COMPUTER SCIENCE 311
SPRING 2019
PROGRAMMING ASSIGNMENT 1
MAINTAINING A POINT OF MAXIMUM OVERLAP
DUE: 11:59 P.M., MARCH 31

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. Suppose we are given a collection of closed intervals $\mathcal{I} = \{I_1, I_2, \ldots, I_n\}$, where $I_j = [a_j, b_j]$. We say that an interval I_j *overlaps* a point x in the real line if $a_j \leq x \leq b_j$. We say that x is a *point of maximum overlap* if it has largest number of intervals in $\mathcal{I}$ overlapping it (see Figure 1). Note that, in general, there may be multiple points of maximum overlap.

The problem of finding a point of maximum overlap has several applications. Suppose, for example, that you run a discussion board on the web. Users log in and log out throughout the day. Each user has an associated time interval, which is given by the start and end times of the user's activity. Finding the peak usage time for the discussion board reduces to finding a point of maximum overlap for this set of intervals.

The goal of this assignment is to design a data structure based on red-black trees that efficiently maintains a point of maximum overlap in a collection of intervals $\mathcal{I}$ under a series of insertions and (optionally) deletions of intervals.

Teamwork. For this programming assignment, you should work in teams of two or three. It is your responsibility to assemble a team.

Reading material. To solve this project, it will be helpful to review Chapters 12, 13, and 14 of

> Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, *Introduction to Algorithms* (3rd edition), MIT Press, 2009.

Copies of these chapters are available on Canvas.

2. The Key Idea

The data structure that you will implement is based on the following easy to verify fact.

> *There always exists a point of maximum overlap that is an endpoint of one of the inrevals.*

See Figure 1 for an illustration.

Based on the above observation, our goal is to find, among the endpoints of the intervals in $\mathcal{I}$, an endpoint that is a point of maximum overlap. To do this, associate with each endpoint e an associated value $p(e)$:

$$p(e) = \begin{cases} +1 & \text{if } e \text{ is a left endpoint and} \\ -1 & \text{if } e \text{ is a right endpoint.} \end{cases}$$

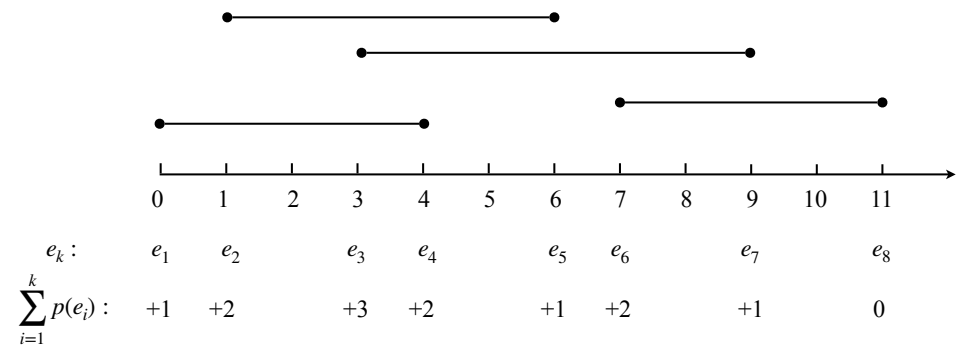

FIGURE 1. A collection of closed intervals $\mathcal{I} = \{I_1, I_2, I_3, I_4\}$. The endpoints are $\langle e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\rangle = \langle 0, 1, 3, 4, 6, 7, 9, 11\rangle$. The maximum overlap is achieved by any point $x \in [3, 4]$. Observe that $s(1, j)$ is maximized for $j = 3$ ($s(1, 3) = 3$), indicating that e_3 is a point of maximum overlap.

Let $e_1, e_2, \ldots, e_{2n}$ be the sorted sequence of endpoints in the set of intervals $\mathcal{I}$. For $1 \le i \le j \le 2n$, let us define $s(i, j)$ as

$$s(i, j) = p(e_i) + p(e_{i+1}) + \cdots + p(e_j).$$

Any endpoint e_i that maximizes $s(1, i)$ is a point of maximum overlap (see Figure 1).

