CS1027

ASSIGNMENT 2

Computer Science Fundamentals II

This assignment is due on Wednesday, June 30, 2021 by 11:55pm. See the bottom for submission details.

Learning Outcomes

To gain experience with

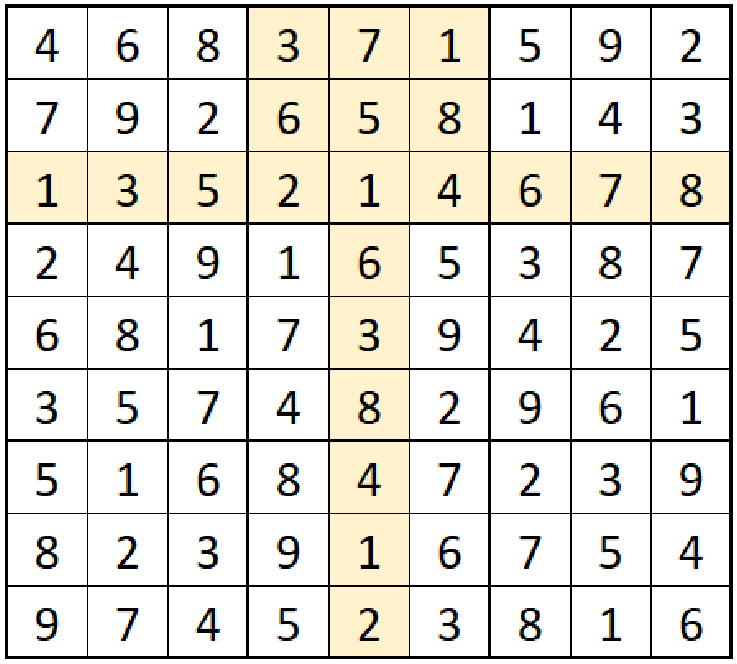
- Creating custom collections
- Working with linked data structures
- Algorithm design

Introduction

Sudoku is a popular type of puzzle in which numbers are filled into a 9 by 9 grid. The rules are that each number from 1 to 9 must be used exactly once (no duplicates) in each row, column, and 3 by 3 box within the grid. A Sudoku is only valid and solved properly if that requirement is true for the entire grid.

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

Figure 1. An example of a valid Sudoku.



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

Figure 2. An example of an invalid Sudoku. Row 3, Column 5, and Box 2 each contain two 1's and no 9's. Cells are highlighted to show these errors.

In this assignment, you will be creating a program that checks to see if a given Sudoku is valid based on the rule stated above. The Sudoku will be implemented as an array of linked data structures for each of the rows. To clarify, there must be an array with 9 slots and each of those slots must contain a LinearNode which is the leftmost cell of that row.

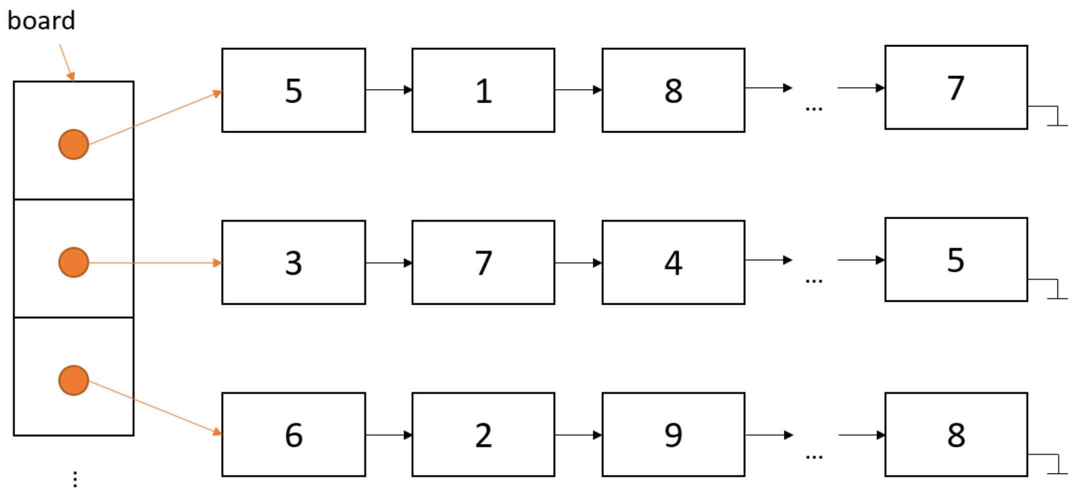


Figure 3. A depiction of the Sudoku board as an array that contains LinearNode elements and

each of those LinearNodes are the leftmost cell in that row. The remainder of each of the rows are connected one by one from the LinearNode objects.

Provided files

The following is a list of files provided to you for this assignment. Please do not alter these files in any way.

- LinearNode.java
- TestUniqueArray.java
- TestSudoku.java

Classes to implement

For this assignment, you must implement 2 Java classes: *UniqueArray* and *Sudoku*. Follow the guidelines for each one below.

In all of these classes, you can implement more private (helper) methods, if you want to, but you may **not** implement more public methods. You may **not** add instance variables other than the ones specified below nor change the variable types or accessibility (i.e. making a variable public when it should be private). Penalties will be applied if you implement additional instance variables or change the variable types or modifiers from what is described here.

UniqueArray.java

This class is a collection that we haven't discussed in lecture but it's something you've likely seen before in other contexts (i.e. a Set in mathematics). This collection is a regular array but with the requirement that duplicate elements are not allowed. This means, when attempting to add a new item, you must check if that element already exists in the array and only add the item if it does not exist there already. Note: this class must work with generic type T.

