CS245-02/CS545, Spring 18
Prof. Karpenko

Project 2: Dictionary / Spell Checker[1].

Due: 03/09/2018, 11:59pm

Overview:
For this project, you will implement a Dictionary ADT with extra functionality of being able to find the closest entries in the dictionary to the given invalid word. You will implement such a dictionary using a *compact prefix tree* (see below). In the starter code, `Dictionary` is an interface that has the following methods:

- `public void add(String word);` Adds a given word to the dictionary.
- `public boolean check(String word);` Checks if a given word is in the dictionary.
- `public boolean checkPrefix(String prefix);` Checks if the given prefix is a prefix of any word in the dictionary.
- `public void print();` Prints all the words in the dictionary in alphabetical order.
- `public String [] suggest(String word, int numSuggestions);` If the given word is in not in the dictionary, returns an array of the closest entries in the dictionary.

We discuss each of these methods in the Implementation section of this document.

You need to fill in the code in the class called `CompactPrefixTree` that implements the `Dictionary` interface and stores words in a **compact prefix tree**. **In addition** to methods of the Dictionary interface, `CompactPrefixTree` should have the following methods:

- `CompactPrefixTree() The` default constructor. Creates an empty compact prefix tree.
- `CompactPrefixTree(String filename).` The constructor that creates a compact prefix tree from the words in the given file.
- `void printTree().` Prints out a human-readable representation of the tree to the console (using a different level of indentation at each level).
- `void printTree(String filename).` Prints out a human-readable representation of the tree to the given file (using a different level of indentation at each level).

You will also need to write some helper methods to implement the above methods (see Implementation section).

If you only needed to write the constructor, add, and check, then the best data structure to use would be a hash table (we will cover hash tables later in the semester). However, getting suggest to work correctly using a hash table would be quite difficult. Binary search may be a little better (since we can relatively easily find words that are close together), but operations on binary search trees take time O(lg n) if you're lucky -- and can be as bad as O(n) if you're not lucky. If keys are entered in sorted order (as the likely would be for a dictionary), then you will get the worst-case performance for a binary search tree.

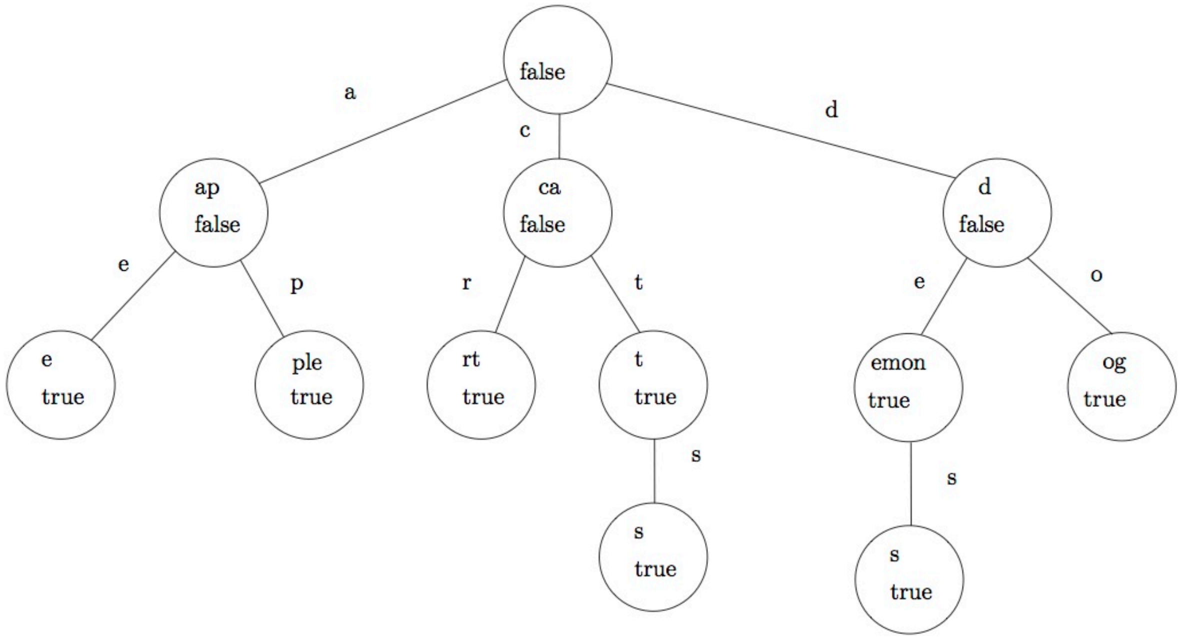
[1] The project is courtesy of Prof. Galles.

