420-406-AB Data Structures
Winter 2019

May 8th, 2019
Worth 11%

Assignment VII: Tries

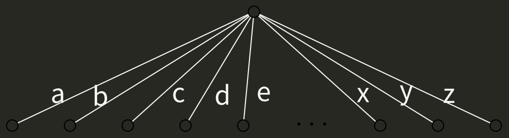
(Due on On Léa)

Overview

In this assignment, we will store a collection of words, called a *lexicon*, in a structure called a *trie*[1]. We will use the lexicon to test if some input string is in the lexicon (Section 4).

1 Representing a trie

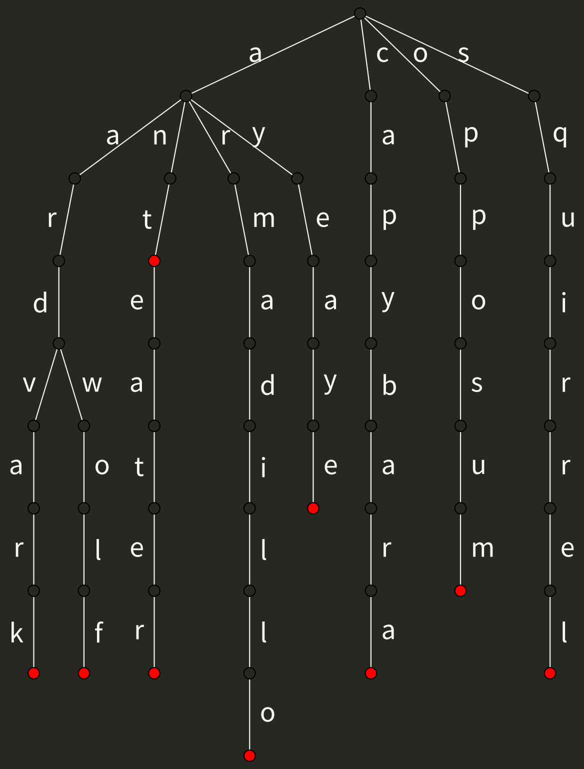
A trie is a tree where each node has 0-26 children labelled with letters of the alphabet (lower-case alphabet characters only).



Starting from the root prefix (the empty string “”), the level 1 subtrees represent the words of the lexicon that start with `a`, `b`, etc…. The subtrees at level 2 represent the words that start with the prefixes `aa`, `ab`, `ac`, etc…. It is possible that no words exist for a particular sequence (ex: `zz`) so that subtree is empty.

[1]pronounced “try”

For example, the following is a trie for the lexicon of interesting animals:



Notice that the nodes corresponding to lexicon words are marked as red. Not all internal trie nodes are words, many are just prefixes to words. All leaf nodes are words.

Create a class `Trie` that implements the interface `Lexicon`. It will use the trie nodes described above. The important part of your node design is correctly storing the child nodes to be able to access them by alphabet characters.

```
public interface Lexicon {

    /**
     * Add a word to the lexicon.
     * @param word the word to add to the lexicon.
     */
    void add(String word);


    /**
     * Test if a word is in the lexicon.
     * @param word the word to check.
     * @return true if the word is in the lexicon, false otherwise.
     */
    boolean contains(String word);
}
```

2 n-ary Trees

The trie you are implementing is an example of an “n-ary” tree, which means that each node can have up to n children. To store an n-ary tree in memory you will need to think of how to design the `Node` class. Hint: use a *collection* to store the child references!