The class must have the following *private* variables:

- array (T[])
- num (int)

The class must have the following *public* (unless otherwise stated) methods:

- public UniqueArray() [constructor]
  - Initialize the array with 5 slots. Use: array = (T[])new Object[5];
  - Set num to 0
- private void expandCapacity()
  - Increase the capacity of the array with 5 additional slots.
- public void addItem(T element)
  - Check if the given element is already within the array. If it is, do nothing. If not, then add the item to the array. You will also have to check if the array is full when adding the new item, and if so, call expandCapacity().
  - Increment num if the item was added.
- public int getLength()
  - Returns the array length
- public int getNumElements()
  - Returns the number of elements in the array
- public String toString()
  - Return a string containing each of the elements in the array separated with a comma and a space, but there should not be any comma or space at the end (i.e. 24, 59, 15, 47)

Sudoku.java

This class represents the Sudoku as an array of linked data structures. This is also where we will be checking the validity of the Sudoku.

The class must have just one *private* variable:

- board (LinearNode<Integer>[])

The class must have the following *public* methods:

- public Sudoku(int[][]) [constructor]
  - Take in a 2D int array containing a Sudoku's values. These values from the array will need to be entered into the array of linked data structures (board).
  - Initialize the board with 9 LinearNode slots.
  - Loop through the 9 slots and initialize them with the values from column 1 of the 2D array. Use a nested loop to create the entire row as a linked structure stemming from the front cell that is stored in the board array. As you create each node, insert into it the value from the 2D array at the corresponding row,col cell.
- public boolean isValidRow(int)
  - Take in a row number from 1 to 9 representing which row is being checked (note: you do not have to handle exceptions by checking if the row is within 1-9, but you may do this if you would like). Recall that the array indices will be 0-8 so you will have to adjust the index.

- Get the corresponding row from the board and loop through the elements and check if it contains all 9 unique numbers from 1 to 9. If so, return true. Otherwise, return false.
    - Hint: the UniqueArray class is very useful here! Although you are not required to do it this way if you can think of an alternate approach.
- public boolean isValidCol(int)
    - The same as isValidRow() except that the parameter represents the column number from 1 to 9, so check if the corresponding column has 9 unique numbers from 1 to 9.
- public boolean isValidBox(int, int)
    - Take in two numbers representing the row and col indices in which the 3x3 box begins. These numbers must be from 1 to 7 (not 9 since the box must go 2 cells down and to the right from the starting cell).
    - Check if the 9 elements in this 3x3 box are the unique digits from 1 to 9. Return true if they are; otherwise return false.
- public boolean isValidSolution()
    - Check if the entire Sudoku is valid by calling the previous 3 methods for ALL rows, columns, and 3x3 boxes in the board. If all 27 of them return true, then this method should also return true. If not, return false here (hint: use loops to avoid writing 27 individual method calls!)
- public String toString()
    - Loop through the Sudoku structure and build a string with all the values in a well-formatted grid.
    - Add 2 spaces after each cell number (but there should not be any spaces at the end of a line), and add a newline character at the end of each line (except the last one) so that the string is formatted like a grid.
    - Return the string that contains this entire grid-like representation of the board.

Marking notes

Marking categories

- Functional specifications
    - Does the program behave according to specifications?
    - Does it produce the correct output and pass all tests?
    - Are the class implemented properly?
    - Are you using appropriate data structures (if applicable)?
    - Does the code run properly on Gradescope (even if it runs on Eclipse, it is up to you to ensure it works on Gradescope to get the test marks)
- Non-functional specifications
    - Are there comments throughout the code (Javadocs or other comments)?

  - Are the variables and methods given appropriate, meaningful names?
  - Is the code clean and readable with proper indenting and white-space?
  - Is the code consistent regarding formatting and naming conventions?
- Penalties
  - Lateness: 10% per day
  - Submission error (i.e. missing files, too many files, etc.): 5%
  - "package" line at the top of a file: 5%
  - Compile or run-time error on Gradescope
  - Failure to follow instructions, (i.e. changing variable types, etc.)

Remember you must do all the work on your own. Do not copy or even look at the work of another student. All submitted code will be run through cheating-detection software.

Submission (due Wednesday, June 30, 2021 at 11:55pm ET)

Assignments must be submitted to Gradescope, not on OWL. If you are new to this platform, see these instructions on submitting on Gradescope.

Rules

- Please only submit the files specified below. Do not attach other files even if they were part of the assignment.
- Do not upload the .class files! Penalties will be applied for this.
- Submit the assignment on time. Late submissions will receive a penalty of 10% per day.
- Forgetting to submit is not a valid excuse for submitting late.
- Submissions must be done through Gradescope and the test marks come directly from there. If your code runs on Eclipse but not on Gradescope, you will NOT get the marks! Make sure it works on Gradescope to get these marks.
- Assignment files are NOT to be emailed to the instructor(s) or TA(s). They will not be marked if sent by email.
- You may re-submit code if your previous submission was not complete or correct, however, re-submissions after the regular assignment deadline will receive a penalty.

Files to submit

- UniqueArray.java
- Sudoku.java

Grading Criteria

- Total Marks: [20]
- Functional Specifications:
  - [3] UniqueArray.java
  - [3] Sudoku.java
  - [10] Passing Tests
- Non-Functional Specifications:
  - [1.5] Meaningful variable names, private instance variables
  - [0.5] Code readability and indentation
  - [2] Code comments