List of preparatory/practice questions for the MidTerm Exam

The MidTerm Exam questions are different from the ones on this list.

Read and understand the code provided – it can help you during the midterm exam. Do not limit yourself to this list of questions, create your own variations and practice!

Use the Test files to verify that your solution is working as expected (add other test cases).

Read the comments in the code, they may help you understand the code and develop your solution.

Again, it is important to note that the midterm exam questions are different from the ones on this list.

Suggestion: REDO ALL LABS as well!

Q1 – BST

Given a binary search tree, implement a method to find the sum of the values of all the nodes that have an odd number of direct children. You can define additional methods of BST and Node classes to complete the task. The method signature is:

public Integer oddNodeSum()

Read and understand the code.

Q2 – BST

Given a binary search tree, implement a method to find the sum of the values of all the nodes that have an even number of direct children. Note that 0 is also an even number. You can define additional methods of BST and Node classes to complete the task. The method signature is:

public Integer evenNodeSum()

Read and understand the code.

Q3 - Red-Black Tree

Implement a checking function in Red-Black Tree to check if the following properties hold:

1. The root and leaf (NIL) nodes are black
2. If a node is red, then its parent is black

You can define additional methods if you need to implement testProp1 and testProp2 methods. The methods signatures are:

public boolean testProp1()
public boolean testProp2()

Read and understand the code.

Q4 - Red-Black Tree

Implement a function in Red-Black Tree to check if the following property hold:

1. All simple paths from any node x to a descendant leaf have the same number of black nodes. You can define additional methods if you need.

The methods signature is:

public boolean testProp3()

Read and understand the code.

Q5 – BST

This question is composed of two items:

- A Binary Search Tree Question (item 1.a) and
- A Branch Complete Question (item 1.b).

Please upload the following files to Wattle after you complete the tasks: `BST.java` and `BSTBranchCompleteTest.java`.

1.a) A Binary Search Tree (BST) is used to **generate a sequence of bases** (A, G, C, T) in a DNA molecule. The sequence is generated based on the information stored in each node (character **base**, see node representation) and the following rules:

1. All bases of all nodes that have an odd number of direct children under the key node (inclusive) are concatenated;
2. If the given key is not found, return "CGTA";
3. If the given key is found, but rule 1 is not satisfied, return **null**.

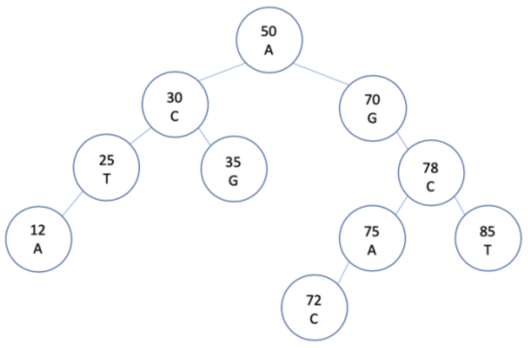
Note that the order in which the nodes are concatenated is irrelevant. As long as the rules are met, the sequence of bases generated will be considered valid.

You can define additional methods in the BST class to complete the task.

The method signature (the method that will be tested) is:

public String DNAGenerator(Integer key)

Example:



Example 1:
Key: 25
Result: T
This example includes the key node 25-T (rule 1), but not the node 12-A as it has no children.

Example 2:
Key: 30
Result: T
In this example only the key node 25-T (rule 1) has an odd number of direct children.

Example 3:
Key: 17
Result: CGTA
In this example the key node is not found (rule 2).

Example 4:
Key: 70
Result: AG (or GA, any possible combination is valid as long as it is correct)
In this example the key node 70-G has an odd number of direct children, and the node 75-A, under the key node, has an odd number of direct children.

Example 5:
Key: 12
Result: null
In this example the key node 12-A has no children (rule 3).

1.b) Implement the minimum number of JUnit test cases for **DNATreeCalc()** that is branch complete. Read the code of **DNATreeCalc()** in the `BranchComplete.java` file and implement the **minimum number of test cases** in the `BSTBranchCompleteTest.java` file.

Note that each execution of an assertion counts as a single test case, therefore loops that execute the same assertion multiple times count as multiple tests.

Q6 – Design Pattern

The given code realises a GreenGrocery using the IteratorDesignPattern. The green grocery has an iterator that can iterate all the fruits that are restocked.

Please refer to the test cases in GroceryTest.java for more details.

You are expected to complete:

- restock(), hasNext(), next() in GreenGrocery.java file

Q7 – Design Pattern

The given code realises a Calculator using the Façade Design Pattern. The calculator can perform four basic arithmetic operations:

- Addition
- Subtraction

- Multiplication
- Division

Please refer to the test cases in CalculatorTest.java for more details.

You are expected to complete:

- add(), subtract(), multiply(), divide() in Calculator.java file
- evaluate() in Division.java file

You should call the evaluate() methods in the basic arithmetic operation classes to achieve add(), subtract(), multiply(), divide() for the Calculator. You are not allowed to directly perform computations in these functions.

PS: Think about what is considered your “complex” system in this calculator to comply to the façade design pattern.