CSCI 3901 Assignment 2

Due date: 11:59pm Friday, February 17, 2023 in Brightspace

Problem 1

Goal

Get practice in writing test cases.

Problem

Write test cases 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( )

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 somewhat mitigates bad insertion orders that would completely unbalance the tree.

Problem

Write a class called “AmortizedTree” 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 AmortizedTree is an unbalanced binary search tree. The tree actually has two underlying data structures: the unbalanced tree and an array of values that are waiting to be added to the unbalanced tree.

New values are added to the array of values. Once that array reaches a sufficient size, we insert all the value of the array into the tree. Since we have all the values in the array, we can select an order of insertion that avoids the worst case unbalancing of the tree. For example, if the array values are 7, 12, 18, 23, and 45 then inserting the values in the order 17, 12, 18, 23, 45 unbalances the tree while inserting in the order 18, 7, 12, 23, 45 has a better chance of keeping the tree balanced. The idea is then to insert the middle element first and then the middle elements of what is left to either side of that element.

The size of the array of values awaiting to be added into the unbalanced tree should be the ceiling of the logarithm (base 2) of the number of values in the unbalanced tree. Start the array with size 1. When the unbalanced tree has 3 values, the array can hold 2 values. When the unbalanced tree has 5 values, the array can hold 3 values. When the unbalanced tree has 9 nodes, the array can hold 4 values. For simplicity, if elements are deleted from the unbalanced tree, you do not need to shrink the array.

If well-implemented, adding all values of the array to the unbalanced tree should result in at most one comparison with each node of the unbalanced tree. (That trick may take some thought…)

Values should only be stored in the tree once.

When we search the AmortizedTree, a value could be found in either the array or the unbalanced tree.

The class implements two interfaces: Searchable and TreeDebug. The Seachable interface contains methods we would expect of any tree while the TreeDebug interface contains methods that are useful for debugging the tree.

The Searchable interface methods are as follows:

- 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.

- boolean find(String key) – search for “key” in the tree. If found, return true. If not in the AmortizedTree then return false.

- boolean remove(String key) – remove the “key” value from the AmortizedTree. Return true if the value was in the tree and is not in the tree after the operation. Otherwise, return false.

- int size() – return the number of values stored in the AmortizedTree.

- boolean rebalance() – restructure the tree so that is as balanced as possible. That balancing should have all the leaves on the same level (or off by one level) and all the leaves on a level should be as far to the left of the tree as possible. Return true if the tree has been rebalanced. Return false otherwise (as in the case of an error).

- boolean rebalanceValue(String key) – as with rebalance(), but with “key” as the root value of the unbalanced tree.

The TreeDebug interface methods are as follows:

- String printTree( ) – create a string of the tree’s content. For each value in the tree (reported in sorted order), print the value, a space, and then the depth of the node in the tree. Separate each value-depth pair with a carriage return (\n). Return a null string if any error occurs.

- String[] awatingInsertion() – returns an array of all the values that are waiting to be added into the unbalanced tree. The values should be in ascending sorted order. Return null in the case of an error.

- String[] treeValues() – returns an array of all the values that are in the unbalanced tree. The values should be in ascending sorted order. Return null in the case of an error.

- int depth(String key) – return the level in the AmortizedTree where the value is found. The root is at depth 1, its children are at depth 2, and so on. If the key is in the array to be inserted into the tree, return its index in the array times -1. If the key is not in the AmortizedTree, return a value greater than the size of the number of values in the tree.

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.

Marking scheme

- Documentation (internal and external) – 3 marks
- Program organization, clarity, modularity, style – 3 marks
- Ability to add an item to the tree correctly – 3 marks
- Ability to find an item – 2 marks (no marks for the functionality of the code from class, but you can use it)
- Ability to merge the contents of the array into the unbalanced tree effectively – 6 marks
- Ability to remove an item – 4 marks
- Ability to rebalance the tree – 4 marks
- Implementation of TreeDebug interface – 4 marks

Test cases

The TreeDebug methods will be used more in the test cases to determine if your code not only did the operation but did it in the way specified.

add

Input validation

- Key value is null
- Key value is an empty string

Boundary case

- Add to an empty AmortizedTree (should go into array)
- Add one element to an AmortizedTree with one element in it (should overflow the array and go to the tree)
- Add 4 elements to force the array to grow by 1 element, then add elements to fill the grown array

Control flow

- Add the same string twice
- Add strings in reverse order to force the tree to be one-sided, but not too badly unbalanced

- Add strings in sorted order to force the tree to be one-sided, but in the opposite direction, and not too badly unbalanced.

