CSCI 136: Fall 2017
Handout L8b
29 October

## 1 Super Lexicon Classes

- `Main.java` - driver program. No need to modify!
- `Lexicon.java` - interface. No need to modify!
- `LexiconTrie.java` - implements interface. You will make changes here.
- `LexiconNode.java` - nodes in Trie. You will make changes here. Methods here do **not** need to be recursive! Also, just use a Vector for children, and when you add, add nodes in order. You don't have to use an OrderedStructure, although an OrderedStructure would also work. **Important:** If you choose to use an OrderedStructure to hold children, you must define an equals() method for LexiconNode!!!

## 2 Miscellaneous Notes

- Unlike our BT implementation, our LexiconTrie consists of LexiconNodes. So in some ways, this is similar to our SLL implementation.
- Only matchRegexp and suggestCorrections **have** to be recursive. Other methods can be done recursively, but you will not be penalized in any way if you choose to implement them iteratively. Do not make the other methods overly complicated! Test your code frequently!

## 3 Suggested approach

1. Start with Setion 2.3 on handout and implement LexiconNode. Pick a data structure (Vector is strongly recommended!!) that will store the children of the node. Work through the methods in LexiconNode. Note that to compare chars, you can just subtract one character from another.
2. After completing LexiconNode.java, move on to LexiconTrie.java. You'll need to add a constructor. The constructor should create a single LexionNode that has the character assigned to be ' ' (just a blank space).
3. Section 2.4 describes constainsWord and containsPrefix. The technique used in both of these methods is basically the same. containsWord performs one additional test before returning to see if the isWord flag is set to be true. You may want to create a helper method called "find(String word)" that returns null or a LexiconNode to be helpful here. Note that you can implement this method with or without recursion! Either way is acceptable.
4. Move on to addWord and addWordsFromFile. addWordsFromFile will use a Scanner to parse an input file (line by line, with a single word per line) and call addWord for each line. Be sure to update size. Convert everything to lowercase, too.
5. removeWord may be implemented recursively or iteratively. If you choose to do it recursively, you may want to use a helper method. Either way, be sure to return true if the word appeared in the lexicon and was removed, and false otherwise. This method is tricky, so think before you type!
6. For the iterator (section 2.7 on handout), create a helper method that recursively builds a Vector of words. Keep in mind that the LexiconNodes already maintain a list of their children in sorted order. That will help you iterate over the trie in alphabetical order easily.
7. Section 3 describes optional extensions for the lab. In these sections, you implement two recursive methods for manipulating the trie. You may create helper methods as needed for both of these methods. Think about printSubsetSum and countSubsetSum from Lab 3 for inspiration.