COMPUTER SCIENCE 311
SPRING 2020
PROGRAMMING ASSIGNMENT 1
INTERVAL TREAPS
DUE: 11:59 P.M., APRIL 5

Note. *This assignment is adapted from Chapter 14 of CLRS[1]. The chief difference with respect to CLRS is that here we use treaps instead of red-black trees.*

1. Overview

A ***closed interval*** is an ordered pair of real numbers $[a, b]$, with $a \leq b$. The interval $[a, b]$ represents the set $\{x \in \mathbb{R} : a \leq x \leq b\}$, where $\mathbb{R}$ denotes the set of all real numbers.

An ***interval database*** is a data structure that maintains a dynamic set of elements, with each element x containing an interval $x.\texttt{interv}$. An interval database supports the following operations:

- `intervalInsert`(x)**:** adds the element x, whose `interv` field references an interval, to the database.
- `intervalDelete`(x)**:** removes the element x from the database.
- `intervalSearch`(i)**:** returns a reference to an element x in the interval database such that $x.\texttt{interv}$ overlaps interval i, or `null` if no such element is in the set.

Intervals are convenient for representing events that each occupy a continuous period of time. We might, for example, wish to query a database of time intervals to find out what events occurred during a given interval. The goal of this assignment is to design a data structure based on treaps that efficiently maintains an interval database.

Teamwork. For this programming assignment, you can work in teams of two people. It is your responsibility to find a teammate, if you wish to have one.

This project is worth 100 points, plus at most 15 extra credit points.

2. The Data Structure

We represent an interval $[a, b]$ as an object i, with attributes $i.\texttt{low} = a$ (the low endpoint) and $i.\texttt{high} = b$ (the high endpoint). We say that intervals i and i' ***overlap*** if $i \cap i' \neq \emptyset$, that is, if $i.\texttt{low} \leq i'.\texttt{high}$ and $i'.\texttt{low} \leq i.\texttt{high}$. The following fact, illustrated in Figure 1, is known as the ***interval trichotomy***.

Fact. *Any two intervals i and i' satisfy exactly one of the following three properties:*

(a) i and i' overlap,
(b) i is to the left of i' (i.e., $i.\texttt{high} < i'.\texttt{low}$),
(c) i is to the right of i' (i.e., $i'.\texttt{high} < i.\texttt{low}$).

---

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms* (3rd edition), MIT Press, 2009.

Originally posted March 3. 1st revision, March 4; changes in blue. 2nd revision, March 16; significant changes in purple. 3rd revision, March 20; modified due date in green.

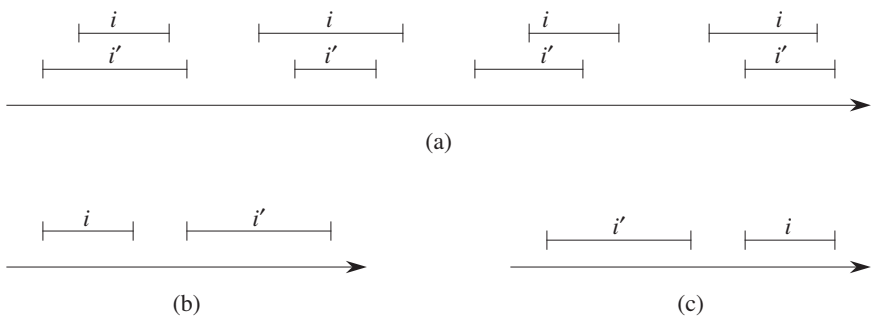

FIGURE 1. The interval trichotomy for two closed intervals i and i'. (a) If i and i' overlap, there are four situations; in each, i.low $\leq i'$.high and i'.low $\leq i$.high.(b) The intervals do not overlap, and i.high $< i'$.low. (c) The intervals do not overlap, and i'.high $< i$.low. (Source: CLRS.)

```
1 intervalSearch(i):
2 |  x = T.root
3 |  while x ≠ null and i does not overlap x.interv do
4 |  |  if x.left ≠ null and x.left.imax ≥ i.low then
5 |  |  |  x = x.left
6 |  |  else
7 |  |  |  x = x.right
8 |  return x
```

Algorithm 1: Searching for an interval in the database that overlaps interval i.

In this project, you will represent an interval database using a treap T, called an ***interval treap***, in which each node x contains an interval x.interv and the key of x is the low endpoint, x.interv.low, of the interval. Thus, an inorder tree walk of T lists the intervals in sorted order by low endpoint. In addition to the intervals themselves, each node x of T contains a value x.imax, which is the maximum value of any interval endpoint stored in the subtree rooted at x. See Figure 2. We use the imax field to implement intervalSearch, as shown in Algorithm 1.

