CMP-5014Y Coursework 2 - Word Auto Completion with Tries

Student number: 100246776. Blackboard ID: 100246776

Wednesday 20th May, 2020 09:55

Contents

1 How to read this report?

1. Short informal descriptions accompany each algorithm. They act as reading guidelines
2. Naming conventions:
   (a) Variable names with single letter,in bold and capitalized indicate abstract data structures. For example, **M** is a map
   (b) Variables names with single letter, in bold and lowercase indicate instances(objects). For example, **n** is a node because it is derived from a user-defined class
   (c) keywords are in bold
3. Comments are either inline. If too long, they are found above a line
4. When calling a function on an object,it is passed explicitly into the function. For example, **M**.add(word,frequency) is add(**M**,word,frequency). Read as, "To **M**, add word and frequency". This separates pseudocode from most programming languages.
   However, to match function signature in specs, a trie implictly passed to every function it is called on. For e.g, outputBreadthFirstSearch() is passed the trie **t** implicitly. The rigorous way is outputBreadthFirstSearch(**t**).
5. Data types used:
   (a) unsigned numeric, boolean, string
   (b) Trie, TrieNode,AutoCompleteTrie, AutoCompleteTrieNode

2 Part 1: Form a Dictionary and Word Frequency Count

2.1 Pseudocode for formDictionary

1. readFile should be a non-empty comma separated values(CSV) file. It contains words separated by commas on a single line. For example, rock,paper,scissors,rock,paper.
2. writeFile is a non-empty CSV file containing word-frequency tuples, separated by line.
3. readWordsFromCSV(readFile) returns a list of words read from readFile.
4. For each word in list, add to map. If encountering word for first time, add to map with frequency=1. If not a new word, retrieve from map and increment frequency.
5. saveToFile(writeFile) saves map to writeFile.
6. Design decisions:
   (a) Map data structures store key-value pairs.
   (b) Implementation of Map(tree map) ensure $\mathcal{O}(log(n))$ runtime complexity for lookup and insertion.

Algorithm 1 formDictionary($readFile$,$writeFile$) **return**

Require: A CSV file $readFile$, an empty CSV file $writeFile$

Ensure: $writeFile$, contains key-frequency tuples where key is a unique word of type string in $readFile$ and frequency of type unsigned numeric, the word's frequency of occurrence in $readFile$

$\mathbf{D} \leftarrow Map()$ ▷ *Initialise map **D** to hold key-frequency tuples*
$\boldsymbol{L} \leftarrow readWordsFromCSV(readFile)$ ▷ **L** is a list of words from readFile
for all s **in L do**
  if $containsKey(\mathbf{D}, s)$ **is false then**
    $put(\mathbf{D}, s, 1)$ ▷ *if key not in **D**,add it and set frequency to 1*
  else ▷ if key in **D**,retrieve and increment frequency
    $frequency \leftarrow get(\mathbf{D}, s) + 1$
    $put(\mathbf{D}, s, frequency)$
$saveToFile(\mathbf{D}, writeFile)$ ▷ *saveToFile writes **D** to a writeFile*

2.2 Algorithm Analysis for formDictionary

In a discussion board thread, Dr. Tony Bagnall stated we need to consider runtime complexity of treemap functions. Below is the runtime complexity analysis for formDictionary with such a consideration.

1. Fundamental operation:
   if $containsKey(\mathbf{D}, s)$ **is false then**
2. Treemap maintains $\mathcal{O}(log(n))$ lookup. Worst, average and best cases are the same. Let size of **L** to be n
3. $$f(n) = \sum_{i=1}^{n} log(n) \tag{1}$$
   $$t(n) = nlog(n) \tag{2}$$
4. The order of the runtime complexity is log linear i.e. nlog(n). The runtime complexity function is $\mathcal{O}(nlog(n))$

Or,consider number of times fundamental operation occurs.

1. Fundamental operation:
   if $containsKey(\mathbf{D}, s)$ **is false then**

2. Let size of **L** to be n

3.

$$f(n) = \sum_{i=1}^{n} 1 \tag{3}$$
$$t(n) = n \tag{4}$$

4. The order of the runtime complexity is linear i.e. n. The runtime complexity function is $\mathcal{O}(n)$

2.3 Pseudocode for saveToFile

1. writeFile is a non-empty CSV file containing keys-frequency tuples, separated by line.
2. D is a non-empty map from saveDictionary. It stores word-frequency pairs.
3. For all entries in D, make a keyValuePair string. For example, "brother,4".
4. On a new line in writeFile, write keyValuePair

Algorithm 2 saveToFile(**D**,$writeFile$) **return**

Require: A map **D** containing word-frequency tuple where word is of type string and frequency of type unsigned numeric, an empty CSV file $writeFile$

Ensure: $writeFile$ contains all word-frequency tuples where word is a key and frequency, the word's frequency of occurrence in $readFile$

```
for all e in D do                                          ▷ for each entry
  ▷ for an entry, retrieve key and value i.e frequency. Format into a word-frequency tuple: "key,value"
    keyValuePair ← getKey(e), getValue(e)
    writeline(writeFile, keyValuePair)                     ▷ write keyValuePair on a newline
```

2.4 Algorithm Analysis for saveToFile

In a discussion board thread, Dr. Tony Bagnall stated we need to consider runtime complexity of treemap functions. Below is the runtime complexity analysis for saveToFile with such a consideration. It is also a more realistic depiction of the runtime because the algorithm's efficiency(or lack of it) depends on the data structure used.

1. Fundamental operation:
   $keyValuePair \leftarrow getKey(\mathbf{e}), getValue(\mathbf{e})$

2. Treemap maintains $\mathcal{O}(log(n))$ lookup. Worst, average and best cases are the same. Allow loop to execute n time, ie, number of entries in **D**

3.

$$f(n) = \sum_{i=1}^{n} log(n) + \sum_{i=1}^{n} log(n) \tag{5}$$
$$t(n) = 2nlog(n) \tag{6}$$

4. Ignoring constants, the order of the runtime complexity is log linear i.e. nlog(n). The runtime complexity function is $\mathcal{O}(nlog(n))$.

Or,consider number of times fundamental operation occurs.

1. Fundamental operation:
   $keyValuePair \leftarrow getKey(\mathbf{e}), getValue(\mathbf{e})$

2. Allow loop to execute n time, ie, number of entries in **D**

3.

$$f(n) = \sum_{i=1}^{n} 1 \tag{7}$$
$$t(n) = n \tag{8}$$

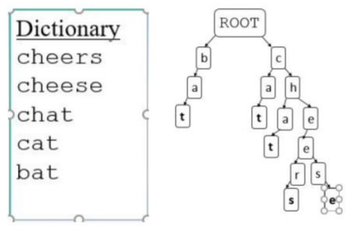
4. The order of the runtime complexity is linear i.e. n. The runtime complexity function is $\mathcal{O}(n)$

3 Part 2: Implement a Trie Data Structure

TrieNode: A TrieNode has an array of size 26. The array stores references to offspring of that node. The indexes of the array each maps to a letter. For example, 'a' maps to 0(1 in pseudocode), 'b' to 2 and so forth.A TrieNode can be a key: it represents a complete word. As such, a TrieNode has getIsKey() and setIsKey(boolean) functionalities. getIsKey() returns true if TrieNode is a key. setIsKey() sets key attribute of TrieNode to true.

Trie: A Trie can be viewed as containing TrieNodes with a root TrieNode at the top. The access point to the rest of the TrieNodes is through the root node.

Figure 1: Trie example



Using bat as an example, the root stores a reference to a node in it's array. The index at which node is stored 2 mapping to 'b'. A key is a node whose isKey attribute is set to true. In the diagram, a key is denoted in bold.

3.1 Algorithm for boolean add(String key)

A key is a unique, non-empty, not null word in the trie. If a valid word is being added and it is already a key in the trie, the algorithm returns false for not added. Otherwise, return true for added.