3 Starter

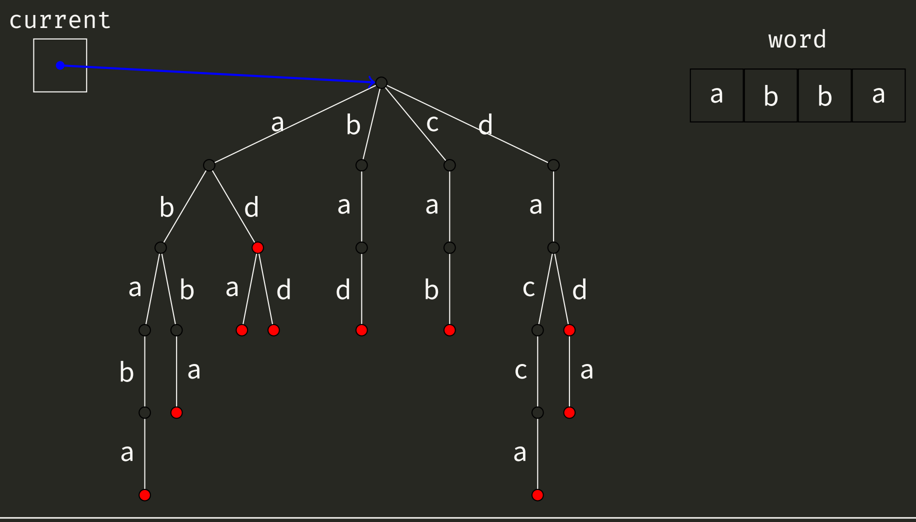
I’ve given you some English lexicon files, one for the complete alphabet (a-z) and another for a limited alphabet (a-e). I’ve also included a `Main` that you can use as a starting point.

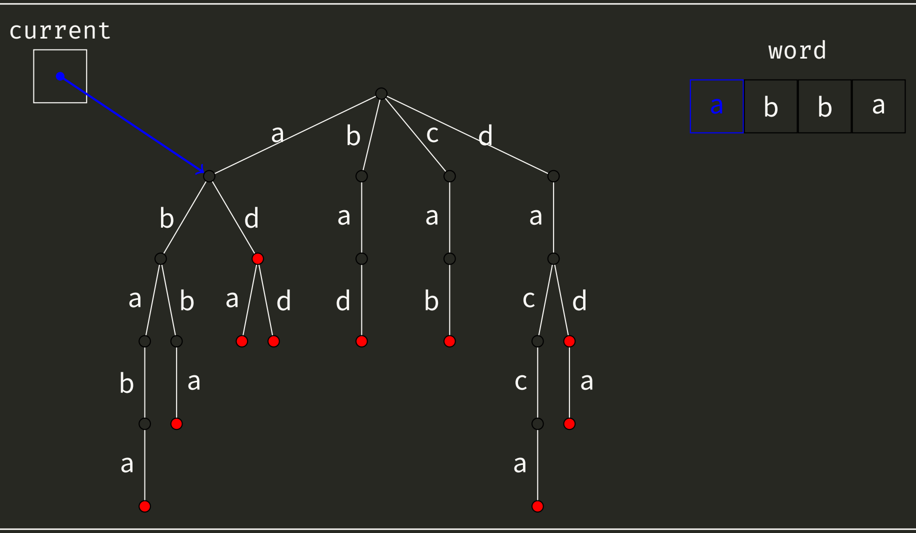
4 Contains

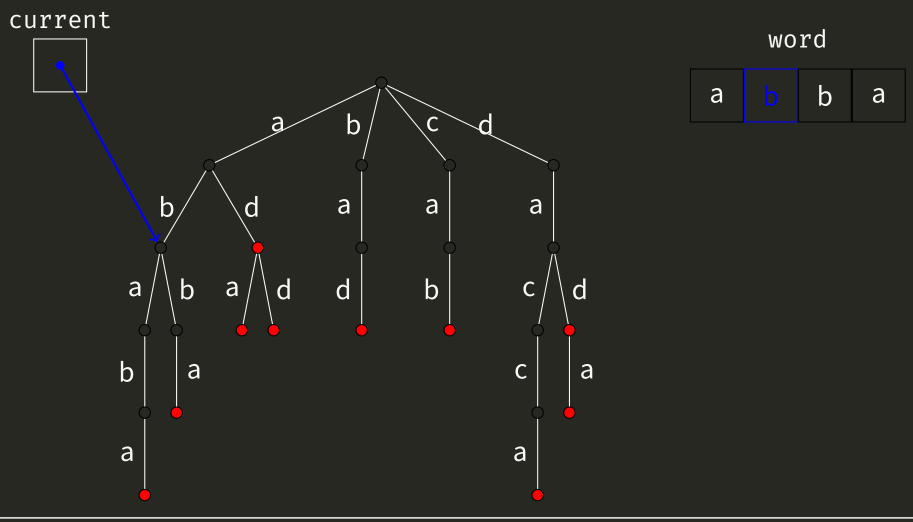
We will now see how useful a trie is in determining if a word is part of a lexicon. Using a trie, it is easy to determine if an input string is a word in the lexicon.

