CS3460: Data Structures Fall 2022
Assignment 06: Fun with Dynamic Sequences
Due Date: Friday, October 04; 23:00 hrs
Total Points: 25

In this assignment, we will build a robust binary tree, balanced using randomized mechanisms, to store a dynamic sequence. We will then use this tree to solve a fun problem of ordering 100000 football teams such that each team i in the ordering defeated team $i + 1$.

1. **Randomly-Built Binary Trees (15 points):** A randomly-built binary search tree (RBST) provides all binary search tree operations in only $O(\log n)$ time with high probability. In this problem, we will build a RBST to encode a dynamic sequence. A dynamic sequence $a_1 \dots a_n$ is a mathematical object that represents a sequence of items. The i^{th} item in the sequence is a_i. A dynamic sequence can be encoded in a RBST, by storing some item a_i at the root, and recursively storing the sequence $a_1 \dots a_{i-1}$ as the left subtree of the root, and the sequence $a_{i+1} \dots a_n$ as the right subtree of the root. If each node in the tree is augmented with subtree sizes, then the i^{th} item can be accessed using the `select` operation.

   Getting Started: Please download all the required files from ASULearn into your working directory. The `Node` class is fully implemented with all the required accessor and mutator methods. It also contains the methods `incSize` which increments the size of the tree and `updateSize` which updates the size of the tree rooted at the node.

   The RBST class: The `RBST` class implements a randonly-built binary search tree. It contains instance fields `root` which is a reference to the root node of the tree, and a `Random` object to make probabilistic decisions in the `insert` routine. The class is currently filled with two constructors, a `getSize()` method that returns the size of the tree, and several public wrapper methods that are described below. Please complete the `RBST` class, so that it fully supports the following operations.

   (a) `insertNormal(team, rank)`: This method inserts the data `team` at position `rank`. This is a simple insert routine without any balancing mechanism.

   (b) `print()`: This method performs an inorder traversal of the tree and prints the sequence stored in the tree.

   (c) `split(rank)`: This method splits the tree at position specified by `rank`. It returns an an array of two RBSTs, the first being the left side of the split, and the second being the right side of the split.

   (d) `insert(team, rank)`: This method inserts the data `team` at position `rank`. The insert routine maintains balance in a probabilistic manner. That is, it inserts the new node at the root of the tree with probability $1/n$, where n is the size of the tree including the new node. This is ideally done by splitting the tree at rank `rank-1`, and attaching the left side and right side of the split as the left and right children of the new node respectively. With probability $1 - 1/n$, the data is recursively inserted to either the left tree or the right tree depending upon the position `rank` (whether `rank` is less or greater than the rank of the root).

   (e) `select(rank)`: This method returns the node in the RBST at position `rank`.

An element has rank k if it is appears in position k in the inorder traversal of the tree. Since the tree encodes a sequence, the element with rank $k = 1$ and $k = n$ are the first and last elements in the sequence respectively.

Note that the above methods are public interfaces, which are wrapper methods that contain calls to corresponding private methods. These are already written and should not be modified. Please fill out places where it says "TODO:", and complete all the private methods.

The above class is designed with wrapper methods because of two reasons. Firstly, all the above methods are recursive, and operate on a single node which represents the tree rooted at that node, instead of a `RBST` object. Secondly, it allows the user to interact much more easily. For instance, the user can simply create and access nodes in the following manner.

```
RBST T = new RBST(); // create an instance of RBST.
T.insert(15, 10); // insert 15 at position 10.
T.select(7); // select or access the node at position 7.
```

Testing: You are provided with a `RBSTTest.java` file that tests all the methods in `RBST.java`. Since the `RBST` class contains many methods, some of which are dependant on others, you are strongly advised to test them one at a time in order. Please do not attempt to test all the methods at the same time. You may want to comment or uncomment certain lines of code in `RBSTTest` to test only those methods you have completely written. The `RBSTTest` class contains a `main` routine that creates the same sequence stored both as an array, and as a `RBST`, and performs the same set of operations, and prints them to screen. Wherever there is a call to the `print` routine in the `RBST`, make sure the output printed matches the corresponding one from the array.

