Computer Science 345
Test 2 Practice problems
March 21, 2023

The test itself will have three questions, one each from the categories of ADTs (drawing from Chapters 1–3, so including bit vectors and heaps for example), graphs (Chapter 4), and BSTs (Chapter 5). This set of practice problems includes two problems from each category.

Pull from the repository. You will find the starter code for these practice problems in a folder called `test2-practice` under the `practice` folder. Make an Eclipse project in the `test2-practice`, because in it each problem has its own package (for example, `sortedArrayBag`), and all the classes for that problem are in that package. This code does **not** use the package names `adt`, `impl`, etc from our convention for the project code base.

A solution to these problems can be found in the folder `test2-practice-soln`.

1. [**ADTs.**] The class `sortedArrayBag.SortedArrayBag` implements a bag like we have seen in class but with the following simplifications or assumptions.

- The keys are of type `String` (the class is *not generic*).
- The `remove()` and `iterator()` methods are *not supported*.
- All (potential) keys are known ahead of time and are passed to the constructor through an iterator that returns them in sorted order. (All the keys have 0 counts until the `add()` method is called for them.)

An example of when a bag like this would be useful is counting word frequencies in a text when one already knows the text's vocabulary.

Finish the class `sortedArrayBag.SortedArrayBag`. The instance variables, constructor, and `isEmpty()` method are provided. The instance variable `keys` contains the given keys in sorted order. Specifically, write the methods `add()` and `count()` so that they run in worst case $O(\lg n)$ time, where n is the number of keys passed to the constructor. Write the method `size()` so that it runs in $O(n)$ time.

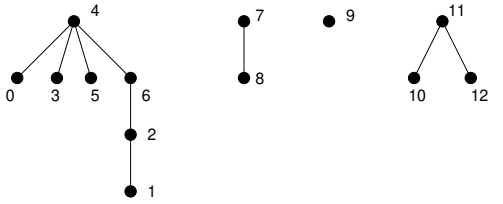
You may assume that `add()` is called only with keys that were initially given to the constructor (you do not need to check for this). If `count()` is called with a key not given to the constructor, then it should return 0.

(Hint: The stub for a suggested helper method, `findIndex()`, is provided. Since `keys` is a sorted array, how can you write this method so that `add()` and `count()` are logarithmic?)

2. [**ADTs.**] Recall that an array forest is a data structure that uses an array to represent a collection of trees (that is, a forest). Specifically, in an array forest with n

nodes, each node in the forest is identified by a whole number in the range $[0, n)$, and an array indexed by the nodes' numbers indicates each node's parent in the forest. A node that is the root of its tree has itself as its parents.

For example, the collection of trees shown below left can be represented by the array shown below right.



4	2	6	4	4	4	4	7	7	9	11	11	11
0	1	2	3	4	5	6	7	8	9	10	11	12

The class `arrayForest.ArrayForest` implements an array forest. Finish the following two methods:

- `depth()` takes a node `p` (indicated by number) and determines `p`'s depth, that is, its number of links `p` is away from the root of its tree. For example, in the array forest above, the depth of node 1 is 3. Use `TestDepth` to test.
- `ancestorChain()` takes a node `p` (indicated by number) and returns an iterator that returns the nodes in the path from `p` to its root. For example, in the array forest above, a call to `ancestorChain(1)`—that is, starting from node 1—returns an iterator which returns 1, 2, 6, and 4 on successive calls to `next()`. Use `testIterator` to test.

3. **[Graphs.]** A *cycle* is a path in a graph that begins and ends at the same vertex. Write the method `checkCycle.CheckCycle.checkCycle()` that takes a graph and a starting vertex and determines whether there is a cycle of length at least two beginning and ending with that vertex. (Since we assume the graph has no self loops, cycles of length one are impossible.)

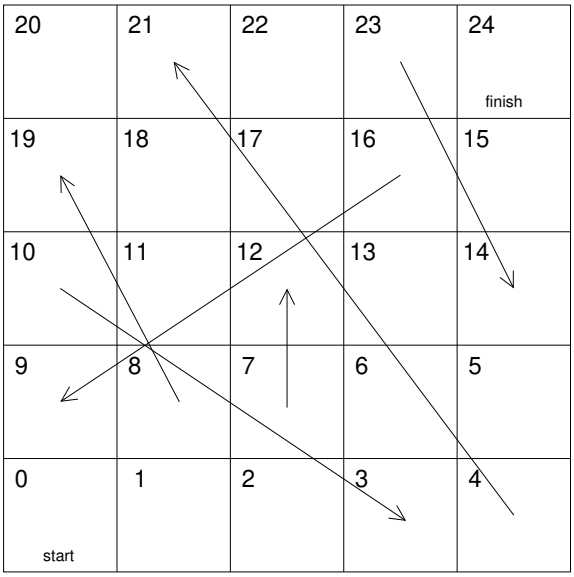
Use `TestCheckCycle` to test.

