CS2040S: Data Structures and Algorithms

Problem Set 5

Due: Monday, March 2, 11:59pm

Collaboration Policy. You are encouraged to work with other students on solving these problems. However, you **must** write up your solution **by yourself**. We may, randomly, ask you questions about your solution, and if you cannot answer them, we will assume you have cheated. In addition, when you write up your solution, you **must** list the names of every collaborator, that is, every other person that you talked to about the problem (even if you only discussed it briefly). You can do so by leaving a comment at the start of your .java file. Any deviation from this policy will be considered cheating, and will be punished severely, including referral to the NUS Board of Discipline.

Problem 5. (Autocomplete)

The goal of this problem is to build a data structure to support searching a dictionary. For example, you might want to build an autocomplete routine, i.e., something that (as you type) will list all the possible completions of your term. Or perhaps you want to be able to search a dictionary based on a search pattern?

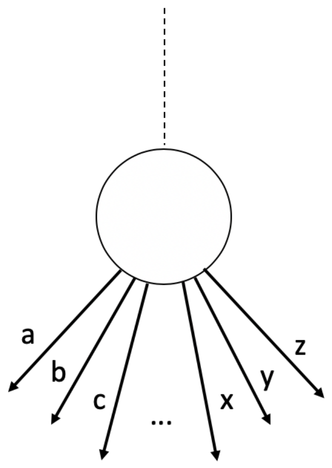
The Trie Data Structure

The main data structure you will build is called trie, which is specially designed for storing strings. (It is also commonly used to store IP addresses, which are just binary strings, and trie data structures are used in routers everywhere to solve the "longest prefix" problem that is used to determine the next hop in a network route.)

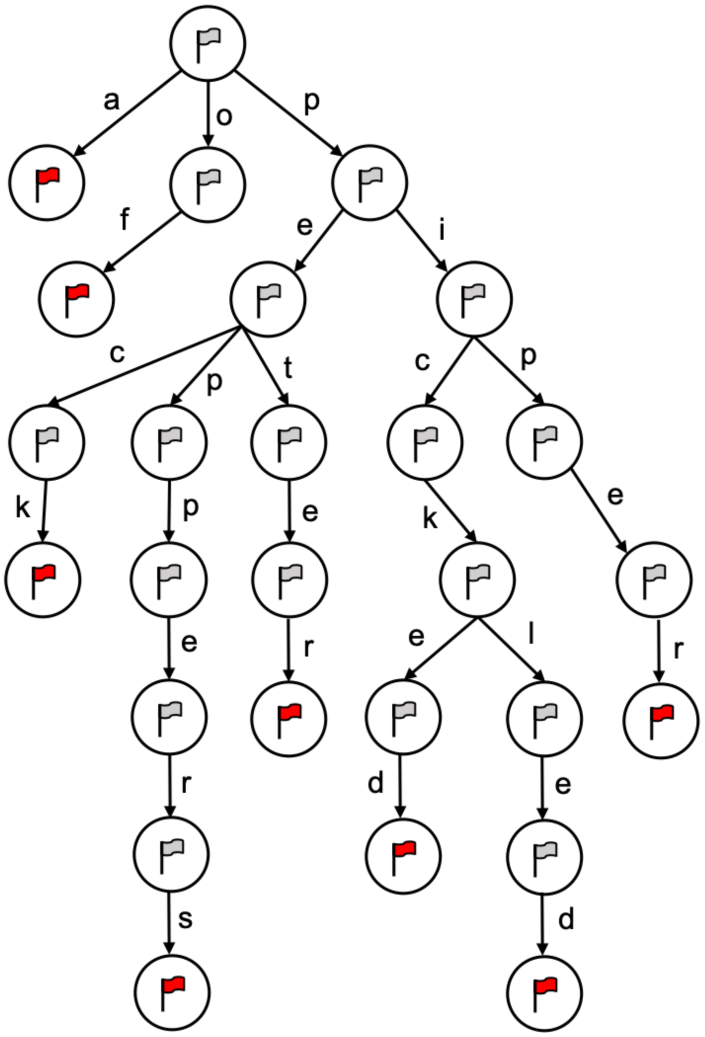
A trie is simply a tree where each node stores one letter or a string. A trie, however, is not a binary tree: each node can have one outgoing edge for each letter in the alphabet. Hence, if a trie supports all 256 possible ASCII characters, each of its nodes can have 256 children!

