CS3460: Data Structures Fall 2022
Assignment 05: Fun with Binary Search Trees
Due Date: Monday, October 24; 23:00 hrs
Total Points: 25

In this assignment, we will build a binary search tree and use it solve a problem involving runway requests by airlines. We will learn that the actual data consisting of large meta-information can be stored outside the data structure, while the binary search tree can be keyed on a particular field within the data. We will also develop several binary search tree operations beyond a dictionary data structure, like computing the minimum element in a set or the predecessor element of a key within the set. Note that in this assignment, it is not required to balance the tree. The input data will be sufficiently random to keep the tree balanced.

1. **Binary Search Tree (15 points):** A binary search tree is a data structure that maintains a collection of keys (along with associated values) which encodes a sorted ordering of the elements withing its inorder traversal. This allows us to perform several operations that are not possible with hash tables.

   Getting Started: We first build a binary search tree that is keyed on an integer `time`. Please download all the necessary files from ASULearn into your working directory. The `Node` class is already implemented with two data fields – `time` and `req_index`. It contains appropriate accessors and mutators. There is no need to modify this file, but feel free to add methods if it is beneficial to your implementation.

   For problem 2, we will be given airline runway requests, which contain meta-data (flight name, source, destination, etc.) for each request. We will store all the requests as an array, and implement a binary search tree keyed on `time` with associated value `req_index` that points to the index of the request in the data array. As a first step, we need to implement all operations in file `BST.java`. Our binary search tree supports the following operations.

   (a) `insert(time, req_index)`: This method inserts an integer `time` along with the associated value `req_index` into the tree. Note that the tree is keyed on `time`.

   (b) `pred(time)`: This returns the node within the binary search tree that is the predecessor of `time`, i.e., the largest integer smaller or equal to `time`. This returns `null` if there is no predecessor.

   (c) `succ(time)`: This returns the node within the binary search tree that is the successor of `time`, i.e., the smallest integer greater or equal to `time`. This returns `null` if there is no successor.

   (d) `min()`: This returns a pointer to the minimum element in the binary search tree, or `null` if the tree is empty.

   (e) `max()`: This returns a pointer to the maximum element in the binary search tree, or `null` if the tree is empty.

   (f) `delete(time)`: This removes the node with key `time` if it is present, otherwise does nothing.

   (g) `print()`: This method prints the contents of the tree in sorted order, by doing an inorder traversal of the tree.

Note that most of the above methods are public wrapper methods to our own private recursive methods. From the user's point of view, only these public methods are called. But since all operations in a binary search tree are recursive, we need to implement them as private methods.

Also note the version of both the `pred` and `succ` methods. We are looking for the deepest ancestor who is smaller and larger than the given `time` respectively. While you traverse from the root of the tree in search of `time`, you need to keep track of this deepest ancestor. You may implement this function iteratively.

The method `delete` can be implemented in a recursive way. Suppose we try to delete from a subtree rooted at node `x`. The recursive method always returns the root of the subtree in which we perform `delete`. As a base case, we return `null` if `x` is `null` (deleting from an empty tree). If `x` contains key `time`, then we have three cases.

(a) **`x` is a leaf:** Then return null. Note that the parent's appropriate child pointer will be updated to `null` therefore removing `x` from the tree.

(b) **`x` has one child `y`:** Then return `y`. Note that the parent's appropriate child pointed will be updated to `y`, thereby removing `x` from the tree.

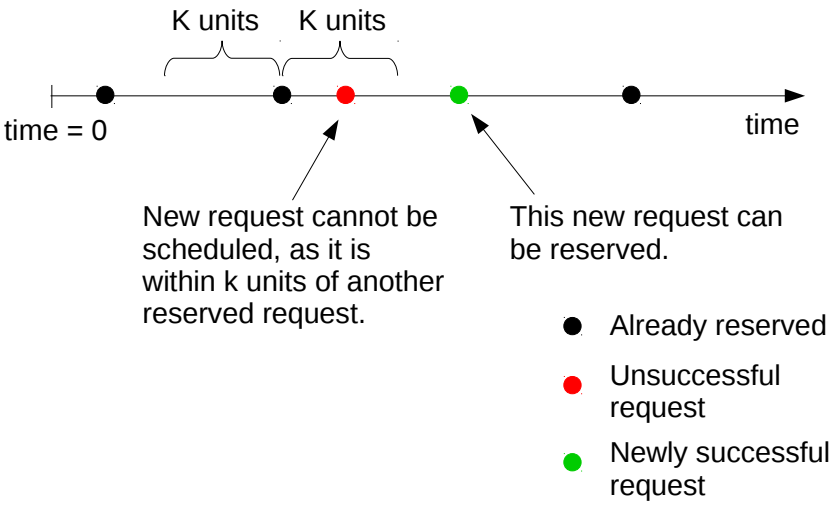
(c) **`x` has two children:** In this case, we find the successor `s` of `x`, then copy `s`'s data fields onto `x`, and delete `s` from the right subtree of `x` (Note that `s` can only be in `x`'s right subtree as its key is larger than `x`). This will cause an update to `x`'s right child.