Algorithm 3 add(*word*) **return** (*added*)

Require: *word* is of type string

Ensure: *added* is a boolean, either **true** or **false**. Indicates if a word is added as key

$added \leftarrow$ **false**

if r is null or *word* **is null then** ▷ ***r*** *is the the root node of the trie*

  return (added)

$asciiOffset \leftarrow 97$ ▷ *97 is 'a' in ascii. Start of small letters*

$\mathbf{p} \leftarrow \mathbf{r}$ ▷ *assign **r**,root of trie ,to **p***

for $i \leftarrow 1$ **to** $size(word)$ **do** ▷ for each character in word

  $character \leftarrow convertToChar(word_i)$ ▷ *converts* $word_i$ *into ascii equivalent* ▷ *see appendix for ascii table*

▷ converts index to a value between 1-26, mapping it between 'a' to 'z'

  $index \leftarrow charToInt(character) - asciiOffset$

▷ *returns an array containing pointers to children of **p***

  $\mathbf{C} \leftarrow getOffspring(\mathbf{p})$

  $\mathbf{n} \leftarrow C_{index}$ ▷ get offspring at mapped index

  if n is null then ▷ *If node is null, word is not in trie*

    $added \leftarrow$ ***true***

    $\boldsymbol{t} \leftarrow TrieNode()$ ▷ create a new trieNode

▷ assign new node to this position. Index of offspring represents a character

    $C_{index} \leftarrow \boldsymbol{t}$

    $\boldsymbol{p} \leftarrow \boldsymbol{t}$ ▷ set parent to the new node created

  else

    $\boldsymbol{p} \leftarrow \boldsymbol{n}$ ▷ If node exists at index, go to it

---

if $getIsKey(\mathbf{p})$ **and** $added$ **is false then** ▷ *If node is key and not already added*
    return $(added)$ ▷ if node already in trie, return false for not added
$setIsKey(\mathbf{p}, \mathbf{true})$
$added \leftarrow$ **true**
return $(added)$

3.2 Algorithm for boolean contains(String word)

A key is a unique, non-empty, not null word in the trie. The algorithm checks if the word passed as a parameter is a key in the trie. If yes, return true for contains. Return false otherwise.

Algorithm 4 contains($word$) **return** $(contains)$

Require: $word$ is of type string
Ensure: $contains$ is a boolean, either true or false. Indicate if a word is a key

$contains \leftarrow$ **false**
if r is null or $word$ **is null then** ▷ ***r*** *is the the root node of the trie*
    return (contains)
$asciiOffset \leftarrow 97$ ▷ *97 is 'a' in ascii. Start of small letters*
$\mathbf{p} \leftarrow \mathbf{r}$ ▷ *assign* ***r****,root of trie ,to* ***p***
for $i \leftarrow 1$ **to** $size(word)$ **do** ▷ looping character by character
    $character \leftarrow convertToChar(key_i)$ ▷ *converts* key_i *into ascii equivalent* ▷ *see appendix for ascii table*
    ▷ converts index to a value between 1-26, mapping it between 'a' to 'z'
    $index \leftarrow charToInt(character) - asciiOffset$
    ▷ *returns an array containing pointers to children of parent node*
    $\mathbf{C} \leftarrow getOffspring(\mathbf{p})$
    $\mathbf{n} \leftarrow C_{index}$ ▷ get offspring at mapped index
    if n is null then ▷ *If node is null, key is not in trie*
        return *(contains)*
    $\boldsymbol{p} \leftarrow n$ ▷ if a reference corresponding to the index is found, go to it.
▷ *If node is key,trie contains it*
if $getIsKey(\mathbf{p})$ **is true then** ▷ getIsKey(node) returns true if node is a key
    $contains \leftarrow true$
    return $(contains)$
▷ A matching word can be found in the trie but it does not mean it is a key
return $(contains)$ ▷ trie does not contain word

3.3 Algorithm for boolean String outputBreadthFirstSearch()

This algorithm goes through a trie and outputs characters in breadth first search order.
Design decision:

1. Use a Queue because it respects first-in first-out rule. This is useful for breadth first search.

2. Underlying implementation of queue is a linkedlist. This ensures ensure O(1) for enqueue and dequeue.

Algorithm 5 outputBreadthFirstSearch() **return** ($result$)

Require:
Ensure: $result$ is of type string and contains all words in breadth first search order

if r is null then ▷ ***r*** *is the the root node of the trie data structure*
  return null
$asciiOffset \leftarrow 97$ ▷ *97 is 'a' in ascii. Start of small letters*
$\mathbf{Q} \leftarrow Queue()$ ▷ *holds a collection of nodes type TrieNode*
$enqueue(\boldsymbol{Q}, \boldsymbol{r})$ ▷ Add root,**r**, to front of queue,**Q**
while Q is not empty do ▷ *dequeue(**Q**) removes top most trie node and assign to temp,**t***
  $\mathbf{t} \leftarrow dequeue(\mathbf{Q})$
  ▷ *getOffspring(node) returns an array containing references to children of that node*
  $\mathbf{C} \leftarrow getOffspring(\mathbf{t})$
  for $i \leftarrow 1$ **to** $size(\mathbf{C})$ **do**
    if C_i **is not null then**
      $enqueue(\mathbf{q}, C_i)$ ▷ add offspring of node to the queue so it goes through them in breadth first order
      $result \leftarrow result + intToChar(i + asciiOffset)$ ▷ *concatenate character represented by i to result*
return ($result$)

3.4 Algorithm for String outputDepthFirstSearch()

This algorithm is an interface method to outputDepthFirstSearch(builder,**r**). It provides an empty string, builder, and the starting node of the depth first search.

Algorithm 6 outputDepthSearchSearch() **return** ($result$)

Require:
Ensure: $result$, a string in depth first search order.
▷ $builder$ is an empty string, and **r** is the root of the Trie.
$builder \leftarrow$ ""
$result \leftarrow depthFirstSearch(builder, \mathbf{r})$
return ($result$)

3.4.1 Recursive implementation of String outputDepthFirstSearch()

This algorithm goes through trie and ouputs characters in depth first search(DFS) order. DFS goes down the left-most leaf and backs up.
Design decision:

1. Recursion as opposed to a iterative version provides a more elegant and shorter solution.

2. No need to handle stack data structure.

Algorithm 7 outputDepthFirstSearch(builder,**r**) **return** ($result$)

Require: builder, an empty string and **r** of type TrieNode

Ensure: $result$ is of type string and contains all words in depth first search order

if r is null then ▷ *r is the the root node of the trie data structure*
  return null
▷ *getOffspring(node) returns an array containing references to children of that node*
$\mathbf{C} \leftarrow getOffspring(\mathbf{r})$
$asciiOffset \leftarrow 97$ ▷ *97 is 'a' in ascii. Start of small letters*
for $i \leftarrow 1$ **to** $size(\mathbf{C})$ **do**
  if C_i **is not null then**
    $result \leftarrow result + intToChar(i + asciiOffset)$ ▷ *convert index to char*
    $outputDepthFirstSearch(result, C_i)$ ▷ recursive call on node's children- depth first
return ($result$)

3.5 Algorithm for getSubTrie(String prefix)

The algorithm returns a new trie rooted at the prefix, or null if the prefix is not present in this trie.

