COSC-211: DATA STRUCTURES
HW4: SKIP LISTS

Due Thursday, March 18, 11:59pm ET

Reminder regarding intellectual responsibility: This is an individual assignment, and the work you submit should be your own. Do not look at anyone else's work, and do not show anyone your work (except for your instructor and the course TAs).

1 Introduction

We've been discussing skip lists in class this week. A *skip list* is a set of linked lists, one on top of the other, where fewer elements are included in the higher levels of the list. The idea is that the higher levels can then serve as "express lanes" by which we can skip ahead many elements all in one step, thereby making our `find(x)` operation more efficient. Skip lists are a randomized data structure, meaning that the construction of the data structure is determined, in part, by random decisions. In the case of skip lists, the "height" of an element (i.e., the highest list level to which the element is promoted) is determined by flipping a coin repeatedly until the coin lands on tails.

In this assignment, you'll complete a skip list implementation. You'll then run some experiments to determine how the choice of `p`, the probability the coin lands on heads, affects the runtime of our skip list operations.

2 Getting Started

Create a new directory/project for yourself, and download the following ZIP archive:

`http://bit.ly/cosc-211-21s-hw4`

Extract/copy the files in this archive into your new directory/project. You will find the following files:

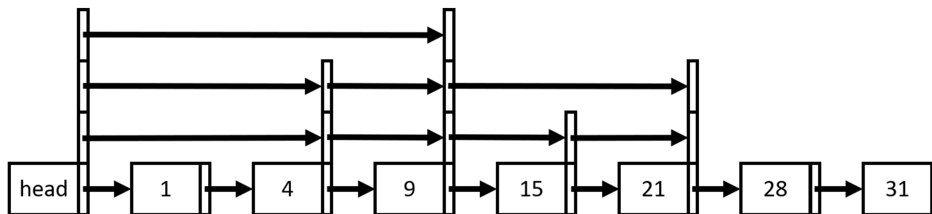
- `AmhSortedSet.java`
  The interface that matches our *Sorted Set* ADT.
- `SkipList.java`
  An incomplete implementation of a Skip List.
- `Node.java`
  A node, used by `SkipList.java`
- `RuntimeExperiment.java`
  Some code to run experiments to see how the probability of promoting an element to the next list level affects the runtime of the `find(x)` operation.

A quick tour of the Skip List implementation. We'll first take a look at the Node class. Each Node stores its data and an array nextNodes of pointers to the next Node on each level of the skip list in which this Node appears. For example, if this Node appears on levels 0-3 of the skip list, then the array nextNodes will have length 4, and the entry at each position i will store the next Node in list L_i.

The Node class contains each of the methods required by the AmhSortedSet interface: add(x), remove(x), find(x), and size(). The find(x) and size() methods are complete; you will need to write some code to complete add(x) and remove(x).

You will also see a few methods that are not part of the AmhSortedSet interface. The chooseHeight() method simulates the process of repeatedly flipping a coin until it lands on heads, thereby determining the height of a new element (i.e., the highest list level to which we want to add the new element). The field p controls the probability that the coin lands on heads (i.e., the probability that the element gets "promoted" to the next list).

The findAllPreds(x) method forms the core of the process by which we add, remove, and find elements: this is the method in which we'll walk from the header to the predecessor of element x, moving down list levels as needed. The findAllPreds(x) method takes an element x as input and returns a Stack<Node<E>> containing the Node objects that immediately precede element x on each level of the list. The predecessor on level 0 should end up at the top of the stack, and the predecessor on the highest level of the list should end up at the bottom of the stack. For example, if our skip list looks like this:



then findAllPreds(30) would return a stack containing, in order from top to bottom, the nodes storing 28, 21, 21 and 9.

You will need to fill in the findAllPreds(x) method.

SkipList includes a number of different constructors that allow you to specify p, the seed for the random number generator, both, or neither. These are meant to help you in the course of your debugging.

3 Your assignment

3.1 Programming

Complete the skip list implementation by filling in the following methods in SkipList.java:

- `findAllPreds(x)`
- `add(x)`
- `remove(x)`

Write yourself some test code (following the example you saw on HW3) to verify that your skip list works properly. You won't submit your test code, but it's always in your best interest to test your code thoroughly before you submit it (and, for this assignment, before you move on to the experimentation component).

3.2 Experimentation

In the second part of this assignment, you'll investigate how the choice of `p` (the probability with which an element gets promoted to the next list) affects the runtime of the `findAllPreds(x)` method.

First, you'll need to add some code to count the number of operations executed in `findAllPreds(x)`. For our purposes, we'll focus on only two types of operations: (1) moving over an element, within the same list level, and (2) moving down to a lower list level. Add some code to your `findAllPreds(x)` method to count how many times these operations are excuted during one run of the method. We've given you a field called `countOps` to help with this. *Be sure to reset* `countOps` *to 0 every time you run* `findAllPreds(x)`.

We've given you a class called `RuntimeExperiment.java`, which contains all the code you need to run the experiments (after you've added a couple lines of code to `findAllPreds(x)` to count the operations). Specifically, the `runTest(n,p)` method takes as input the desired list size (`n`) and the probability of promoting an item to the next list level (`p`). The method then adds `n` elements to a skip list and, after every power-of-2-th add, does a `find` and prints out the number of operations executed during the `find`. By default, we set `n` to 2^{21}; you can change this if you like, but don't make it too much smaller because our runtime analysis is all about what happen when `n` is large.

`RuntimeExperiment` can take the parameter `p` as a command line argument; if you do not input a command line argument then `p` will be set to 0.5 by default.

When you run `RuntimeExperiment`, the `go()` method will execute. All `go()` does is call `runTest(n,p)` five times. Repetition is helpful with this kind of experiment because, as you'll see, the runtimes can vary quite a bit from run to run. This is because the runtime depends on the current set of elements in the skip list, the heights of the nodes in the skip list, and the element we're finding, all of which are generated randomly. Running the same experiment five (or more) times and averaging the results will help to smooth out some of the "noise" that you'll see due to this randomization. You're welcome to modify `go()` to run more repetitions of the experiment if you like.

Your task: Your goal is to determine how the choice of `p` affects the runtime of the `find(x)` operation. To that end, run `RuntimeExperiment` several times with different choices of `p`. You should include a few very small values of `p` (e.g., 0.001), a few moderate values of `p` (e.g., 0.5), and a few large values of `p` (e.g., 0.95).

After you've done your experiments, write up your findings. Your writeup should answer the questions:

- **What** did your results show? That is, how does the runtime of `find` compare, as `n` grows, for the low, moderate, and high values of `p`?
- **Why** do you think the results looked the way they did? What does a skip list look like when `p` is very low? What about when `p` is very high? How do those list structures lead to the results that you saw?
- Did you observe anything else interesting in your skip list experiments?

4 Submit your work

Submit your work via Gradescope. You should submit your `SkipList.java` and your writeup of the findings from your experiment (as a pdf).

This assignment is due Thursday, March 18, 11:59pm ET.