Instead of a hash table or a binary search tree, you will use **a compact prefix tree**: a tree where each node has:

- a string *prefix*
- up to 26 children (one for each letter of the alphabet),
- a valid bit,

If you start at a the root, and follow a path to any node *whose valid bit is true*, and *concatenate all of the prefixes together*, you will get a word stored in the dictionary.

For example, consider the following tree:



This tree stores the words **ape, apple, cart, cat, cats, demon, demons, dog.**

Example: in order to get the word "apple" we need to start at the root of the tree (that stores an empty string) and concatenate prefixes "", "ap", and "ple". The word "ap" is not in the dictionary, since the bit stored at that node is false.

This dictionary structure has three main advantages:

- The methods `add` and `check` can be performed in constant time (with respect to the number of words in the dictionary -- both these operations are linear in the length of the word being added or checked)

- `Suggest` can be done (relatively) easily

- Determining if a string *is a valid prefix* of some word can be done in constant time (relative to the size of the dictionary)

Implementation Details

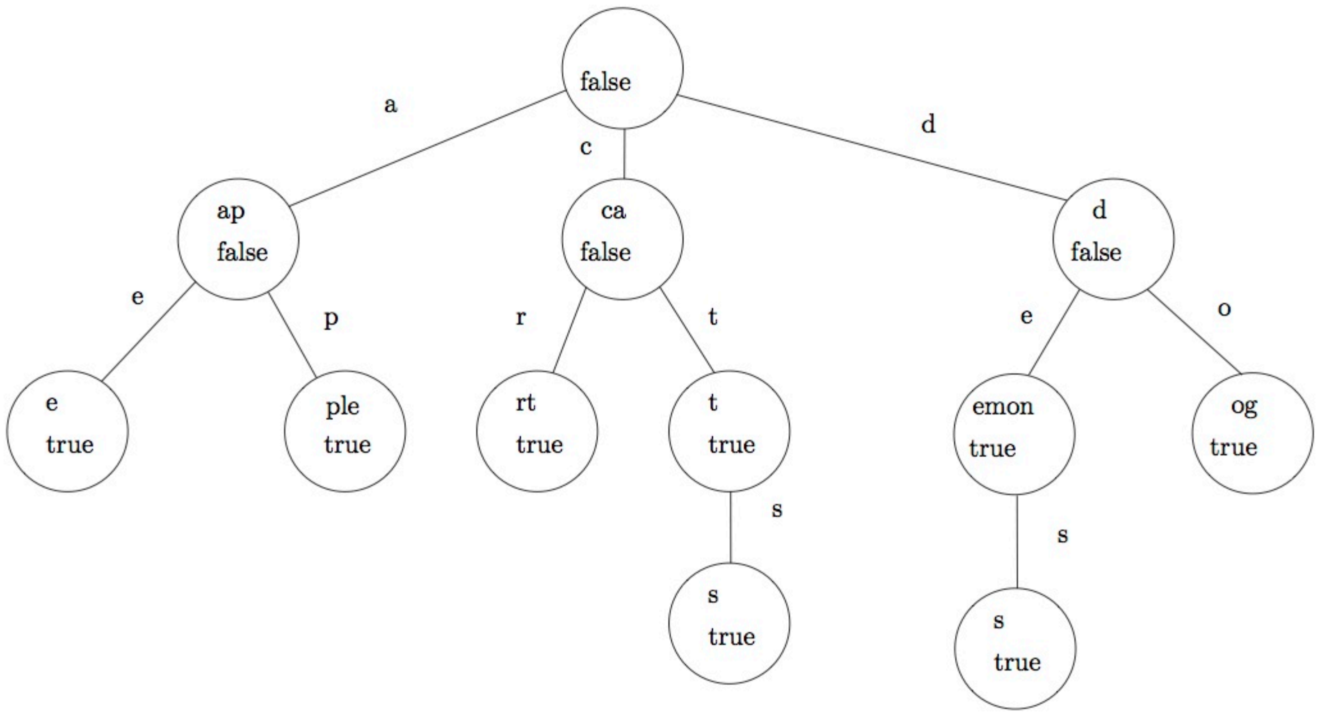
Methods `add`, `check`, `checkPrefix`, `suggest` and `print` need to be implemented **recursively.** No credit will be given for non-recursive implementations. You will also need to write recursive helper methods that take as one input parameter a tree, and your required methods should call these helper methods passing in the root of the tree - just like we did for the binary search trees (see under Examples on github). So your `add` method will likely look like the following:

```
void add(String word) {
    root = add(word, root);
}
```

