COMP 2402 AB- Fall 2023
Assignment #3

Due: Wednesday, November 8, 23:59

Submit early and often. Late submissions (up to 12 hours) will be accepted.

Academic Integrity

You may:

- Discuss general approaches with course staff and your classmates,
- Use code and/or ideas from the textbook,
- Use a search engine / the internet to look up basic Java syntax.

You may not:

- Send or otherwise share code or code snippets with classmates,
- Use code not written by you, unless it is code from the textbook (and you should cite it in comments),
- Use a search engine / the internet to look up approaches to the assignment,
- Use code from previous iterations of the course, unless it was solely written by you,
- Use the internet to find source code or videos that give solutions to the assignment.

If you ever have any questions about what is or is not allowable regarding academic integrity, please do not hesitate to reach out to course staff. We will be happy to answer. Sometimes it is difficult to determine the exact line, but if you cross it the punishment is severe and out of our hands. Any student caught violating academic integrity, whether intentionally or not, will be reported to the Dean and be penalized. Please see Carleton University's Academic Integrity page.

Grading

This assignment will be tested and graded by a computer program (and **you can submit as many times as you like; your highest score is counted towards your grade for the assignment**). For this to work, there are some important rules you must follow:

- Keep the directory structure of the provided **zip** file. If you find a file in the subdirectory `comp2402a3`, leave it there.
- Keep the package structure of the provided **zip** file. If you find a package `comp2402a3` directive at the top of a file, leave it there.

- Do not modify any of the provided interfaces.
- Do not rename/delete any of the supplied files.
- Do not rename or change the visibility of any methods already present. If a method or class is public, leave it that way.
- Submit early and often. The submission server compiles and runs your code and gives you a mark. You can submit as often as you like; only your best submission will count. There is no excuse for submitting code that does not compile or does not pass tests.
- Write efficient code. The submission server places a limit on how much time it will spend executing your code, even on inputs with a million lines. For some questions it also places a limit on how much memory your code can use. If you choose and use your data structures correctly, your code will easily execute within the time limit. Choose the wrong data structure or use it the wrong way, and your code will be too slow for the submission server to grade (resulting in a grade of 0).

Submitting and Testing

For every assignment on Brightspace, you will find a URL "Assignment # submission server" (replace # with the corresponding assignment number) that will take you to the submission page. For more details on how to submit your work, follow the instructions on the document "Submission instructions". If you have issues, please post to Discord to the teaching team (or the class), and we'll see if we can help.

Warning: Do not wait until the last minute to submit your assignment. There is a hard 5-second limit on the time each test has to complete. For the largest tests, even an optimal implementation takes 3 seconds and may take longer if the server is heavily loaded.

Start by downloading and decompressing the Assignment 3 Zip File (comp2402a3.zip), which contains a skeleton of the code you need to write. The skeleton code in the **zip** file compiles fine. Here's what it looks like when you unzip and compile it from the command line:

```
alina@euclid:~$ unzip comp2402a3.zip
Archive:  comp2402a3.zip
  inflating: comp2402a3/SSet.java
  inflating: comp2402a3/IndexedSSet.java
  inflating: comp2402a3/SkippitySlow.java
  inflating: comp2402a3/SkippityFast.java
  inflating: comp2402a3/DefaultComparator.java
  inflating: comp2402a3/BinaryTree.java
  inflating: comp2402a3/Tester.java
alina@euclid:~$ javac comp2402a3/*.java
```

For this assignment you need to implement `SkippityFast` and `BinaryTree`. The `Tester` class, included in the zip file gives a very basic demonstration of the code. Despite the name, `Tester` does not do thorough testing. The submission server will do thorough testing. To run the `Tester` from the command line: `alina@euclid:~$ java comp2402a3.Tester`

The Assignment

This assignment contains two main parts:

Part 1 [40 marks] – extension of the SSet interface that supports two extra operations
The `IndexedSSet` interface is an extension of the `SSet` interface described in the textbook that supports two extra operations

- `get(i)`: return the value at index `i`, in other words, return the value `x` in the set that is larger than exactly `i` other elements in the set.
- `rangecount(x, y)`: return the number of elements in the set that are between `x` and `y` inclusive (`x` is not necessarily smaller than `y`).