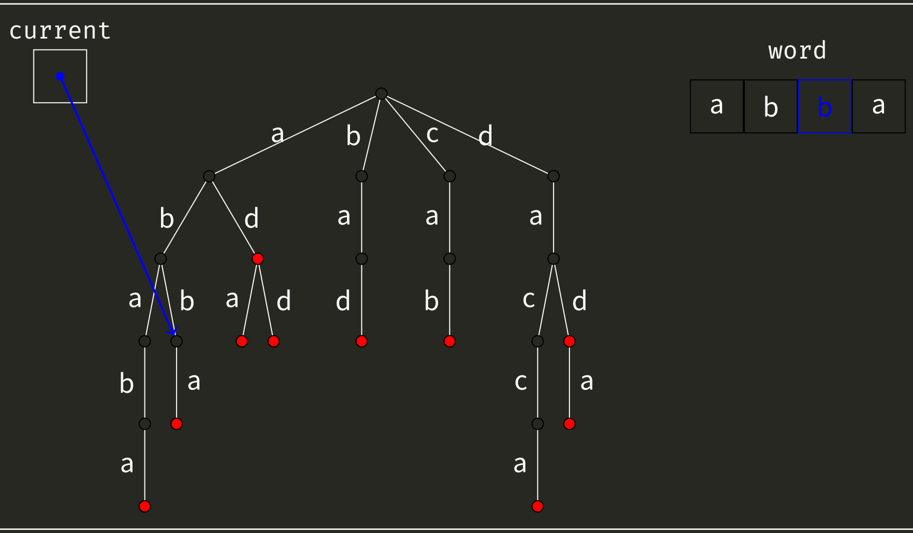
1. Start at the root of the trie.
2. If there are no more letters in the input word, then check that the current node is marked as a lexicon word and stop.
3. Read the next letter of the word.
4. If the current node has a child corresponding to that letter, move to that node and repeat the process from step 2. If there is no child node for this letter, the input word is not a lexicon word.

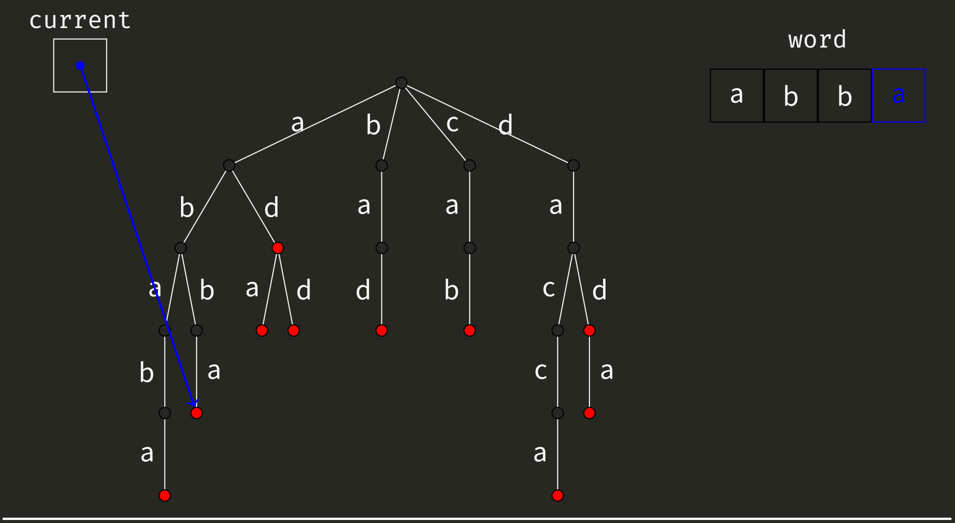
Successful search for “abba”:


