**CS2040S: Data Structures and Algorithms**

# Problem Set 4

*Due: Monday, February 14, 3:00pm*

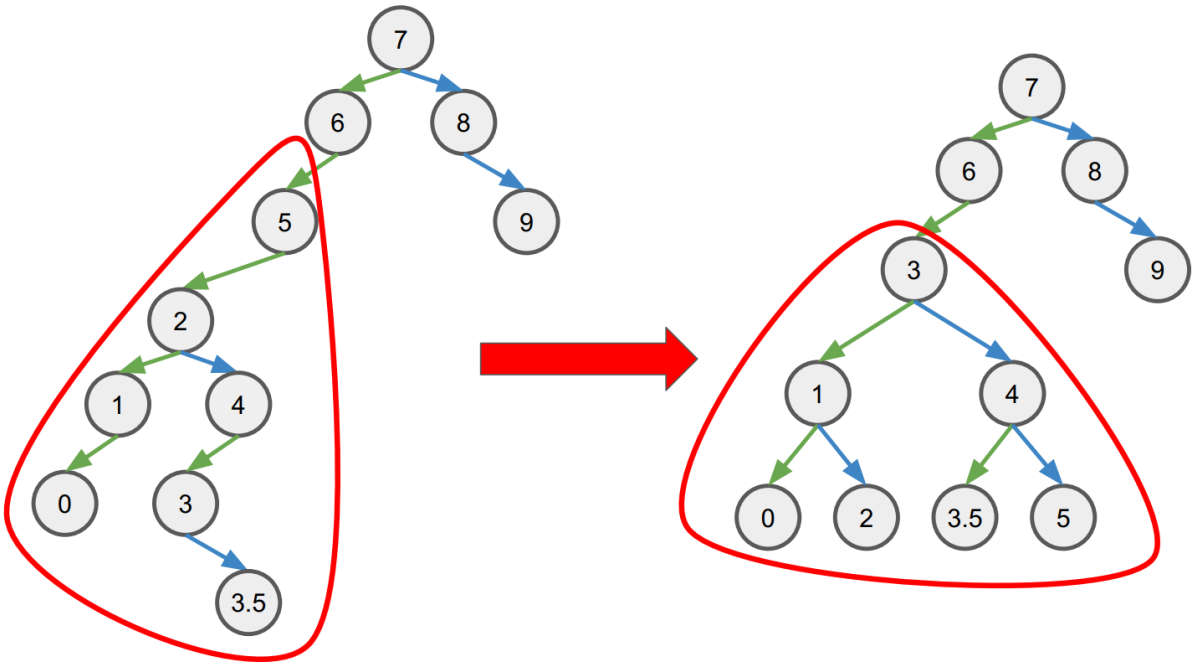
**Collaboration Policy.** You are encouraged to work with other students on solving these problems. However, you **must** write up your solution **by yourself**. We may, randomly, ask you questions about your solution, and if you cannot answer them, we will assume you have cheated. In addition, when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

Self-balancing trees can seem complicated at times, with various rotations and splits and merges needed to keep everything in balance. Scapegoat Trees, by contrast, are simple. On insertion and deletion, you do not do any anything extra at all! Just do the insert or delete operation as you would in an unbalanced binary search tree.

We do, however, do need to do some work when and if parts of the tree become too unbalanced. Whenever things get too bad, a scapegoat node is chosen, and the entire subtree rooted at that node is rebuilt. So most of the time, our tree operations are very fast and simple. And every so often, we have to do some cleanup work to compensate.

In this problem, you are going to implement the key step in a scapegoat tree: rebuilding a node. (We are going to postpone the part where we choose which nodes to scapegoat and when to perform this rebuild operation.) So your goal in this problem set is to implement an operation `void rebuild(TreeNode node, Child child)` that rebuilds either the specified left or right child of the node in question.




Beware that a key property of the rebalance operation is that it has to be efficient. (Otherwise, the cost of the occasional rebalancing might dominate the cost of the entire data structure!) In particular, if we are rebuilding a subtree containing $k$ nodes, it should only take $O(k)$ time. Thus your implementation of the following methods should be sufficiently efficient to satisfy this goal. (As always, you should make your solutions as efficient as possible.)

