CSC 325 Adv Data Structures

Program #3

Tries

Your objective is to implement the main operations of a Trie data structure from the lecture. To be clear, this entails creating a class for your nodes, then creating another class for your Trie.

Create a class called **`Node`** that has the following fields:

- Fields
    - children
        - This is an array of Node pointers covering all potential children this node could have. This should be private.
    - isTerminal
        - This is true if the node is the last character in a word stored in this Trie, false otherwise. Note that this may be set to true but still be a part of a bigger word. For example, if the words "ban" and "band" were in the Trie, the node for "n" would have isTerminal set to true because of "ban" but would be part of a bigger word in "band", so the node corresponding to "d" would also have isTerminal set to true. This should be private.
- Methods
    - Constructor
        - Allocates the children array to be the same size as the alphabet and initializes it to contain all null values.
        - It also initializes the isTerminal field.
    - Accessors
        - All private fields should be given an accessor
    - Mutators
        - All private fields should be given a mutator

Put this class into the same file as your Trie (described below).

Create a class called **`Trie`** that has the following:

- Fields
    - A pointer to the root node
    - A constant int that is set to our alphabet size
        - Set it to 128 since we only need to store ASCII values from 0 to 128
        - Refer to this constant in the constructor of the Node class above
- Public methods
    - A default constructor (i.e. a constructor that doesn't take any parameters). This will initialize `root` to null.

- `insert(word)`
    - Takes a String word to insert into our trie
- `search(word)`
    - Returns true if word is in the trie, false otherwise
- `printAllWords()`
    - Prints all words to the console with a comma separating the words
    - Print them on one line. If it word wraps in the terminal, that is fine.
    - Due to the nature of a trie, words should be printed in alphabetical order
- `printAllWords(prefix)`
    - Takes a prefix and prints all words that start with that prefix (including the prefix itself if it is a valid word in the trie)
    - Due to the nature of a trie, words should be printed in alphabetical order
- `Other methods as needed`
    - You may need other methods (for example an additional method to help with the recursion part of printAllWords).
    - Feel free to add additional helper methods as needed but do make sure they are marked as "private" since they will only be called from within the class itself.

Main part:
Make a separate public class called **TrieMain** and put it into the file with your Node and Trie classes. Put your entry point inside this Main class.

You will need to create two modes for this application. One will be for testing your trie to make sure it works properly. The other is for seeing the memory usage.

First, the user will select which mode they want to be in on the command line. Typing the following into the terminal (after TrieMain.java has been compiled that is):

```
java TrieMain test
```

should bring up test mode. To get this to work, you will need to use command line arguments. You know that "String[] args" part of the main method? This is there so that if you add data on the command line (aka terminal) when you run the program, that data can be passed into your program and grabbed in your Java code. So inside of your main function, you will need to examine the first value in args and see if it is the string "test". If it is, you should treat this as a request from the user to go into "test mode".

Test mode

When in test mode, you should create a new trie object and fill it with a few words. To help me in grading, I want you to fill the trie with the following words:

```
"banana", "bandana", "bandaid", "bandage", "letter", "lettuce", "let",
"tool", "toy", "toilet"
```

Do this in the code before showing the menu. Now, show the following menu to the user:

```
{add WORD}: Add a word
{printAll}: Print all words
{startsWith PREFIX}: Print words that start with
{search WORD}: Match a specific word
{quit}: Quit

?
```

This is different from the menu in the last assignment. Instead of just typing a number, the user will type a word (or words). If they type `add hotdog` your application should add the word hotdog to the trie. If they type `printAll` your application should print all the words currently in the trie (in alphabetical order, which should happen if you call your printAll function). If they type `startsWith ba` your application should print all the words that start with the prefix "ba". So, basically, the user will type one of the words from the menu (either add, printAll, startsWith, search, or quit) and then, if required, will type a word or prefix (with no quotes). The application will then perform that action. After preforming an action, the application should clear the screen and re-print the menu, waiting for another command from the user.

Note that since some of these actions require printing to the screen, for those actions we should pause and wait for the user to press enter after we have printed everything to the screen. So, for example, if I type `printAll` and press enter, the application should print all the words, then wait for me to press enter again before clearing and re-drawing the menu.

