Data Structures and Algorithms
Coursework Assignment 2

Patryk Dobbek
stp18rsu
100023818

April 23, 2020

1 DictionaryMaker

This algorithm populates and returns a dictionary structure created from the the.

1.1 The 'formDictionary' algorithm

Algorithm 1 formDictionary(**doc**,n) **return** $(sortedDict, m)$

Require: list of strings **doc** of length n
Ensure: $sortedDict$:=, Dictionary of string keys and integer values of size m.
1: $sortedDict$ ▷ *empty dictionary with string keys and integer values.*
2: **for** $i \leftarrow 1$ **to** n **do**
3:     $temp \leftarrow doc_i$
4:     **if** $sortedDict$ **contains** $temp$ **then**
5:         $sortedDict$ **put** $(key \leftarrow temp, value \leftarrow value + 1)$
6:     **else**
7:         $sortedDict$ **put** $(key \leftarrow temp, value \leftarrow 1)$
8:     **end if**
9: **end for**
10: **return** $(sortedDict, m)$

1.2 Algorithm analysis for 'formDictionary' algorithm

Fundamental operation: (Line 3) **If** $sortedDict$ **contains** $temp$

Contain operations for the dictionary used in this algorithm (Java's red-black map-based tree, TreeMap) are $\mathbf{O(log(m))}$.

1.2.1 Worst case

The worst case for this algorithm is when every word in **doc** is unique, thereby making $\mathbf{m} = \mathbf{n}$ and **contains** operation on line 3 equal to $\mathbf{O(log(n))}$.

$$t(n) = \sum_{i=1}^{n} log(n) = n\ log(n)$$

$$t(n) = n\ log(n)$$

Which gives us linearithmic running time and $\mathbf{O(n\ log(n))}$ as the order of the algorithm.

1.2.2 Best case

The best case for this algorithm is for **doc** to contain repetitions of the same word, thereby making $\mathbf{m = 1}$ and **contains** operation on line 3 equal to $\mathbf{O(1)}$.

$$t(n) = \sum_{i=1}^{n} 1 = n$$

From here we can categorise the run-time function as linear and give $\mathbf{O(n)}$ as the order of the algorithm.

2 Trie

2.1 The 'add' algorithm

This function adds a string into the trie. It ensures space efficiency by initialising nodes and child arrays only when needed.

Algorithm 2 add(**key**, n, **root**) **return** *boolean*

Require: lower case alphabetical character array **key** of length n.
Require: reference to TrieNode **root** (root of a Trie structure)
Ensure: *boolean*, true if operation successful, false if key already in Trie.

```
curr ← root                                  ▷ saving a reference to the root node in this Trie.
for i ← 1 to n do
    charIndex ← key_i − 98                   ▷ should give range between 1-26
    if curr.children equals null then
        curr.children :=                     ▷ initialise space for references to other TrieNodes
    end if
    if curr.children_charIndex equals null then
        curr.children_charIndex :=           ▷ initialise TrieNode
    end if
    curr ← curr.children_charIndex
end for
if curr.isWord equals false then
    curr.isWord ← true
else return false
end if
return true
```

2.2 Algorithm analysis for 'add' algorithm

Fundamental operation: (Line 8) $curr.children_{charIndex} :=$

Fundamental operation is constant time.

2.2.1 Worst case

None of the characters present in **key** are present in the Trie in the same order. This case would require n fundamental operations.

$$t(n) = \sum_{i=1}^{n} 1 = n$$

Which gives us $\mathbf{O(n)}$ as the order of the algorithm and linear running time.

2.3 The 'contains' algorithm

This algorithm searches for the specified word and returns a boolean value depending on whether it is found within the structure.

Algorithm 3 contains(**key**, n, **root**) **return** *boolean*

Require: lower case ASCII character array **key** of length n.
Require: reference to a TrieNode **root** (root of a Trie structure)
Ensure: *boolean*, true if the word passed in is in the Trie as a while word, not just prefix. Otherwise, false.