**Problem 1.a.** First, implement an operation `int countNodes(TreeNode node, Child child)` that counts the number of nodes in the specified `child` of `node`, where `child` can be `LEFT` or `RIGHT`.

**Problem 1.b.** Second, implement an operation `TreeNode[] enumerateNodes(TreeNode node, Child child)` that returns an array of nodes in the **inorder traversal** of the subtree. (You should make no assumption about the ordering of keys in the tree, but instead simply rely on the tree structure to infer the order. The tree itself may or may not be sorted in the way you expect.)

You are only allowed to use default Java arrays for your solution, i.e. Java API such as `List` and `ArrayList` will not be allowed.

*Hint: You may want to use the `countNodes` method from the previous part to determine the size of the array to return in advance. If you resize the array frequently, it is likely to not be as efficient.*

**Problem 1.c.** Finally, implement an operation `TreeNode buildTree(TreeNode[] nodeList)` that builds a perfectly balanced tree (definition below) from the list of nodes previously constructed. Combined with the previous parts, this should give you everything you need to rebuild a subtree.

Do remember that for each node, after you've balanced the two subtrees directly below it, you will need to update that node's left and right child to be the new root of the balanced left and right subtrees respectively.

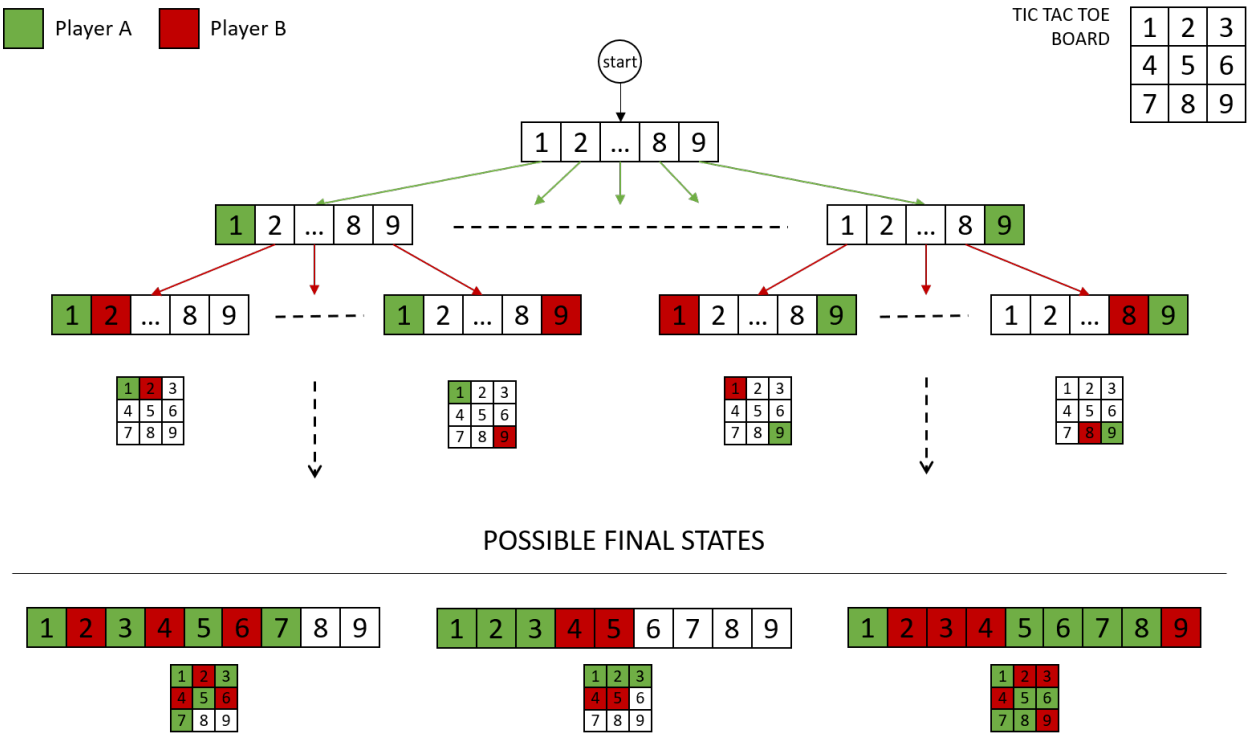
*Let us define a perfectly balanced tree. For any two sibling nodes in a perfectly balanced tree, their size (the number of nodes in their subtree, not the height of their subtree) should differ by at most 1.*