Example usage:

```
> java TrieMain test

* press enter *

{add WORD}: Add a word
{printAll}: Print all words
{startsWith PREFIX}: Print words that start with
{search WORD}: Match a specific word
{quit}: Quit

? printAll

* press enter *

{add WORD}: Add a word
{printAll}: Print all words
{startsWith PREFIX}: Print words that start with
{search WORD}: Match a specific word
{quit}: Quit

? printAll
All words: 'banana', 'bandage', 'bandaid', 'bandana', 'let', 'letter',
'lettuce', 'toilet', 'tool', 'toy',
* press enter *

{add WORD}: Add a word
{printAll}: Print all words
{startsWith PREFIX}: Print words that start with
{search WORD}: Match a specific word
{quit}: Quit

? startsWith ba


* press enter *
{add WORD}: Add a word
{printAll}: Print all words
{startsWith PREFIX}: Print words that start with
{search WORD}: Match a specific word
{quit}: Quit

? startsWith ba

All words that start with 'ba': 'banana', 'bandage', 'bandaid',
'bandana',


* press enter *

{add WORD}: Add a word
{printAll}: Print all words
{startsWith PREFIX}: Print words that start with
{search WORD}: Match a specific word
{quit}: Quit

? add toybox


* press enter *

{add WORD}: Add a word
{printAll}: Print all words
{startsWith PREFIX}: Print words that start with
{search WORD}: Match a specific word
{quit}: Quit

? startsWith toy


* press enter *

{add WORD}: Add a word
{printAll}: Print all words
{startsWith PREFIX}: Print words that start with
{search WORD}: Match a specific word
{quit}: Quit

? startsWith toy

All words that start with 'toy': 'toy', 'toybox',
* press enter *

{add WORD}: Add a word
{printAll}: Print all words
{startsWith PREFIX}: Print words that start with
{search WORD}: Match a specific word
{quit}: Quit

? search letter


* press enter *
{add WORD}: Add a word
{printAll}: Print all words
{startsWith PREFIX}: Print words that start with
{search WORD}: Match a specific word
{quit}: Quit

? search letter
word 'letter' was found in the trie

* press enter *

{add WORD}: Add a word
{printAll}: Print all words
{startsWith PREFIX}: Print words that start with
{search WORD}: Match a specific word
{quit}: Quit

? quit


* press enter *
```

The exact text that is printed with each command doesn't have to match the example output exactly, although there should be some delimiter between words printed and it should be on the same line unless the terminal word wraps it.

MemUse mode

The other mode you will create is called "memuse" and is activated like this:

```
  java TrieMain memuse < words.dict
```

This will test the memory usage of your trie data structure. First, the "< words.dict" part allows the program to grab words from a file (that you are given) called "words.dict". You will read these words the same as if the user typed them in, one at a time, on separate lines. Each time you read in a word, you will add it to your trie. As you add words, the amount of heap memory used will increase. I've written a fun visualization so you can see how the memory grows as you add items to your trie.

To get the visualization to work correctly, first I would recommend zooming out on the your terminal (by pressing ctrl and minus a few times, possibly several times). Now, in your code, you will need to call the following, which are in the Display class inside the Display.class file:

```
// replace 900 with whatever the max y value is that you want to
// display. This is in MB so 900 would be 900MB
Display.setYMax(900);

// replace 15 with whatever you want the y increment to be. For
// example, at 15 you will see 0mb, 15mb, 30mb, 45mb, 60mb, and so on
// on the y axis, up to the y max
Display.setYInc(15);
```

Now, as you add words into your trie, keep track of the count of words you have added so far and call the show function each time you add a word:

```
Display.show(trie, heapMemoryInBytes, wordCount);
```

This function takes in your trie data structure object, the number of bytes currently allocated on the heap (more on this in a bit) and the number of words currently added into our trie. How do you access the number of bytes currently allocated in your program? Java makes this easy! Just grab a pointer to the runtime environment and call two functions:

```
Runtime rt = Runtime.getRuntime(); // do this once
long heapMemoryInBytes; // create this once

// now do this EVERY iteration as you add more words to the trie
heapMemoryInBytes = rt.totalMemory() - rt.freeMemory();
```

This will give you the total amount of memory allocated in bytes (minus the memory marked as free space that the garbage collector hasn't collected yet).

Now you should see a nice bar graph showing the memory usage of the trie as you add more words to it. We want to compare this to the memory usage of another data structure, the HashSet. This is a data structure built into Java and is similar to a dictionary but only contains keys, no values. We want to add these same words, one at a time, into our HashSet object and observe the memory usage difference.

To do this, first we need to cleanup the memory from the trie. Set your trie object to null and call `rt.gc()` to force the garbage collector to clean up the memory. Recall that setting trie to null with reduce the reference count to zero and allow the garbage collector to claim that memory.

Now create a new HashSet object (it is a generic class so give it String as the generic type) and, just like we did with the trie, insert the words from the file one at a time, calling the show method each time. The call will look like this:

```
Display.show(hashSet, heapMemoryInBytes, wordCount);
```

When you run this, you should see the memory usage go up for the trie until all words are added, and then see the memory usage go up for the hashset, until all words are added.

If the visualization doesn't show up correctly, you may have to zoom out some more. Try keeping the values for ymax and yinc at 900 and 15 respectively and then zooming out to see the chart.

Note that to get this to work properly, you will need to store the words from the file in another data structure (like an array or a list). You can then iterate over this structure when inserting values into your Trie and then again when inserting values into your HashSet. Yes this structure will show up in memory but both the Trie and HashSet will include this extra memory so the comparison in memory usage is still valid (i.e. both take the hit, so their relative values can still be compared).

Bonus

Just like in the last assignment, there is a lot of new stuff here. Getting your trie to work is only half the battle as there is a lot here to setup. Focus on getting everything working first, then find the secret phrase hidden in the notes. Put this phrase into the sea of words and see what happens.

Rubric:

#	ITEM	POINTS
1	insert method	10
2	printAllWords method	10
3	printAllWords(prefix) method	10
4	search method	10
5	Test mode works	5
6	MemUse mode works	5
	TOTAL	**50**

Note that if test mode doesn't work, it will make it harder for me to test and points may be deducted in multiple areas because of it.

#	PENALTIES	POINTS
1	Doesn't compile	-50%
2	Doesn't execute once compiled (i.e. it crashes)	-25%
3	Late up to 1 day	-25%
4	Late up to 2 days	-50%
5	Late after 2 days	-100%