Programming Assignment #4: SkipLists

COP 3503, Spring 2020

Due: Sunday, March 1, *before* 11:59 PM

Abstract

In this assignment, you will implement probabilistic skip lists. You will gain experience working with a probabilistic data structure and implementing reasonably complex insertion and deletion algorithms. You will also solidify your understanding of generics in Java by making your skip lists capable of holding any type of Comparable object. When you're finished, you'll have a very powerful and well-constructed container class.

Don't be too intimidated by this assignment (unless that's what it takes to get you to start early, in which case you should be **VERY INTIMIDATED!**). This will be a big challenge – probably our most challenging assignment this semester – but it's very much within your grasp. I've already done a lot of the heavy lifting for you by giving you the basic structure of the program and creating lots of test cases, so you get to focus on the fun part, which is taking your knowledge of how the insertion, deletion, and search algorithms for skip lists work, and translating them into code.

Deliverables

SkipList.java

Note! The capitalization and spelling of your filename matter!

Note! Code must be tested on Eustis, but submitted via Webcourses.

1. Preliminaries

In this assignment, you will implement the probabilistic skip list data structure. It is important that you implement the insertion, deletion, and search operations as described in class. Also, your skip list class must be generic. Since there is an ordering property in skip lists, you must also restrict the type parameter (e.g., AnyType) to classes that implement Comparable.

2. Node and SkipList: Multiple Class Definitions in One Source File

You will have to implement a Node class, but since you are only submitting one source file via Webcourses (*SkipList.java*), you must tuck the Node class into *SkipList.java*. That means the Node class cannot be public. I have provided a sample *SkipList.java* file that demonstrates how to structure your code so that it will work with my test cases.

3. Node Class

3.1. Method and Class Requirements

Implement the following methods in a class named `Node`. You may replace `AnyType` with something else, if you wish (e.g., `T`), as long as the method names and return types stay the same.

`Node(int height)`

This constructor creates a new node with the specified height, which will be greater than zero. Initially, all of the node's *next* references should be initialized to *null*. This constructor will be particularly useful when creating a head node, which does not store anything meaningful in its *data* field. This constructor may be useful to you in other ways as well, depending how you choose to implement certain of your skip list methods.

`Node(AnyType data, int height)`

This constructor creates a new node with the specified height, which will be greater than zero, and initializes the node's value to *data*. Initially, all of the node's *next* references should be initialized to *null*.

`public AnyType value()`

An O(1) method that returns the value stored at this node.

`public int height()`

An O(1) method that returns the height of this node.

For example, if a node has three references (numbered 0 through 2), the height of that node is 3 (even if some of those references are *null*).

```
public Node<AnyType> next(int level)
```

An O(1) method that returns a reference to the next node in the skip list at this particular level. Levels are numbered 0 through (*height* – 1), from bottom to top.

If *level* is less than 0 or greater than (*height* – 1), this method should return *null*.

3.2. Suggested Methods

I found the following methods helpful in implementing my skip lists. You might find them helpful as well, but you aren't required to implement them. You may also choose to implement these suggested methods with different return types or parameter types. You're not bound by these method signatures in the same way that you're bound by the required method signatures above.

```
public void setNext(int level, Node<AnyType> node)
```

Set the *next* reference at the given level within this node to *node*.

```
public void grow()
```

Grow this node by exactly one level. (I.e., add a *null* reference to the top of its tower of *next* references). This is useful for forcing the skip list's *head* node to grow when inserting into the skip list causes the list's maximum height to increase.

```
public void maybeGrow()
```

Grow this node by exactly one level with a probability of 50%. (I.e., add a *null* reference to the top of its tower of *next* references). This is useful for when inserting into the skip list causes the list's maximum height to increase.

```
public void trim(int height)
```

Remove references from the top of this node's tower of *next* references until this node's height has been reduced to the value given in the *height* parameter. This is useful for when deleting from the skip list causes the list's maximum height to decrease.

4. SkipList Class

4.1. A Few Notes about the Head Node

The height of your skip list's *head* node should almost always be $\lceil \log_2 n \rceil$ (the ceiling of $\log_2 n$), where n is the number of elements in the skip list (i.e., the number of nodes in the skip list, excluding the *head* node). Recall that a skip list's *head* node does not contain an actual value, and therefore does not count as a node in your skip list when calling the *size()* method described below.

There are three exceptions to this rule for the height of the *head* node:

1. A skip list that has exactly one element should have a height of 1 (not $\lceil \log_2(1) \rceil$, which is 0).
2. If the skip list contains no elements, you may set the height of the *head* node to either 0 or 1. That choice is yours. You should pick the value that plays nicely with however you're implementing the other methods in your SkipList class.
3. One of the SkipList constructors described below allows the programmer to manually set the height of the *head* node when the skip list is first created. This is useful if you know that you're going to insert enough elements to cause the skip list to grow several times. In that case, you might as well start off with the skip list's height being greater than 1, even though there are no elements in the skip list to begin with. (This behavior is clarified in a number of test cases included with this assignment.)