The search for an interval that overlaps i starts with x at the root of the tree and proceeds downward. It terminates when either it finds an overlapping interval or x == null. The key idea behind intervalSearch is that at any node x, if x.interv does not overlap i, the search always proceeds in a safe direction:

- Suppose x.left $\neq$ null and x.left.imax $\geq i$.low. Then, if there exists an interval that overlaps i, the left subtree must contain such an interval.[2] Thus, we proceed to the left.
- Suppose x.left == null or x.left.imax $< i$.low. Then, no interval in x's left subtree overlaps i. Thus, we proceed to the right.

Thus, the search will find an overlapping interval if the interval treap contains one. Since each iteration of the **while** loop takes $O(1)$ time and the expected height of an n-node treap is $O(\log n)$, intervalSearch takes expected $O(\log n)$ time.

[2]Note that it may also be an interval in the right subtree that overlaps i.

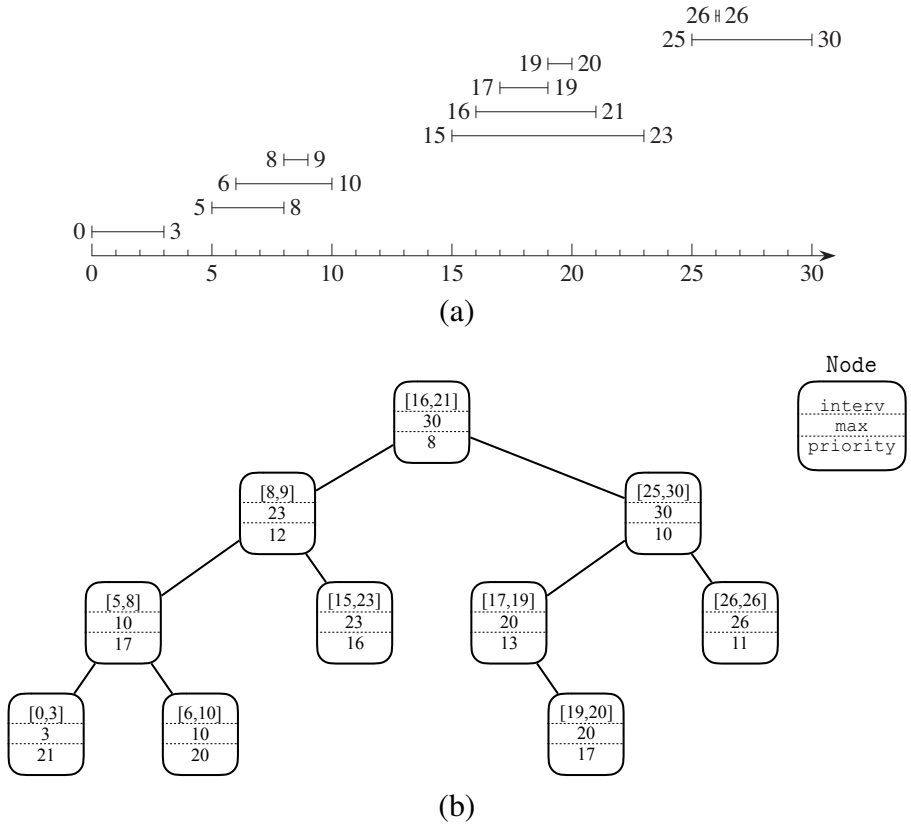

FIGURE 2. An interval treap. (a) A set of 10 intervals, shown sorted bottom to top by left endpoint. (Source: CLRS.) (b) The interval treap that represents them. Each node x contains, from top to bottom, an interval, the maximum value of any interval endpoint in the subtree rooted at x, and a priority field. An inorder tree walk of the tree lists the nodes in sorted order by left endpoint.

Example 1: Successful search. Suppose we wish to find an interval that overlaps the interval $i = [22, 25]$ in the interval treap of Figure 2. We begin with x as the root, which contains $[16, 21]$ and does not overlap i. Since $x.\texttt{left.imax} = 23$ is greater than $i.\texttt{low} = 22$, the loop continues with x as the left child of the root — the node containing $[8, 9]$, which also does not overlap i. This time, $x.\texttt{left.imax} = 10$ is less than $i.\texttt{low} = 22$, and so the loop continues with the right child of x as the new x. Because the interval $[15, 23]$ stored in this node overlaps i, the procedure returns this node.

