Practical 8: Tries

A trie is a tree-like data structure whose nodes store the letters of an alphabet. By structuring the nodes in a particular way, words and strings can be retrieved from the structure by traversing down a branch path of the tree.Tries can be extremely useful in searching for a word in a dictionary of words. It is important to note that a trie will only work if there are no words that are prefixes of another word in the input set.

What am I doing today?

Today's practical focuses on:

1. Implementing a Trie by hand
2. Implementing a Trie in code

Trie Development

Part 1

Let's start by creating a Trie by hand.

Build a trie by hand:

Let's take an example set of strings: S = { bank, book, bar, bring, film, filter, simple, silt, silver}

So work out a trie by hand for this set of words.
You can consult the lecture videos to help you.

Steps:

1. Start at the root note which has no character associated with it
2. Begin with the first word in the set and add child nodes to the root until you reach the last character in the set - adding the leaf node
3. Work your way through all the words in the set adding characters as needed, taking advantage of shared prefixes where possible.

Step 2 Code a Simple Trie in Java

A Trie is a tree in which each node has many children. The value at each node consists of 2 things: 1) A character 2) A boolean to say whether this character represents the end of a word.

Take the starter code (below or from the master repo) and implement a basic Trie structure that does the following:

1. Implements a static method that can insert new keys (as nodes and leaf nodes) and take advantage of existing prefixes in the trie
2. Implements a static boolean method that can search the Trie for a key and return true or false

Inserting Elements

The first operation that we'll describe is the insertion of new nodes.

1. Set a current node as a root node
2. Set the current letter as the first letter of the word
3. If the current node has already an existing reference to the current letter (through one of the elements in the "children" field), then set the current node to that referenced node. Otherwise, create a new node, set the letter equal to the current letter, and also initialize current node to this new node

4. Repeat step 3 until the key is traversed

Searching Elements

Let's now add a method to check whether a particular element is already present in a trie:

1. Get children of the root
2. Iterate through each character of the *String*
3. Check whether that character is already a part of a sub-trie. If it isn't present anywhere in the trie, then stop the search and return *false*
4. Repeat the second and the third step until there isn't any character left in the *String.* If the end of the *String* is reached, return *true*

Sample starter code (but feel free to code your own)

```
import Trie.TrieNode;

public class Trie{

// Alphabet size (# of symbols) we pick 26 for English alphabet
static final int ALPHABET_SIZE = 26;


// class for Trie node
static class TrieNode {
TrieNode[] children = new TrieNode[ALPHABET_SIZE];
// isEndOfWord is true if the node represents end of a word i.e. leaf node
boolean isEndOfWord;

TrieNode(){
isEndOfWord = false;

for (int i = 0; i < ALPHABET_SIZE; i++)
children[i] = null;
}
}

static TrieNode root;
// If not key present, inserts into trie
// If the key is prefix of Trie node,
//  marks leaf node
static void insert(String key){



}
```

```
// Returns true if key presents in trie, else false
static boolean search(String key) {


}


// Driver
public static void main(String args[]) {

// Input keys (use only 'a' through 'z' and lower case)
String keys[] = {"bank", "book", "bar", "bring", "film", "filter", "simple", "silt",
"silver"};


String output[] = {"Not present in trie", "Present in trie"};

root = new TrieNode();

// Construct trie
int i;
for (i = 0; i < keys.length ; i++) {
insert(keys[i]);
}

// Search for different keys


}

//end of class
}
```