Data Structures and Algorithms

Lab Practice 3: Trees

2019-2020 course

Contents

1 Introduction

This practice aims to achieve the following objective:

- Apply the theoretical-practical concepts learned in theory to implement the Abstract Data Type (ADT) prefix tree or *Trie*.

2 Prefix tree or *Trie*

A prefix tree or *Trie* is a hierarchical tree-like data structure that allows to efficiently store and retrieve (*"reTRIEve"*) information from large collections of associative elements (e.g. dictionaries), where search keys are typically text strings. Unlike binary search trees (BST), (1) internal nodes do not necessarily contain an associated element or key of interest, (2) arcs are labeled with symbols (characters), and (3) the position of a node in the tree with respect to the root (the arcs that are crossed along the path) defines the key or prefix to which they are associated. The way information is retrieved from a Trie is done similarly to how we would look for a word in a paper-based language dictionary. Figure 1 shows a Trie that stores objects of type `String` using the string's own value as a key (although they can store any other type of data).

2.1 Properties

The main properties of a Trie are the following:

- Prefixes and keys are built from the arcs of the tree.
  - All arcs are tagged with the character that is added to the prefix/key in each step.
- The root node is associated with the empty prefix.
- The depth of the nodes in the tree are equal to the length of their corresponding prefix or key.
- All descendants of a node share the same prefix.
- Leaf nodes: always store an item (e.g. `there`, `does`).
- Internal nodes: can store elements (e.g. `the`, `do`) or not (prefix nodes, e.g. `th`, `d`).

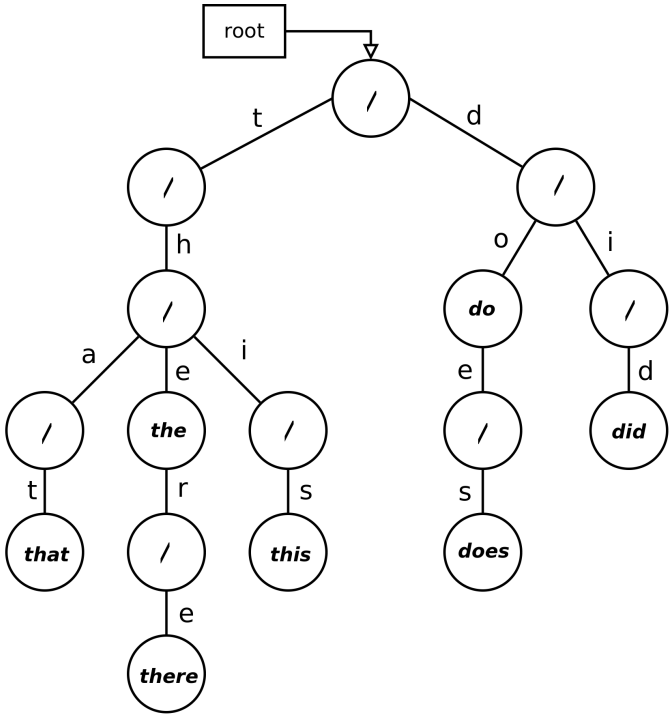

Figure 1: Example of a Trie that stores String objects using string's values as keys.

2.2 Temporal complexity

Unlike BSTs, temporal complexity does not depend on the number of elements n stored in the tree and their disposition (balanced vs. unbalanced tree). It depends solely and exclusively on the length $|k|$ of the keys k associated to the elements This means that, regardless of the number of items stored in a Trie, the cost of basic operations will always be the same: linear to the length of the $O(|k|)$ key.

For example, in the Trie of Figure 1, the cost of accessing to key $k =$ `do`: $O(|k|) = O(2)$, as 2 steps are needed to reach the corresponding node from the root of the tree (arcs: {`d`,`o`}); and not, being $n = 7$ (as the Trie contains 7 elements), e.g. $O(n) = O(7)$.

For this reason, Tries are very convenient to retrieve information from large repositories of elements.

2.3 Practical applications

Tries have many applications, and are very convenient in contexts where it have to be dealt with long data collections.

