CS2040S: Data Structures and Algorithms

Problem Set 5

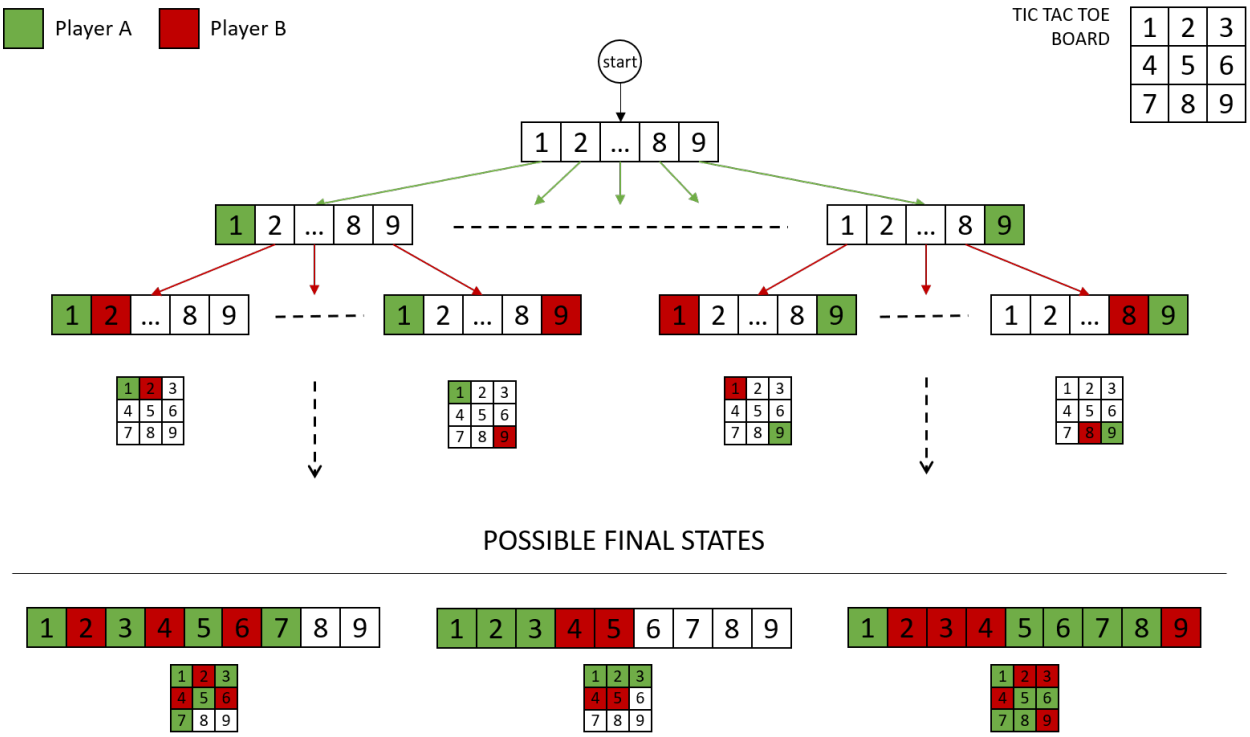
Collaboration Policy. You are encouraged to work with other students on solving these problems. However, you **must** write up your solution **by yourself**. We may, randomly, ask you questions about your solution, and if you cannot answer them, we will assume you have cheated. In addition, when you 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.

Problem 1. (Game Trees)

Trees can be a convenient way to represent a game. Imagine you have a two-player game where players take turns making moves. The root node (at depth 0) can represent the initial state of the game, and each of its children can represent a choice made by the first player. The nodes at depth 1 represent the position after the first move, and the children of the depth 1 nodes can represent choices made by the second player. And so forth.

The leaves represent the final state of the game, where either one of the players has won, or there are no further moves to make.

For example, let us consider a classic game of Tic-Tac-Toe (though we can use a tree to represent any game where players take turns!).



You can think of every node as belonging to either Player A (the first player) or Player B (the second player). As you walk down the tree, the nodes will alternate between A-nodes and B-nodes.