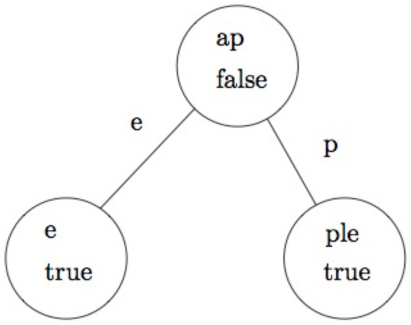
And you will have another `add` method that takes a String and a Node. When you have two methods with the same name but different parameters, it is called `overloading` in Java. The same applies to other methods of Dictionary - you will have two methods with the name `check`, two methods with the name `checkPrefix` and so on.

Finding a word (check method)

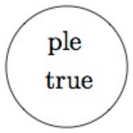
Consider finding a word in a tree. The word "apple" is in the following tree:



If and only if the word "apple" is in the "a"-subtree of the root (because "apple" starts with an "a"):



If and only if the word "ple" is in the "p"-subtree of the node '"ap" (because "ple" starts with a "p"):



Which is in the tree! We know that because "ple" is stored in the node, and the bit is set to true.

Let's look a little closer at how to find a word in the tree (and hence how you would implement a `check` method, recursively):

Base Cases:

- If the tree is empty, then the word is not in the tree
- If the prefix stored at the root of the tree is *not* a prefix of the word, the word is not in the tree
- If the word we are looking for is the same as the prefix stored in the root of the tree, and the valid bit for the root is set to false, then the word is not in the tree. (This base case is not strictly necessary, since it can be covered by the recursive case and the empty tree base case.)
- If the string we are looking for is the same as the prefix stored in the root of the tree *and* the valid bit for the root is set to true, then the word is in the tree

Recursive Case:

If none of the base cases hold (that is, the prefix stored at the root of the tree is a proper prefix of the word we are looking for) then:

- Let **suffix** be the portion of the word that is not part of the prefix stored at the root. So, if the word we are looking for is "green", and the prefix stored at the root of the tree is "gre", then **suffix** would be "en"

- The word is in the tree if and only if **suffix** in the tree corresponding to the child labeled with the first letter of **suffix**. In the above example, if we are looking for "green" in a root that contains the prefix "gre", then "green" is in this tree if and only if "en" is in the 'e' subtree of this tree.

Adding a word

Adding a word to the dictionary is much like finding a word -- you should still recursively go down the tree (just like in find) until you get to one of the base cases. What to do next depends upon which base case you reach:

- An empty tree: Create a new node whose prefix is the word you are looking for, and whose valid bit is true. Return this node.
- A node whose prefix is the same as the word you are looking for, with the valid bit set to false. Set this bit to true, and return the tree.
- A node whose prefix is the same as the word you are looking for, with the valid bit set to true. The word is already in the tree! Return the tree unchanged.

- A node whose prefix is not the prefix of the word you are looking for. This is the hard case. Example: if you were inserting "hamster" into a node whose prefix was "hamburger". You need to:

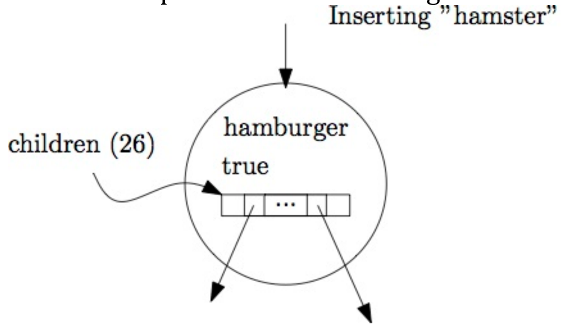
    - Create a new node. The prefix stored in this node is **the longest common prefix** of the word you are inserting and the prefix stored at the original root. Thus, if you were inserting "hamster" into a node whose prefix was "hamburger", then the prefix of this new node would be "ham"

    - Let **suffix** and **suffixWord** be the suffix of the original prefix and the suffix of the word you are adding, after extracting the common prefix. So, if you are inserting hamster into a node whose prefix was hamburger, then **suffixWord** would be "ster" and **suffix** would be "burger".