Example 2: Unsuccessful search. Suppose we wish to find an interval that overlaps $i = [11, 14]$ in the interval treap of Figure 2. We once again begin with x as the root. Since the root's interval $[16, 21]$ does not overlap i, and since $x.\texttt{left.imax} = 23$ is greater than $i.\texttt{low} = 11$, we go left to the node containing $[8, 9]$. Interval $[8, 9]$ does not overlap i, and $x.\texttt{left.imax} = 10$ is less than $i.\texttt{low} = 11$, and so we go right. (Note that no interval in the left subtree overlaps i.) Interval $[15, 23]$ does not overlap i, and its left child is `null`, so again we go right, the loop terminates, and we return `null`.

3. Updating the Interval Treap

You must implement interval insertion and interval deletion on an interval treap so that these operations run in expected $O(\log n)$ time, where n is the number of intervals. This time bound includes the work in updating all necessary data fields in the interval treap. Insertions and deletions on an interval treap are performed by modifying the corresponding operations for an ordinary treap. We outline these modifications next — you must supply the additional details.

3.1. **Insertion.** As seen in class, inserting a node z into an ordinary treap is done in two phases. The first phase assigns a random priority to z and then goes down the tree from the root, following a path determined by z.`key`, inserting z as a child of an existing node. The second phase goes up the tree, performing rotations to satisfy the constraint that v.`priority` > v.`parent`.`priority` for every node v in the treap, except the root.

Now suppose we want to insert a node z, where z.`interv` references an interval and z.`priority` has been assigned a random integer, into an interval treap. We begin by making z.`imax` = z.`interv`.`high`. We then proceed to the first phase, which is conducted as in an ordinary treap, except that we now use z.`interv`.`low` as the key of z, and we make x.`imax` = max(x.`imax`, z.`interv`.`high`) for each node x on the simple path traversed from the root down toward the leaves. Since the expected number of nodes on the traversed path is $O(\log n)$, the additional cost of maintaining the `imax` fields is $O(\log n)$.

In the second phase, the only structural changes to the treap are caused by rotations, which depend only on the node priorities. A rotation is a local operation: only two nodes have their `imax` fields invalidated. We can update these fields in $O(1)$ time per rotation using the fact that the value of x.`imax` for any node x can be determined from x.`interv` and the `imax` values of node x's children as follows.

$$x.\texttt{imax} = \begin{cases} x.\texttt{interv}.\texttt{high} & \text{if } x \text{ is a leaf} \\ \max\{x.\texttt{interv}.\texttt{high},\ x.\texttt{left}.\texttt{imax}\} & \text{if } x.\texttt{right} == \texttt{null} \\ \max\{x.\texttt{interv}.\texttt{high},\ x.\texttt{right}.\texttt{imax}\} & \text{if } x.\texttt{left} == \texttt{null} \\ \max\{x.\texttt{interv}.\texttt{high},\ x.\texttt{left}.\texttt{imax},\ x.\texttt{right}.\texttt{imax}\} & \text{otherwise.} \end{cases}$$

The expected number of rotations is bounded by the expected height of the treap, which is $O(\log n)$. Thus, the expected time to perform an insertion, including maintaining the `imax` fields, is $O(\log n)$. Figure 3 shows an example of interval insertion.

3.2. **Deletion.** Deleting of a node z from an ordinary treap is also done in two phases. The first phase is exactly like deletion in an ordinary binary search tree, and has three cases[3]:

(1) z *has no left child:* Then, replace z by its right child, which may be `null`.
(2) z *has a left child, but no right child:* Then, replace z by its left child.
(3) z *has two children:* Then, replace z by its successor y = `Minimum`(z.`right`).

The second phase rotates z's replacement y (assuming y is not null) down the tree as far as necessary to satisfy the constraint that v.`priority` > v.`parent`.`priority` for every node v.

Now, suppose we want to delete a node z from an interval treap. We begin by deleting z as in the first phase of ordinary treap deletion. Next, we update the necessary `imax` fields by traversing a path up the treap.

[3]Refer to CLRS, Chapter 12, and pages 15–22 of the slides on binary search trees.