3. The Data Structure

Your program must maintain a red-black tree T of all the endpoints. Let v be the node of T that stores endpoint e. Node v has fields v.`key` and v.`p`, which equal e and $p(e)$, respectively. It also has the usual `parent`, `left`, `right`, and `color` fields. Node v will contain additional fields in order to maintain the point of maximum overlap. Let ℓ_v and r_v denote the indices of the leftmost and rightmost endpoints in the subtree rooted at v. Then, x has the following three fields (see Figure 2).

- v.`val` is the sum of the `p`-values of the nodes in the subtree rooted at v (including v itself); that is, v.`val` $= s(\ell_v, r_v)$.
- v.`maxval` is the maximum value obtained by the expression $s(\ell_v, i)$ for $\ell_v \le i \le r_v$.
- v.`emax` is a reference to an endpoint e_m, where m is the value of i that maximizes $s(\ell_v, i)$ over all i such that $\ell_v \le i \le r_v$.

Assuming these fields are maintained correctly, the root of the tree, T.`root`, provides the answer to a point of maximum overlap query in $O(1)$ time: T.`root.emax` is a reference to a point of maximum overlap and T.`root.maxval` is the number of intervals that overlap this point.

We can compute the `val`, `maxval`, and `emax` fields in a bottom-up fashion. For example, we can compute v.`val` using the following recursive definition:

$$v.\texttt{val} = \begin{cases} 0 & \text{if } v == T.\texttt{nil}, \\ v.\texttt{left.val} + v.\texttt{p} + v.\texttt{right.val} & \text{otherwise.} \end{cases}$$

To compute v.`maxval`, there are two possibilities. If v == T.`nil`, then v.`maxval` = 0 . Otherwise, there are three cases:

(1) the maximum is in v's left subtree,
(2) the maximum is at v, or
(3) the maximum is in v's right subtree.

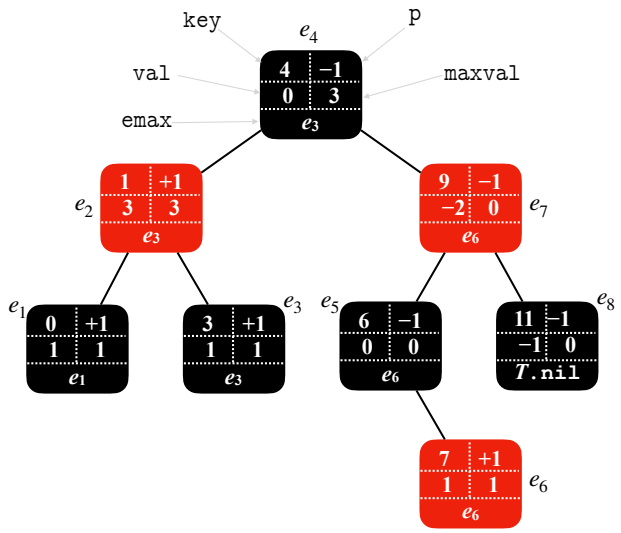

FIGURE 2. A red-black tree representation of the set of intervals of Figure 1. Not shown are the `parent`, `left`, `right`, and `color` fields. T.`nil` is also not shown.

This leads to the following expression for v.`maxval`:

$$v.\texttt{maxval} = \max\{\underbrace{v.\texttt{left.maxval}}_{\text{Case 1}}, \underbrace{v.\texttt{left.val} + v.\texttt{p}}_{\text{Case 2}}, \underbrace{v.\texttt{left.val} + v.\texttt{p} + v.\texttt{right.maxval}}_{\text{Case 3}}.\}$$

We leave the derivation of an expression for v.`emax` as an exercise..

Updating the fields dynamically. When a new interval $I = [a, b]$ is added, we must insert two new nodes into T: one for a and one for b. In doing so, we must update the `val`, `maxval`, and `emax` fields (in addition to the `parent`, `left`, `right`, and `color` fields) of various other nodes.