Throughout this document, the height of the *head* node is considered to be synonymous with the height of your skip list.

4.2. Method and Class Requirements

Implement the following methods in a class named SkipList. You may replace `AnyType` with something else if you wish (e.g., `T`), as long as the function names and return types stay the same.

`SkipList()`

This constructor creates a new skip list. The height of the skip list is initialized to either 0 or 1, as you see fit. (See note about the height of the *head* node, above.)

`SkipList(int height)`

This constructor creates a new skip list and initializes the *head* node to have the height specified by the *height* parameter. If the height is less than the default height you have chosen for empty lists (0 or 1), you may instead default to your chosen value of 0 or 1.

The skip list should remain this tall until either it contains so many elements that the height must grow (i.e., when $\lceil \log_2 n \rceil$ exceeds the given height of the skip list), or when an element is successfully deleted from the skip list. For further details, see the *insert()* and *delete()* method descriptions, below.

`public int size()`

In O(1) time, return the number of nodes in the skip list (excluding the head node, since it does not contain a value).

`public int height()`

In O(1) time, return the current height of the skip list, which is also the height of the *head* node. Note that this might be greater than $\lceil \log_2 n \rceil$ if the skip list was initialized with a height greater than 1 using the second constructor listed above.

`public Node<AnyType> head()`

Return the head of the skip list.

`public void insert(AnyType data)`

Insert *data* into the skip list with an expected (average-case) runtime of O(log n). If the value being inserted already appears in the skip list, this new copy should be inserted *before* the first occurrence of that value in the skip list. (See test cases #14 and #15 for further clarification on the order in which duplicate values should be inserted.)

You will have to generate a random height for this node in a way that respects both the maximum height of the skip list and the expected distribution of node heights. (I.e., there should be a 50% chance that the new node has a height of 1, a 25% chance that it has a height of 2, and so on, up to the maximum height allowable for this skip list.)

The maximum possible height of this new node should be either $\lceil \log_2 n \rceil$ or the current height of the skip list – whichever is greater. (Recall that the height of the skip list can exceed $\lceil \log_2 n \rceil$ if the list was initialized using the second SkipList constructor described above.)

If inserting this node causes $\lceil \log_2 n \rceil$ to exceed the skip list's current height, you must increase the overall height of the skip list. Recall that our procedure for growing the skip list is as follows: (1) the height of the *head* node increases by 1, and (2) each node whose height was maxed out now has a 50% chance of seeing its height increased by 1. For example, if the height of the skip list is increasing from 4 to 5, the head node must grow to height 5, and each other node of height 4 has (independently) a 50% chance of growing to height 5.

Test Cases #1 through #3 demonstrate how the *insert()* method should affect the height of a skip list under a variety of circumstances. Test Cases #4 and #5 speak to the expected height distribution of nodes in a skip list.

`public void insert(AnyType data, int height)`

Insert *data* into the skip list using the procedure described above, but do not generate a random height for the new node. Instead, set the node's height to the value passed in via this method's *height* parameter. This will be super handy for testing your code with your own sequence of not-so-random values and node heights.[1]

`public void delete(AnyType data)`

Delete a single occurrence of *data* from the skip list (if it is present). If there are multiple copies of *data* in the skip list, delete the first node (that is, the leftmost node) that contains *data*. The expected runtime for this method should be O(log n), so you cannot perform a linear search for this node along the bottom-most level of node references. You must use the search algorithm for skip lists described in class.

If this method call results in the deletion of a node, and if the resulting number of elements in the list, *n*, causes $\lceil \log_2 n \rceil$ to fall below the current height of the skip list, you should trim the maximum height of the skip list to $\lceil \log_2 n \rceil$. When doing so, all nodes that exceed the new maximum height should simply be

1 The *height* parameter passed to this method will always be on the range $[1, \lceil \log_2(n + 1) \rceil]$, where *n* is the number of nodes in the skip list before adding this new node. This accounts for the fact that the new node (the +1 in that statement) might cause the height of the skip list to grow. For example, if the current height is 3 and the new element would cause the height to grow to 4, the valid range for that height parameter is 1 through 4.

trimmed down to the new maximum height.

If this method call does not actually result in a node being deleted (i.e., if *data* is not present in the skip list), you should *not* modify the height of the skip list.