Currently there are two identical implementations of the `IndexedSSet` interface using **skiplists** included in the assignment called `SkippitySlow` and `SkippityFast`. Both of them are slow: each of them has an implementation of `get(i)` and `rangecount(x, y)` that runs in $\Omega(n)$ time, in the worst case.

Modify the `SkippityFast` implementation so that the `get(i)` and `rangecount(x, y)` implementations each run in $O(logn)$ expected time and none of the other operations (`add(x)`, `remove(x)`, `find(x)`, etc.) have their running-time increased by more than a small constant factor. Look at the `SkiplistList` implementation discussed in class for inspiration on how to achieve this.

Testing: For this assignment, you will have to **test your own code thoroughly**. The submission server will perform a range of tests on your implementation. In case your code throws an exception, the server will tell you which method threw the exception and what kind of exception it was. Other than that, the server will consume anything that you try to print to `System.out` or `System.err`, so the testing done on the server will be almost completely opaque. It will only give you a generic error of the form "get(i)/ rangecount()/… returns incorrect value", or "Program timed out". On the positive side, testing is something you can do easily because `SkippitySlow` is a correct implementation that you can use to test against. Be sure to test a good mix of operations that includes and interleaves `add(x)`, `remove(x)`, `get(i)`, `find(x)`, `size()`, and `rangecount(x, y)`.

Part 2 [60 marks] – Binary Tree Traversals and extra operations

For this part of the assignment, you will work with the `BinaryTree` class provided in the zip file. It contains four uncompleted functions, which you are supposed to complete. For full marks, each of these functions should run in $O(n)$ time and **must not use recursion**. See the function `traverse2()` in `BinaryTree.java`, which we also discussed in class, for an example of how to do tree traversal without recursion. Note that this is just one example, and there could be other functions that you may take inspiration from.

You need to implement the following:

1. The method `leafAndOnlyLeaf()` should return the number of leaves in the tree, or `0` if the tree has no nodes at all. (A leaf is a node that has no left child and no right child.)
2. The method `dawnOfSpring()` should return the smallest depth in the tree that has leaves in it, or `-1` if the tree has no nodes.
3. The method `monkeyLand()` should return the number of nodes in the most populated depth (the depth where there is the largest number of nodes). If there are multiple depths with the maximum number of nodes, it should return the smallest of them.
4. The method `bracketSequence()` should return a string that gives the dot-bracket representation of the binary tree. The dot-bracket representation of a binary tree can be defined as follows:
    - the dot-bracket representation of a tree with no nodes is the string "."
    - the dot-bracket representation of a binary tree with root node `r` consists of an open bracket `(` followed by the dot-bracket representation of `r.left` followed by the dot-bracket representation of `r.right` followed by a closing bracket `)`

Some examples:

- the dot-bracket representation of the binary tree with only one node is `(..)`
- the dot-bracket representation of a 2-node binary tree is either `((..).)` or `(.(..))` depending on whether the root has no right child or no left child
- the dot-bracket representation of the height-1 binary tree with two leaves is `((..)(..))`

Testing: The `BinaryTree` class implements a static method called `randomBST(n)` that returns a random-looking binary tree. `BinaryTree` also implements the `toString()` method, so you can use `System.out.println(t)` to view a representation of a `BinaryTree` that is closely related to the method `bracketSequence()`. You can use this for testing your functions.