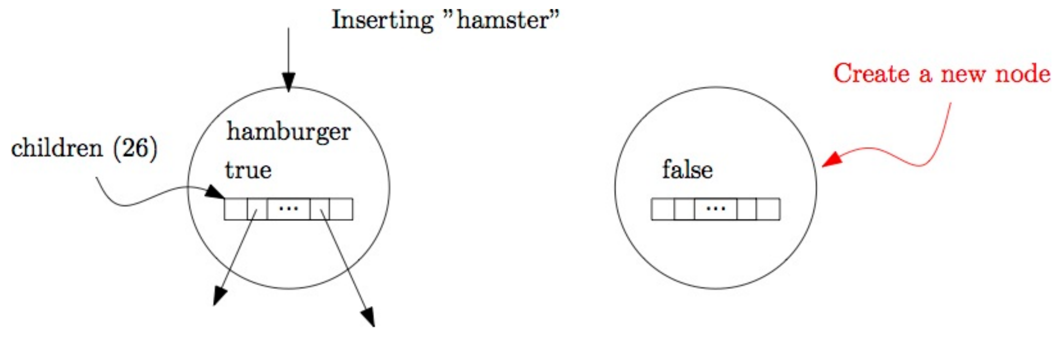
    - Set the prefix of the original tree to **suffix**, and set the child of the new node corresponding to the first letter of **suffix** to the original tree

    - Recursively insert **suffixWord** into the appropriate child of the new node
    - Return the new node

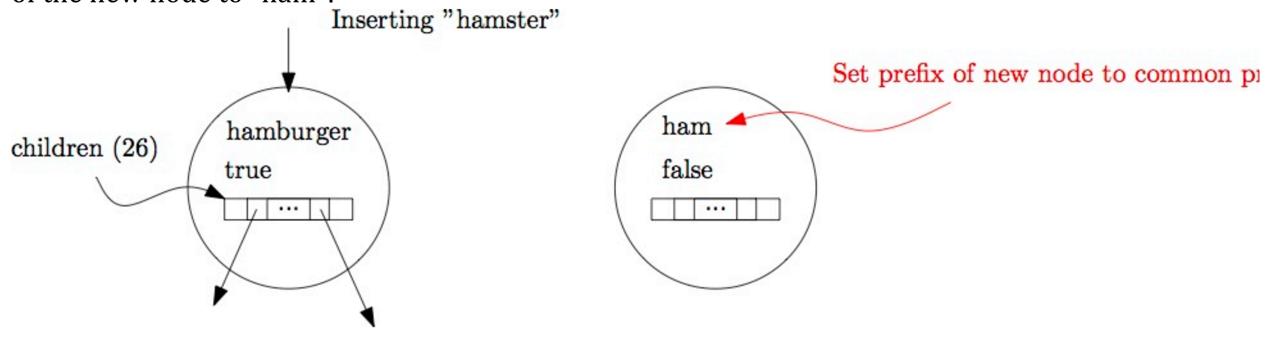
Let's see an example: If we are inserting hamster into a root that contains hamburger:



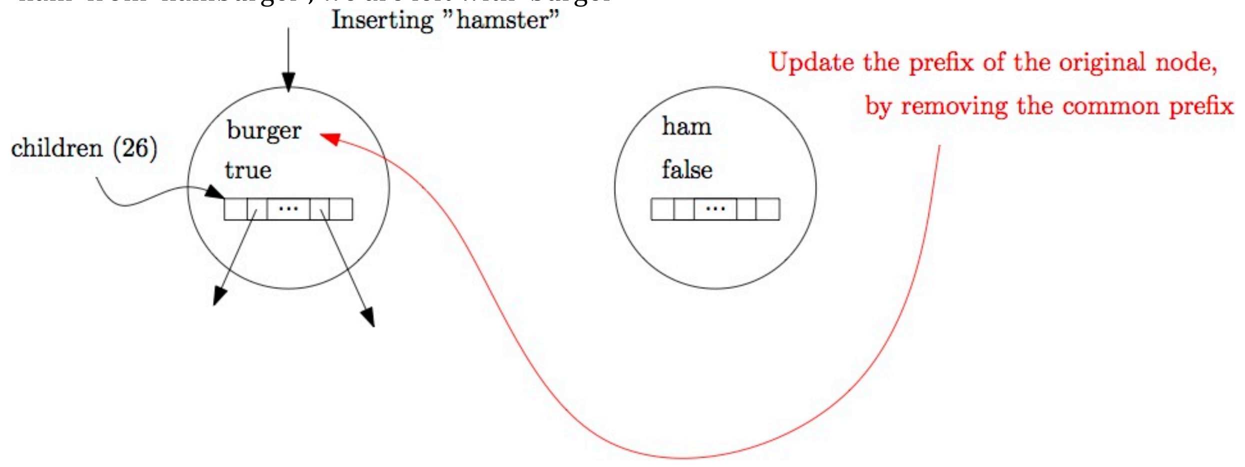
We note that 'hamburger' is not a prefix of 'hamster'. So, we create a new node (be sure that the 'valid' bit of this new node is false!):