Algorithm 8 getSubTrie($prefix$) **return** (**t**)

Require: **r**,root node of trie and $prefix$, a string representing incomplete word

Ensure: **t** is a trie data structure rooted at $prefix$

if r is null or $prefix$ **is null then**
  r ← **null** ▷ *root is the the root node of the trie data structure*
$asciiOffset \leftarrow 97$ ▷ *97 is 'a' in ascii. Start of small letters*
$\mathbf{p} \leftarrow \mathbf{r}$
for $i \leftarrow 1$ **to** $size(prefix)$ **do**
▷ *converts* $prefix_i$ *into ascii equivalent*
  $character \leftarrow convertToChar(prefix_i)$
▷ converts index to a value between 1-26, mapping it between 'a' to 'z'
  $index \leftarrow charToInt(character) - asciiOffset$ ▷ see appendix for ascii table
▷ *getOffspring(node) returns an array containing references to children of that node*
  $\mathbf{C} \leftarrow getOffspring(\mathbf{p})$
  if C_{index} **is null then** ▷ prefix does not exist
    return (**null**)
  $\mathbf{p} \leftarrow C_{index}$ ▷ if node is not null, go to it
return $Trie(\mathbf{p})$ ▷ return a trie rooted at the last node assigned to **p**

3.6 Algorithm for getAllWords()

This algorithm is an interface method to getAllWords(**L**,textBuilder,**r**). It provides an empty string, builder, the starting node and an empty list.

Algorithm 9 getAllWords() **return** (**L**)

Require:
Ensure: **L**, a list of keys from trie.

$getAllWords(\mathbf{L}, textBuilder, \mathbf{r})$ ▷ *textBuilder* is an empty string, and **r** is the root of the Trie.
if $size(\mathbf{L})$ **is null then** ▷ prefix does not exist
  return (**null**)
return (**L**)

This algorithm returns a list containing all words in the trie. It is called by getAllWorld().
Design decision:

1. Underlying implementation used for list, **L** is a linked list. Linked list provides O(1) runtime complexity for insertion. Linked List is always added to in the algorithm.

Algorithm 10 getAllWords(**L**,**textBuilder**,**r**) **return**

Require: **L** is a list of words that are also keys of type string, **textBuilder**,string that can be appended to, and **r**, the root node of the trie
Ensure:

$asciiOffset \leftarrow 97$ ▷ *97 is 'a' in ascii. Start of small letters*
if $getIsKey(\mathbf{r})$ **then**
▷ *If node is a key, textBuilder contains a complete word that is also a key*
  $add(\mathbf{L}, textBuilder)$
▷ *getOffspring(node) returns an array of references to children of that node*
$\mathbf{C} \leftarrow getOffspring(\mathbf{r})$
for $i \leftarrow 1$ **to** $size(\mathbf{C})$ **do**
  $\mathbf{o} \leftarrow C_i$
  if o is not null then
    $textBuilder \leftarrow textBuilder + intToChar(i + asciiOffset)$ ▷ *intToChar converts the int to it's char equivalent*
    $getAllWords(\mathbf{L}, textBuilder, \mathbf{o})$ ▷ recursive call with updated parameters
▷ *Backtracks one level up the trie. Say trie contains keys ab and ac. Once ab is appended to **L**, remove last character 'b' from 'ab' in order to append 'c'*
    $removeLastCharacter(textBuilder)$

---

4 Part 3: Word Auto Completion Application

AutoCompletionTrieNode: An AutoCompletionTrieNode is conceptually similar to a TrieNode. It is made of an array of size 26 to hold children nodes and contains a wordFrequency attribute. wordFrequency is updated whenever the node is set to a key. As such, it also contains a setter and getter for wordFrequency.

AutoCompletionTrie: This is same as a regular Trie.

WordAndFrequency: WordAndFrequency is a blueprint for an entity containing a key and frequency.

4.1 Algorithm for add(word,wordFrequency)

This algorithm adds a word, frequency pair to the AutoCompletionTrie. If word is not already a key in the AutoCompletionTrie, mark the node that holds last character to word as a key. Set its frequency. If a node is already a key, update its frequency.

Algorithm 11 add(*word*,*wordFrequency*) **return** (*added*)

Require: *word* is a unique word and of type string,*wordFrequency* is the frequency of occurrence of a word

Ensure: *added* is a boolean, either **true** or **false**. Indicate if a word is added as key

$added \leftarrow$ **false**
if r is null or *word* **is null then** ▷ ***r*** *is the the root node of the trie*
  return (added)
$asciiOffset \leftarrow 97$ ▷ *97 is 'a' in ascii. Start of small letters*
$\mathbf{p} \leftarrow \mathbf{r}$ ▷ *assign **r**,root of AutoCompletionTrie ,to **p***
for $i \leftarrow 1$ **to** $size(word)$ **do** ▷ for each character in key
  $character \leftarrow convertToChar(word_i)$ ▷ *converts* $word_i$ *into ascii equivalent* ▷ *see appendix for ascii table*
  ▷ converts index to a value between 1-26, mapping it between 'a' to 'z'
  $index \leftarrow charToInt(character) - asciiOffset$
  ▷ *returns an array containing pointers to children of **p***
  $\mathbf{C} \leftarrow getOffspring(\mathbf{p})$
  $\mathbf{n} \leftarrow C_{index}$ ▷ get offspring at mapped index
  if n is null then ▷ *If node is null, word is definitely not in trie*
    $added \leftarrow$ ***true***
    $\boldsymbol{t} \leftarrow AutoCompletetionTrieNode()$ ▷ create a new AutoCompletionTrieNode
  ▷ assign new node to this position. Index of offspring represents a character
    $C_{index} \leftarrow \boldsymbol{t}$
    $\boldsymbol{p} \leftarrow \boldsymbol{t}$ ▷ set parent to the new node created
  else
    $\boldsymbol{p} \leftarrow \boldsymbol{n}$ ▷ If node exists at index, go to it
▷ If node is key and not already added
if $getIsKey(\boldsymbol{p})$ ***and*** *added* ***is false then***
  ▷ if word is a key, update its frequency of occurrence
  $newFrequency \leftarrow wordFrequency + getWordFrequency(\mathbf{p})$ ▷ if word is a key, update its frequency of occurrence
  $setWordFrequency(p, newFrequency)$
  return (*added*) ▷ if node already in trie, return false for not added
$setIsKey(\mathbf{p}, \mathbf{true})$
$setWordFrequency(\mathbf{p}, wordFrequency)$; ▷ *set frequency of word when it's a key*
$added \leftarrow$ ***true***
return (*added*)

4.2 Algorithm for addDictionaryToTrie(df)

This algorithm takes a valid CSV file as parameter. Each line in a valid file has the following format: word,frequency. The algorithm extracts word-frequency pairs from each line and calls add(word,frequency) on AutoCompletionTrie **t** to add them.

Algorithm 12 addDictionaryToTrie(*df*) **return**

Require: A dictionary CSV file *df*,where each line represents a unique word and frequency of occurrence

Ensure:

while *df* **not EOF do** ▷ *while not end of file*

  $line \leftarrow readLine(df)$ ▷ read line by line

  $word \leftarrow getWord(line)$ ▷ Implementation is different. Get word from line

  $wordFrequency \leftarrow getWordFrequency(line)$ ▷ Implementation is different. Get wordFrequency from line

▷ **t** is the AutoCompletionTrie being operated on. Refer to line 4 in section 1: How to read this report?

  $add(\mathbf{t}, word, wordFrequency)$

Note: Pseudocode and implementation for contains through to getSubTrie are the same as in Trie

4.3 Algorithm for boolean contains(String word)

Algorithm 13 contains(*key*) **return** (*contains*)

Require: *key* is a word and of type string

Ensure: *contains* is a boolean, either true or false. Indicate if a word is a key

contains ← **false**

if r is null or *key* **is null then** ▷ ***r*** *is the the root node of the AutoCompletionTrie*

  return (contains)

$asciiOffset \leftarrow 97$ ▷ *97 is 'a' in ascii. Start of small letters*

$\mathbf{p} \leftarrow \mathbf{r}$ ▷ *assign* ***r****,root of trie ,to* ***p***

for $i \leftarrow 1$ **to** $size(key)$ **do** ▷ looping character by character

  $character \leftarrow convertToChar(key_i)$ ▷ *converts* key_i *into ascii equivalent* ▷ *see appendix for ascii table*

▷ converts index to a value between 1-26, mapping it between 'a' to 'z'

  $index \leftarrow charToInt(character) - asciiOffset$

▷ *returns an array containing pointers to children of parent node*

  $\mathbf{C} \leftarrow getOffspring(\mathbf{p})$

  $\mathbf{n} \leftarrow C_{index}$ ▷ get offspring at mapped index

  if n is null then ▷ *If node is null, key is not in trie*

    contains ← ***false***

    return *(contains)*

  $\boldsymbol{p} \leftarrow n$ ▷ if a reference corresponding to the index is found, go to it.

