CSCI 3901 Winter 2021

Assignment 1

Due date: 23:59 AST Friday, February 12th, 2021 in Brightspace

Problem 1

Goal

Get practice in writing test cases.

Problem

Write test cases for Alice's Sudoku class.

A Sudoku puzzle is a square grid of $n \times n$ sub-grids, each of which is itself a grid of $n \times n$ cells, where n is a positive integer. In total, there are $n^2 \times n^2$ individual cells. An example is shown in fig. 1. The goal of the puzzle is to enter the numbers 1 to n^2 into the cells of the grid such that:

- Every row contains all numbers from 1 to n^2
- Every column contains all numbers from 1 to n^2
- Every sub-grid contains all numbers from 1 to n^2

A slight generalization of Sudoku is to use a collection of n^2 distinct items instead of the numbers for 1 to n^2. For instance, in a 9×9 Sudoku, we can use the letters `a` to `i` to fill the cells instead of numbers. In the version of Sudoku we consider in this assignment, any set of n^2 unique characters will be allowed.

The Sudoku class Alice is tasked with writing has the following methods:

`public Sudoku(int size)` A constructor for a Sudoku grid with $\texttt{size}^2 \times \texttt{size}^2$ cells.

`public boolean setPossibleValues(String values)` Takes a string of length $\texttt{size}^2$ where the characters of `values` are the unique items allowed in the cells. Returns false if any error occurred and true otherwise.

`public boolean setCellValue(int x, int y, char letter)` Sets the value in cell `(x, y)` to the character `letter`.

`public boolean solve()` Solve the puzzle, returning true if a solution was found and false otherwise.

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Unsolved Sudoku

5	3	*4*	*6*	7	*8*	*9*	*1*	*2*
6	*7*	*2*	1	9	5	*3*	*4*	*8*
1	9	8	*3*	*4*	*2*	*5*	6	*9*
8	*5*	*9*	*7*	6	*1*	*4*	*2*	3
4	*2*	*6*	8	*5*	3	*7*	*9*	1
7	*1*	*3*	*9*	2	*4*	*8*	*5*	6
9	6	*1*	*5*	*3*	*7*	2	8	*4*
2	*8*	*7*	4	1	9	*6*	*3*	5
3	*4*	*5*	*2*	8	*6*	*1*	7	9

Solved Sudoku

Figure 1: Sudoku puzzle from https://en.wikipedia.org/wiki/Sudoku

`public String toPrintString(char emptyCellLetter)` Return a string representing the current state of the grid. The grid is returned with no spaces between cells, the character `emptyCellLetter` used for any empty cells, and lines separated by carriage returns (`\n`).

The normal course of operation is for a user to

1. Create a puzzle using the constructor
2. Define the symbols for the grid using `setPossibleValues`
3. Constrain the puzzle by setting some of the cell values using `setCellValue`
4. Solve the puzzle using the `solve` method

Notes

- You are **not asked to write code** to solve this problem.
- Only write test cases for this problem.
- **Do not provide test data for your test cases**. Do provide what the method is expected to return (qualitatively for `toPrintString`).
- I am looking for distinct test cases. Do not duplicate conditions across cases.
- Ensure that your test case description is short but also clear on what is being tested. Cases where we can't tell what is being tested will be discarded.

Marking scheme (out of 5)

- List of test cases, assessed based on completeness of coverage for the problem and distinctness of the cases – 5 marks

Problem 2

Goal

Implement a data structure from basic objects.

Background

Balanced binary search trees can be tricky to code and expensive to maintain. However, we don't always need a perfectly balanced tree. We are often content with an approximation to the tree or a heuristic structure that mimics the balanced binary tree.

In this assignment, you will implement a data structure that has the structure of an unbalanced binary search tree, but that is designed to make searches for frequently accessed items happen quickly.

Problem

Write a class called `SearchTree` that accepts and stores ordered keys (`String`s for this assignment), and then lets you search to see if a particular key is in the tree. The `SearchTree` should be programmed as an *unbalanced* binary search tree. New keys are added to the bottom of the tree, maintaining the structure of a binary search tree. Keys should only be stored in the tree once.

A counter is stored along with each key to keep track of how many previous times a key has been accessed. When a key is accessed (i.e. searched for), its counter is increased. If a key's count is higher than its parent in the tree, then this key is moved up one level in the tree and its parent is moved down a level. More details on this follow.

The public methods of the `SearchTree` class are as follows:

`SearchTree( )` Constructor.

`boolean add(String key)` Add the key to the tree. Return true if added. Return false if already in the tree or some problem occurs.

`int find(String key)` Search for `key` in the tree. If found, return the depth of the node in the tree (Note: the root node has depth 1). If not found or if some error happens, return 0.

`void reset()` Change all of the counters for searches in the tree to 0.

`String printTree()` Create a string of the tree's content. For each key in the tree (reported in sorted order), print the key, a space, and then the depth of the node in the tree. Separate each key with a carriage return (`\n`). Return a null string if any error occurs.

`find` operation

Searching in a binary search tree finds information that is closer to the root of the tree faster than information that is lower in the tree. Our `find` method will move items that we search for frequently closer to the root of the tree so that we can find them more quickly in later searches.

Suppose that we have a node with key `egg` and a child node `carrot` (see fig. 2). We have searched for `egg` twice and `carrot` once. When we search for `carrot` again, we increment its count to 2. When we search for `carrot` one more time, its count now becomes 3, which is more than its parent `egg`, so we want to move `carrot` up the tree by one level. Figure 2 shows how the tree is reshaped around this move. This reshaping must preserve the structure of the binary search tree. A consequence of the reshaping is that the sibling of `carrot` and all its descendants become further from the root.