To simplify things for this assignment, our trie will only support **alphanumeric characters**.

Here's a picture of a trie node (for simplicity, this is a trie that stores only lowercase letters):



Each root-to-leaf path in a trie represents a single string. And when two strings share a prefix, they will overlap in the trie! Here's a picture of a trie that contains a collection of overlapping strings:



By following the edges, you can verify that this trie contains all of the following strings: {“peter”, “piper”, “picked”, “a”, “peck”, “of”, “pickled”, “peppers”}.

Notice that two words in a trie might completely overlap. For example, you could have both the words “pick” and “picked” in your trie. To indicate that “pick” is also a valid word, we can add an additional special flag to indicate that a node represents the last letter in some string. In the image above, we use the red flag to denote the end of string.

Search. Searching a trie for a string s is simply a matter of starting at the root, and iterating through the string one character at a time. For each character in s, follow the child node indicated by the character. If there is no such child node, then the string is not in the tree. When you get to the end of s, and if the last node has the special flag indicating the end of a word, then s is in the tree!

Insert. To insert a string into a trie, do the same thing as a search. Now, however, when you find a node in the trie where you cannot keep following the string (because the indicated character leads to a non-existent child), then you create nodes representing the rest of the string.

Implementation hints: When you implement a trie, it is useful to have a TrieNode class to contain the information stored at a node in the trie. In a binary tree, your node class would have a left and a right child. Here, you will need an array of children, one per possible child. For simplicity, you are encouraged to just use a *fixed size* array with one entry for each of the possible children. (Recall: we know exactly how many children a trie node can potentially have!) There are more efficient solutions, but we can ignore them for now.

Your trie class will manipulate Java strings that use alphanumeric characters. One useful method that a string supports is `charAt(j)` which returns the j'th character of the string. For example, if we have previously declared `String name = Iphigenia`, then `name.charAt(2)` will return 'h'.

Additionally, recall that characters are internally represented as integers. The table below shows each character and their corresponding ASCII value.

Character	Integer Value	Character	Integer Value	Character	Integer Value
0	48	A	65	a	97
1	49	B	66	b	98
...	...	...	...	...	...
9	57	Z	90	z	122

Problem 5.a. Implement the trie data structure.
By editing Trie.java, implement a trie data structure based on the description given above.

In addition, you should also implement the following functions on the trie:

- `void insert(String s)`
  Inserts a string into the trie data structure.
- `boolean contains(String s)`
  Returns true if the specified string is inside the trie data structure, false otherwise.

Pattern Matching

Once you have a dictionary, you want to be able to search it! And you really want to be able to search it using some sort of “search pattern” so you can find words that you do not already know. For example, you might want to know all the words in the dictionary that begin with the substring “beho”, for example.

One very common way to specify such search patterns is with regular expressions (or people usually just call it regex). Regular expressions are a powerful way of expressing a search. (In fact, they allow you to search for any pattern specified by a finite automaton!) We will not implement complete regular expression searching here, but we will use a small subset of the regular expression language:

- ’.’: a period can match any character. For example, the string ‘b.d’ would match the words: ‘bad’ and ‘bid’ and ‘bud’.
- ’*’: a star modifies the preceding character, which can be repeated zero or more times. For example, the string ‘ho*p’ would match the words: ‘hp’, ‘hop’, ‘hoop’, ‘hooop’, ‘hoooop’, etc.
- ’+’: a plus modifies the preceding character, which can be repeated one or more times. For example, the string ‘ho+p’ would match the words: ‘hop’, ‘hoop’, ‘hooop’, ‘hoooop’, etc. It would not match ‘hp’.
- ’?’: a question mark modifies the preceding character, which can appear either zero or one time. For example, the string ‘colou?r’ would match the words: ‘color’ and ‘colour’ (but nothing else).

For the purpose of our string matching algorithm, we also want to support ‘.’ followed by one or more special characters.