1: $curr \leftarrow root$ ▷ *saving a reference to the root node in this Trie.*
2: **for** $i \leftarrow 1$ **to** n **do**
3:     $charIndex \leftarrow key_i - 98$ ▷ *should give range between 1-26*
4:     **if** $curr.children$ **equals** $null$ **then**
5:         **return** $false$
6:     **end if**
7:     **if** $curr.children_{charIndex}$ **equals** $null$ **then**
8:         **return** $false$
9:     **end if**
10:     $curr \leftarrow curr.children_{charIndex}$
11: **end for**
12: **if** $curr.isWord$ **equals** $false$ **then**
13:     **return** $false$
14: **end if**
15: **return** *true*

2.4 Algorithm analysis for 'contains' algorithm

Fundamental operation: (Line 7) **If** *curr.children* **equals** *null*

Fundamental operation is constant time.

2.4.1 Worst case

In the worst case the series of characters passed in as a character array are present in the Trie. This case would require n fundamental operations.

$$t(n) = \sum_{i=1}^{n} 1 = n$$

Which gives us $\mathbf{O(n)}$ as the order of the algorithm and linear running time.

2.5 The 'getSubTrie' algorithm

This algorithm returns a trie structure that is a subset of another trie. Returns null when prefix string cannot be found with the original trie.

Algorithm 4 getSubTrie(**prefix**, n, **root**) **return** $(subTrie, m)$

Require: lower case ASCII character array **key** of length n.
Require: reference to TrieNode **root** (root of a Trie structure).
Ensure: $subTrie$, Trie data structure of size m.

$curr \leftarrow root$ ▷ *saving a reference to the root node in this Trie.*
for $i \leftarrow 1$ **to** n **do**
    $charIndex \leftarrow key_i - 98$ ▷ *should give range between 1-26*
    if $curr.children$ **equals** $null$ **then**
        return $null$
    end if
    if $curr.children_{charIndex}$ **equals** $null$ **then**
        return $null$
    end if
    $curr \leftarrow curr.children_{charIndex}$
end for
$subTrie :=$ ▷ initialising an empty Trie
$subTrie.root \leftarrow curr$ ▷ setting the root node in subTrie
return $subTrie$

2.6 The 'outputBreadthFirstSearch' algorithm

Algorithm 5 outputBreadthFirstSearch(**root**,**q**) **return** (str, n)

Require: reference to TrieNode **root** (root of a Trie structure). Every TrieNode can hold a fixed array of 26 references to other TrieNodes called **children**

Ensure: *str*, string of lower-case alphabetical characters of size n (n equals to the size of the Trie).

q ▷ empty queue structure that can hold TrieNodes
add *root* to q (enqueue)
str := ▷ declaring a new empty string
$TrieNode$ ▷ declaring a new TrieNode
while q is not empty **do**
    $TrieNode \leftarrow$ dequeue q
    if $TrieNode.children$ **is not** *null* **then**
        for $i \leftarrow 1$ **to** 26 **do**
            if $TrieNode.children_i$ **is not** *null* **then**
                str add ASCII char equal to $i + 98$
                add *child* to q (enqueue)
            end if
        end for
    end if
end while
return (str, n)

2.7 The 'outputDepthFirstSearch' algorithm

The Trie function '$outputDepthFirstSearch$' is not described here because it simply calls on a recursive function '$getDepthFirstString$' of its root TrieNode.

Algorithm 6 getDepthFirstString() **return** (str, n)

Require: this TrieNode's fixed-sized array of 26 references to other TrieNodes, **children**.

Ensure: **str**, string of lower-case alphabetical characters of size n (n equals to the amount of offspring TrieNodes).

str := ▷ declare a new string
if *children* **is** *null* **then return** $(str, 0)$ ▷ return an empty string
end if
for $i \leftarrow 1$ **to** 26 **do**
    if $children_i$ **is** *not null* **then**
        str add ASCII char equal to $i + 98$
        str add $children_i.getDepthFirstString()$ ▷ recursive call
    end if
end for
return (str, n)

2.8 The 'findAllWords' algorithm

The Trie function '$findAllWords$' is not described here because it simply calls on a recursive function '$findWords$' of its root TrieNode.