▷ *If node is key,trie contains it*

if $getIsKey(\mathbf{p})$ **is true then**

  contains ← *true*

  return (*contains*)

return (*contains*) ▷ trie does not contain word

4.4 Algorithm for boolean String outputBreadthFirstSearch()

Algorithm 14 outputBreadthFirstSearch() **return** (*result*)

Require:

Ensure: *result* is of type string and contains all words in breadth first search order

if r is null then ▷ ***r*** *is the the root node of the trie data structure*

  return null

$asciiOffset \leftarrow 97$ ▷ *97 is 'a' in ascii. Start of small letters*

$\mathbf{Q} \leftarrow Queue()$ ▷ holds a collection of nodes type TrieNode

$enqueue(\mathbf{Q}, \mathbf{r})$ ▷ Add root,**r**, to front of queue,**Q**

while Q is not empty do

▷ *dequeue(**Q**) removes top most trie node and assign to temp,**t***

  $\mathbf{t} \leftarrow dequeue(\mathbf{Q})$

▷ *getOffspring(node) returns an array containing references to children of that node*

  $\mathbf{C} \leftarrow getOffspring(\mathbf{t})$

  for $i \leftarrow 1$ **to** $size(\mathbf{C})$ **do**

    if C_i **is not null then**

      $add(\mathbf{q}, C_i)$ ▷ add offspring of node to the queue so it goes thorough them in breadth first order

▷ *concatenate character represented by i to result*

      $result \leftarrow result + intToChar(i + asciiOffset)$

return (*result*)

4.5 Algorithm for String outputDepthFirstSearch()

Algorithm 15 outputDepthSearchSearch() **return** (*result*)

Require:

Ensure: *result*, a string in depth first search order. ▷ *builder* is an empty string, and **r** is the root of the Trie.

$result \leftarrow depthFirstSearch(builder, \mathbf{r})$

return (*result*)

4.5.1 Recursive implementation of String outputDepthFirstSearch()

Algorithm 16 outputDepthFirstSearch(builder,**r**) **return** (*result*)

Require: builder, an empty string and **r** of type TrieNode

Ensure: *result* is of type string and contains all words in depth first search order

if r is null then ▷ ***r*** *is the the root node of the trie data structure*

  return null

▷ *getOffspring(node) returns an array containing references to children of that node*

$\mathbf{C} \leftarrow getOffspring(\mathbf{r})$

$asciiOffset \leftarrow 97$ ▷ *97 is 'a' in ascii. Start of small letters*

for $i \leftarrow 1$ **to** $size(\mathbf{C})$ **do**

  if C_i **is not null then**

    $result \leftarrow result + intToChar(i + asciiOffset)$ ▷ *convert index to char(0 maps to 'a')*

    $outputDepthFirstSearch(result, C_i)$ ▷ recursive call on node's children- depth first

return (*result*)

4.6 Algorithm for getSubTrie(String prefix)

Algorithm 17 getSubTrie($prefix$) **return** (**t**)

Require: **r**,root node of AutoCompletionTrie and $prefix$, a string representing part of a word

Ensure: **t** is a AutoCompletionTrie data structure rooted at $prefix$

if r is null or $prefix$ **is null then**
  r $\leftarrow$ **null** ▷ *root is the the root node of the trie data structure*
$asciiOffset \leftarrow 97$ ▷ *97 is 'a' in ascii. Start of small letters*
p $\leftarrow$ **r**
for $i \leftarrow 1$ **to** $size(prefix)$ **do**
  ▷ *converts* $prefix_i$ *into ascii equivalent*
  $character \leftarrow convertToChar(prefix_i)$
  ▷ converts index to a value between 1-26, mapping it between 'a' to 'z'
  $index \leftarrow charToInt(character) - asciiOffset$ ▷ see appendix for ascii table
  ▷ *getOffspring(node) returns an array containing references to children of that node*
  C $\leftarrow getOffspring(\mathbf{p})$
  if C_i **is null then** ▷ prefix does not exist
    return (**null**)
  p $\leftarrow C_{index}$
return ($AutoCompletionTrie(\mathbf{p})$) ▷ return an AutoCompletionTrie rooted at the last node assigned to **p**

4.7 Algorithm for wordFrequencyInfo()

wordFrequencyInfo() is an interface function. It passes an empty queue, empty string and starting node(usually root) to wordFrequencyInfo(**Q**, *textBuilder*, **r**). The queue returned should contain WordAndFrequency entities(For definition,see part 3 introduction).

Algorithm 18 wordFrequencyInfo() **return** (**Q**)

Require:

Ensure: **Q**, a queue that guarantees head will contain word with highest frequency or if all elements have same frequency, shortest key length first.

▷ ***Q*** *guarantees sorted order for head of queue*
$\boldsymbol{Q} \leftarrow Queue()$ ▷ *textBuilder is an empty string, and* ***r*** *is the root of the AutoCompletetionTrie*
$wordFrequencyInfo(\boldsymbol{Q}, textBuilder, \boldsymbol{r})$
return ($\boldsymbol{Q}$)

4.8 Algorithm for wordFrequencyInfo(Q, *textBuilder*, r)

wordFrequencyInfo(**Q**, *textBuilder*, **r**) is called by wordFrequencyInfo(). The algorithm goes through the AutoCompletionTrie recursively. When it encounters a node that is a key, it means that the textBuilder contains a complete word. A WordAndFrequency entity is then made and added to the queue.

The implementation of this pseudocode uses a priority queue.

1. Ability to specify priority of an element. As such, no need to sort
2. Adding to priority queue is log(n) which is fairly fast
3. Polling/removing from priority queue is O(1)

Algorithm 19 wordFrequencyInfo(**Q**, $textBuilder$, **r**) **return**

Require: **Q**, an empty queue,$textBuilder$, a string to append to, and **r**, the root of the AutoCompletionTrie
Ensure: **Q** contains all WordFrequency entities from AutoCompletionTrie. Head guaranteed to be in order

if $getIsKey(\mathbf{r})$ **then** ▷ Make a WordAndFrequency entity with word and frequency of occurrence
  $\mathbf{t} \leftarrow WordAndFrequency(textBuilder, getFrequency(\mathbf{r}))$
  $add(\mathbf{Q}, t)$
▷ *getOffspring(node) returns an array containing references to children of that node*
$\mathbf{C} \leftarrow getOffspring(\mathbf{r})$
for $i \leftarrow 1$ to $size(C)$ **do**
  if C_i *is* **not null then**
    $textBuilder \leftarrow textBuilder + intToChar(97 + i)$ ▷ *intToChar converts an int to it's char equivalent*
    $wordFrequencyInfo(\mathbf{Q}, textBuilder, C_i)$
    $removeLastCharacter(textBuilder)$ ▷ *backtracks one level up the AutoCompletionTrie*

4.9 Algorithm for prefixMatches(P)

This function finds the most probable matches for each prefix in the list passed to it and writes them to a CSV file.

Algorithm 20 prefixMatches(**P**) **return**

Require: **P**, a list of prefixes
Ensure: Adding top three most probable words for each prefix in **P**. Prefix, words and probability should be written to a CSV file.