The case where the searched node is the right child of the parent is symmetric.

A node is moved up at most one level per search, even if its new parent has smaller count than it.

Inputs

All the inputs will be determined by the parameters used in calling your methods.

Assumptions

No special assumptions provided for this assignment.

Constraints

- **You may not use any data structures from the Java Collection Framework, including ArrayLists.**
- If in doubt for testing, I will be running your program on `timberlea.cs.dal.ca`. Correct operation of your program shouldn't rely on any packages that aren't available on that system.
- String comparisons should **not** be case sensitive.

Notes

- Start with some basic methods. Get add and search (without the changes to the data structure) working.
- Your nodes for the binary tree will likely benefit from having a reference to the parent node. The root of the tree will have a null value for its parent.

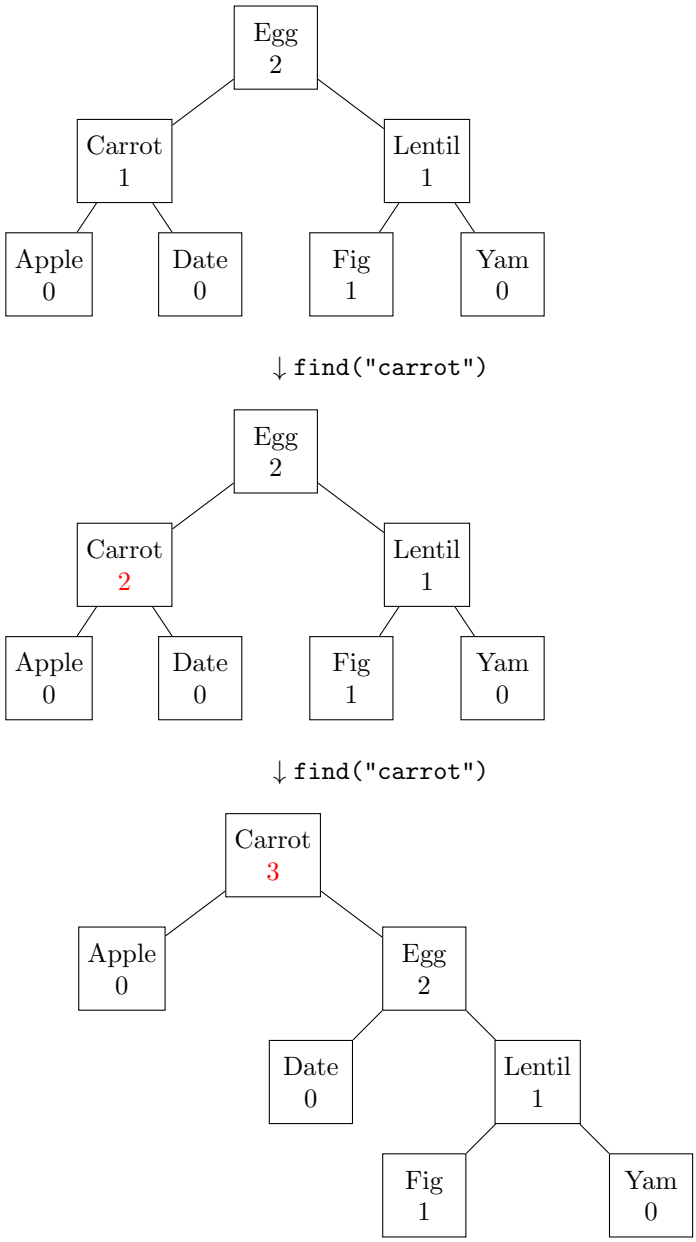

Figure 2: Sequence of `find` operations for `carrot` that eventually shift the tree structure

Marking scheme (out of 25)

- Documentation (internal and external) – 3 marks
- Program organization, clarity, modularity, style – 3 marks
- Ability to search for an item in the tree correctly – 5 marks
- Ability to add an item to the bottom of the tree correctly – 3 marks
- Ability to change the structure of the tree as we search for the same item repeatedly – 6 marks
- Ability to reset the counts for the nodes – 2 marks
- Ability to print the tree – 3 marks

Test cases

Input Validation

- Send a null string to the add method
- Send an empty string to the add method
- Send a null string to the add method
- Send a null string to the find method

Boundary Cases

- Add a string of 1 character
- Add a long string
- Find a string of 1 character
- Find a long string
- Reset an empty tree
- Reset a tree with 1 node
- Reset a big tree
- Print an empty tree
- Print a tree with 1 node
- Print a big tree

Control Flow

- `add`
  - Add to an empty tree
  - Add to a tree with 1 node
  - Add strings in alphabetical order
  - Add strings in reverse alphabetical order
  - Add a smallest string

  - Add a largest string
  - Add a string that forces each of the following search sequences:
    * Left child then left child
    * Left child then right child
    * Right child then left child
    * Right child then right child
  - Add an item that is already in the tree
- `find`
  - Find in an empty tree
  - Find in a tree with 1 node
  - Find the smallest string in the tree
  - Find the largest string in the tree
  - Find a middle string
  - Search for an item that requires us to follow:
    * Left child then left child
    * Left child then right child
    * Right child then left child
    * Right child then right child
  - Find an item that is in the tree
  - Find an item that is not in the tree
- `reset`
  - Covered under boundary cases
- `printTree`
  - Some covered under boundary cases
  - Print a tree that is a linked list
  - Print a tree that has many levels and has nodes with both right and left children

Data Flow

- Call find before calling add (mentioned earlier for empty tree)
- Call reset before calling add (mentioned earlier for empty tree)
- Call printTree before calling add (mentioned earlier for empty tree)
- Call reset twice in a row
- Call find for a string not in the tree, add it to the tree, then find it again