CS2020: Data Structures and Algorithms (Accelerated)

Problems 6–7

Due: February 7th, 23:59

Overview. You have two tasks this week. The first involves implementing a basic binary search tree. In another problem set later this semester, you will improve your search tree so that it always remains balanced (using a new weight-balancing technique different from that seen in class). The second task involves using a balanced search tree to implement an airport scheduling system.

Collaboration Policy. As always, you are encouraged to work with other students on solving these problems. However, you **must** write up your solution **by yourself**. In addition, when you write up your solution, you **must** list the names of every collaborator, that is, every other person that you talked to about the problem (even if you only discussed it briefly). Any deviation from this policy will be considered cheating, and will be punished severely, including referral to the NUS Board of Discipline.

Problem 6. Basic Binary Search Trees

For this question, your task is to implement a basic binary search tree in Java. Notably, your search tree must satisfy the binary search tree property: every key in a node's left sub-tree must be smaller than the key stored at the node itself, and every key in a node's right sub-tree must be larger than the key stored at the node itself. Inserting an element into the tree and searching for an element in the tree should take time $O(h)$, where h is the height of the tree.

Your solution to this problem set should consist of a class which implements the `ITreeNode` interface distributed with the problem set. A tree node should contain a key, along with a reference to its parent (if any), its left child (if any) and its right child (if any). The `ITreeNode` interface contains the following methods:

- `getLeftChild`, `getRightChild`, `getParent`: returns the left child, the right child, or the parent (or null, if the specified child/parent does not exist).
- `getWeight`: returns the total number of keys stored in the sub-tree rooted at the tree node (see below)
- `getKey`: returns the key associated with the tree node
- `getTreeKeys`: returns a sorted list of every key in the sub-tree rooted at the tree node
- `insert`: inserts an integer into the sub-tree rooted at the tree node
- `search`: queries whether a given integer appears in the sub-tree rooted at the tree node

Each of these functions should be implemented as efficiently as possible.

Notice that one of the required functions returns the *weight* of a node. We define the weight of a tree node as the total number of keys that are stored in that sub-tree. For example, the weight of a leaf node is 1. In general, the weight of a node v is equal to: `v.left.weight + v.right.weight + 1` (where we define `v.left.weight = 0` if v has no left child, and `v.right.weight = 0` if v has no right child). While the weight can be re-calculated on every call to `getWeight`, that can be quite expensive (costing $O(n)$ for querying the weight of the root of a tree of n nodes). Instead, each node in the tree should store its weight. When a node is inserted, the weight is updated accordingly.

Your tree node class should have one constructor which takes two parameters: an integer key and an ITreeNode parent, and initializes the left and right children to `null`.

In addition to the tree node class, you also must implement a static function that builds a *balanced* tree from a *sorted* list of integers and returns the root of the tree:

```
static ITreeNode buildTree(List<Integer> a)
```

Your function for building a tree should take run in $O(n)$ time, and have the minimum height possible. (Note that the original list is already sorted, so there is no need to sort the list.) Ideally, every node in the tree that your function returns should be *perfectly weight balanced*: the weight of the left child and the weight of the right child differ by at most one. See Figure 1 for an example of a tree built by the `buildTree` routine. (Partial credit will be given for slower solutions.)
Hint: think about divide-and-conquer solutions.

Several of the routines in this problem rely on the Java `List<Integer>` interface. (See below for more discussion of this interface.) You may assume that standard operations on list objects are

