CS2040S: Data Structures and Algorithms

Problem Set 5

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 you talked to about the problem (even if you only discussed it briefly). You can do so in Coursemology by adding a Comment. Any deviation from this policy will be considered cheating and will be punished severely, including referral to the NUS Board of Discipline.

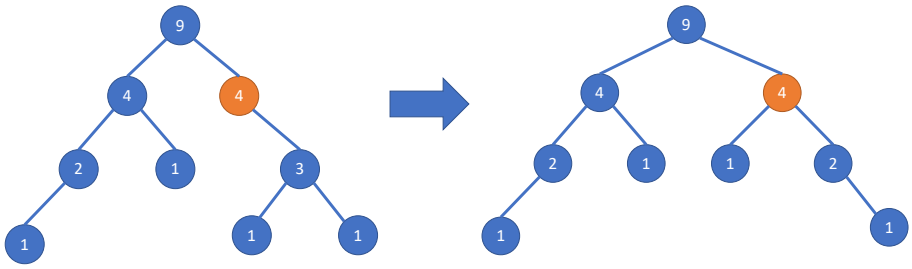


Figure 1: Example of a tree being rebalanced. The number in each node represents its weight. Notice the orange node is violating the balance condition as $3 > (2/3) \times 4$, and that all the other nodes are in balance. The rebalance operation rebuilds the subtree rooted at the orange node, using the rebuild routine from PS4.

Problem 1. (Scapegoat Trees, Part 2)

This week, you will complete the implementation of Scapegoat trees, building on the balancing routine that you wrote last week. The basic idea of a Scapegoat Tree is that whenever a node is out of balance, we simply rebuild the entire subtree as a balanced tree.

To decide if a node is out of balance, we are going to maintain the ***weight*** of every node: the weight of a node u is the number of nodes in the subtree rooted at u (including u itself). In other words, leaf nodes have a weight of 1 by definition.

We say that a non-root node u is ***unbalanced*** when one of its children has more than 2/3 of the total weight, i.e., if v is a child of u and $weight(v) > (2/3)weight(u)$.

After every insertion, we can check whether any of the nodes on the insertion path became unbalanced, and if so, rebuild the relevant subtree. Your main job in this first problem is to modify the implementation of the `insert` routine to perform such checks and rebuild as needed.

A few important notes about solving this problem:

- The solution to this problem will rely on your solution to Problem Set 4 to rebuild the tree. However, mistakes from Problem Set 4 will not be considered mistakes in this problem set, so you can solve this problem even if your previous code had mistakes. It may, though, be harder to debug your solution if your `rebuild` routine does not work correctly, and you may want to implement a simpler and less efficient version for the purpose of this problem.
- It is possible to use the `rebuild` routine from Problem Set 4 as a black-box without changing it at all, and that is acceptable. However, you may find it easier (or slightly more efficient) to modify the `rebuild` routine from the previous problem set in small ways. That is also okay. (Please **do not modify nor remove any method signature**, however.)
- Recall that the `void rebuild(TreeNode node, Child child)` routine implement in Problem Set 4 is designed to rebuild the *child* of a node. Thus, it cannot be used to rebuild the

root of the tree, when it is out of balance. For the purpose of this problem set, **the root is a special node that is never rebalanced**.

- Be careful when multiplying or dividing integers by `float` or `double` (non-integer) numbers. Typically, Java correctly interprets the result as a `float` or a `double`, but occasionally problems can arise if the result is automatically converted back into an `int`.
- For the purpose of this problem, you may assume that every integer inserted into the tree is unique, and there will never be a duplicate insertion. (For more of a challenge, you can implement your code so that a duplicate insertion has no effect on the tree, i.e., it leaves the tree unchanged with a single copy of the node in the tree.)

Problem 1.a. The first step is to modify the implementation of the `TreeNode` class so that each node maintains the weight of its subtree. (Unlike in Problem Set 4, we do not want to have to repeatedly count the number of nodes in a subtree, as that can be slow.) Add a variable `weight` to the class, ensure that it is properly initialized whenever a new node is created, and update the `insert` method to ensure that after every insertion, the weights of all relevant nodes have been updated.

Problem 1.b. You will need to update the weights of the nodes when a subtree is rebuilt. You may want to modify the `rebuild` method from Problem Set 4 to update the weights. Alternatively, you may want to implement a new routine `fixWeights(TreeNode u, Child child)` that updates the weights of every node in the subtree of the specified child.

Problem 1.c. Implement the method `boolean checkBalance(TreeNode u)` which determines whether the specified tree node is unbalanced.

Problem 1.d. Update the `insert` method so that it works as follows:

- Insert the specified key in the tree using a typical binary search tree insertion (notice that this new node will be inserted as a leaf).
- Identify the **highest** unbalanced node on the root-to-leaf path to the newly inserted node. (Why is it sufficient for us to check only the nodes along the root-to-newly-inserted-leaf path?)
- If there is no such unbalanced node, then we are done. If there is an unbalanced node, then rebuild it.