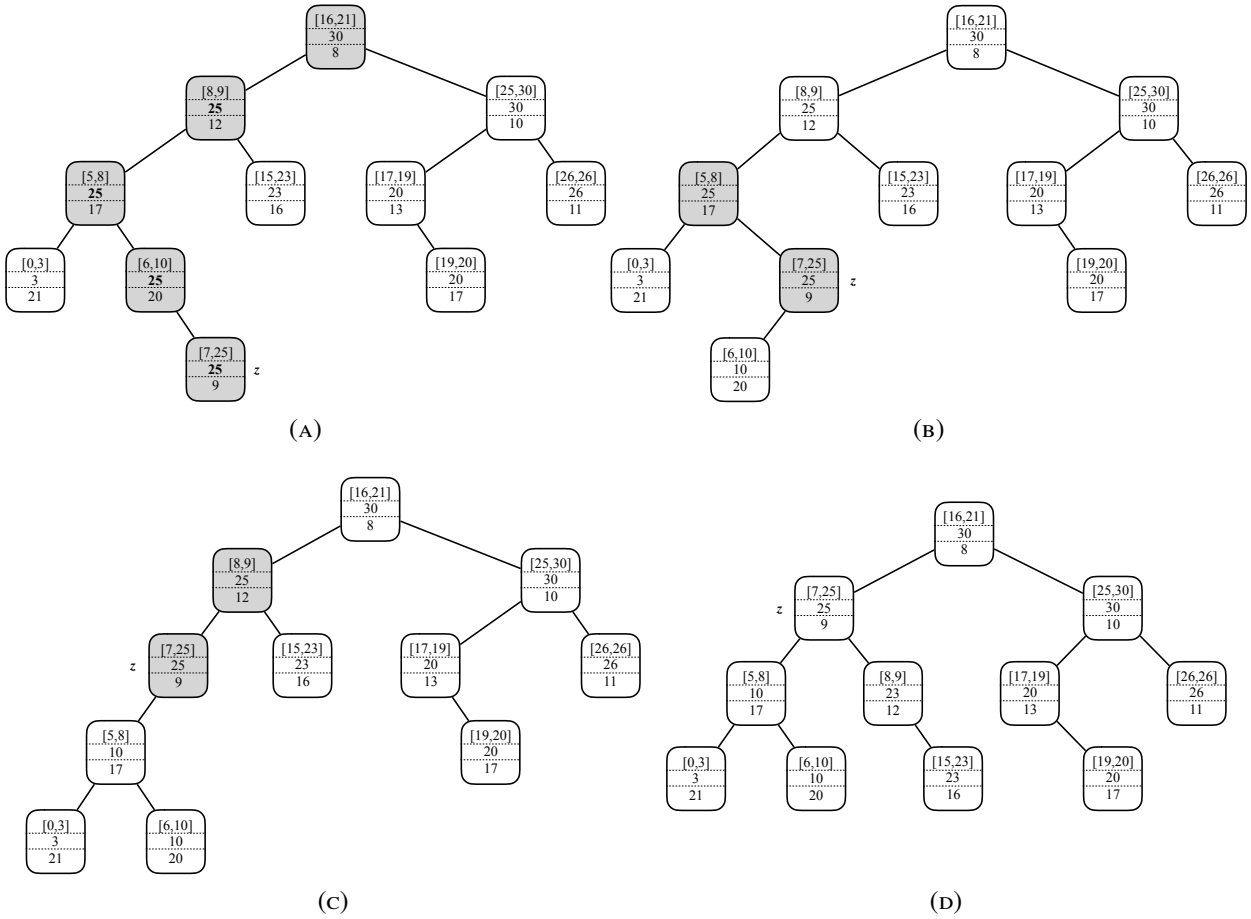

FIGURE 3. Inserting node z, with $z.\texttt{interv} = [7,25]$, into the interval treap of Figure 1(b). (A) The interval treap after phase 1. The nodes traversed are shaded, and the modified `imax` values are shown in boldface. Node z is assigned a priority of 9, which is lower than the priority of z's parent. (B) After the first rotation, $z.\texttt{priority} < z.\texttt{parent.priority}$. (C) After the second rotation, we still have $z.\texttt{priority} < z.\texttt{parent.priority}$. (D) After the third rotation, all the priority constraints are satisfied. Notice how `imax` fields are updated after each rotation.

- If z has been replaced by `null`, the path goes from the former parent of z to the root of the treap.
- If z has been replaced by a node y, the path starts at y's original position in the treap and goes up to the root.

In both cases, we decrement the `imax` field of each node on the path, as needed. Since the expected length of this path is $O(\log n)$ in an n-node treap, the time spent in traversing the path to update `imax` fields is $O(\log n)$.