Test Cases #8 through #11 and Test Case #13 clarify how the *delete()* method should affect the height of a skip list under a variety of circumstances.

```
public boolean contains(AnyType data)
```

Return *true* if the skip list contains *data*. Otherwise, return *false*. The expected (average-case) runtime of this function must be O(log n), and you must use the search algorithm for skip lists described in class. Test Case #6 demonstrates how the runtime of this method might be tested.

```
public Node<AnyType> get(AnyType data)
```

Return a reference to a node in the skip list that contains *data*. If no such node exists, return *null*. If multiple such nodes exist, return the first such node that would be found by a properly implemented *contains()* method.

```
public static double difficultyRating()
```

Return a double indicating how difficult you found this assignment on a scale of 1.0 (ridiculously easy) through 5.0 (insanely difficult).

```
public static double hoursSpent()
```

Return a realistic estimate (greater than zero) of the number of hours you spent on this assignment.

4.3. Suggested Methods

I found the following methods helpful in implementing my skip lists. You might find them helpful as well, but you aren't required to implement them. You may also choose to implement these suggested methods with different return types or parameter types. You're not bound by these method signatures in the same way that you're bound by the required method signatures above.

```
private static int getMaxHeight(int n)
```

A method that returns the max height of a skip list with *n* nodes.

```
private static int generateRandomHeight(int maxHeight)
```

Returns 1 with 50% probability, 2 with 25% probability, 3 with 12.5% probability, and so on, without exceeding *maxHeight*.

```
private void growSkipList()
```

Grow the skip list using the procedure described above for the *insert()* method.

```
private void trimSkipList()
```

Trim the skip list using the procedure described above for the *delete()* method.

5. Special Requirements: No Output or Compiler Warnings (*Super Important!*)

Here are some special restrictions regarding compile-time warnings:

- ★ None of the methods described above should print anything to the screen. Printing anything to the screen will likely resolve in tragic test case failure.
- ★ Your code must not produce any warnings when compiled on Eustis. For this particular assignment, it's especially important not to have any *-Xlint:unchecked* warnings. Please note that some systems don't produce *-Xlint:unchecked* warnings! The safest way to check whether your code is producing such warnings is to compile and run your program on Eustis with the *test-all.sh* script.
- ★ You cannot use the `@SuppressWarnings` annotation (or any similar annotations) to suppress warnings in this assignment.

Please do not give your code to classmates and ask them to check whether their compilers generate compile-time warnings for your code. Remember, sharing code in this course is out of bounds for assignments.

6. Style Restrictions (Same as in Program #1) (*Super Important!*)

Please conform as closely as possible to the style I use while coding in class. To encourage everyone to develop a commitment to writing consistent and readable code, the following restrictions will be strictly enforced:

- ★ Capitalize the first letter of all class names. Use lowercase for the first letter of all method names.
- ★ Any time you open a curly brace, that curly brace should start on a new line.
- ★ Any time you open a new code block, indent all the code within that code block one level deeper than you were already indenting.
- ★ Be consistent with the amount of indentation you're using, and be consistent in using either spaces or tabs for indentation throughout your source file. If you're using spaces for indentation, please use at least two spaces for each new level of indentation, because trying to read code that uses just a single space for each level of indentation is downright painful.
- ★ Please avoid block-style comments: /* *comment* */
- ★ Instead, please use inline-style comments: // *comment*
- ★ Always include a space after the "//" in your comments: "// *comment*" instead of "//*comment*"
- ★ The header comments introducing your source file (including the comment(s) with your name, course number, semester, NID, and so on), should always be placed *above* your import statements.
- ★ Use end-of-line comments sparingly. Comments longer than three words should always be placed *above* the lines of code to which they refer. Furthermore, such comments should be indented to properly align with the code to which they refer. For example, if line 16 of your code is indented with two tabs, and line 15 contains a comment referring to line 16, then line 15 should also be intended with two tabs.

- ★ Please do not write excessively long lines of code. Lines must be no longer than 100 characters wide.
- ★ Avoid excessive consecutive blank lines. In general, you should never have more than one or two consecutive blank lines.
- ★ Please leave a space on both sides of any binary operators you use in your code (i.e., operators that take two operands). For example, use *(a + b) - c* instead of *(a+b)-c*. (The only place you do *not* have to follow this restriction is within the square brackets used to access an array index, as in: *array[i+j]*.)
- ★ When defining or calling a method, do not leave a space before its opening parenthesis. For example: use *System.out.println("Hi!")* instead of *System.out.println ("Hi!")*.
- ★ Do leave a space before the opening parenthesis in an *if* statement or a loop. For example, use use *for (i = 0; i < n; i++)* instead of *for(i = 0; i < n; i++)*, and use *if (condition)* instead of *if(condition)* or *if(condition)*.
- ★ Use meaningful variable names that convey the purpose of your variables. (The exceptions here are when using variables like *i*, *j*, and *k* for looping variables or *m* and *n* for the sizes of some inputs.)
- ★ Do not use *var* to declare variables.