current
word
a
b
b
a
a
b
c
d
b
d
a
a
a
a
b
a
d
d
b
c
d
b
a
c
a
a
a

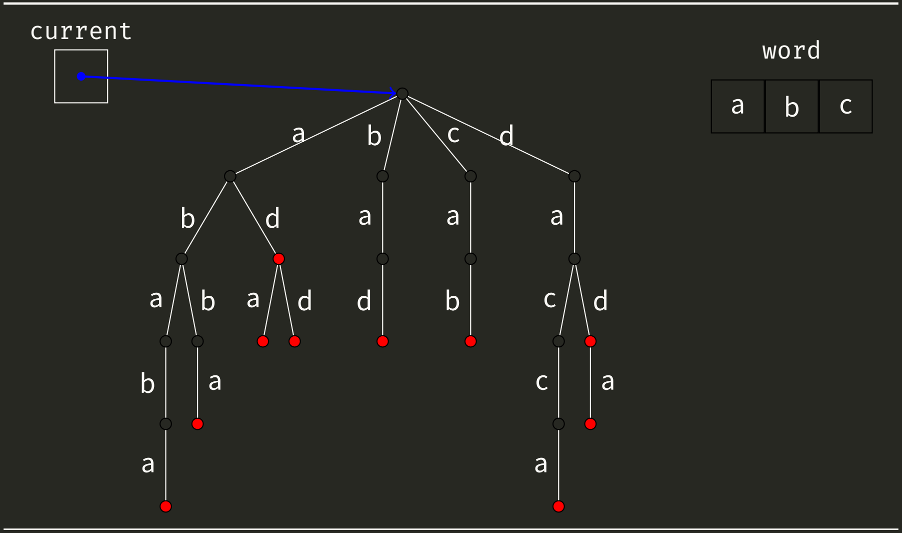

current
word
a
b
b
a
a
b
c
d
b
d
a
a
a
a
b
a
d
d
b
c
d
b
a
c
a
a
a

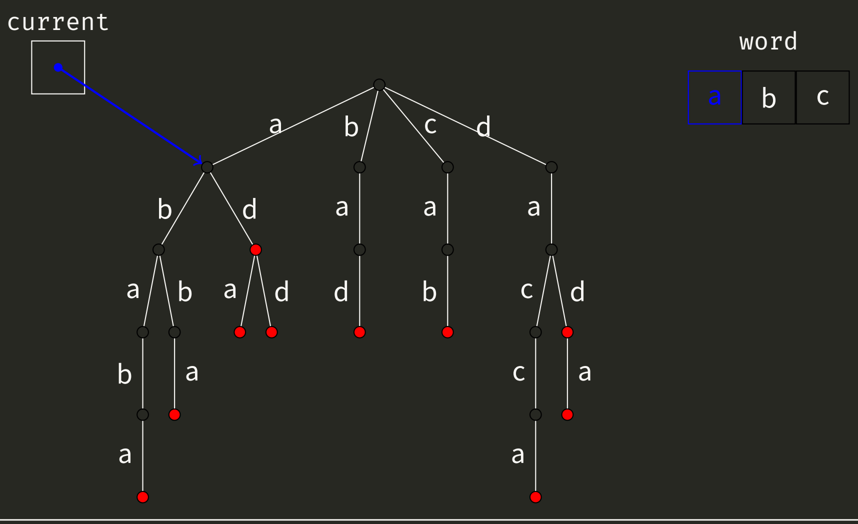

current
word
a
b
b
a
a
b
c
d
b
d
a
a
a
a
b
a
d
d
b
c
d
b
a
c
a
a
a

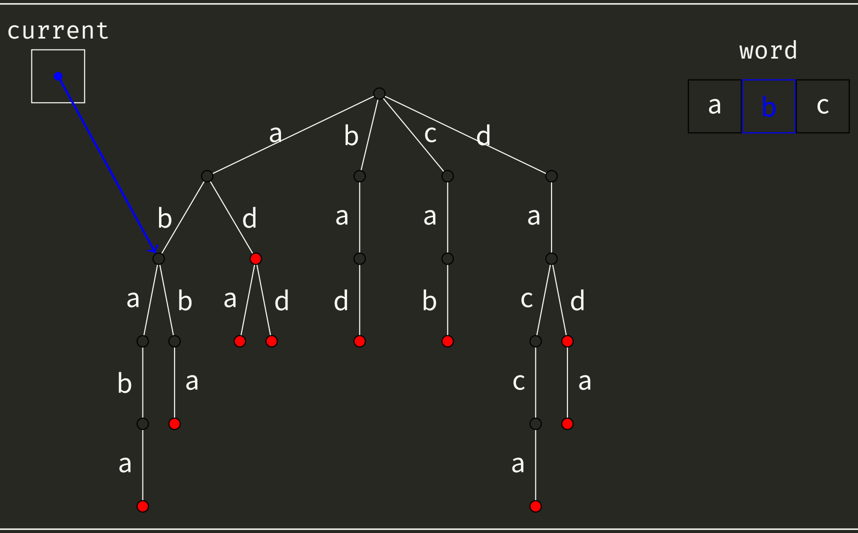

current
word
a
b
b
a
a
b
c
d
b
d
a
a
a
a
b
a
d
d
b
c
d
b
a
c
a
a
a

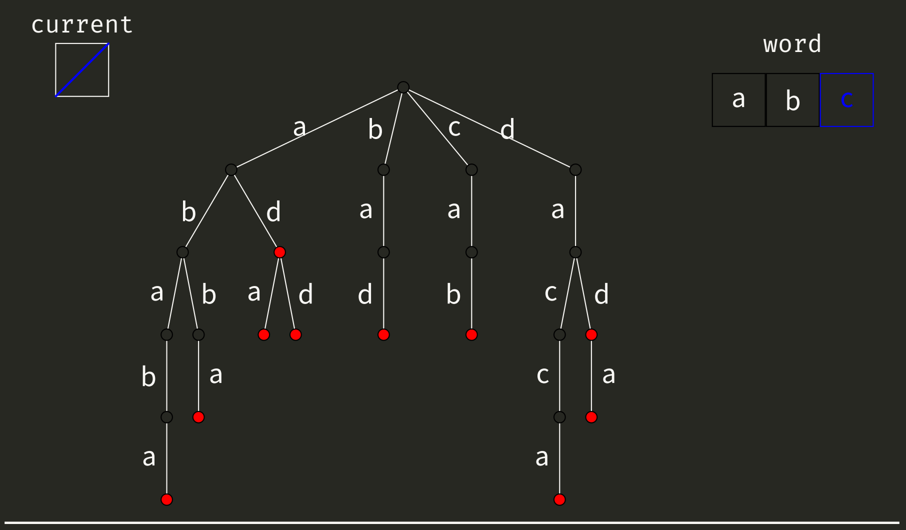


Failed search for “abc”:




current
word
a
b
c
a
b
c
d
b
d
a
a
a
a
b
a
d
d
b
c
d
b
a
c
a
a
a


current
word
a
b
c
a
b
c
d
b
d
a
a
a
a
b
a
d
d
b
c
d
b
a
c
a
a
a



5 Requirements

Your program *must* meet the following requirements:

1. Your program should be clear and well commented. It must follow the "420-406 Style Guidelines" (on Léa).
2. Your class `Trie` must implement the `Lexicon` interface.
3. Implement `add` and `contains` *recursively*.
4. Submit using Git by following the instructions (on Léa).