A ’.’ can follow an **arbitrary number** of other ’.’ characters. For example:

- ‘..’: This matches exactly two arbitrary characters. For example, the string ‘a..d’ would match ‘abcd’ but not ‘abd’ or ‘abcbd‘.
- ‘...’: This matches exactly three arbitrary characters. For example, the string ‘a...e’ would match ‘abcde’ as well as ‘azyxe’.

A ’.’ can also be followed by **at most one** other special character before encountering another alphanumeric character in the string. For example:

- ’.*’: This matches an arbitrary number (zero or more) of arbitrary characters. For example the string ‘t.*r’ would match ‘their’, ‘tr’, and ‘tabcdefghr’, but would not match ‘tabcd’.
- ’.+’: This matches an arbitrary number (one or more) of arbitrary characters. For example, the string ’t.+r’ would match ’their’, but would not match ’tr’.
- ’.?’: This matches zero or one arbitrary character. For example, the string ‘ab.?cd’ would match the strings ‘abcd’ and ‘abxcd’, but would not match ’abcdcd’.

We will also not support other combinations of special characters (what would '*+' mean?). We will not support any other regular expression features.

How to search? A trie is a great data structure for searching for a pattern. For example, imagine if you want to search for all strings with prefix 'abc'. Then you can simply walk down the trie until you find the node at the path 'abc' and recursively print out every string in the remaining subtree!

Similarly, if you want to match the pattern 'a.c', then first you follow the edge to node 'a'. Then, you follow *all* the outgoing edges from that node, i.e., recursing on strings with prefix 'aa', 'ab', 'ac', etc. Then, from each of those nodes, you follow the outgoing edge to 'c'.

Also, for pattern 'ab*c', you first follow the edge to node 'a' and then recursively go to 'ab', 'abb', 'abbb', 'abbbb', etc., before checking character 'c'.

In general, at every step in the tree, you can continue your pattern search recursively at one or more children, using a (possibly modified) version of the string pattern that you are currently searching for.

Beware. The hardest cases probably are combining two special characters, e.g., '.*' and '.+', where you may have to recurse on a lot of characters at a lot of levels!

This Problem, similar to PS4, has easy and hard versions. We will only take the maximum between your score for 5.b (max 15 points) and 5.c (max 25 points). For both versions, you are encouraged to implement a solution with linear running time in terms of the number of entries returned to get full marks.

Problem 5.b. (Easier) Implement prefix search for the trie data structure.
You have to complete the implementation for the following function:
`prefixSearch(String s, ArrayList<String> results, int limit)`.

This should return all the strings in the trie with **prefix** matching the given pattern `s` and **sorted in ASCII order**. If there are more entries than the `limit`, then it should just stop and return the first `limit` entries. The entries should be put in the `results`.

Your implementation only needs to handle the '.' special character in the pattern (although there can be an arbitrary number of them, and not necessarily contiguous).

Problem 5.c. (Harder) Implement pattern search for the trie data structure.
You have to complete the implementation for the following function:
`patternSearch(String s, ArrayList<String> results, int limit)`.

This should return all the strings in the trie that exactly match the pattern `s`, up to the `limit`. If there are more entries than the `limit`, then it should just stop and return that many entries. The entries should be put in the `results`. In this case, although it is preferred if your output is in ASCII order, it is not required. Hence, if the number of possible entries is bigger than the `limit`, you may output any subset with size equal to the `limit`.

Your implementation needs to support all the special characters and their combinations as stated on the previous page. For testing purposes, **each string will only match the pattern in (at most) one way**.

Scoring for this problem is also affected by the number of possible patterns that you manage to handle. Hence, you may still get good marks even if you cannot handle everything and pass all the private test cases.

Autocomplete. If you implement your trie properly, you can test it using the Autocomplete application. If loads a large dictionary of English words. (To test it out, you might want to use a smaller list of words.) Then, as you type, it lists all the words that have a prefix that matches your pattern so far. When you hit "enter" it shows exactly the words that match your pattern.

ArrayList. `ArrayList<String>` is the Java implementation of a resizable array for strings. The angle brackets specify that the array only supports String entries. (This is related to Java generics, which you can ignore for now.) To insert something into a ArrayList, simply perform `results.add(myString)` to add the element at the end. You can find more documentation for the ArrayList in the Java documentation.