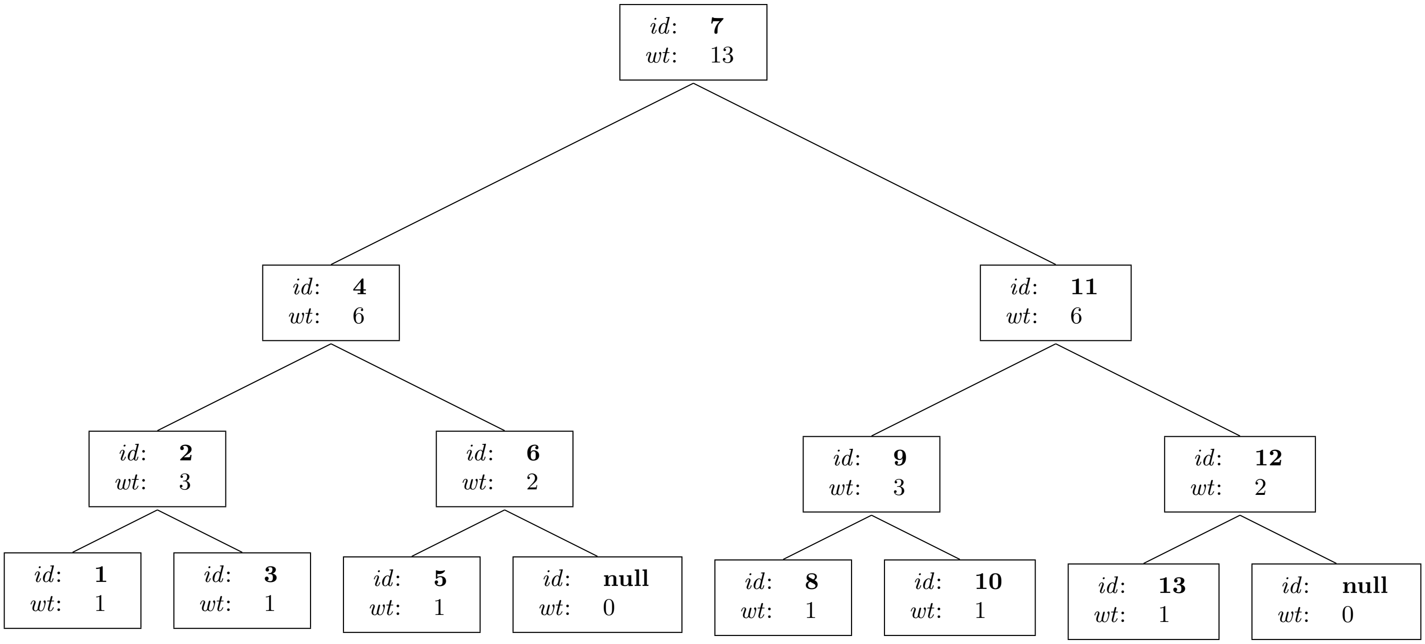

Figure 1: Balanced tree produced by the `buildTree` routine on thirteen nodes. In the diagram, each node contains its identifier (abbreviated *id*) and its weight (abbreviated *wt*). Nodes with identifier **null** indicate a non-existent leaf. Note that the weight of two children differs by at most one.

$O(1)$, for example: `get(...)` and `sublist(...)`. This corresponds to an `ArrayList` implementation of the list interface.

Finally, along with your new class, submit a set of routines for testing your binary search tree. Include several tests that verify whether it is working correctly, and explain why these tests are sufficient to indicate that your code works.

Using the Java List Class. This problem set relies on the java.util.List class, which is part of the Java libraries. (As stated above, you may assume that the list interface is instantiated via java.util.ArrayList, and you may use ArrayList for lists that you need to generate. Using a list, instead of an array, should simplify your code. It automatically manages the size of the list, and you do not have to worry about sizing the array properly. In this problem set, you will use a list as part of the `getTreeKeys` routine and the constructor for the search tree.

By including the statement: `import java.util.List;` and `import java.util.ArrayList;`, you can use the List class and the ArrayList class in your code. You can use a list to store different types of elements. If you declare a variable to be of type `List<Integer>`, then it stores a list of Integers; if you declare a variable to be of type `List<String>`, then it stores a list of Strings. For this problem, you will be using lists of Integers. (Recall that an `Integer` is a class that contains an `int` and can be used much like an `int`.)

The List class (and the ArrayList class) contain a lot of different useful functionality. For the purpose of this problem, you mainly need to know how to add an element to the end of the list, how to access elements in the list, and how to access a sublist of a list. You can find a complete reference to the List class at:

`http://download.oracle.com/javase/6/docs/api/java/util/List.html`

Here, we list a few hints as to how to use a list. First, declare a list of integers:

```
List<Integer> intArray = new ArrayList<Integer>();
```

You can then add elements to the list:

```
intArray.add(7);
```

This code adds the integer 7 to the end of the list. You can also get access to individual elements of the list:

```
int k = 7;
int i = intArray.get(k);
```

which retrieves the seventh element of the list. You can also check if the list is empty (`isEmpty()`), and you can remove an element from the list (`remove(k)` remove the kth element). You can check the size of the list (`size()`). Also notably, you can get a sublist of the original list:

```
int start = 7;
int end = 10;
List<Integer> smallList = intArray.subList(start, end);
```

This returns a view of a portion of the list from `begin` (inclusive) to `end` (exclusive). That is, in this example, the resulting list has three elements.

If you have other questions on how to use a Java list, please ask your tutor during the discussion groups.

Problem 7. (Scheduling Airplanes)

We began our discussion of dynamic dictionaries with an example application: scheduling an airport runway. Consider an airport with a single runway. Pilots call in to the control tower and request a landing time. The air traffic controller must respond by either accepting the request (and scheduling the plane) or rejecting the request. Of critical importance, no two planes can be scheduled within three minutes of each other. Throughout this problem, we will be representing times as integers. (If planes land at one minute intervals, there are at most 10,080 minutes in a week, and hence we can measure time as the number of minutes since the beginning of the week.)