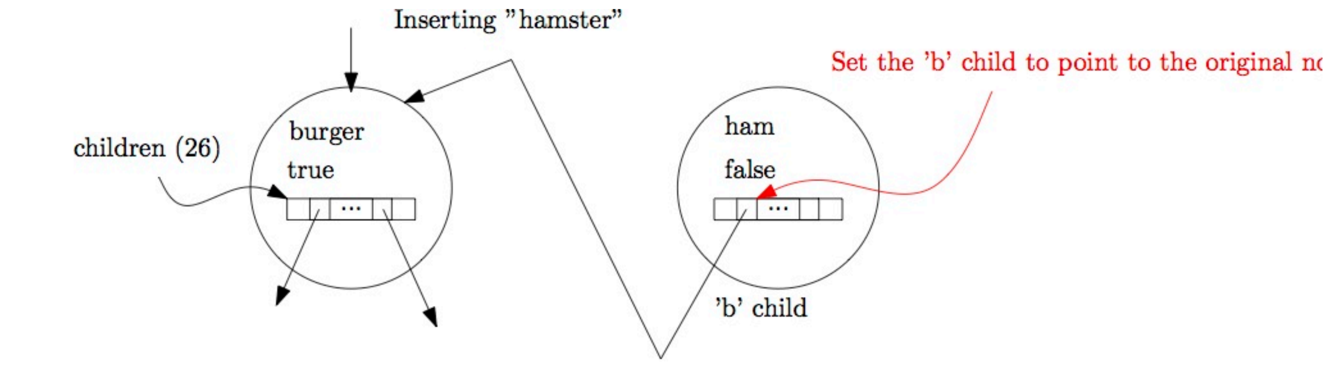
The longest common prefix of "hamburger" and "hamster" is "ham". We set the prefix of the new node to "ham".



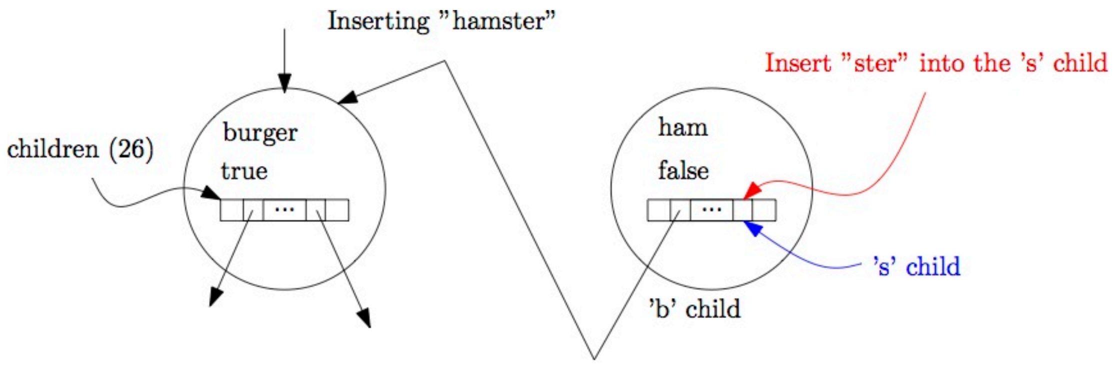
We now update the prefix of our original node. After removing the common prefix of 'ham' from 'hamburger', we are left with 'burger'


Inserting "hamster"
Update the prefix of the original node,
by removing the common prefix
children (26)
burger
true
...
ham
false
...

