally pronounceable following "aft")

LetterSample 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". By making this a public static final variable you will be able to access it from other classes like `LetterSample.STOP` to know which letter means "stop". You should use this public static variable in all reasonable places instead of using '.' directly – this makes it easier to change in the future, if the need arises.

Secondly:

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

This function is in charge of taking a string and generating letter samples from it. So for example `toSamples("apple", 3)` would return the array `[\"\"->a, \"a\"->p, \"ap\"->p, \"app\"->l, \"ppl\"->e, \"ple\"->.]`

5.4 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(String[])` which should for each string:
  - Use the LetterSample class to generate LetterSamples, and then, for each resulting LetterSample:
    - add this 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 getSampleCount()` that gets the number of distinct LetterSamples 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. As a note – make sure you remover the STOP character from the strings before returning them!

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 third of a second (336ms) and can generate 2000 words in less than 20 ms.

6 Files on Canvas

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

- `LetterSample.java` - a simple class to represent one "Data point" for the program we are making
- `CharBagTest.java` - tests for the CharBag class
- `Dictionary.java` - a file for loading the words list into a String array. This class also has a main method which you can use to test if the words file is in the right place.
- `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.

- `GibberisherTests.java` - a file to perform some very basic tests against the Gibberisher class.
- `TrieNodeTest.java` - a test file for TrieNode
- `TrieTest.java` - a test file for Trie

7 Submission and grading

This information is still being developed. As usual – there will be a small autograded component based on the provided tests and a large manual grading component. A rough breakdown on points is as follows:

- 10 points for code style
- 20 points for tests
- 25 points for CharBag, including 6 points on general class design and efficiency
- 25 points for Trie and TrieNode
- around 20 points for Gibberisher