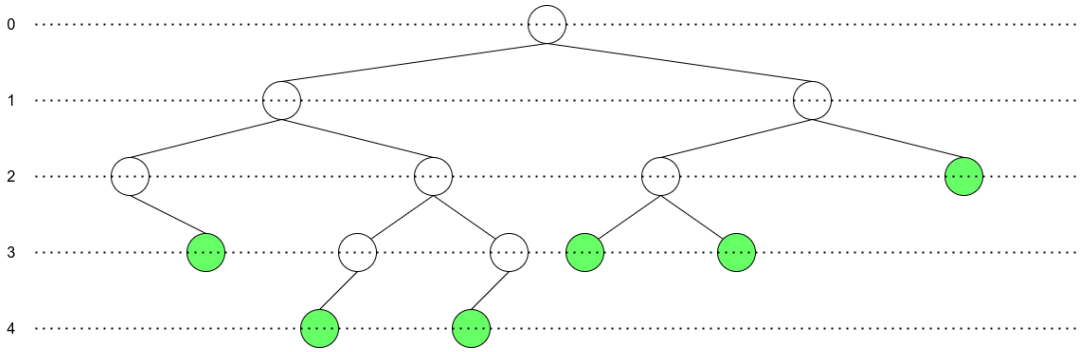
You should test your own code thoroughly since the testing done on the server will be mostly opaque. On the server, your code will be tested on a Java virtual machine with a limited stack size. You can do similar testing yourself by using the `java -Xss` argument (although you won't have to worry about this if you don't use recursion).

Tips, Tricks, and FAQs

How should I approach each problem?

- For part 1, it's best first to study how random access can be performed on skiplists (specifically, ODS section 4.3) and then attempt to solve it. Also, look at the skeleton code and try to understand how various operations work. It's okay if you do not understand every detail of the code as long as you understand enough to do what you need to do.
- For part 2, you can look up the ODS textbook on how binary trees can be traversed in various ways without recursion (specifically, ODS section 6.1.2). You can find these traversals already

implemented in the skeleton code as well. If you are unsure what the operations are supposed to do, the following figure might help you understand them better.



leafAndOnlyLeaf() = Number of leaves (marked green) = 6
dawnOfSpring() = Smallest level where there are leaves (level 2) = 2
monkeyLand() = Number of nodes in the most populated level (level 3) = 5

- Make sure you understand what the operations do. Construct **small** examples and compute (by hand) the expected output. If you aren't sure what the output should be, go no further until you get clarification.
- Now that you understand what you are supposed to output, and you've been able to solve it by hand, think about how you solved it and whether you could explain it to someone. How about explaining it to a computer?
- If it still seems challenging, what about a simpler case? Can you solve a similar or simplified problem? Maybe a special case? If you were allowed to make certain assumptions, could you do it then? Try constructing your code incrementally, solving the smaller or simpler problems, and then only expanding the scope once you're sure your simplified problems are solved.

How should I test my code?

- You can modify the `Tester` class. For example, you can change the "20" in `skippityTest(20);` to a smaller/bigger number to test less/more operations.
- The `Tester` class provided with the assignment does a sequence of `add`s, followed by a sequence of `get`s and `rangecount`s. This is a very basic test and far from an exhaustive one. In fact, the provided tester is meant to be more of a user guide to get started with the skeleton code and by no means, a representative of the tests performed on the server side. Design your own test cases, and throw in a good mix of `add`, `remove`, `get`, `find`, `rangecount`, and `size`. Try to break your solution with the tester in every way you can think of. And, of course, don't forget to test with large test sizes.
- You should be testing your code as you go along. The slow (but correct) solutions can always be taken as a reference when testing.
- Use small tests first so that you can compute the correct solution by hand.
- You will be dealing with pointer manipulation. Beware of null pointer exceptions.

- If you have print statements in your program that use `System.out`, the server will execute those statements but ultimately will remove the output generated by those statements from the response. You probably already know by now how console output can be painfully slow. Even worse, it could cause the testing program to crash if the output is large. So, be sure to remove such statements you may have used for debugging or any other reason before submitting your work.
- Test for speed.