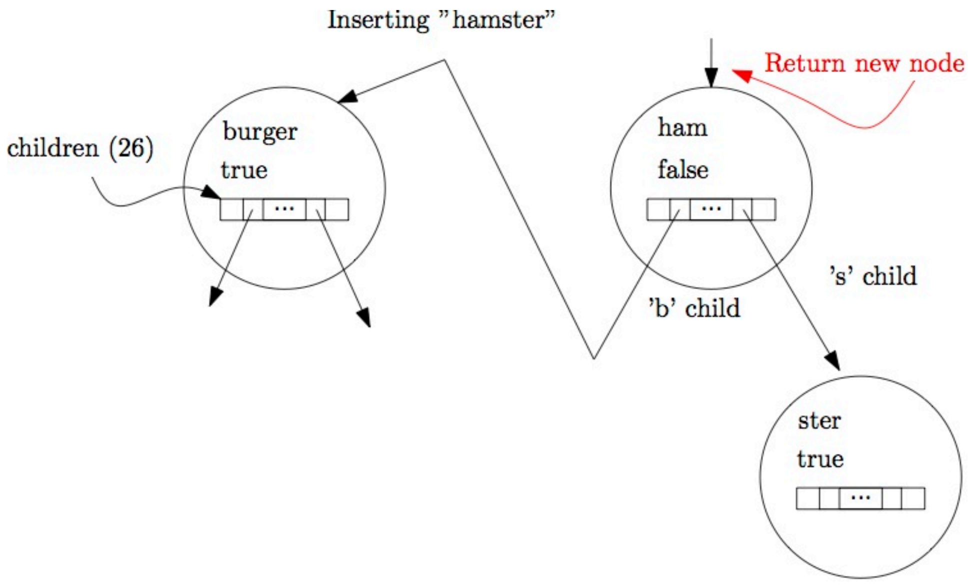
The first letter of the updated prefix is 'b', so we set the 'b' child of our new node to point to our original tree that has a "burger".



After removing the common prefix 'ham' from 'hamster', we are left with 'ster'. Recursively insert 'ster' into the 's' child of the new node (note: this will immediately hit a base case, which will create a new node)



Finally, return a pointer to the new node. We're done!



Suggest

You have some flexibility as to how to implement suggest. The easiest way to implement suggest is to return entries with the same prefix as the word passed in, trying to make the common prefix as long as possible. There are a number of other ways to implement suggest -- missing letters, added letters, transposed letters, near misses on the keyboard, phonetic equivalents, and the like. Make your suggest as good as you can!

Printing all the words in the dictionary

To print out all the words in your dictionary (recursively), you may want to write a private helper method `print(Node node, String soFar)` that takes two parameters: the root of the tree to print, and *the word built up so far*. When you first call the print function, you would pass in the root of the tree and an empty string (""). In the method, if the valid bit of the node is *true*, you should print the concatenation of the word built so far with the prefix stored at the node. You should also iterate over the children of the node, and call the print method recursively on each child with correct parameters.

Printing out the structure of the tree

There are two `printTree` methods that you need to write: one that outputs the tree to the console, and another that outputs the tree to the file. Both should output a tree in a human readable form that uses indentations. For example, for a tree that contains: **car, carted, carts, doge, doges and doghouse**, your `printTree` should print out (the first line contains the prefix of the root which is an empty string):

```

car*
  t
    ed*
    s*
dog
  e*
    s*
  house*
```

I recommend creating a *private helper method toString()* that returns the string representation of the tree. In this private method, do a preorder traversal of the tree, keeping track of the current indentation level -- just like we did for printing out the structure of a binary tree. You should denote which nodes have a valid bit set to true by concatenating an asterisk ("*") after the prefix of such nodes.

Indexing Child Array

The easiest way to have a child for each letter of the alphabet is to have each tree node store an array of 26 pointers: index 0 represents the 'a' child, index 1 represents the 'b' child, and so on. We will thus need to convert from a letter between 'a' and 'z' to a number between 0 and 25. If we cast a character to an integer, we will get the ASCII code for the character. To get this number in the range 0 - 25, we just need to subtract the ASCII code for 'a', as follows:

```
char ch = 'e';
int index = (int) ch - (int) 'a';
```

Submission

Submit project2 code to your private github repository created for you automatically by Github Classroom when you click on the assignment link on github. The project must be submitted to this github repository by the deadline. No credit for late projects.

Testing

You are responsible for testing your code. The instructor will provide a test file that tests some of the methods of the project, but you are expected to perform additional testing on your own.

.