$\mathbf{T} \leftarrow List()$
for all prefix in **P** **do**
  $add(\mathbf{T}, prefix)$ ▷ **T** is will be saved to a CSV lotrMatches. To respect format of file, add prefix
  $\mathbf{s} \leftarrow getSubTrie(prefix)$
  $\mathbf{Q} \leftarrow wordFrequencyInfo()$ ▷ **Q** contains WordAndFrequency entities for that prefix
  $totalFreq \leftarrow 0$
  for all w in **Q** **do** ▷ For each WordAndFrequency entity in **Q**
    $totalFreq = totalFreq + getFreq(\mathbf{w})$ ▷ Add frequency of WordAndFrequency entity to totalFreq
  if $size(\mathbf{Q}) < 3$ **then** ▷ if queue has less than 3 elements, set limit to size of **Q**
    $limit \leftarrow size(\mathbf{Q})$
  else
    $limit \leftarrow 3$ ▷ else we want top 3 most probable words
  for $i \leftarrow 1$ to $limit$ **do**
    $\mathbf{w} \leftarrow dequeue(\mathbf{Q})$ ▷ remove top element
    if w *is not null* **then**
      $prob \leftarrow getFreq(\mathbf{w})/totalFreq$ ▷ find probability of given word
      $completeKey \leftarrow prefix + \mathbf{w}$ ▷ Need to add prefix to make a complete word. This is due to getSubTrie()
      $\mathbf{T} \leftarrow add(\mathbf{T}, completeKey)$
      $\mathbf{T} \leftarrow add(\mathbf{T}, prob)$
  $saveToFile("lotrMatches.csv", \mathbf{T})$
  clear(**T**) ▷ clear contents of **T** to add information for next prefix

Figure 2: lotrMatches format

```
ab,about,0.566667,above,0.3,able,0.1
go,going,0.277778,go,0.240741,good,0.166667
the,the,0.626703,they,0.153951,them,0.06812
mer,merry,0.947368,merely,0.026316,merrily,0.026316
fro,frodo,0.490909,from,0.436364,front,0.072727
gr,great,0.19697,ground,0.181818,grass,0.151515
gol,goldberry,0.6,golden,0.4
sam,sam,1
```

Note: pseudocode for saveToFile("lotrMatches.csv",T) is not very relevant

5 Code Listing

Listing 1: DictionaryMaker.java

```java
/*****************************************************************************

 File        : DictionaryMaker.java

 Date        : Wednesday 18th March 2020

 Author      : 100246776

 Description : Provide implementation for formDictionary and saveToFile in part 1

 Last update : 18th May 2020

 *****************************************************************************/

package dsacoursework2;
import java.io.*;
import java.util.*;
public class DictionaryMaker {
    /**
     * Reads all the words in a comma separated text document into an Array
     *
     * @param
     */
    public static ArrayList<String> readWordsFromCSV(String file, String delimiter) throws FileNotFoundException {
        Scanner sc = new Scanner(new File(file));
        sc.useDelimiter(delimiter);
        ArrayList<String> words = new ArrayList<>();
        String str;
        while (sc.hasNext()) {
            str = sc.next();
            str = str.trim();
            str = str.toLowerCase();
            words.add(str);
        }
        return words;
    }

    /**
     * Forms a dictionary of word,frequency tuples. A word should be unique
     * @param readFile, the file to be read
     * @param writeFile, the file to be written to
     * @throws IOException
     */
    public void formDictionary(String readFile, String writeFile) throws IOException {

        ArrayList<String> listOfWords = readWordsFromCSV(readFile, ",");
        TreeMap<String, Integer> frequencyDictionary = new TreeMap<>();//treeMap maintains sorted order
        for (String s : listOfWords) {
            if (!frequencyDictionary.containsKey(s)) {//if treemap does not contain the key
                frequencyDictionary.put(s, 1);//if key not already in map, set frequency to 1
            } else {
                int frequency=frequencyDictionary.get(s) + 1;
```

```
                frequencyDictionary.put(s,frequency);//if key already in map, increase its frequency by 1
            }
        }
        System.out.println(frequencyDictionary);//testing purposes
        saveToFile(frequencyDictionary, writeFile);
        System.out.println("file saved");
    }

    /**
     * Writes a map to a CSV file
     * @param frequencyDictionary, a map of word,frequency pair from formDictionary
     * @param writeFile, an empty CSV file to write to
     * @throws IOException
     */
    public static void saveToFile(TreeMap<String, Integer> frequencyDictionary, String writeFile) throws IOException {
        FileWriter fileWriter = new FileWriter(writeFile);
        PrintWriter printWriter = new PrintWriter(fileWriter);
        for (Map.Entry<String, Integer> entry : frequencyDictionary.entrySet())//loop through TreeMap
        {
            printWriter.println(entry.getKey() + "," + entry.getValue());
        }
        printWriter.close();
    }

    public static void main(String[] args) throws Exception {
        DictionaryMaker df = new DictionaryMaker();
        /*
        Testing done by comparing test.csv to testDictionary, provided in zip file
         */
        df.formDictionary("testDocument.csv", "test.csv");
    }
}
```

Listing 2: TrieNode.java

```java
/*****************************************************************************

 File        : TrieNode.java

 Date        : Wednesday 18th March 2020

 Author      : 100246776

 Description : Implementation of TrieNode

 Last update : 18th May 2020

 *****************************************************************************/
package dsacoursework2;

public class TrieNode {
    private boolean isKey;//isKey=true if node completes a word
    private final TrieNode[] offspring;

    public TrieNode() {
        this.offspring = new TrieNode[26];
    }//An array to hold the references to offspring

    /**
     *
     * @return an array of reference to children nodes
     */
    public TrieNode[] getOffspring() {
        return offspring;
    }

    /**
     *
     * @return either true of false if node is a key
     */
    public boolean getIsKey() {
        return this.isKey;
    }

    /**
     * Setter method to set isKey to true or false
     * @param bool
     */
    public void setIsKey(boolean bool) {
        this.isKey = bool;
    }//set isKey attribute of the trie node
}
```