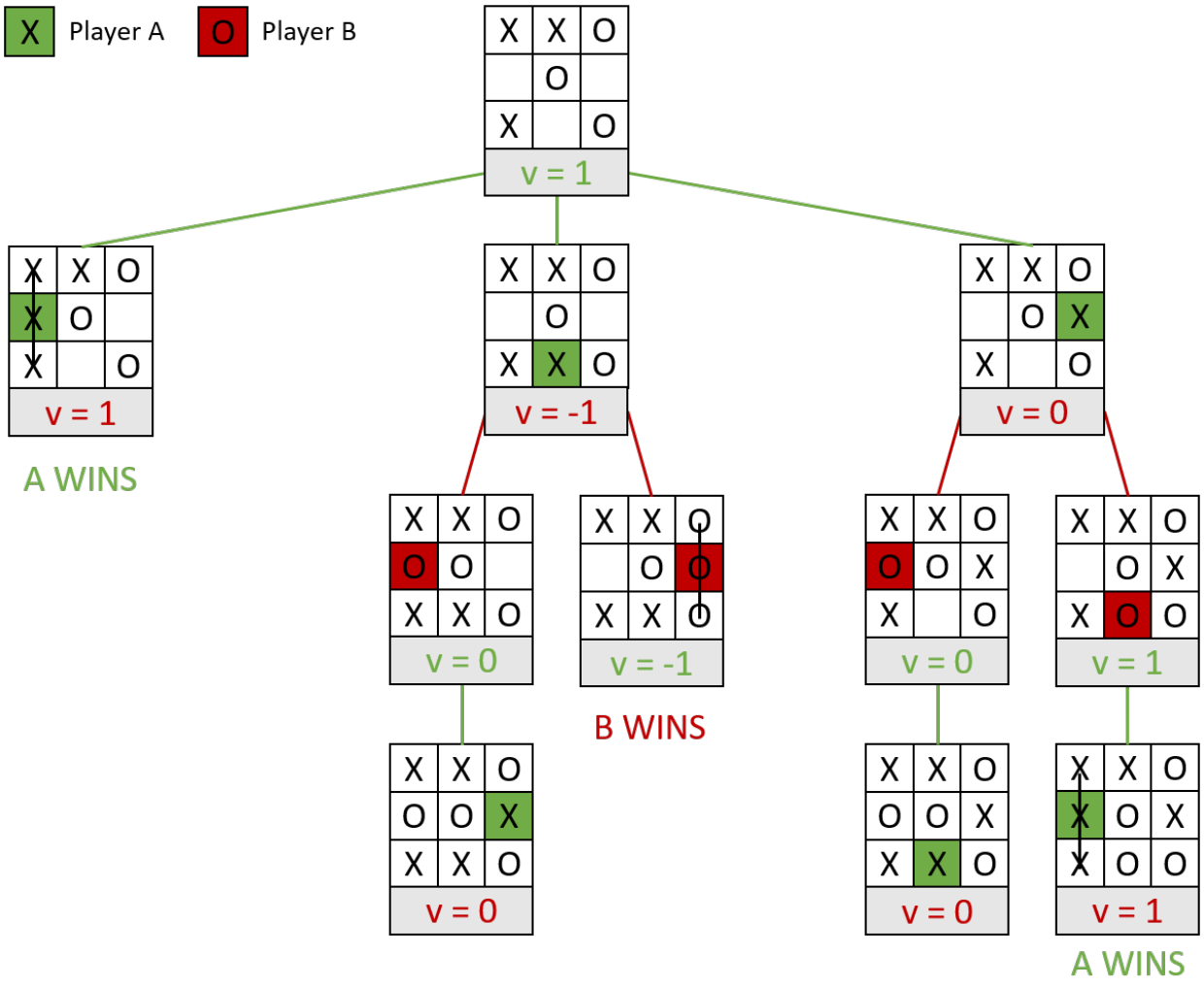
The value of a node indicates the best score that Player A can hope for. For example, a leaf where Player A wins may have a value of 1, while a leaf where Player A loses may have a value of -1 (We will use positive numbers to indicate a win for Player A and negative numbers to indicate a loss for Player A). A draw would be represented by a value of 0.

Notice that if we are at a B-node and all the children are leaves where Player A wins, then Player B is out of luck! We can assign that B-node a value of 1 as well, since no matter what Player B does, Player A is going to win. Similarly, if we are at an A-node and any one of the children is a

leaf where Player A wins, then we can designate that A-node as having value 1 also, since Player A can make the move that leads to that child node, and will definitely win.

In this manner, we can go through and assign a value to every node. At a B-node, we can assume that Player B is going to choose the child node that is worst for A, i.e., the node with the minimum value. So we can assign a value to a B-node by taking the minimum value among its children. For an A-node, we can assume that Player A is going to choose the child node that is best for A, i.e., the node with the maximum value. So we can assign a value to an A-node by taking the maximum value among its children.

Why minimum and maximum? In our example, we are looking at a game with only three possible values for end states: $\{-1, 0, 1\}$*. In other games, we may have end states with other possible values, e.g.* -9*,* 45*, etc.*



Problem 1.a. In this problem, we will give you a `GameTree`. The `GameTree` has a root node, and each node in the tree has an array of children. The template code includes a routine for reading in the entire `GameTree` from a file. We have also included a few other variants that you can try! (See Part (b) below.)

Each node in the tree contains a `name` field, a `String` that represents the state of the game at that node. Each node also contains a `value` field, which is used to represent the best score for Player ONE for that state. Note that Player ONE refers to the player who is taking the turn at the root node.

Your job in this problem is to implement the function `int findValue()` which determines the best score that Player ONE can hope to achieve for the game represented by the tree. Specifically, the `findValue` routine should fill in the `value` field in every node in the tree, and it should return the final value of the entire game, i.e. the value at the root. Note that the leaf nodes are already initialised with their values.