Algorithm 7 findWords(**strList**,n,**str**,m) **return** *void*

Require: **strList**, list of n complete word strings found so far.
Require: **str**, string of lower-case alphabetical characters of size m.
Require: this TrieNode's fixed-sized array of 26 references to other TrieNodes, **children**.
Require: **isWord**, a boolean indicating whether this node is marks a complete word.
Ensure: **strList** populated with with words found.

```
str :=                                          ▷ declare a new string
if children is null then
    for i ← 1 to 26 do
        if children_i is not null then
            temp ← str add ASCII char equal to i + 98
            if children_i isWord then
                strList add temp
            end if
            children_i.findWords(strList, temp)
        end if
    end for
end if
return
```

---

3 AutoCompleteTrie

3.1 The 'populateFrequencyMap' algorithm

This is a recursive function retrieves all words and word frequencies within the AutoCompleteTrie using a depth first search. Those words are stored in a map of integer keys (frequencies) and string values (words).

This function is called by AutoCompleteTrie's '$getFrequencyMap()$' method, which is not described here because it simply declares a Sorted Dictionary and calls its root node's '$popupaleFrequencyMap$' function.

Algorithm 8 populateFrequencyMap(**sortedDict**,n,**str**,m) **return** *void*

Require: this TrieNode's fixed-sized array of 26 references to other TrieNodes, **children**.
Require: **str**, string of lower-case alphabetical characters of size m.
Require: dictionary with integer keys and values of list of strings (representing words found so far), **sortedDict**, of size n.
Ensure: *sortedDict* populated with integer keys (corresponding to words frequency) and list of strings values.

if *children* **is** *not null* **then**
  for $i \leftarrow 1$ **to** 26 **do**
    if $children_i$ **is** *not null* **then**
      if $children_i$ $isWord$ **then**
        $childFrequency \leftarrow children_i.frequency$
        $list \leftarrow sortedDict_{childFrequency}$
        if *list* **is** *null* **then**
          $list :=$ ▷ initialise list
        end if
        list add (*str* add ASCII char equal to i + 98)
        $sortedDict_{childFrequency} \leftarrow list$
      end if
      str add ASCII char equal to i + 98)
      $children_i.populateFrequencyMap(sortedDict, str)$ ▷ recursive call
    end if
  end for
end if
return

3.2 The 'writeAutoComplete' algorithm

Algorithm 9 writeAutoComplete(**prefix**,n,**writer**) **return** *void*

Require: reference to this AutoCompleteTrie, **trie**.
Require: **prefix**, string of lower-case alphabetical characters of size n, representing the query.
Require: **writer**, file writer object.

```
subTrie ← trie.getSubTrie(prefix)
allWords ← subTrie.getFrequencyMap()    ▷ map of frequencies and words
totalWordFreq ← 0
for i ← 1 to allWords.pairsCount do    ▷ count all frequencies
    totalWordFreq ← totalWordFreq + allWords_i
end for

    ▷ If the prefix is a complete word, insert it as one of the results
subTrieRoot ← subTrie.root
entryValue    ▷ declare list of strings
if subTrieRoot isWord then
    entryValue ← allWords_subTrieRoot.frequency
    if entryValue is null then
        entryValue :=    ▷ initialise
    end if
    entryValue push empty string
    allWord_subTrieRoot.frequency ← entryValue
end if
writer write prefix and ","

entry    ▷ declare an integer/list of string pair
probability    ▷ declare a float
writeCount ← 0
while allWords has entries do
    entry ← allWords.pop    ▷ integer/list pair
    entryValue ← entry.value    ▷ list of strings
    if entryValue has more than 1 element then
        sort entryValue in lexicographical order
    end if

    for i ← to entryValue.size do
        probability ← (entry.key/totalWordFreq)
        print prefix and entryValue_i and probability in format "word
        (probability 0.5)"

        writer write prefix and entryaValue_i and probability in format
        "word,0.5,"

        wordCount ← wordCount + 1
        if wordCount is 3 then
            print new line
            writer write new line
            return
        end if
    end for
end while
print new lane
writer write new line
return
```