Data flow

- Add elements to grow the array, remove all elements from the unbalanced tree, then add elements to verify that the array only grows once the binary tree is back to a large size.

find

Input validation

- Key value is null
- Key value is an empty string

Boundary case

- Call find on an empty AmortizedTree
- Call find on an element in the array when the array has a single entry in it
- Call find on an element in the array when the array has many elements and the element to be found is the last in the array
- Call find when an element to be found is the root of the unbalanced tree
- Call find when an element to be found is the smallest element of the unbalanced tree
- Call find when an element to be found is the largest element of the unbalanced tree

Control flow

- Call find on an element in the middle of the array
- Call find on an element that is some intermediate node of the unbalanced tree
- Call find on an element not in the tree or the array

Data flow

- Call find before any items are added to the AmortizedTree
- Call find after all elements in AmortizedTree have been removed

remove

Input validation

- Key value is null
- Key value is an empty string

Boundary case

- Call remove on an empty AmortizedTree
- Call remove on an element in the array when the array has a single entry in it
- Call remove on an element in the array when the array has many elements and the element to be removed is the last in the array
- Call remove when an element to be removed is the root of the unbalanced tree
- Call remove when an element to be removed is the smallest element of the unbalanced tree

Control flow

- Call remove on an element in the middle of the array
- Call remove on an element that is some intermediate node of the unbalanced tree

- Call remove on an element not in the tree or the array

Data flow

- Call remove before anything is added to the AmortizedTree

size

Input validation

- none

Boundary case

- call size on an empty AmortizedTree

Control flow

- call size when there are just elements in the array
- call size when there are just elements in the unbalanced tree
- call size when there are elements in both the array and the unbalanced tree

Data flow

rebalance

Input validation

- none

Boundary case

- rebalance an empty AmortizedTree
- rebalance an AmortizedTree with just one element, currently in the array
- rebalance an AmortizedTree with just one element in the unbalanced tree

Control flow

- rebalance a tree where the root has no left children
- rebalance a tree where the root has no right children
- rebalance a tree when the array is empty
- rebalance a tree when the array contains strings

Data flow

- rebalance a tree before any items are added

rebalanceValue

Input validation

- Key value is null
- Key value is an empty string

Boundary case

- rebalance an empty AmortizedTree
- rebalance an AmortizedTree with just one element, currently in the array
- rebalance an AmortizedTree with just one element in the unbalanced tree

Control flow

- rebalance a tree when the key to rebalance around isn't in the data structure
- rebalance a tree when the key to rebalance around is in the array
- rebalance a tree when the key to rebalance around is already the root of the unbalanced tree
- rebalance a tree when the key to rebalance around is inside the unbalanced binary tree
- rebalance a tree when the key to rebalance around is a leaf of the unbalanced binary tree
- rebalance the tree when the key is the smallest element in the AmortizedTree
- rebalance the tree when the key is the largest element in the AmortizedTree

Data flow

printTree

Input validation

- none

Boundary case

- print an empty tree
- print a tree with just the root

Control flow

- print a tree that is has more than 1 level

Data flow

awaitingInsertion

Input validation

- none

Boundary case

- print the list when the array is empty

Control flow

- print the list when the array has 1 entry
- print the list when the array has size > 2 and is full
- print the list when data was inserted in reverse alphabetical order

Data flow

treeValues

Input validation

- none

Boundary case

- print an empty tree
- print a tree with just one element in it

Control flow

- print a tree that has several levels

Data flow

depth

Input validation

- Key value is null
- Key value is an empty string

Boundary case

- Ask for the depth of the root
- Ask for the depth of the deepest leaf

Control flow

- Ask for the depth of a middle node in the left side of the root
- Ask for the depth of a middle node in the right side of the root
- Ask for the depth of an item that is in the array
- Ask for the depth of a key that is not in the AmortizedTree

Data flow