CSCI 3901 Assignment 2

Due date: 11:59pm Friday, January 31, 2020 in Brightspace

Problem 1

Goal
Get practice in writing test cases.

Problem
<u>Write test cases</u> for the following code.

A Sudoku puzzle is a square grid of cells n^2 x n^2 where n is a positive integer (for the purposes of this assignment). The goal of the puzzle is to enter the numbers 1- n^2 into the cells of the grid such that

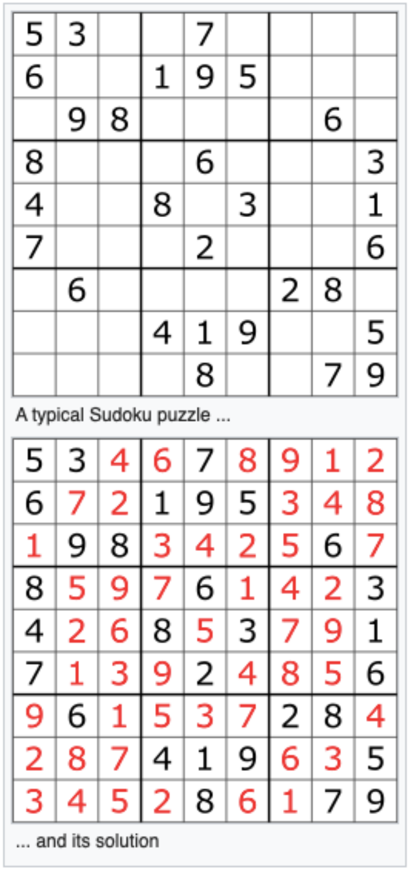
- Every row contains all numbers from 1- n^2
- Every column contains all numbers from 1- n^2.
- Every mini-grid contains all numbers from 1- n^2.

The mini-grid reference in this description is an n x n sub-grid of the larger puzzle and where n^2 mini-grids completely cover the grid (see Figure 1).

Notice that we don't make use of the fact that we are putting numbers into the cells. Our problem will let you put other characters into the cells, like letters. An implementation will have a method to let the puzzle know which characters (letters, numbers, or symbols) are allowed in the cells.

The implementation of a solution is a class called Sudoku with the following methods:

- A constructor for the grid that defines the number of rows and columns in the square puzzle:
  public Sudoku(int size)
- setPossibleValues to identify which characters can be stored in the puzzle, returning true when the values are set for th there is some error:
  public boolean setPossibleValues(String values)
- setCellValue to fill in some cell values before solving the puzzle
  public boolean setCellValue(int x, int y, char letter)
- solve to solve the puzzle, returning true if the method was able to find a solution to the puzzle and false if there is no solution or if there is some error in the processing:
  public boolean solve( )



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

A typical Sudoku puzzle ...

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	7
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

... and its solution

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

- toPrintString to get a printable version of the puzzle where we can specify the character to place in unfilled cell values; a string of the grid is returned, with no spaces between characters and with lines separated by carriage returns (\n):
  public String toPrintString(char emptyCellLetter)

The normal flow 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, to not force you to write or create Sudoku puzzles).
- 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

- 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 data values and then lets you search the list to see if a value is in the list. The key part of the class is in the data structure that it uses to store the data.

At its core, the SearchTree is an unbalanced binary search tree. New values are added to the bottom of the tree. Values should only be stored in the tree once.

The important feature to the tree is in the search method. In addition to searching for an item in the tree, we store the number of times that an item has been searched for previously. When find the item, we compare the number of searches for the item with the number of searches for the parent. If the item is searched for more than its parent then we move that item up one level in the tree and the parent goes down a level. We elaborate on this action later in this assignment.

The set of methods 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 (with the root as 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

We find 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 Figure 2). We have searched for "egg" twice and "carrot" once. When we search for "carrot" again, we increment its count to having been searched for twice. 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. 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.

When we promote a node up the tree, we only move it one level at a time, even if its new parent has a search count that is smaller than the node. This action is just for simplicity.

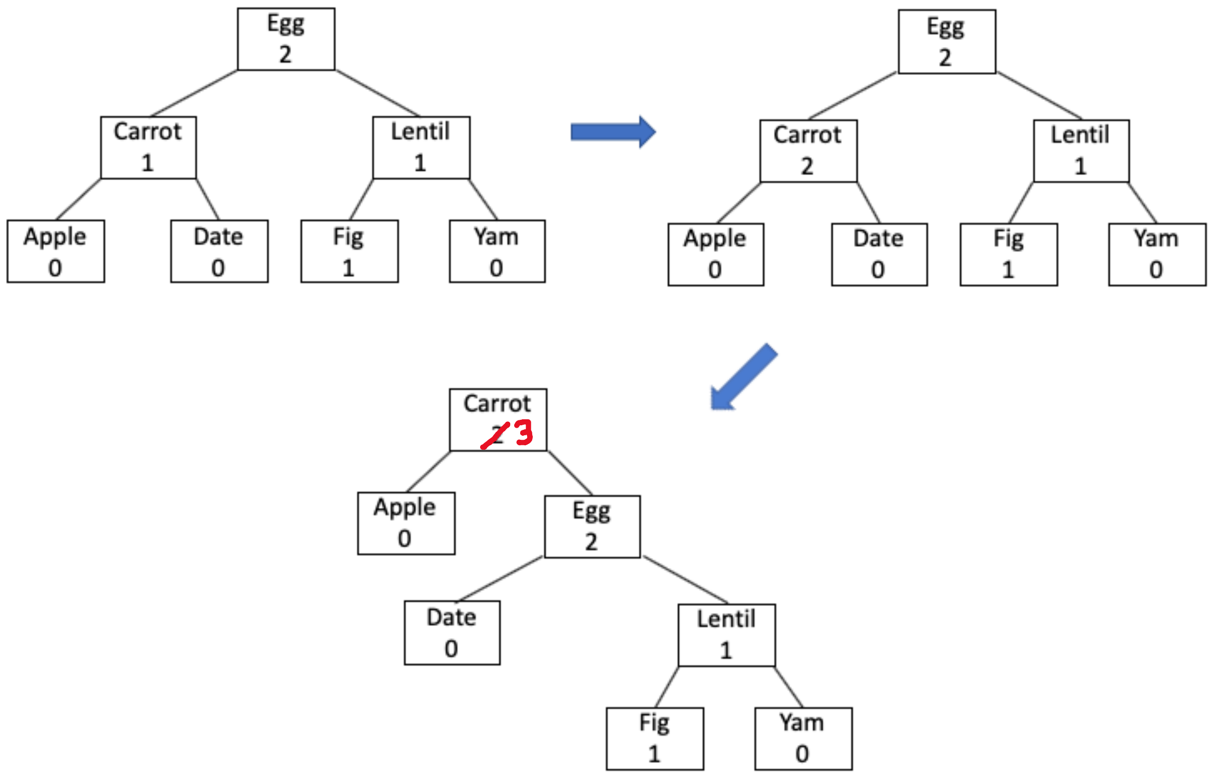


Figure 2 sequence of "find" operations for "Carrot" that eventually shift the tree structure

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 <u>not</u> use any data structures from the Java Collection Framework, including ArrayLists.
- If in doubt for testing, I will be running your program on bluenose.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.

Marking scheme

- Documentation (internal and external) – 3 marks
- Program organization, clarity, modularity, style – 3 marks
- Ability to add an item to the bottom of the tree correctly – 3 marks
- Ability to find an item without changing the structure of the tree – 0 marks (code from class)
- 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