IMPORTANT SUBMISSION INSTRUCTIONS

You will be uploading your submission using the assignment server. The link is posted to cuLearn. A short video is posted there also in case you require instructions. You may submit as many times as you wish, but must wait at least 5 minutes between submissions (to protect the server). Your highest mark is kept. It would help us out if you try the upload early, even using just the unmodified assignment skeleton. That way potential problems can be found before the deadline. It will also ensure that YOU know how to submit your assignment well before the deadline. If your attempt to obtain your secret code is unsuccessful, please email me at darrylhill@cunet.carleton.ca. This may happen if you have registered late, etc.

You must adhere to the following rules *(as your submission will be subjected to automatic marking system):*

- ***Download*** *the compressed file* ***"comp2402a3.zip"*** *from cuLearn.*
- ***Retain the directory structure*** *(i.e., if you find a file in subfolder "comp2402a3", you must not remove it).*
- ***Retain the package directives*** *(i.e., if you see "package comp2402a3;" in a file, you must not remove it).*
- ***Do not rename any of the methods already present*** *in the files provided.*
- ***Do not change the visibility of any of the methods provided*** *(e.g., do not change private to public).*
- ***You may use the main method of FastDefaultList for test code****; this is a different specification from previous assignments.*
- ***Upload*** *a compressed file* ***"comp2402a3.zip"*** *to the assignment server to submit your assignment as receive your mark immediately* ***(highest mark of all submissions will be your assignment mark).***

Please also note that ***your code may be marked for efficiency as well as correctness*** *– this is accomplished by placing a hard limit on the amount of time your code wil be permitted for execution. If you select/apply your data structures correctly, your code will easily execute within the time limit, but* ***if your choice or usage of a data structure is incorrect****, your code may be* ***judged to be too slow*** *and* ***it may receive a grade of zero****.*

You are expected to ***demonstrate good programming practices at all times*** *(e.g., choosing appropriate variable names, provide comments in your code, etc.) and* ***your code may be penalized if it is poorly written****. The server won't judge this, but in case of discrepancies, this will be evaluated.*

Instructions

Start by downloading the comp2402a3 Zip file from cuLearn, which contains a skeleton of the code you need to write. This assignment is about modifying the SkiplistList class to implement a FastDefaultList. The folder contains two files (java classes); DumbDefaultList.java and FastDefaultList.java. You are to complete FastDefaultList.java according to the specification provided below. DumbDefaultList.java is not to be modified. It is to help you test the correctness of your class. It is a correct but slow implementation of the behaviour required.

Specification for Assignment 3 of 4

Part 1 of 1 – A Fast Default List

Starting with the **SkiplistList** class, implement a **FastDefaultList** class that represents an infinite list with indices 0,1,2,3,…,∞. When we start, every value in this list is assigned the default value **null**. Otherwise, this class behaves just like a **List**; it has the **add(i,x)**, **remove(i)**, **set(i,x)**, and **get(i)** that behave just like the same methods in a list. Each of these operations should run in **O(log n)** time. The **size()** method is already implemented for you, it returns the largest value you can store in an int.

A **DumbDefaultList** class has already been implemented for you. You can use it to help test the correctness of your implementation. Your **FastDefaultList** should have the same output for a given input as **DumbDefaultList,** but it should run much faster.

Warning: This is not a hack job. First understand how the **SkiplistList** class works and figure out how you can modify it to do what this question asks.

Hint: Work slowly. There is not much code you have to change in the **FastDefaultList.java** file (which is mostly just a copy of **SkiplistList**), but it is delicate. Checkpoint your work often and test carefully as you go.

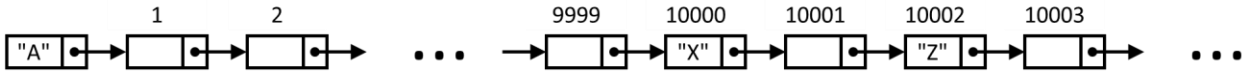
Here is a short example to illustrate the behaviour required:

1) I make a new **FastDefaultList** of **Strings.**
2) I call **get(1000)** and it returns **null**.
3) I call **add(1000, "hello").**
4) **get(1000)** now returns **"hello".**
5) A call of **get(500)** returns null.
6) I call **add(500, "goodbye").**
7) **get(1000)** now returns null.
8) **get(1001)** returns **"hello"** (because every entry from 500 to ∞ was shifted up one index to make room for **"goodbye"** at index 500).
9) I call **remove(20).**
10) **get(500)** returns null.
11) **get(499)** returns**"goodbye"**, and **get(1000)** returns **"hello"** (because all the items from index 21 to ∞ were shifted one index down when we removed index 20).

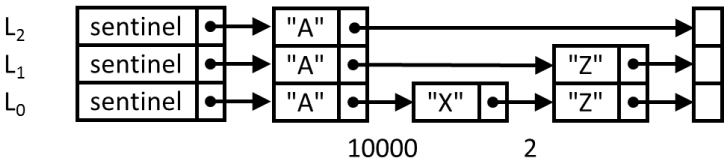
Addendum

A linear collection that is (for all intents and purposes) of infinite length, is a difficult structure to conceptualize. Obviously, an array of infinite length is not feasible, and although an infinitely long singly or doubly-linked list is conceivable, it would exhibit very poor performance. (Imagine, for instance, a linked list into which elements were added at indices 0 and 10,000, respectively. In spite of there being only two elements in the list, traversing the indices 1 to 9,999 would still be necessary to visit every element...)

That said, it is possible to create a skiplist representation that can represent an infinite list quite effectively, without actually containing an infinite number of nodes. So although the user might envision a structure like this...



...in actually, it would be represented using a skiplist like this (where "1000" and "2" now represent the lengths of the edges)...



Since the underlying structure remains very much like a skiplist, this structure would still yield logarithmic expected time complexities for get(i), set(I, x), add(u, x), and remove(i) operations, and would waste space on the storage of nonexistent values.

Consider the following additional details:

- If you were to attempt to get the value from a particular index in the infinite list, and that index is not yet associated with a value, your get function would need to return a null value (even though there isn't actually a null value explicitly being stored at that index).
- If you were to attempt to set the value for a particular index in the infinite list, and that index is not yet associated with a value, your set function would actually need to insert a new node into the skiplist (meaning that your set method call would actually be more like a call to add).
- Although the "lengths" of the "edges" in L0 (i.e., the underlying singly-linked list) in a standard skiplist would always have a value of one (because, in L0, the next node is always one unit away from the current node), this would NOT be the case in the structure being described here. The magnitude of each of the "lengths" in L0 can now be any positive integer, and methods like findPredecessor would be expected to return both the predecessor node and the length of the edge to its successor.
- Infinite here is being operationally defined as Integer.Max_Value. Although this technically means that there is an edge case of what happens when there is something at the end of the list and something is inserted before it (causing the list to overflow) you do NOT need to account for this case (and the submission server will not test for it).