Finally, if z has been replaced by a node y, we rotate y downward as in an ordinary treap, in order to reestablish the correct priority relationships. We handle the rotations in the same manner as for insertion — that is, we update `imax` fields after each rotation. Thus, like insertion, deletion takes expected $O(\log n)$ time for an n-node interval treap.

Note. *We strongly encourage you to draw several examples of deletion from an interval treap, in order to understand the work involved. Once you have developed an intuition for the process, you can develop the pseudocode for deletion.*

4. Requirements

Here we list the Java classes and methods that your program must support. You may, of course, implement other auxiliary classes and methods, as needed.

4.1. **`Interval`.** The `Interval` class represents intervals. To avoid problems with numerical precision, the endpoints of intervals will be integers. The class must provide the following methods, all of which must run in constant time.

- `Interval(int low, int high)`: Constructor with two parameters: the `low` and `high` endpoints.
- `int getLow()`: Returns the low endpoint of the interval.
- `int getHigh()`: Returns the high endpoint of the interval.

4.2. **`Node`.** The `Node` class represents the nodes of the treap. The following methods must be provided, all of which must run in constant time.

- `Node (Interval i)`: Constructor that takes an `Interval` object i as its parameter. The constructor must generate a priority for the node. Therefore, after creation of a `Node` object, `getPriority()` (defined below) must return the priority of this node.
- `Node getParent()`: Returns the parent of this node.
- `Node getLeft()`: Returns the left child of this node.
- `Node getRight()`: Returns the right child of this node.
- `Interval getInterv()`: Returns the interval object stored in this node.
- `int getIMax()`: Returns the value of the `imax` field of this node.
- `int getPriority()`: Returns the priority of this node.

4.3. **`IntervalTreap`.** The `IntervalTreap` class represents an interval treap. The following methods must be provided, the first four of which should run in constant time.

- `IntervalTreap()`: Constructor with no parameters.
- `Node getRoot()`: Returns a reference to the root node.
- `int getSize()`: Returns the number of nodes in the treap.
- `int getHeight()`: Returns the height of the treap.
- `void intervalInsert(Node z)`: adds node `z`, whose `interv` attribute references an `Interval` object, to the interval treap. This operation must maintain the required interval treap properties. The expected running time of this method should be $O(\log n)$ on an n-node interval treap.
- `void intervalDelete(Node z)`: removes node `z` from the interval treap. This operation must maintain the required interval treap properties. The expected running time of this method should be $O(\log n)$ on an n-node interval treap.
- `Node intervalSearch(Interval i)`: returns a reference to a node `x` in the interval treap such that `x.interv` overlaps interval `i`, or `null` if no such element is in the treap. This method must not modify the interval treap. The expected running time of this method should be $O(\log n)$ on an n-node interval treap.

5. Extra Credit

For extra credit, add the following two methods to `IntervalTreap`.

- `Node intervalSearchExactly(Interval i)`: Returns a reference to a `Node` object `x` in the treap such that `x.interv.low = i.low` and `x.interv.high = i.high`, or `null` if no such node exists. The expected running time of this method should be $O(\log n)$ on an n-node interval treap. *(5 extra credit points)*
- `List<Interval> overlappingIntervals(Interval i)`: returns a list all intervals in the treap that overlap `i`. This method must not modify the interval treap. The expected running time of this method should be $O(\min(n, k \log n))$, where k is the number of intervals in the output list. *(10 extra credit points)*

6. Guidelines on Code Submission

- Use the Java default package (unnamed package). While this is not good coding practice for larger applications, it is more convenient for testing.
- Make all the methods and constructors explicitly public.
- You may design helper classes and methods in addition to those listed in Sections 4 and 5. However, every class and method listed in that section must be implemented ***exactly as specified***. This includes
  - the names of methods and classes (*remember that Java is case-sensitive*),
  - the return types of the methods, and
  - the types and order of parameters of each method/constructor.

  If you fail to follow these requirements, you will lose a significant portion of the points, even if your program is correct.
- You are not allowed to have external JARs as dependencies. You may use built-in packages such as `java.util.List`.
- Please include all team members names as a JavaDoc comment in each of the Java files.
- Create the project folder as follows: `<directory-name>/src/*.java`. This folder should nclude *only* `.java` files. Do not include any `.class` or other files.
- Create a zip file of your project folder.
- Upload your zip file on Canvas. Only one submission per team.

Your grade will depend on adherence to specifications, correctness, and efficiency.

Programs that do not compile will receive zero credit.

Important Note

Some aspects of this specification are subject to change in response to issues detected by students or the course staff. ***Check Canvas and Piazza regularly for updates and clarifications.***