7. Compiling and Testing SkipList on Eustis (and the *test-all.sh* Script!)

Recall that your code must compile, run, and produce precisely the correct output on Eustis in order to receive full credit. Here's how to make that happen:

1. To compile your program with one of my test cases:

   ```
   javac SkipList.java TestCase01.java
   ```

2. To run this test case and redirect your output to a text file:

   ```
   java TestCase01 > myoutput01.txt
   ```

3. To compare your program's output against the sample output file I've provided for this test case:

   ```
   diff myoutput01.txt sample_output/TestCase01-output.txt
   ```

   If the contents of *myoutput01.txt* and *TestCase01-output.txt* are exactly the same, *diff* won't print anything to the screen. It will just look like this:

   ```
   seansz@eustis:~$ diff myoutput01.txt sample_output/TestCase01-output.txt
   seansz@eustis:~$ _
   ```

   Otherwise, if the files differ, *diff* will spit out some information about the lines that aren't the same.

4. I've also included a script, *test-all.sh*, that will compile and run all test cases for you. You can run it on Eustis by placing it in a directory with *SkipList.java* and all the test case files and typing:

```
bash test-all.sh
```

Super Important: Using the *test-all.sh* script to test your code on Eustis is the safest, most sure-fire way to make sure your code is working properly before submitting.

Note! Because TestCase06 generates a lot of data, you might not be able to run it on Eustis in a reasonable amount of time. It's totally acceptable to make sure everything is compiling and running on Eustis, but to then use your own personal computer to test the actual runtime of TestCase06.

8. Grading Criteria and Miscellaneous Requirements

> ***Important Note:*** When grading your programs, we will use different test cases from the ones we've released with this assignment, to ensure that no one can game the system and earn credit by simply hard-coding the expected output for the test cases we've released to you. You should create additional test cases of your own in order to thoroughly test your code. In creating your own test cases, you should always ask yourself, "What kinds of inputs could be passed to this program that don't violate any of the input specifications, but which haven't already been covered in the test cases included with the assignment?"

The *tentative* scoring breakdown (not set in stone) for this programming assignment is:

80%	Passes test cases with 100% correct output formatting. This portion of the grade includes tests of the *difficultyRating()* and *hoursSpent()* methods.
10%	Program compiles without warnings. To be eligible for these points, generics with Comparable must be implemented properly, and the code must not use any warning suppression annotations.
10%	Adequate comments and whitespace. To earn these points, you must adhere to the style restrictions set forth above. We will likely impose huge penalties for small deviations, because we really want you to develop good style habits in this class. Please include a header comment with your name and NID, and please be sure to name your file correctly.

Your program must be submitted via Webcourses.

Please be sure to submit your *.java* file, not a *.class* file (and certainly not a *.doc* or *.pdf* file). Your best bet is to submit your program in advance of the deadline, then download the source code from Webcourses, re-compile, and re-test your code in order to ensure that you uploaded the correct version of your source code.

Important! Programs that do not compile on Eustis will receive zero credit. When testing your code, you should ensure that you place *SkipList.java* alone in a directory with the test case files (source files, the *sample_output* directory, and the *test-all.sh* script), and no other files. That will help ensure that your *SkipList.java* is not relying on external support classes that you've written in separate *.java* files but won't be including with your program submission.

Important! You might want to remove *main()* and then double check that your program compiles without

it before submitting. Including a *main()* method can cause compilation issues if it includes references to home-brewed classes that you are not submitting with the assignment. Please remove.

Important! **You should not print anything to the screen.** Extraneous output will result in severe point deductions. The required methods you write in *SkipList.java* should not print anything to the screen.

Important! **No file I/O.** Please do not read or write to any files.

Important! **Please do not create a java package.** Articulating a *package* in your source code could prevent it from compiling with our test cases, resulting in severe point deductions.

Important! **Name your source file, class(es), and method(s) correctly.** Minor errors in spelling and/or capitalization could be hugely disruptive to the grading process and may result in severe point deductions. Similarly, failing to implement a required method, or failing to make certain methods *public*, *private*, *static*, and/or non-static (as required), may cause test case failure. Please double check your work!

Input specifications are a contract. We promise that we will work within the confines of the problem statement when creating the test cases that we'll use for grading. Please reflect carefully on the kinds of edge cases that might cause unusual behaviors for any of the methods you're implementing.

Test your code thoroughly. Please be sure to create your own test cases and thoroughly test your code. You're welcome to share test cases with each other, as long as your test cases don't include any solution code for the assignment itself.

Start early! Work hard! Ask questions! Good luck!