```java
/**
 * COSC 311 - Program 3 (threaded)
 *
 * This file defines the Index data type as a threaded binary search tree.
 *  Index instances can be added to, searched, and deleted from. Additionally,
 *  a pointer is stored so that the Index may be traversed efficiently.
 *
 * @author Mordechai Sadowsky
 * @version 25-mar-2014
 *
 */
public class Index {

    /**
     * This inner class describes the <code>IndexRecord</code> data type.
     *  IndexRecords contain a key value (e.g. first name); a reference number
     *  so they can be associated with the main record in the database; and
     *  references to its left and right children in the tree.
     *
     * Furthermore, as nodes in a threaded tree, each child reference has a
     *  corresponding boolean value designating whether it is a thread reference
     *  or a hard-linked edge.
     *
     */
    private class IndexRecord {

        String key;
        int recordNumber;
        IndexRecord left, right;
        boolean leftIsThread, rightIsThread;

        IndexRecord(String k, int recNum) {
            this.key = k;
            this.recordNumber= recNum;
            left = null;
            right = null;
            leftIsThread = true;
            rightIsThread = true;
        }
    }

    private IndexRecord root;
    private IndexRecord iterator;
    private int size;

    public Index() {
        root = null;
        iterator = null;
        size = 0;
    }

    public boolean isEmpty() {
        return size == 0;
    }

    /**
```

```
     * Adds a new student <code>IndexRecord</code> to the <code>Index</code>.
     *
     * @param k is the new key value (e.g. first name)
     * @param rN is the new <code>recordNumber</code> reference to the full
     *  student record in the <code>database</code>
     */
    public void insert(String k, int rN) {
        IndexRecord newRecord = new IndexRecord(k, rN);
        size++;
        if (this.isEmpty()) {
            root = newRecord;
            return;
        }
        IndexRecord rover = root;
        int comparison = 0;
        boolean wentLeft = false;

        while (true) {
            comparison = newRecord.key.compareTo(rover.key);
            if (comparison >= 0) {
                newRecord.left = rover; //setting left thread as rover traverses
                wentLeft = false;
                if (rover.rightIsThread) //reached bottom of tree
                    break;
                else
                    rover = rover.right;
            }
            else {
                newRecord.right = rover;//setting right thread as rover
                    traverses
                wentLeft = true;
                if (rover.leftIsThread) //reached bottom of tree
                    break;
                else
                    rover = rover.left;
            }
        }
        if (wentLeft) {
            rover.left = newRecord;
            rover.leftIsThread = false;
        }
        else {
            rover.right = newRecord;
            rover.rightIsThread = false;
        }
    }

    /**
     * Removes a student record from the <code>Index</code>
     *
     * @param recNum is the reference <code>recordNumber</code> of the student
     *  to be deleted
     */
    public void delete(int recNum) {
        this.size--;
        IndexRecord rover = root, roverParent = root;
        boolean wentLeft = false;

        while (true) { //first find the IndexRecord that matches recNum...
            if (recNum == rover.recordNumber)
                break;

            roverParent = rover;
            if (recNum > rover.recordNumber) {
                wentLeft = false;
                rover = rover.right;
            }
            else {
                wentLeft = true;
                rover = rover.left;
            }
        }

        //...Then delete it. Now rover is the IndexRecord to delete. Three
            cases:
        if (rover.leftIsThread && rover.rightIsThread) { //1. Rover is a leaf
            if (wentLeft) {
                roverParent.left = rover.left;
                roverParent.leftIsThread = true;
            }
            else {
                roverParent.right = rover.right;
                roverParent.rightIsThread = true;
            }
        }
        else if (rover.leftIsThread || rover.rightIsThread) { //2. Has one child
            if (wentLeft) {
                if (rover.leftIsThread)
                    roverParent.left = rover.right;
                else
                    roverParent.left = rover.left;
            }
            else {
                if (rover.leftIsThread)
                    roverParent.right = rover.right;
                else
                    roverParent.right = rover.left;
            }
        }
        else { //3. Rover has two children

            /* Find in-order successor of rover, move its children up to its
             * parent and set its right child to rover's right*/
            IndexRecord successor = getSuccessorForDeletion(rover);

            //set rover's parent to point to rover's in-order successor instead
            if (rover == root)
                root = successor;
            else if (wentLeft)
                roverParent.left = successor;
            else
                roverParent.right = successor;
            //Finish setting rover's successor's children to rover's children
            successor.left = rover.left;
            successor.leftIsThread = false;
        }
    }

    /**
     * Resets the tree pointer <code>iterator</code> to the smallest
     *  <code>IndexRecord</code> in the tree.