Warning: You should implement a generic `GameTree` that does not make any assumptions about the game being played. In other words, you should not design your `GameTree` and/or `findValue` to only work for the standard Tic-Tac-Toe games used in our example. This also implies that the leaf nodes can have arbitrary values, instead of only $\{-1, 0, 1\}$.

Problem 1.b. (Optional)

To experiment with your `findValue` routine, let us think about Tic-Tac-Toe. You are likely familiar with the standard version, which most people know ends inevitably in a tie (if both players play properly). How about some alternate versions?

- *Misere-Tac-Toe*: In this variant, the first person to create a line of three-X's or three-O's loses (instead of wins, as in the classic version).
- *NoTie-Tac-Toe*: In this variant, if the game ends in a tie, each player gets one point for every X or O they have on the board, except for in the middle square; any player with an X or O in the middle squares gets -1 points. For example, if Player ONE has four Xs, with one in the middle square, their total number of points is 2. Since the number of points totals to 7, there will always be a winner!
- *Arbitrary-Tac-Toe*: In this variant, much like in the previous NoTie variant, each square has an arbitrary point value. If the game ends in a tie, then each player gets points for every square in which they have an X or an O.

It is not quite as obvious who will win each of these variants, nor the best way to play! We have given you the complete game tree for these variants. For each of these games, can you determine whether Player ONE or Player TWO wins (or whether the game is necessarily a tie)? Optionally (and to test your code), you might implement a small interactive game that would allow Player

ONE or Player TWO to play against the computer (which would use the values in the game tree to determine the optimal move).

Another possible question is whether you can find (different) sets of values for the arbitrary variant so that (a) Player ONE is guaranteed a victory, and (b) Player TWO is guaranteed a victory?

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?

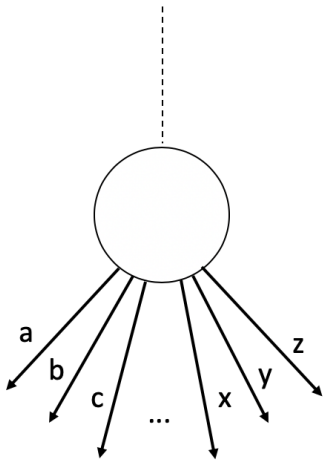
The Trie Data Structure

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.)

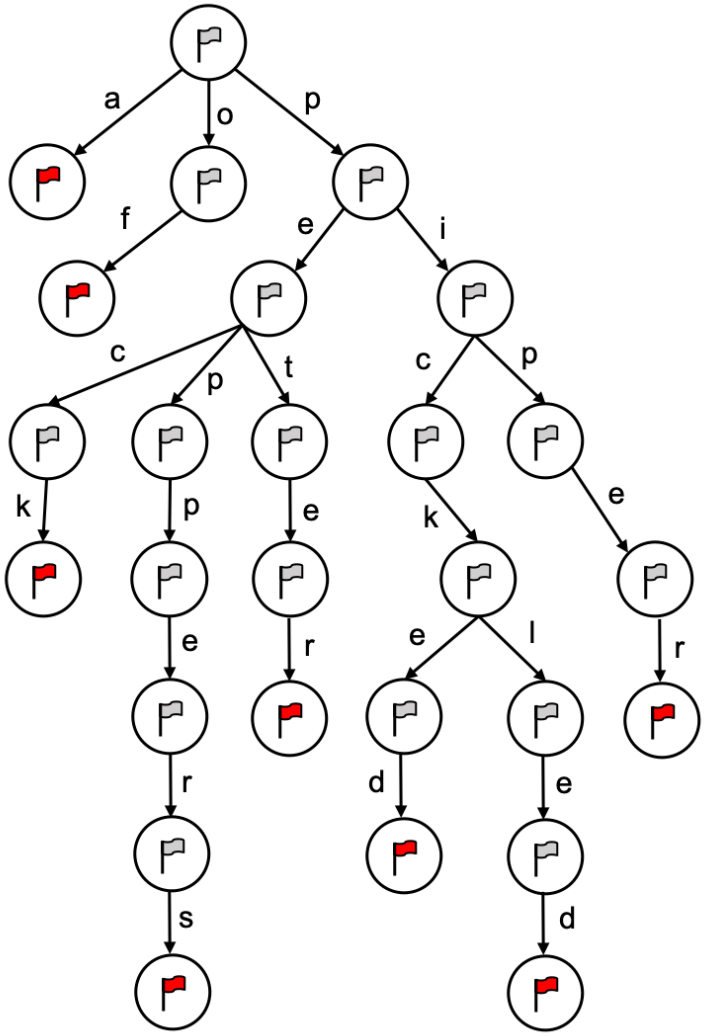
A trie is simply a tree where each node stores one letter or a string. A trie, however, is not a binary tree: each node can have one outgoing edge for each letter in the alphabet. Hence, if a trie supports all 256 possible ASCII characters, each of its nodes can have 256 children!

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.