> *You must implement the interval insertion so that it runs in $O(\log n)$ time, where n is the number of intervals. This time bound includes the work in updating all necessary data fields in the tree.*

To understand how to achieve $O(\log n)$ insertion time, it will be helpful to read Chapter 14 of Cormen, Leiserson, Rivest, and Stein. Theorem 14.1 of that text outlines the approach to follow, and Section 14.3 describes an application of red-black trees to a different problem involving intervals.

> *You are not required to implement interval deletion, but bonus points will be given to those who do. If you do implement deletion, your code should handle it in $O(\log n)$ time.*

Endpoints with the same value. Your program should allow the possibility of multiple endpoints having the same value. To handle such endpoints, you should give precedence to the left endpoints over any right endpoints with the same value.

4. SPECIFICATIONS

Your submission must be a Java program that contains the four classes described in this section: `Intervals`, `Endpoint`, `Node`, and `RBTree`. You *must* make these classes and their methods public. Each of the classes may contain additional methods apart from the required ones. Be judicious in designing the classes and the data structures.

4.1. **Intervals.** The `Intervals` class represents a collection of intervals. It must have the following methods.

- `Intervals()`: Constructor with no parameters.
- `void intervalInsert(int a, int b)`: Adds the interval with left endpoint a and right endpoint b to the collection of intervals. Each newly inserted interval must be assigned an ID. The IDs should be consecutive; that is, the ID of the interval inserted on the ith call of this method should be i. For example if `intervalInsert` is called successively to insert intervals $[5, 7], [4, 9], [1, 8]$, then the IDs of these intervals should be $1, 2, 3$, respectively. These IDs are permanent for the respective intervals. Keep track of the IDs, as multiple intervals that have the same endpoints on both sides can be added. `intervalInsert` should run in $O(\log n)$ time.
- `boolean intervalDelete(int intervalID)`: Deletes the interval whose ID (generated by `intervalInsert`) is `intervalID`. Returns `true` if deletion was successful. This method should run in $O(\log n)$ time.

  Note. The `intervalDelete` method is *optional*; that is, you are not required to implement it. However, your code ***must*** provide an `intervalDelete` method even if you choose not to implement interval deletion. If you do not implement deletion, the `intervalDelete` method should consist of just one line that returns `false`.
- `int findPOM()`: Finds the endpoint that has maximum overlap and returns its value. This method should run in constant time.
- `RBTree getRBTree()`: Returns the red-black tree used, which is an object of type `RBTree`.

4.2. **Endpoint.** The `Endpoint` class represents an endpoint and its value. It must have the following method:

- `int getValue()`: Returns the endpoint value. For example if the `Endpoint` object represents the left endpoint of the interval $[1, 3]$, this would return 1.

4.3. **Node.** The `Node` rclass epresents the nodes of the red-black tree. Use 0 and 1 to represent the colors red and black, respectively. The following methods must be provided.

- `Node getParent()`: Returns the parent of this node.
- `Node getLeft()`: Returns the left child.
- `Node getRight()`: Returns the right child.
- `int getKey()`: Returns the endpoint value, which is an integer.
- `int getP()`: Returns the value of the function `p` based on this endpoint.
- `int getVal()`: Returns the `val` of the node as described in this assignment.
- `int getMaxVal()`: Returns the `maxval` of the node as described in this assignment.
- `Endpoint getEndpoint()`: Returns the `Endpoint` object that this node represents.
- `Endpoint getEmax()`: Returns an `Endpoint` object that represents e_{max}. Calling this method on the root node will give the `Endpoint` object whose `getValue()` provides a point of maximum overlap.
- `int getColor()`: Returns 0 if red. Returns 1 if black.

4.4. **RBTree.** The `RBTree` class represents the red-black tree. The following methods must be provided.

- `RBTree()`: Constructor with no parameters.

- `Node getRoot()`: Returns the root node.
- `Node getNILNode()`: Returns the `nil` node.

  Note. *A red-black tree T must contain* ***exactly one*** *instance of T*.`nil`.
- `int getSize()`: Returns the number of internal nodes in the tree.
- `int getHeight()`: Returns the height of the tree.

5. 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 Section 4. However, every class and method listed in Section 4 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 inbuilt 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.***