We proposed a solution in which the landing times were stored in a binary search tree. However, we now want some additional functionality. This additional functionality will require associating some additional data with each key in the search tree. (The precise interface can be found in `IRunwaySearch`.) For example, if you are using the data structure to track the ranking of students:

```
insert(1, "Hermione Granger"); // Everyone knows that Hermione is number one
insert(10, "Harry Potter"); // Harry Potter is the 10th ranked student
String student = dataSearch(1) // Who is the top ranked student?
System.out.println("The number one student is: " + student); // Hermione is!
String student = dataSuccessor(3) // After 3, who is next?
System.out.println("The closest student following number 3 is: " + student); // Harry!
int rank = keySuccessor(3) // After 3, who is next?
System.out.println("The closest student following number 3 is number: " + rank); // 10!
```

Assume that you are given a binary search tree implementation with the following interface:

```
void insert(typeA key, typeB data);
boolean search(typeA key);
typeB dataSearch(typeA key);
typeB dataSuccessor(typeA key);
typeB dataPredecessor(typeA key);
typeA keySuccessor(typeA key);
typeA keyPredecessor(typeA key);
void delete(typeA key);
```

You may instantiate this data structure using whatever two types `typeA` and `typeB` that you choose. When you insert a key, at the same time, you may specify some additional data. When you search for the key, it returns whether any such key has been previously inserted into the tree. When you perform a `dataSearch` for a given key, it returns the data associated with that key, and when you perform a `successor` query, it returns the data associated with the smallest key larger than the value queried[1]. You may assume that every operation is performed in time $O(\log n)$.

Your task is to give a data structure that supports the following functionality, which is defined in the included interface `IRunwayScheduler`:

[1] Note that the tree implementation does not support in-order-traversal, and you will have to make do with successor and predecessor queries.

```
boolean requestTimeSlot(int time, String pilot)
int getNextFreeSlot(int time)
int getNextPlane(int time)
String getPilot(int time)
List<String> getPilots()
List<Integer> getPilotSchedule(String pilot)
```

When scheduling a plane via `requestTimeSlot`, you are given the time at which the plane wishes to land and the name of the pilot. You may either schedule the plane and return `true`, if the time slot is available, or reject the plane and return `false`. Sometimes, a pilot may ask for the next available slot. In that case, the air traffic controller will call `getNextFreeSlot` (with the earliest acceptable time specified). This function should return the first time at which the plane can be scheduled. Note that `getNextFreeSlot` does not actually schedule the plane; it only notifies the pilot of a good time to request.

The air traffic controller also sometimes wants to enumerate the scheduled planes for the day, and it will accomplish this by calling `getNextPlane`, which returns the time at which the next plane will land. The air traffic controller can then call `getPilot` to determine the pilot of the plane that is scheduled to land at that time.

Finally, the air traffic controller sometimes likes to generate an alphabetic list of all the pilots and their schedule landing times. For this purpose, you should implement the `getPilots` routine which returns a list of all the pilots (alphabetically, by name), and the `getPilotSchedule` routine which returns all the times at which a given pilot has been scheduled.

For the purpose of this problem, you may assume that each pilot has a unique name. However, some pilots may be scheduled to land more than one plane. That is, a given pilot may request multiple time slots. Design each of these routines to be as efficient as possible. For this problem, you may submit your solution in pseudocode, rather than Java (though we have provided Java interfaces for those who want to use Java). Along with the code/pseudocode, be sure to explain the overall design of your data structure and the precise behavior of each of the functions. (Do not forget to specify what happens when errors occur.) The following is an example execution:

```
boolean answer = requestTimeSlot(300, "Seth"); // Request slot 300
System.out.println("Answer: " + answer); // Success!

answer = requestTimeSlot(400, "Steven"); // Request slot 400
System.out.println("Answer: " + answer); // Success!

answer = requestTimeSlot(302, "Steven"); // Request slot 302
System.out.println("Answer: " + answer); // Failure!

answer = requestTimeSlot(398, "Seth"); // Request slot 398
System.out.println("Answer: " + answer); // Failure!

int nextFree = getNextFreeSlot(398); // When can I land?
System.out.println("Next free slot: " + nextFree); // I can land at 403
```

```
answer = requestTimeSlot(nextFree, "Seth"); // Request next free slot
System.out.println("Answer: " + answer); // Success!

int nextPlane = getNextPlane(350); // When is the next plane landing?
System.out.println("Next plane: " + nextPlane); // At time=400

String pilot = getPilot(nextPlane); // Who is landing then?
System.out.println("Next pilot: " + pilot); // Steven is landing at 400

List<String> pilots = getPilots(); // List all the pilots
for (int i=0; i<pilots.size(); i++){
    System.out.println("Pilot " + pilots.get(i));
}
// Outputs Seth and Steven (in sorted order, alphabetically)

List<Integer> landingTimes = getPilotSchedule("Seth"); // When is Seth landing?
for (int i=0; i<landingTimes.size(); i++){
    System.out.println("Time: " + landingTimes.get(i));
}
// Outputs that Seth is landing at slot 300 and 403.
```