CS3460: Data Structures

Lab 04: Binary Search Trees

Total Points: 25

Problems

Notes on Grading: Unless otherwise stated, all programs will receive input via `System.in` and will output solutions via `System.out`.

To simplify the grading process, all grading will be automated. When applicable, you will be provided with sample input/output files for testing. You can ensure that your program will receive full marks by testing it with these provided files.

```
$ java YourProgram < input.txt > output.txt
$ diff output.txt correct.txt
```

The first line executes the Java program, redirecting input from a file `input.txt` and writing the output to a file `output.txt`. The second line compares your program's output (now stored in `output.txt`) with the correct answer (stored in `correct.txt`). If these files match exactly, the `diff` program will print nothing. Otherwise, it will list the differences.

Important Note: You are not allowed to use any classes or code from the Java Collections library. While the classes defined in that library would not be an ideal fit for most of our tasks, the purpose of these assignments is to build these data structures from first principles. Programs which import any of these libraries will receive zero points.

Submission: Please submit any `.java` files necessary to compile/run your project. Please do not include any other files (such as input/output files).

1. **Building a Binary Search Tree (15 points):** A *binary search tree* is a node-based data structure that encodes a sorted ordering on its inserted keys. It is capable of answering queries beyond that of what hash tables are capable of.

   For this assignment we will be building a binary search tree keyed on an integer `time`. You have been provided a `Node` class with two data fields: `time` and `req`, the second being a request object that will be relevant in the second half of the assignment. There is no need to modify this file, but feel free to make any changes you think might help you.

   You will write a file `BST.java` that implements a binary search tree using this `Node` class. You will need to implement the following methods:

   (a) `public void insert(int time, Request req)` : insert an integer `time` (with the associated request `req`) into the binary search tree, which is keyed on `time`.

   (b) `public Node pred(int time)` : return the `Node` within the binary search tree that is the predecessor of the value `time` (i.e., the largest integer smaller than `time`). Return `null` if there is no predecessor. This can be implemented iteratively.

   (c) `public Node succ(int time)` : return the `Node` within the binary search tree that is the successor of the value `time` (i.e., the smallest integer larger than `time`). Return `null` if there is no successor. This can be implemented iteratively.

   (d) `public Node min()` : return the `Node` with the smallest key in the tree. Return `null` if the tree is empty. This can be implemented iteratively.

   (e) `public Node max()` : return the `Node` with the largest key in the tree. Return `null` if the tree is empty. This can be implemented iteratively.

   (f) `public void delete(int time)` : remove the `Node` with the key `time` (if such a node is present, otherwise do nothing).

   (g) `public void print()` : print the contents of the tree in sorted order (by doing an in-order traversal)

   For many of these methods, they may be best implemented as a public wrapper that calls a private, recursive implementation of the method. The public interface for the class will not have references to internal nodes generally, so any method that takes a `Node` as an argument will likely be private.

   `delete(time)` can be implemented recursively. Suppose we try to delete from a subtree rooted at x. The recursive method will return the root of the subtree resulting from calling `delete(node, time)`. If `x == null`, we return `null`, otherwise:

(a) x **is a leaf:** Then x has no children. Therefore, we return `null`. This gets updated by the parent node to effectively remove the node from the tree.

(b) x **has one child** y**:** Then return y. This will update the parent's child pointer to y to remove x from the tree.

(c) x **has two children:** Find $s =$ `succ(x)`, then copy the data fields from s into x, and delete the value of s from x's right subtree.

Two recursive cases remain, to walk down the subtree to the left or right to find the node in question, based on a comparison of the node's key and the value being searched for. Please write your own driver program to test the operations of the binary search tree.

2. **Runway Reservation (10 points):** Now we will implement a single-runway reservation system. Airlines make request reservations for the runway at specific times. In order to accommodate these requests, and to account for delays and buffer, we allow a grace period of length k between successive uses of the runway. All requests are fulfilled first-come, first-served. You will write a program to output all successful requests for runway allocation. That is to say, some requests will be unsuccessful.

   The provided file `Request.java` contains a class representing a single event. This includes a command (either a reservation request or a time command, explained below), the associated time, and an Airline object to store meta-data about an airline (flight name, number, source, and destination). There should be no need to modify this class, but be sure you understand how the class works.

   The provided file `RunwayReservation.java` contains some startup code that reads through an input file (through `System.in`) and loads the events into an array of `Request` objects. The input file contains two numbers, n and k on the first line, where n represents the total number of events and k represents the grace period allowed between reservations. What follows are n lines containing the events described above. These events are either reservation requests (denoted by the "r" command) or an advance time event (denoted by the "t" command). Each "r" command includes information about the time requested and the airline information for the request. Each "t" command includes some integer amount to advance the *current time* by. See the sample input/output below to provide clarity.

   The provided code automatically loads these requests into an array. Use a binary search tree keyed on times to store all valid, successful requests. Requests are always processed in order.

   For each `Request`, do the following:

   - If event is `r`-type, check if the request is valid with respect to already allocated reservations (taking into account the length-k grace period). If the request is valid, add it to the BST.
   - If event is `t`-type, update the current time, and **print and remove** all requests from the BST with times up to and including the current time. These are successful requests who used the runway already.

   For each reservation request for a time of t, we can check if it is valid in $O(\lg n)$ time by checking `pred(t)` and `succ(t)` to determine if there is enough buffer. Keep in mind, reservations that have already passed due to a `t`-type event are still afforded a length-k buffer. Throughout the entirety of the algorithm, each request is inserted at most one time, and removed at most once, which takes $O(\lg n)$ each, for a total running time of $O(n \lg n)$.

The output format can be seen below. For each *advance time* event, we print the new current time, followed by all successful flights that have used the runway since the previous `t`-type event. The final step of the algorithm is an implied `t`-type event to advance the current time to the last valid request time, to output the remaining successful reservations.

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
```

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
```