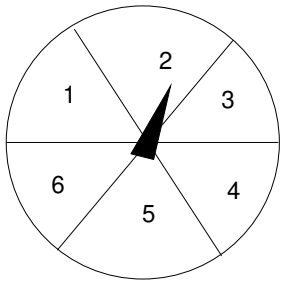
4. **[Graphs.]** The game *Chutes and Ladders* consists in a game board and a spinner. The game board has a sequence of spaces, and additionally some spaces have ladders that connect them to spaces further on the board, and some spaces have chutes that connect them to spaces earlier on the board.

Each player starts on space 0, and on each turn the player uses the spinner to determine how many spaces he or she may move ahead, from 1 to 6 spaces. However, if the player lands on a space that is at the bottom of a ladder, then the player moves to the space on the top of that ladder on the same turn; and similarly if the player lands on a space that is the top of a chute, then the player moves back to the space

at the bottom of the chute on the same turn. A player wins by landing on the last space. (See the next page for an example.)

The class `cheatsAndLadders.ChutesAndLaddersBoard` represents a board for this game. The two operations are `size()` which gives the number of spaces and `reachableSpaces()` which, for a given space, returns an iterator over the spaces reachable from that space in one spin. For example, in the board shown below, the spaces reachable from 5 are 6, 12, 8, 9, 3, and 11. That is, from space 5 one could, by spinning values from 1 through 6, move to spaces 6, 7, 8, 9, 10, and 11, except that landing on 7 takes you ahead to space 12 by a ladder, and landing on space 10 takes you back to space 3 by a chute.





Write the static method `fewestMoves()` in class `cheatsAndLadders.CheatsAndLadders` that takes a board and determines the fewest number of moves that are required to get from the start (position 0) to the final space (position `board.size() - 1`). For example, in the board shown above, space 24 can be reached in just two moves: Spinning a 4 on the first move allows one to take the ladder from space 4 to space 21; then spinning a 3 on the second move takes one to space 24.

User `TestCL` to test.

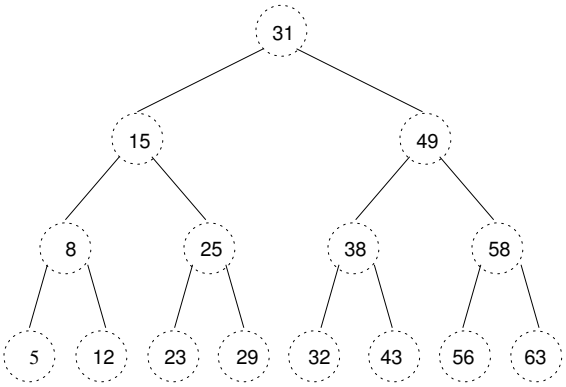
5. [**BSTs.**] Complete the method `verifyAVL.VerifyAVL.verifyAVL()`, which takes a `Node` as the root of a binary tree and determines whether the tree rooted at the node fulfills the AVL property. Notice that in order for a tree to fulfill the AVL property, all of its subtrees much fulfill the AVL property also.

For simplicity, the class `verifyAVL.Node` does not contain any keys or values, only links to children. Also, there is no "nully" object; the Java value `null` is used for empty (sub)trees.

Recursion is recommended. You may want to write one or more helper methods. Also, the `Node` class has a public instance variable `info` of type `Object`, where you can store any information you want about the node. (My own solution did *not* use the `info` instance variable. Also in my own solution `verifyAVL()` is not itself recursive, but called a recursive helper method.)

6. **[BSTs.]** Recall that any sorted array can be turned into a binary search tree. For example, if the length of the array were one less than a power of 2, then we could turn it into a balanced binary search tree this way:

5	8	12	15	23	25	29	31	32	38	43	49	56	58	63

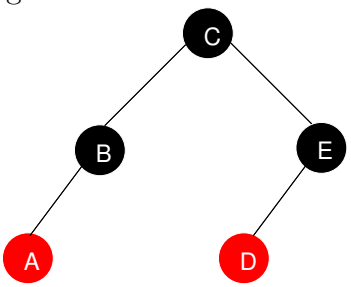


If the length of the array were *not* one less than a power of 2, we could convert the array into one of the varieties of nearly-balanced trees.

Consider the simplified implementation of red-black trees represented by the class `sorted2RB.RBNode`. This represents only the data (keys, links, and redness). Any operations such as lookup, insertion, and rotations would need to be implemented externally.

Complete the method `sorted2RB.Sorted2RB.sorted2RB()` which takes a sorted array of Strings and returns an equivalent red-black tree represented by its root, an instance of `RBNode`. For example, given the array on the left, `sorted2RB()` could return the tree on the right.

A	B	C	D	E



Recursion is recommended, though you may choose to make a recursive helper method rather than make `sorted2RB()` itself be recursive. Also, in the example above the red-black tree happens to be left-leaning, and in my own solution the resulting tree always is a left-leaning red-black tree. That's not required, though—your solution

could produce different trees from mine, as long as they are valid red-black trees, valid BSTs, and contain all and only the given keys.

Use `TestSTRB` to test.