- They can be used to replace hash tables (dictionaries). In comparison:
  - No need of hash functions.
  - Far more efficient that hash tables in the worst case.
  - Absence of the key collision problem.
  - No need of costly auto-expansion and entry reallocation methods .
  - Keys can be recovered in alphabetical order.
- Auto-complete/predictive text and spell checking tasks.
- Efficient searches using text strings in extensive databases (e.g. catalogs of audiovisual content such as movies or books, being able to search by title, ISBN, etc.).

3 The *Trie* ADT

Below we define the interface *Trie* with a set of public methods (operations) that must be implemented. This interface will allow working with generic type objects `E`.

```
package P3;
import java.util.List;
public interface Trie<E> {
    public void insert(String key, E data) throws ExistingElementException;
    public E search(String key) throws ElementNotFoundException;
    public void delete(String key) throws ElementNotFoundException;
    public List<E> list();
    public List<E> prefixSearch(String prefix);
    public int size();
    public int numNodes();
}
```

As example, with this interface we could perform the following operations:

```
import P3.*;
Trie<String> t = ...; // we will store String objects in it!
t.insert("that", "tHaT");
t.insert("these", "THESE");
t.insert("there", "there");
String s1 = t.search("that"); // s1 = "tHaT"
List<String> l1 = t.list(); // l1 = ["tHaT", "THESE", "there"]
List<String> l2 = t.prefixSearch("the"); // l2 = ["THESE", "there"]
```

Below we describe in detail the expected behaviour of each method and some implementation hints:

- `void insert(String key, E data)`: inserts element `data` by key `key`.
  - This method will create, if necessary, the corresponding node sequence in the tree to add the element.
  - I.e. `insert("doe", "doe")` would not add any additional node to the Trie of Figure 1, whilst `insert("that is", "that is")` would insert the nodes corresponding to characters `" "`, `"i"` y `"s"`.
  - If `key` exists and its node already contains an element (regardless it is equal or not to `data`), an *ExistingElementException* will be thrown.

- `E search(String key)`: looks for an element with key `key`.
  - Provided that there is no sequence of nodes to reach the key `key` provided, or that it exists but the corresponding node does not store any element (= it is a prefix node), an *ElementNotFoundException* will be thrown.
- `void delete(String key)`: deletes an element with key `key`.
  - Provided that there is no sequence of nodes to reach the key `key` provided, or that it exists but the corresponding node does not store any element (= it is a prefix node), an *ElementNotFoundException* will be thrown.
  - If the element is found, in addition to deleting (dereferencing) it, it will recursively prune, if necessary, all those nodes that, as a consequence of deleting the element, become leaf nodes that do not store data.
  - Therefore, deleting an element does not necessarily imply deleting the node that stored it.
  - I.e. `delete("do")` would not prune any node from the tree in Figure 1, while `delete("does")` would involve a recursive pruning of two nodes: first `s`, and then `e`, as they would recursively become leaf nodes that do not store any element.
- `List<E> list()`: returns a list with all the elements stored in the Trie.
  - We recommend to use class *LinkedList* from the standard library of Java.
  - If there are no elements in the Trie, it will return an empty list.
- `List<E> prefixSearch(String prefix)`: returns a list with all the elements stored in the Trie sharing a given `prefix`.
  - We recommend to use class *LinkedList* from the standard library of Java.
  - If the prefix does not exist in the tree, it will return an empty list.
- `int size()`: returns the number of elements stored in the Trie ($\neq$ number of nodes!).
- `int numNodes()`: returns the number of nodes of the Trie.

3.1 The *TrieImpl* class

We will implement the interface *Trie* with a class called *TrieImpl*:

```
package P3;
import java.util.List;
import java.util.LinkedList;
public class TrieImpl<E> implements Trie<E> {
    ...
}
```

This class will store a reference to the root node (i.e. `root`) of the tree, a counter of the number of elements stored in it (i.e. `n`), and a counter of the number of nodes that compose the tree structure (i.e. `nNodes`). It will contain as many private methods as necessary, either to avoid code duplication, to define recursive methods, etc.