Listing 3: Trie.java

```java
/****************************************************************************

 File        : Trie.java

 Date        : Wednesday 25th March 2020

 Author      : 100246776

 Description : Implementation of Trie, Solutions to part 1, Testing

 Last update : 18th May 2020

 ****************************************************************************/
package dsacoursework2;
import java.util.LinkedList;
import java.util.Queue;

public class Trie {
    private static final int ASCII_OFFSET = 97;
    private TrieNode root;

    public Trie() {
        root = new TrieNode();
    }//A trie is constructed with a root node

    public Trie(TrieNode trieNode) {
        root = trieNode;
    }

    /**
     * Adds a word to a trie if word not already a key in the trie.
     * @param word should be valie i.e not null or empty
     * @return True for added. False for not added
     */
    boolean add(String word) {
        boolean added = false;
        if (root == null || word==null || word.equals(""))
        {
            System.out.println("Some null reference");
            return added;
        }
        TrieNode parent = root;
        for (int i = 0; i < word.length(); i++) //for each character in the key
        {
            char character = word.charAt(i);//converts i to ascii equivalent
            int index = character - ASCII_OFFSET;//b being 98 and a being 97, then the index of b maps to 1(98-97)
            TrieNode nodeReference=parent.getOffspring()[index];
            if (nodeReference == null)//if no reference to any node
            {
                added = true;
                TrieNode temp = new TrieNode();//create a new node
                parent.getOffspring()[index] = temp; //add temp to this index position
                parent = temp;//set parent to the new node created
            } else {
                parent = nodeReference;//if node exists at particular index, go to it
            }
        }
        if (parent.getIsKey() && !added) {//if node is a key, return false for not
```

```java
added
            System.out.println(word+" already in trie");
            return added;//if node already in trie, return false for not added
        }
        parent.setIsKey(true);
        added=true;
        System.out.println(word+" added");
        return added;
    }

    /**
     * If word is a key in trie, return true. Otherwise return false.
     * @param word should be valid i.e not null and not empty
     * @return true for word in trie. False for word not in trie
     */
    public boolean contains(String word) {
        boolean contains=false;
        if (root == null  || word ==null || word.equals("")) {
            System.out.println("Some null reference");
            return contains;
        }
        TrieNode p = root;
        for (int i = 0; i < word.length(); i++) {
            char c = word.charAt(i);
            int index = c - ASCII_OFFSET;
            TrieNode[] children=p.getOffspring();
            TrieNode n=children[index];
            if (n== null) {
                /*
                Say, we look for word cat in trie. In the offspring array of root, there is no reference at index 2
                representing character 'c', this means word 'cat' is not present in trie. Return false.
                Same applies for trie nodes further down the tree
                 */
                System.out.println(word+ " is not in trie");
                return contains;
            }
            p = n;//if a reference corresponding to the index is found, go to it.
        }
        /*
        Return true only when word is a key
         */
        if (p.getIsKey()) {
            System.out.println(word+ " is in trie");
            contains= true;
            return contains;
        }
        /*
        Say,we look for word 'cat'.The characters 'c','a','t' are in the trie. However, node that holds 't' is not a
        key. This means 'cat' in the trie is part of another word that is a key. For e.g catalogue. In this case, return
        false
         */
        System.out.println(word+ " is not in trie");
        return contains;//trie does not contain the key
    }

    /**
```

```
     * Goes through trie and outputs each character in breadth first search order
     * @return A string containing characters in that order
     * Design decision:
     * Use a Queue because it respects first-in first-out rule. This is useful for breadth first search.
     * Underlying implementation of queue is a linkedlist. This ensures ensure O(1) for enqueue and dequeue.
     */
    public String outputBreadthFirstSearch() {
        if (root == null)
            return null;
        StringBuilder sb = new StringBuilder();
        Queue<TrieNode> queue = new LinkedList<>();
        queue.add(root);
        while (!queue.isEmpty()) {
            TrieNode temp = queue.remove();//remove oldest node added
            TrieNode[] c=temp.getOffspring();
            for (int i = 0; i < c.length; i++) {
                if (c[i] != null) {
                    queue.add(c[i]);//add offspring of node to the queue so it goes through them in breadth first order
                    sb.append((char) (i + ASCII_OFFSET));//index of offspring reference corresponds to character
                }
            }
        }
        System.out.println(sb.toString());//testing purposes
        return sb.toString();//toString method returns a string object
    }

    /**
     * Interface method to outputDepthFirstSearch(StringBuilder sb, TrieNode trieNode)
     * @return A string of characters in depth first search order
     */
    public String outputDepthFirstSearch()
    {
        StringBuilder sb=new StringBuilder();
        String result=outputDepthFirstSearch(sb,root);
        return result;
    }

    /**
     * Goes through trie and outputs character in depth first search order
     * @param sb An empty stringbuilder to store characters
     * @param trieNode Start point of depth first Search. Can specify any node as returned by getSubTrie(String prefix)
     * @return A string of characters in depth first search order
     * Design decision:
     * Recursion as opposed to a iterative version provides a more elegant and shorter solution
     * No need to handle stack data structure
     */
    private String outputDepthFirstSearch(StringBuilder sb, TrieNode trieNode) {
        if (trieNode == null) {//base case for non=existing trie
            return null;
        }
        TrieNode[] children = trieNode.getOffspring();
        for (int i = 0; i < children.length; i++) {
            if (children[i] != null)//ignore nulls
            {
```

```java
                sb.append((char) (i + ASCII_OFFSET));//convert index to char
                outputDepthFirstSearch(sb, children[i]);//recursive call on node's children- depth first
            }
        }
        return sb.toString();
    }

    /**
     * @param prefix, an incomplete word
     * @return A Trie rooted at prefix
     */
    public Trie getSubTrie(String prefix) {
        int index;
        if (root == null || prefix==null || prefix.equals("")) {
            return null;
        }
        TrieNode parent = root;
        for (int i = 0; i < prefix.length(); i++) {
            index = prefix.charAt(i) - ASCII_OFFSET;
            if (parent.getOffspring()[index] == null) {//prefix does not exist
                return null;
            }
            parent = parent.getOffspring()[index];//if node is not null, go to it
        }
        return new Trie(parent);//return a trie rooted at the last node assigned to parent
    }

    /**
     * Interface method to getAllWords(LinkedList listOfWords, StringBuilder sb, TrieNode root)
     * @return A LinkedList containing all keys in the trie
     */
    public LinkedList<String> getAllWords()//interface method for private getAllWords method
    {
        if(root==null){
            return null;
        }
        LinkedList<String> listOfWords = new LinkedList<>();
        getAllWords(listOfWords, new StringBuilder(), root);
        if(listOfWords.size()==0){
            return null;
        }
        return listOfWords;
    }

    /**
     * Populates a LinkedList with keys from the trie.
     * @param listOfWords A linkedlist
     * @param sb An empty StringBuilder
     * @param root Root node of Trie
     */
    private void getAllWords(LinkedList listOfWords, StringBuilder sb, TrieNode root) {
        if (root.getIsKey()) {
            System.out.println(sb.toString());//for test purposes
            listOfWords.add(sb.toString());//if key is found, add to listOfWords
        }
```

```
        TrieNode[] offspring = root.getOffspring();
        for (int i = 0; i < offspring.length; i++) {
            TrieNode o=offspring[i];
            if (o != null) {
                getAllWords(listOfWords, sb.append((char) (ASCII_OFFSET + i)), o);
                /*
                Say, the key 'bat'is added to stringBuilder, there might still be other keys to find such as bass
                Once function returns, we continue adding to the string "ba" if there any other keys.
                 */
                sb.setLength(sb.length() - 1);//backtracks one level
            }
        }
    }


    public static void main(String[] args) {
        Trie trie = new Trie();

        System.out.println("Testing add(key)...");
        if (!trie.add("bat")) throw new AssertionError();
        if (!trie.add("cat")) throw new AssertionError();
        if (!trie.add("chat")) throw new AssertionError();
        if (!trie.add("cheese")) throw new AssertionError();
        if (!trie.add("cheers")) throw new AssertionError();
        if (!trie.add("cheer")) throw new AssertionError();
        if (trie.add("bat")) throw new AssertionError();
        if (trie.add("cat")) throw new AssertionError();
        if (trie.add("chat")) throw new AssertionError();
        if (trie.add("cheese")) throw new AssertionError();
        if (trie.add("cheers")) throw new AssertionError();
        if (trie.add(null)) throw new AssertionError();
        if (trie.add("")) throw new AssertionError();

        System.out.println("\nTesting contains(key)...");
        if (!trie.contains("bat")) throw new AssertionError();
        if (!trie.contains("cat")) throw new AssertionError();
        if (!trie.contains("cat")) throw new AssertionError();
        if (!trie.contains("cheer")) throw new AssertionError();
        if (!trie.contains("cheers")) throw new AssertionError();
        if (trie.contains("coronavirus")) throw new AssertionError();
        if (trie.contains("china")) throw new AssertionError();
        if (trie.contains("ch")) throw new AssertionError();
        if (trie.contains("")) throw new AssertionError();
        if (trie.contains(null)) throw new AssertionError();

        System.out.println("\nTesting outputBreadthFirstSearch()");
        if(!trie.outputBreadthFirstSearch().equals("bcaahttaetersse")) throw new AssertionError();

        System.out.println("\nTesting outputDepthFirstSearch()");
        System.out.println(trie.outputDepthFirstSearch());
        if(!trie.outputDepthFirstSearch().equals("batcathateersse")) throw new AssertionError();

        System.out.println("\nTesting getAllWords()");
        trie.getAllWords();
        System.out.println("Test passed. If adding more keys, check if getAllWords() finds all keys");
```

```
        System.out.println("\nTesting getSubTrie(String prefix)");
        Trie subtrie=trie.getSubTrie("ch");
        subtrie.getAllWords();
        System.out.println("Test passed. If adding more keys, check if getAllWords() finds all keys");

        System.out.println("\nTesting getSubTrie(String prefix)");
        Trie subtrie1=trie.getSubTrie("coron");
        if(subtrie1==null){
            System.out.println("Invalid prefix");
        }

    }
}
```

