Data Structures and Algorithms
Coursework Assignment 1

Patryk Dobbek
stp18rsu
100023818

April 4, 2020

1 DictionaryMaker

1.1 The 'formDictionary' algorithm

This algorithm assumes that the language it is written in has a sorted dictionary or map data structure (such Java's TreeMap or Cs's SortedDictionary). For languages that do not support such structures an additional sorting operations must be performed.

Furthermore, time complexity for operations such as *add*, *remove* or *contain* might be language-specific. While this report was written with the intention of being language-ambiguous, its main purpose is to analyse algorithms written in Java. For this reason its necessary for this section to reference a Java-specific data structure ($TreeMap$) to describe time complexity of $formDictionary$.

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

Require: list of strings **doc** of length n
Ensure: $dict$, Dictionary of string keys and integer values of size m.

1: $dict$ ▷ *empty dictionary with string keys and integer values.*
2: **for** $i \leftarrow 1$ to n **do**
3:     $temp \leftarrow doc_i$
4:     **if** $dict$ **contains** $temp$ **then**
5:         $dict$ **put** ($key \leftarrow temp$, $value \leftarrow value + 1$)
6:     **else**
7:         $dict$ **put** ($key \leftarrow temp$, $value \leftarrow 1$)
8:     **end if**
9: **end for**
10: **return** $(dict, m)$

1.2 Algorithm analysis for 'formDictionary' algorithm

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

Contain operations for TreeMaps 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 $\mathbf{O}(\mathbf{log}(\mathbf{n}))$.

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

Which gives us linearithmic running time and $\mathbf{O}(\mathbf{n}\ \mathbf{log}(\mathbf{n}))$ 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 $\mathbf{m} = \mathbf{1}$ and **contains** operation on line 3 equal to $\mathbf{O}(\mathbf{1})$.

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

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

2 Trie

2.1 The 'add' algorithm

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.

$curr \leftarrow root$ ▷ *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**
    $curr.children :=$ ▷ *initialise space for references to other TrieNodes*
  end if
  if $curr.children_{charIndex}$ **equals** $null$ **then**
    $curr.children_{charIndex} :=$ ▷ *initialise TrieNode*
  end if
  $curr \leftarrow curr.children_{charIndex}$
end for
if $curr.isWord$ **equals** $false$ **then**
  $curr.isWord \leftarrow true$
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

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

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 ← dequeue q
    if TrieNode.children is not null then
        for i ← 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 ← 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
```