It is convenient to implement the method `toString()` to check the actual contents of the tree and debug your implementation. Representing a tree in a text string is a task that has its certain complexity. Below it is shown an example/suggestion of a text string representation of the Trie from Figure 1. In this representation, each line represents a node in the tree, with this format:

```
(depth level) -- arc --> {stored element} , [children_arcs_list]
```

```
(0) -- @ --> {/} , [t, d]
  (1) -- t --> {/} , [h]
    (2) -- h --> {/} , [a, e, i]
      (3) -- a --> {/} , [t]
        (4) -- t --> {that} , []
      (3) -- e --> {the} , [r]
        (4) -- r --> {/} , [e]
          (5) -- e --> {there} , []
      (3) -- i --> {/} , [s]
        (4) -- s --> {this} , []
  (1) -- d --> {/} , [i, o]
    (2) -- i --> {/} , [d]
      (3) -- d --> {did} , []
    (2) -- o --> {do} , [e]
      (3) -- e --> {/} , [s]
        (4) -- s --> {does} , []
```

3.2 The *TrieNode* class

Internally, the *TrieImpl* class will work with the *TrieNode* class, that will represent the tree nodes. Each node will store two sources of information: a reference to an element of type `E` (e.g. `data`), and a set of references to its child nodes (e.g. `children`).

Since arcs in a Trie are labeled with a symbol (character), pointers to child nodes must be retrievable by the corresponding arc symbol. For example, in the Trie from Figure 1, the root node has two child nodes, indexed by `"t"` and `"r"`, respectively.

There are different ways to maintain this indexing. Probably the simplest and most versatile, and one of the most efficient (although not the most one), is to use a hash table, so we will use the *HashMap* class from the standard Java library. This class will be instantiated using keys of type `Character` and values of type `TrieNode<E>`.

In addition, *TrieNode* must implement a series of public methods detailed below, which will be used by the *TrieImpl* class for all operations concerning the management of the structure and data storage in the tree.

```
package P3;
import java.util.HashMap;
public class TrieNode<E> {
    HashMap<Character, TrieNode<E>> children;
    E data;
    public TrieNode(){ ... }
    public Character[] getArcs(){ ... }
    public TrieNode<E> getChildNode(Character arc) { ... }
    public void addChildNode(Character arc, TrieNode<E> node) { ... }
    public void removeChildNode(Character arc) { ... }
    public int numChildrenNodes() { ... }
    ...
}
```

- `Character[] getArcs()`: returns an array of characters corresponding to the arcs leaving the node and linking to the corresponding childs.
  - I.e. `getArcs()` executed on the root node from Figure 1 would return `['t','d']`.
  - Internally, will make a call to the method *keySet()* of the hash table object.
- `TrieNode<E> getChildNode(Character arc)`: returns the child node that follows the arc (character) `arc`, or `null` if it does not have and outbound arc `arc`.
  - I.e. `getChildNode("o")` executed on the node `"d"` from Figure 1 would return the node that stores the element (string) `"do"`.
  - Internally, will make a call to the method *get()* of the hash table object.
- `void addChildNode(Character arc, TrieNode<E> node)`: adds a new child node `node` following the arc `arc`.
  - Internally, will make a call to the method *put()* of the hash table object.

- `void removeChildNode(Character arc)`: deletes from the children set the node that follows the arc `arc`.
  - Internally, will make a call to the method *remove()* of the hash table object.
- `int numChildrenNodes()`: returns the number of children nodes.
  - I.e. `numChildrenNodes()` executed on the root node from Figure 1 would return 2.
  - Internally, will make a call to the method *size()* of the hash table object.

4 Deliverables

As part of the assessment of the lab practice 3, the following exercises must be done:

1. Full implementation of the interface `Trie<E>`.

These exercises will be bundled into a single compressed file (`.zip`, `.tgz`, etc.) containing the source code of the project, and delivered via a poliformaT task created for this purpose. All `.java` files should be encapsulated within the same package called `P2` (`package P3;`).