Listing 4: AutoCompletionTrieNode.java

```java
/****************************************************************************

 File        : AutoCompletionTrieNode.java

 Date        : Monday 30th March 2020

 Author      : 100246776

 Description : Implementation of AutoCompletionTrieNode

 Last update : 18th May 2020

 ****************************************************************************/
package dsacoursework2;

public class AutoCompletionTrieNode {
    private boolean isKey;
    private final AutoCompletionTrieNode[] offspring;
    private int wordFrequency;

    /**
     * wordFrequency is only set when the node is a key
     */
    public AutoCompletionTrieNode()
    {
        this.offspring = new AutoCompletionTrieNode[26];
        this.wordFrequency = 0;
    }

    /**
     *
     * @return an array of offspring
     */
    public AutoCompletionTrieNode[] getOffSpring() {
        return offspring;
    }

    /**
     *
     * @return true if isKey is set to true. False otherwise
     */
    public boolean getIsKey() {
        return this.isKey;
    }

    /**
     *
     * @param bool, true when node needs to be a key. False otherwise.
     */
    public void setIsKey(boolean bool) {
        this.isKey = bool;
    }

    /**
     *
     * @param wordFrequency an integer that specifies the frequency of occurrence of
         ↪ word
     */
    public void setWordFrequency(int wordFrequency) {
```

```
        this.wordFrequency = wordFrequency;
    }


    public int getWordFrequency()
    {
        return wordFrequency;
    }
}
```

Listing 5: AutoCompletionTrie.java

```java
/*****************************************************************************

 File        : AutoCompletionTrie.java

 Date        : Wednesday 18th March 2020

 Author      : 100246776

 Description : Solutions for part 3

 Last update : 18th May 2020

 *****************************************************************************/
package dsacoursework2;

import java.io.*;
import java.text.DecimalFormat;
import java.util.*;

public class AutoCompletionTrie {
    private static final int ASCII_OFFSET = 97;
    private AutoCompletionTrieNode root;

    public AutoCompletionTrie() {
        root = new AutoCompletionTrieNode();
    }

    public AutoCompletionTrie(AutoCompletionTrieNode trieNode) {
        root = trieNode;
    }

    /**
     * Extracts word-frequency pairs for each line of a CSV file and call add method to add to AutoCompletionTrie
     * @param dictionary, a CSV file with each line of the format word,frequency
     * @throws IOException
     */
    public void addDictionaryToTrie(String dictionary) throws IOException {
        BufferedReader csvReader = new BufferedReader(new FileReader(dictionary));
        String line;
        String[] dictionaryEntry;
        while ((line = csvReader.readLine()) != null) {//reads line by line until end of document
            try {
                /*
                E.g. Alive,2 becomes dictionaryEntry[0]="Alive", dictionaryEntry[1]=2
                 */
                dictionaryEntry = line.split(",");
                String word = dictionaryEntry[0];
                int wordFrequency = Integer.parseInt(dictionaryEntry[1]);
                boolean isAdded = this.add(word, wordFrequency);//add word and frequency to trie
                if (isAdded) {
                    System.out.printf(" %s added to trie\n", dictionaryEntry[0]);//test purposes
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
```

```java
        }
        csvReader.close();
    }


    /**
     * If word is a key , update its wordFrequency. Otherwise, add the word and wordFrequency to the trie
     * @param word
     * @param wordFrequency
     * @return True for added and false for not added
     */
    public boolean add(String word, int wordFrequency) {
        boolean added = false;
        if (root == null || word == null) {
            return added;
        }
        AutoCompletionTrieNode p = root; //the root
        for (int i = 0; i < word.length(); i++) //for each character in the key
        {
            char c = word.charAt(i);//converts i to char
            int index = c - ASCII_OFFSET;//b being 98 and a being 97, then the index of b maps to 1(98-97)
            AutoCompletionTrieNode[] offSpring = p.getOffSpring();
            AutoCompletionTrieNode n = offSpring[index];
            if (n == null)//if no reference to any node
            {
                added = true;
                AutoCompletionTrieNode temp = new AutoCompletionTrieNode();//create a new node
                p.getOffSpring()[index] = temp;
                p = temp;//set p to the new node created
            } else {
                p = n;//if node exists at particular index, go to it
            }
        }
        if (p.getIsKey() && !added) {
            //if word is a key, update its frequency of occurrence
            int newFrequency = wordFrequency + p.getWordFrequency();
            p.setWordFrequency(newFrequency);
            System.out.printf("%s is a key in trie. Frequency is now %d\n", word, newFrequency);
            return added;//added is still false. No new key added
        }
        p.setIsKey(true);
        added = true;
        p.setWordFrequency(wordFrequency);//set frequency of word when it is a key
        return added;
    }


    public boolean contains(String key) {
        boolean contains = false;
        if (root == null || key == null || key.equals("")) {
            System.out.println("Some null reference");
            return contains;
        }
        AutoCompletionTrieNode p = root;//starts at the root
        for (int i = 0; i < key.length(); i++) {
            char c = key.charAt(i);//converts i to char
```

```java
            int index = c - ASCII_OFFSET;//b being 98 and a being 97, then the index of b maps to 1(98-97)
            AutoCompletionTrieNode[] children = p.getOffSpring();
            AutoCompletionTrieNode n = children[index];
            if (n == null) {//if a index that maps to a character is not found, key is not in trie
                /*
                Say, we look for word cat in trie. In the offspring array of root, there is no reference at index 2
                representing character 'c', this means word 'cat' is not present in trie. Return false.
                Same applies for trie nodes further down the tree
                 */
                System.out.println(key + " is not in trie");
                return contains;
            }
            p = n;//if index exists, go to the node
        }
        /*
        Return true only when word is a key
         */
        if (p.getIsKey()) {
            System.out.println(key + " is in trie");
            contains = true;
            return contains;
        }
        /*
        Say,we look for word 'cat'.The characters 'c','a','t' are in the trie. However, node that holds 't' is not a
        key. This means 'cat' in the trie is part of another word that is a key. For e.g catalogue. In this case, return
        false
         */
        System.out.println(key + " is not in trie");
        return contains;//trie does not contain the key
    }


    public String outputBreadthFirstSearch() {
        if (root == null)
            return null;
        StringBuilder sb = new StringBuilder();
        Queue<AutoCompletionTrieNode> queue = new LinkedList<>();
        queue.add(root);
        while (!queue.isEmpty()) {
            AutoCompletionTrieNode temp = queue.remove();//remove oldest node added
            AutoCompletionTrieNode[] c = temp.getOffSpring();
            for (int i = 0; i < c.length; i++) {
                if (c[i] != null) {
                    queue.add(c[i]);//add offspring of node to the queue so it goes through them in breadth first order
                    sb.append((char) (i + ASCII_OFFSET));//index of offspring corresponds to character
                }
            }
        }
        System.out.println(sb.toString());//for testing purposes
        return sb.toString();
    }