The two recursive cases remain, that of deleting `time` from either the left or right subtree of the root depending on the comparison between `time` and the key stored in node `x`. Please write your own driver program to test all the operations of the binary search tree.

2. **Runway Reservation System (10 points):** In this problem, we will be implementing a runway reservation system. Suppose that there is a single runway in an airport. Airlines can make reservations on the runway based on the times they need to use the runway. In order to use the runway efficiently and to accommodate for slight changes in flight schedules, we allow a grace period of k units of time between successive use of the runway. All runway reservation requests are to be processed on a first-come-first-serve basis. In this problem, we will print out all the requests that have been successful in using the runway. Note that the above constraints mean that some requests may be unsuccessful (see Figure below).

   The file `Requests.java` contains the class template for a single reservation request. It contains a `command` (either a request or a time command, see below for explanation), the request `time`, and an `Airline` object to store the meta-information about an airline (flight name, number, source and destination airports). This file also contains appropriate constructors, accessor methods and other methods. Please make sure you understand these methods. There is no need to modify this class.

   You are given some startup code in `RunwayReservation.java`. This code currently reads through the input file (from `System.in`) and loads each request into an array `reqs`. The input file contains two numbers n and k on the first line that specifies the total number of requests and the grace time period between successive requests respectively. The next n lines provide information about a request. Each request consists of one of two commands – a 'r' or 'request reservation' command to request use of the runway for an airline or a 't' or 'advance time' command to move time forwards by some units. Each 'r' command follows with information about the airline – namely the time to use the runway, the airline flight



name, number, its source and destination airports, all space separated on the same line. Each 't' request follows with some integer time units to advance the current time. See below for sample input and output.

Since all the requests are stored as an array, we are now ready to solve the runway reservation problem. We need to use a binary search tree keyed on time (so the inorder traversal of the tree prints the requests sorted on time) to store all valid or successful requests. We process all the requests in order. The following pseudocode solves this problem at a high lived.

(a) For each request `reqs[i]`, do the following.

   i. If the request is 'r', then check if it is a valid request (there is a grace period of k units between this request and its previous and next reserved requests). If it is valid, then insert it into a BST T.

   ii. If the request is 't', update the current time, and remove and print all requests from tree T with times that are strictly less than the current time. These are the successful requests who have used the runway.

For each request, we can check for validity in only $O(\log n)$ time. Throughout the entirety of the algorithm, each request is inserted into the tree at most once, and removed at most once, both of which take only $O(\log n)$ time per element. So the total running time of the algorithm is $O(n \log n)$.

The output will be formatted as follows. For each 't' command, we print the current time followed by all successful airlines that have used the runway after the previous 't' command. We assume that there is a 't' command at the end of input that advances current time to the last successful request. See below for sample input and output. They are also contained in files `small-flight.in` and `small-flight.out` respectively.

Sample Input (file small-flight.in):

```
11 5
r 20 UA  1545  CLT IAH
r 9 UA  1714  CLT IAH
r 6 AA  1141  CLT MIA
r 5 B6  725  CLT BQN
r 10 DL  461  CLT ATL
r 6 B6  79  CLT MCO
t 7
r 25 UA  1696  CLT ORD
r 7 B6  507  CLT FLL
t 10
r 24 EV  5708  CLT IAD
```

Sample Output (file small-flight.out):

```
Current time = 7 units
Current time = 17 units
UA 1714 CLT IAH
Current time = 25 units
UA 1545 CLT IAH
UA 1696 CLT ORD
```

Please also test your program for the input provided in `large-flight.in`. It's corresponding output is in `large-flight.out`.

Submission: Please submit all the files as a single zip archive `h05.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.