2. **Ordering Football Teams (10 points):** Suppose n football teams, numbered $1 \ldots n$, all play each other in a large tournament, where there are no ties. In file `Ordering.java`, you will find a method called

```
boolean didXBeatY(int x, int y)
```

which returns `true` if team x won its game against team y, `false` otherwise. Your task is to compute an ordering of the teams, say $a_1 \ldots a_n$, so that team a_1 defeated team a_2, team a_2 defeated team a_3, and so on. Lets call such a sequence valid.

Note that there could be many possible valid solutions. For example, if team 1 beat team 2, team 2 beat team 3, etc., and team n beat team 1, then any "cyclic rotation" of the sequence $1, 2, 3, \ldots n$ (e.g., $3, 4, 5, \ldots n-1, n, 1, 2$) will be a valid solution.

The algorithm we will use to solve this problem is straightforward. It involves building up a sequence (stored in a `RBST`) to which we add teams $1 \ldots n$ one by one. Suppose we have built up a valid sequence of teams $1 \ldots k-1$ and we want to add team k in a position so that the entire sequence is valid. It turns out that there always exists such a position, and that we can find it quickly using our balanced `RBST`. If team k defeated the first team in our sequence, then we can just add it to the beginning. Likewise, if team k lost to the last team in the sequence, we can add it to the end. If neither of these cases holds, then let us imagine the sequence with arrows going from the winning team to the losing team (see Figure 1). That is, $a \rightarrow b$ means that team a won against team b. Since we know that team k lost to the first

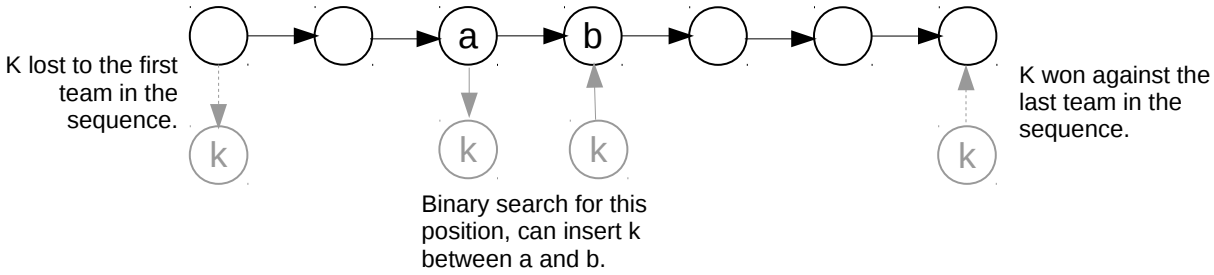

Figure 1: Inserting team k into a valid ordering of teams $1 \dots k-1$. Team k lost to the first team, and won against the last team. Therefore, we can binary search to find the position for k in the ordering. Team k can be placed between a an b, since $a \to k$ and $k \to b$.

team, and beat the last team, there must be some location in the ordering where there is an adjacent pair of teams (a, b) for which $a \to k$ and $k \to b$. It is thus feasible to place k between a and b. Moreover, we can binary search for such a position with only $O(\log k) \leq O(\log n)$ calls to `select`. Since each call to select takes only $O(\log n)$ time with high probability, this gives a total running time of $O(n \log^2 n)$ with high probability to solve the entire problem. Please implement the algorithm above in the method `orderTeams`. We have already provided some code to get you started. In the `main` method, we have included testing code to see if your method returns a tree encoding a valid sequence.

Submission: Please submit all the files as a single zip archive `h06.zip` through ASULearn. The zip archive should only contain the individual files of the assignment, and these files should not be inside folders. Moreover, please do not include other extraneous files. Only include all files that belong to your solution.

Input/ Output Instructions: For all programs, until and otherwise stated, we will be taking the input from standard input (`System.in`) and will be sending the output to standard output (`System.out`). Since we will be using an automatic grading system, please make sure that your output format exactly matches the description above. So make sure that there are no extraneous output in terms of spaces, newlines and other characters.

Notes on Coding: Please do not include user-defined packages in your code. Your code should run in the Unix/Linux machine using the commands `javac` and `java`.

Please note that you are **not allowed to use Java Collections** which contain pre-defined libraries for many of the data structures that we learn in class. The purpose of these assignments is to build these data structures from first principles. Programs that includes these objects will receive 0 points.