**Problem 2. (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.

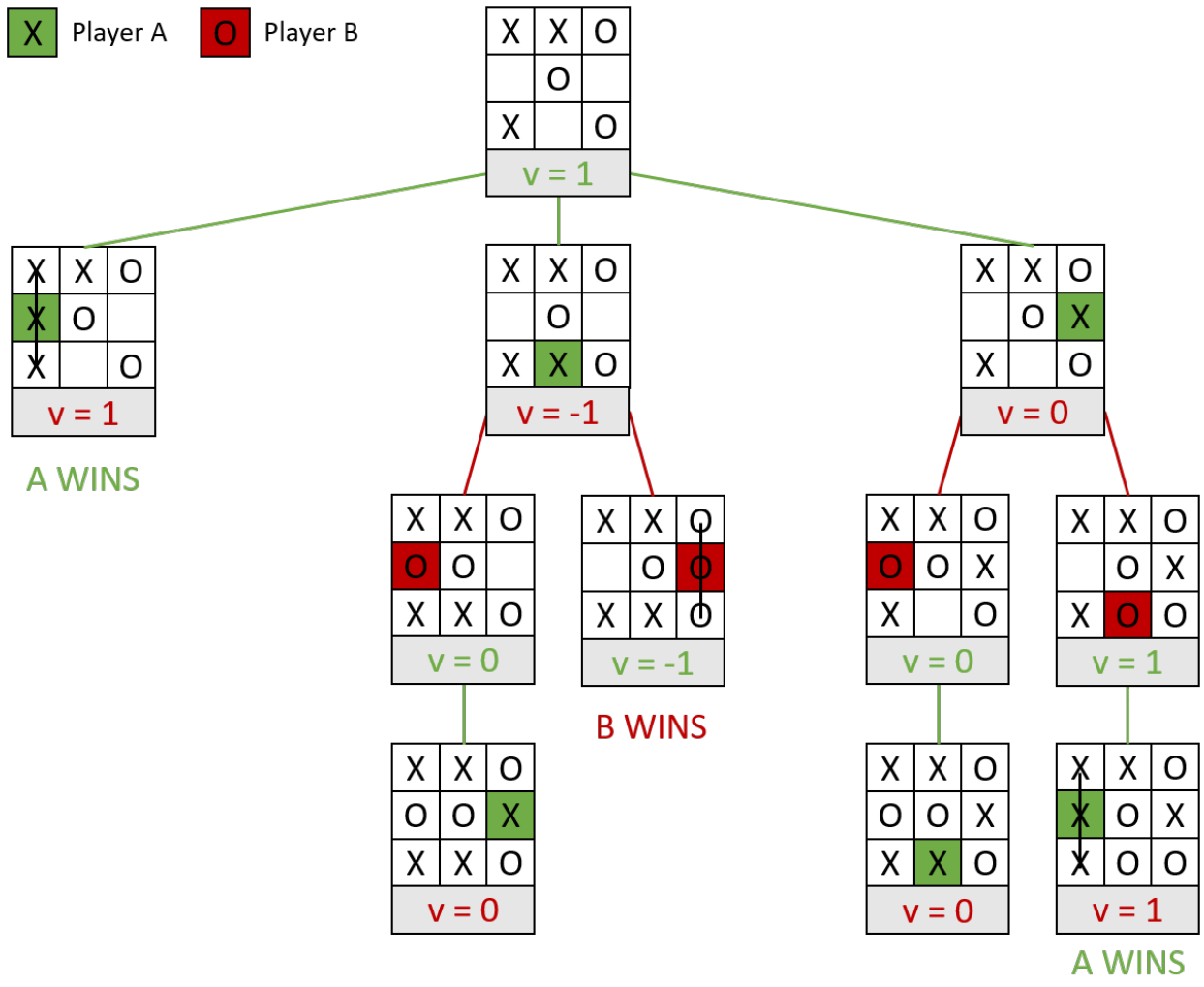
The value of a node indicates the best score that Player A can hope for. For example, a leaf where Player A wins may have a value of 1, while a leaf where Player A loses may have a value of -1 (We will use positive numbers to indicate a win for Player A and negative numbers to indicate a loss for Player 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 2.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 2.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?