CS201 Discussion 10

DRAWTREE + TRIES
Problem Statement

Given a tree, you are to return a representation of that tree as a String[]. The tree will be specified by an int[], where element $i$ gives the index of the parent of node $i$ (indexed from 0). Exactly one element of this int[] will be -1, corresponding to the root of the tree. Element $i$ of the int[] corresponds to element $i$ of names, which gives the name of node $i$.

The String[] you return should have the root on the first line. Following the root, add elements to the return one subtree at a time. The subtrees should be sorted by the indices of their roots and drawn in the same way as the whole tree. However, the root of each subtree should be indented two more characters than its parent. After indenting, each node should be indicated by a plus ‘+’ or a minus ‘-’, and then the name of the node from names. The plusses of siblings should be connected by vertical pipes ‘|’ as shown in the examples. For example, the tree given by the input:

```
parents = {-1,0,1,1,0,0,5,5}
names = {"Root","SubB","LEAF1","LEAF2","LEAF3","SUBA","LEAF4","LEAF5"}
```

would result in the following return.

```
+-Root
  +-SubB
   |  +-LEAF1
   |  +-LEAF2
   |     +-LEAF3
   |     +-SubA
   |        +-LEAF4
   |               +-LEAF5
```
First instinct: recursion

As a very generic structure, we could tackle this problem as follows:

draw():

    Find the root

draw(root)

draw(root):

    Write the line for the root

    For each child element of root:
        draw(child)
Recursive arguments

First, we should either keep the parents and names arrays as global variables, or pass them to our recursive method so it can access them. However, this is not enough.

Looking at the Strings in each subtree – what do all the Strings in any subtree have in common?
Recursive arguments

All elements (besides the root) in any subtree have Strings starting with the same substring.

Thus, we should pass this “prefix” as one of our arguments. The root will also start with prefix, but minus the last two characters.
Updated pseudo-code

draw():
    Find the root
    draw(root, """) //note the initial value

draw(root, prefix):
    Write the line for the root, using prefix minus last two chars
    For each child element of root:
        draw(child, updated prefix)
Updating the prefix

How does the prefix change for each child?
Updating the prefix

How does the prefix change for each child?

For every child but the last, we append "| " to the prefix. For the last child, we add " ".
Updated pseudocode

```
draw():
    Find the root
    draw(root, "  ")

draw(root, prefix):
    Write the line for the root, using prefix minus last two chars
    For each child element of root but the last:
        draw(child, prefix + "| ")
    For the last child (if there is one):
        draw(child, prefix + " "
```
Implementation details

Probably the easiest way to handle writing the line is to keep a global ArrayList. Where the pseudocode says “write the line”, just add the String to the ArrayList.

In addition, it may be easiest to pass the index of the root value, rather than the value itself. Then, to find all children of the j\textsuperscript{th} value, we just find all indices such that parents[i] == j.
Final pseudocode

draw(...):
    Find the root’s index, i
    Initialize empty ArrayList L globally. Also keep parents and names globally.
    draw(i, "  ")

draw(ind, prefix):
    Add to L (prefix minus last two characters plus “+-” plus names[ind])
    Children = all indices j such that parents[j] == ind
    For each element of children but the last, call draw(child, prefix+“| “)
    For the last child (if there one), call draw(child, prefix + “ “)
A trie is a data structure used to store words. It’s like a tree, except instead of each node having a left and right child, it can have a child for each character of the alphabet.

Each node represents a String, which is made of the characters on the path to it from the root.

- e.g. the bottom left node represents the word “as”
Storing words

In order to keep track of which nodes represent Strings that are actually words, each node will have a boolean marker. If the boolean is true, the String that node represents is a word.

◦ e.g if the red nodes are those with the marker set to true, this trie has the words “a”, “as”, “at”, and “be”.
◦ Note that there is a node representing the String “b”, but it is not a word.
Prefix property

Tries have the useful property that if given a prefix, all words which start with the prefix are in the trie rooted at the node representing that prefix.

- e.g. the trie rooted at the node outlined blue contains all the words in the trie starting with the letter a

For this reason, tries are sometimes called “prefix trees”

In this next assignment, Autocomplete, this property will prove very useful...
Checking for words

To check if a word is in the trie, start from the root.

For each character in the word, see if there is a child corresponding to the character. If there is, move to it. If not, you know the word doesn’t exist.

After this, you’ve arrived at the node representing the word. Then, simply check its boolean marker.
Checking for words

The blue arrows show the result of checking for “as”. We successfully traverse to the node representing “as”, and its marker is true, so we know “as” is a word in our trie.
Checking for words

The blue arrows show the result of checking for “beast”. The node for “be” has no child for the letter A, so we know “beast” isn’t a word since a node representing it does not exist.

(You can equivalently think of this as running off of the trie when we move to the A child since it doesn’t exist)
Adding words

The process for adding words is very similar to the process for seeing if they exist in the trie: We start at the root, and then for each letter of the word, we move on to the child corresponding to that letter.

The difference is, when a child we’re trying to move to does not exist, we simply create that child before moving to it.

Then, we set the last node’s boolean marker to true.
Adding words

For example, suppose we’re adding the word “bat” to this trie. We set a pointer current and start it at the root. We’ll use the blue outline to represent the node current is pointing to.
Adding words

Our first letter is “b”, so we move our pointer to the child corresponding to b.
Adding words

Our next character is “a”. Our current node has no child for “a”, so we create a new node and have the “b” node point to it.
Adding words

Our next character is “a”. Our current node has no child for “a”, so we create a new node and have the “b” node point to it.

Then, we move the pointer down to it.
Adding words

We repeat this process for the character “t” – since we don’t have a “t” child, we create a new node and set that as the “t” child, and then move our pointer to that child.
Adding words

We’ve finished moving to the node representing our target word at this point.

Now, all that’s left to do is set the current node’s boolean marker to true. Then, we’ve successfully added the word “bat”.

![Diagram showing the addition of the word “bat” to the node structure.](image)
Add: Pseudocode

current = root
for each character in the input word:
    if current doesn’t have a child for that character:
        create a node and set it as current’s child
        current = current’s child for that character
    set current’s marker to true
Actual Java implementation

To actually represent the nodes in the Trie, we’ll use an implementation like this:

class TrieNode{
    boolean isWord;
    HashMap<Character, TrieNode> children;
    //constructors
}

If we have a node current and we want to access the child of current corresponding to the character ch we can do so using current.children.get(ch)

Similarly, to add a new child to current, we can use current.children.put(ch, new TrieNode())
Discussion activity: TrieExample

Snarf the code for today, and complete the class TrieExample.

The add method is partially written for you. Complete this first before moving on to other methods, as the tester needs it to work.

Remember, Maps come with a ton of useful built in methods like containsKey, keySet, and values.

For firstWord, think of the most efficient way to complete the method. You could just find every word, sort them, and return the first one, but this is inefficient and will time out the tester.