Note that you already have all the parts you need to solve this last part, just that you will have to now put them together.

Also, notice that to rebuild an unbalanced node u, we actually need to call the `rebuild` method on its parent, and we will also need to know if the unbalanced node is the left child or the right child.

Problem 2. (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?

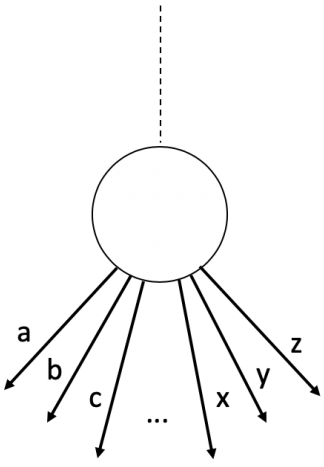
The Trie Data Structure

The main data structure you will build is a trie (was originally pronounced as "tree" as in re**trie**val, but now more commonly pronounced as "try"), 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.)

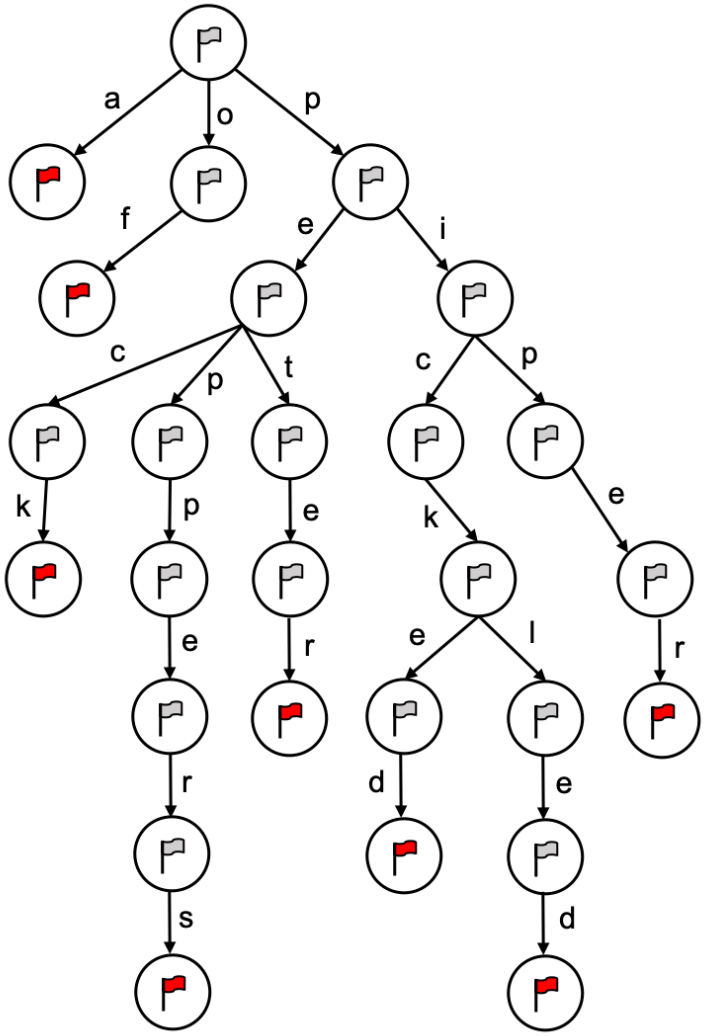
A trie is simply a tree where each node stores one letter or a string. A trie, however, is not a binary tree: each node can have one outgoing edge for each letter in the alphabet. Hence, if a trie supports all 256 possible ASCII characters, each of its nodes can have 256 children!

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 trie. 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 trie!

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 2.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 methods for 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!) For the purpose of our string matching algorithm, we will only support the '.' character. For example:

- '.': a period can match any character. For example, the string 'b.d' would match the words: 'bad' and 'bid' and 'bud', and the string 'a.d' would match 'and' and 'add' but not 'abc'.
- '..': 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'.

We **will not** support other special characters or any other regular expression feature, as it is somewhat more complicated, but just for your information, the following are standard regex patterns:

- '*': 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).

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'.

Problem 2.b. Complete the implementation for the following method:
`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` array. Your implementation should handle the '.' special character in the pattern (although there can be an arbitrary number of them, and not necessarily contiguous).

Autocomplete. If you have implemented your trie properly, you can test it using the Autocomplete application. It 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 to the end of the array. You can find more documentation for `ArrayList` in the Java documentation.

String concatenation. As we have seen from Problem 4(g) of Tutorial 1, concatenating n characters one by one is an $O(n^2)$ operation, which isn't very efficient. Luckily, Java provides us with the `StringBuilder` class, which allows us to build strings in $O(n)$ time. You can find more documentation for `StringBuilder` in the Java documentation.