     */
    public void setIteratorFront() {
        iterator = root;
        while (iterator.left != null) {
            iterator = iterator.left;
        }
    }

    /**
     * Resets the tree pointer <code>iterator</code> to the largest
     *  <code>IndexRecord</code> in the tree.
     */
    public void setIteratorBack() {
        iterator = root;
        while (iterator.right != null) {
            iterator = iterator.right;
        }
    }

    /**
     * Moves the tree pointer <code>iterator</code> to the next larger
     *  <code>IndexRecord</code> in the tree.
     */
    public void iterateForward() {
        if (!iterator.rightIsThread)
            iterator = getSuccessor(iterator);
        else
            iterator = iterator.right;
    }

    /**
     * Moves the tree pointer <code>iterator</code> to the next smaller
     *  <code>IndexRecord</code> in the tree.
     */
    public void iterateBackward() {
        if (!iterator.leftIsThread)
            iterator = getPredecessor(iterator);
        else
            iterator = iterator.left;
    }

    /**
     * @return the <code>recordNumber</code> of the <code>IndexRecord</code>
     *  to which the <code>iterator</code> currently points.
     */
    public int getIteratorRecNum() {
        return iterator.recordNumber;
    }


    /**
     * Finds the in-order successor of a given <code>IndexRecord</code>, passes
     *  the successor's children to the successor's parent and takes on the
         right
     *  child of the given <code>IndexRecord</code>.
     *
     * @param n an <code>IndexRecord</code> whose right child is not a thread
     * @return the in-order successor of <code>n</code>
     */
    public IndexRecord getSuccessorForDeletion(IndexRecord n) {
        IndexRecord successor = n, successorParent = n, rover = n.right;

        while (rover != n) { //interesting artifact of threading
            successorParent = successor;
            successor = rover;
            rover = rover.left;
        }
        if (successor != n.right) {
            successorParent.left = successor.right;
            if (successor.rightIsThread) {
                successorParent.leftIsThread = true;
                successor.rightIsThread = false;
            }
            successor.right = n.right;
        }
        return successor;
    }

    /**
     * Finds the in-order successor of a given <code>IndexRecord</code>
     *
     * @param n an <code>IndexRecord</code> whose right child is not a thread
     * @return the in-order successor of <code>n</code>
     */
    public IndexRecord getSuccessor(IndexRecord n) {
        IndexRecord successor = n.right;

        while (successor.left != n) {
            successor = successor.left;
        }
        return successor;
    }

    /**
     * Finds the in-order predecessor of a given <code>IndexRecord</code>
     *
     * @param n an <code>IndexRecord</code> whose left child is not a thread
     * @return the in-order predecessor of <code>n</code>
     */
    public IndexRecord getPredecessor(IndexRecord n) {
        IndexRecord predecessor = n.left;
        while (predecessor.right != n) {
```

```java
            predecessor = predecessor.right;
        }
        return predecessor;
    }

    /**
     * Searches for a record in the tree of <code>IndexRecord</code>s
     * @param key is the value (e.g. student ID) that is searched for
     * @return <code>recordNumber</code> of goal; -1 if not found
     */
    public int find(String key) {
        IndexRecord rover = root;
        int recNum = -1, comparison = 0;

        while (true) {
            comparison = key.compareTo(rover.key);
            if (comparison == 0) {
                recNum = rover.recordNumber;
                break;
            }
            else if (comparison > 0) {
                if (rover.rightIsThread) //reached end of Index without finding
                    break;
                else
                    rover = rover.right;
            }
            else {
                if (rover.leftIsThread) //reached end of Index without finding
                    break;
                else
                    rover = rover.left;
            }
        }
        return recNum;
    }

    /**
     * @return the number of IndexRecords in the Index
     */
    public int getLength() {
        return size;
    }
}
```