```

```java
    public String outputDepthFirstSearch()//interface function
    {
        StringBuilder sb = new StringBuilder();
        String result = outputDepthFirstSearch(sb, root);
        return result;
    }


    private String outputDepthFirstSearch(StringBuilder sb, AutoCompletionTrieNode trieNode) {
        if (trieNode == null) {//base case for non=existing trie
            return null;
        }
        AutoCompletionTrieNode[] children = trieNode.getOffSpring();
        for (int i = 0; i < children.length; i++) {
            if (children[i] != null)//ignore nulls
            {
                sb.append((char) (i + ASCII_OFFSET));//convert index to char(0 maps to 'a')
                outputDepthFirstSearch(sb, children[i]);//recursive call on node's children- depth first
            }
        }
        return sb.toString();
    }


    public AutoCompletionTrie getSubTrie(String prefix) {
        int index;
        if (root == null || prefix == null || prefix.equals("")) {
            return null;
        }
        AutoCompletionTrieNode p = root;
        for (int i = 0; i < prefix.length(); i++) {
            index = prefix.charAt(i) - ASCII_OFFSET;
            if (p.getOffSpring()[index] == null) {//prefix does not exist
                return null;
            }
            p = p.getOffSpring()[index];
        }
        return new AutoCompletionTrie(p);
    }


    private void getAllWords(ArrayList listOfWords, StringBuilder sb, AutoCompletionTrieNode root) {
        if (root.getIsKey()) {
            System.out.println(sb.toString());
            listOfWords.add(sb.toString());
        }
        AutoCompletionTrieNode[] children = root.getOffSpring();
        for (int i = 0; i < children.length; i++) {
            if (children[i] != null) {
                getAllWords(listOfWords, sb.append((char) (ASCII_OFFSET + i)), children[i]);
                /*
                Say, the key 'bat'is added to stringBuilder, there might still be other keys to find such as bass
                Once function returns, we continue adding to the string "ba" if there any other keys.
```

```java
                 */
                sb.setLength(sb.length() - 1);//backtracks one level
            }
        }
    }

    public ArrayList<String> getAllWords()
    {
        ArrayList<String> listOfWords = new ArrayList<>();
        getAllWords(listOfWords, new StringBuilder(), root);
        return listOfWords;
    }


    public static class WordAndFrequency {
        /*
        Objects of WordAndFrequency hold a word string and frequency of that key
         */
        private String key;
        private int freq;

        WordAndFrequency(String key, int freq)
        {
            this.key = key;
            this.freq = freq;
        }
        WordAndFrequency(String key, String freq) {
            this.key = key;
            this.freq = Integer.parseInt(freq);
        }

        int getFreq() {
            return freq;
        }

        String getKey() {
            return key;
        }
    }

    /*
    Design decision for priority queue:
    1. Ability to specify priority of an element. As such, no need to sort
    2. Adding to priority queue is log(n) which is fairly fast
    3. Poll() is O(1) time complexity
    4. Better space complexity than using an arraylist that doubles in size to accommodate more elements
    5. Priority queue guarantees maintaining order for highest priority value. It does not necessarily maintain sorted order.
       As we only ever use it to find highest priorities, it is better to use it instead of a treeset,
        which maintains sorted order throughout
       It is also more computationally expensive to maintain trees
     */
    /**
     * An interface function
     * @return A queue, q, containing WordAndFrequency objects
     */
    private PriorityQueue<WordAndFrequency> wordFrequencyInfo() {
        //Each WordAndFrequency object has a key and its frequency
```

```java
        PriorityQueue<WordAndFrequency> q = new PriorityQueue<>(new CompareByFreqAndAlphabet());
        wordFrequencyInfo(q, new StringBuilder(), root);//get all words in the trie and their associated frequency
        return q;
    }

    /**
     * Comparator logic for priority queue
     */
    static class CompareByFreqAndAlphabet implements Comparator<WordAndFrequency> {
        public int compare(WordAndFrequency m1, WordAndFrequency m2) {
            if (m1.getFreq() < m2.getFreq()) {
                return 1;
            } else if (m1.getFreq() > m2.getFreq()) {
                return -1;
            } else {
                return m1.getKey().compareTo(m2.getKey());//if same frequency, sort by key length i.e choose shorter word
            }
        }
    }


    /**
     * The algorithm goes through the AutoCompletionTrie recursively.
     * When it encounters a node that is a key, it means that the stringbuilder contains a complete word.
     * A WordAndFrequency entity is then made and added to the priority queue.
     * @param queue, an empty priority queue
     * @param sb, an empty stringbuilder
     * @param root, starting node in an AutoCompletionTrie
     */
    private void wordFrequencyInfo(PriorityQueue queue, StringBuilder sb, AutoCompletionTrieNode root) {
        if (root.getIsKey()) {
            queue.add(new WordAndFrequency(sb.toString(), root.getWordFrequency()));
        }
        AutoCompletionTrieNode[] offspring = root.getOffSpring();
        for (int i = 0; i < offspring.length; i++) {
            if (offspring[i] != null) {
                wordFrequencyInfo(queue, sb.append((char) (97 + i)), offspring[i]);
                sb.setLength(sb.length() - 1);//backtracks one level
            }
        }
    }

    /**
     * This function finds the most probable matches for each prefix in the list
     * Writes matches to a CSV file.
     * @param prefixes, an ArrayList of prefixes
     * @throws IOException
     */
    public void prefixMatches(ArrayList<String> prefixes) throws IOException {
        final DecimalFormat df = new DecimalFormat("#.######");
        ArrayList<String> toWrite=new ArrayList<>();
        for (String prefix : prefixes) {
            toWrite.add(prefix);
            AutoCompletionTrie currentSubTrie = this.getSubTrie(prefix);//get subtrie rooted at prefix
```

```java
            /*
            temp contains a WordAndFrequency objects. The object contains a unique word and frequency of occurrence
             */
            PriorityQueue<WordAndFrequency> temp = currentSubTrie.wordFrequencyInfo();
            int totalFreq = 0;
            for (WordAndFrequency wordAndFrequency : temp) {
                /*
                adding frequency attribute of each object to find totalFrequency
                 */
                totalFreq += wordAndFrequency.getFreq();
            }

            int limit = Math.min(temp.size(), 3);

            for (int i = 0; i < limit; i++)
            {
                WordAndFrequency wf=temp.poll();//removes and return top most object in priority queue
                if(wf !=null){
                    toWrite.add(prefix+wf.getKey());
                    toWrite.add(df.format((double)wf.getFreq()/totalFreq));
                }
            }
            saveToFile("lotrMatches.csv", toWrite);
            toWrite.clear();
        }
    }

    /**
     * Writes ArrayList to a file in a specific format
     * @param writeFile, name of file to write to
     * @param toWrite, an Arraylist to write to file
     * @throws IOException
     */
    private static void saveToFile(String writeFile, ArrayList<String> toWrite) throws IOException
    {
        BufferedWriter writer = new BufferedWriter(
                new FileWriter(writeFile, true));  //with append set to true, file data is not overwritten
        for(int i=0;i<toWrite.size();i++)
        {
            if(i != toWrite.size()-1)
            {
                writer.write(toWrite.get(i)+",");
            }
            else {
                writer.write(toWrite.get(i)+System.lineSeparator());
            }
        }
        writer.close();
    }


    public static void main(String[] args) throws IOException {

        AutoCompletionTrie trie = new AutoCompletionTrie();
        DictionaryMaker df = new DictionaryMaker();
        /*
```

```
        Writes lotr as a dictionary in csv format
         */
        df.formDictionary("lotr.csv", "dictionaryOfWords.csv");

        trie.addDictionaryToTrie("dictionaryOfWords.csv");
        /*
        Load queries into an ArrayList
         */
        ArrayList<String> lotrQueries = DictionaryMaker.readWordsFromCSV("lotrQueries.csv", "\n");

        trie.prefixMatches(lotrQueries);

    }
}
```

6 Appendix

[b]

Figure 3: Ascii Table

ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	|
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

© w3resource.com

References

[1] Dr. Anthony Bagnall. *Tries*. Lecture notes. University of East Anglia,Norwich

[2] Dr. Anthony Bagnall. *Trees*. Lecture notes. University of East Anglia,Norwich

[3] Dr. Anthony Bagnall,Jason Lines. *Algorithm design and recursion*. Labsheet. University of East Anglia,Norwich

[4] Dr. Anthony Bagnall. *Lab 9: Trees* . Labsheet. University of East Anglia,Norwich

[5] Dr. Anthony Bagnall. *Lab 10: Trie* . Labsheet. University of East Anglia,Norwich