```
import java.util.Scanner;

/**
 * COSC311 - Program 3 (threaded)
 *
 * This is the driver for Program 3, a database implementation.
 *  This program reads in data from an external file of 68 records.
 *  Each record is composed of three fields:
 *      (String lastName) (String firstName) (String ID)
 *
 * This menu driven command line program offers options to add a
 *  record to the database, delete a record, search for a record
 *  and print out the entire database in different orders.
 *
 * @author Bill Sverdlik, Mordechai Sadowsky
 * @version Version 2.0-SNAPSHOT, 23-feb-2014
 */

public class COSC311Driver {

    private static Scanner keyboard;
    private static DataStructure myStructure;

    public static void main(String[] args) {
        keyboard = new Scanner(System.in);
        myStructure = new DataStructure();
        int response;

        System.out.println("Welcome to YourStudentRoster");
        do {
            System.out.println("\nMain menu:\n");
            System.out.println(" 1 Add a new student");
            System.out.println(" 2 Delete a student");
            System.out.println(" 3 Find a student by ID");
            System.out.println(" 4 List students by ID increasing");
            System.out.println(" 5 List students by first name increasing");
            System.out.println(" 6 List students by last name increasing");
            System.out.println(" 7 List students by ID decreasing");
            System.out.println(" 8 List students by first name decreasing");
            System.out.println(" 9 List students by last name decreasing\n");
            System.out.println(" 0 End");
            System.out.print("\nMenu selection: ");

            response = keyboard.nextInt();
            keyboard.nextLine();

            switch (response) {
                case 1: addIt();
                    break;
                case 2: deleteIt();
                    break;
                case 3: findIt();
                    break;
                case 4: myStructure.listIt(1, 1);
                    break;
                case 5: myStructure.listIt(2, 1);
```

```
                    break;
                case 6: myStructure.listIt(3, 1);
                    break;
                case 7: myStructure.listIt(1, 2);
                    break;
                case 8: myStructure.listIt(2, 2);
                    break;
                case 9: myStructure.listIt(3, 2);
                    break;
                default:
            }
        } while (response != 0);

        System.out.println("\nThank you, goodbye!");
    }

    /**
     * Menu option 1: add a new student to the database.
     */
    public static void addIt() {
        String name1, name2, tempID;
        boolean found;

        do {
            System.out.print("Enter a unique student ID number: ");
            tempID = keyboard.nextLine();

            //is it unique?
            found = (myStructure.search(tempID) > -1);
            if (found) {
                System.out.println("ID already in use.");
            }
        } while (found);

        // We found a unique ID. Now ask for first and last name
        System.out.print("Enter first name: ");
        name1 = keyboard.nextLine();
        System.out.print("Enter last name: ");
        name2 = keyboard.nextLine();
        System.out.println();

        // add to our data structure
        if (!(myStructure.insert(name1,name2,tempID)))
            System.out.println("Error, database full!");
    }

    /**
     * Menu option 2: delete a student from the database.
     */
    public static void deleteIt() {
        String tempID;
        boolean found;

        do {
            System.out.println("\nEnter the ID number of student to delete: ");
            tempID = keyboard.nextLine();
```

```
            //is it in the database?
            found = (myStructure.search(tempID) > -1);
            if (!found) {
                System.out.println("ID not found.");
                System.out.print("Please re-enter an ID to delete: ");
            }
        } while (!found);

        myStructure.delete(tempID);
    }

    /**
     * Menu option 3: find a student
     *  If found, prints out the record.
     */
    public static void findIt() {
        String tempID;
        boolean found;
        int recNum;

        do {
            System.out.print("\nEnter an ID number: ");
            tempID = keyboard.nextLine();

            //is it in the database?
            recNum = myStructure.search(tempID);
            found = (recNum != -1);
            if (!found) {
                System.out.println("ID not found.");
            }
        } while (!found);

        myStructure.print(recNum);
    }
}
```

```
import java.util.Scanner;
import java.io.FileInputStream;
import java.io.FileNotFoundException;

/**
 * COSC311 - Program 3 (threaded)
 *
 * This file defines the <code>DataStructure</code> type with an array of
 *  <code>DatabaseRecords</code> and three <code>Index</code> objects. Instances
 *  initially read in a list of records from an external file to populate the
 *  database. The structures can then be searched, added to, deleted from,
 *  displayed, and can print individual records.
 *
 * Student records are referenced to by their position in the database, which
 *  is stored in each <code>Index</code> as the <code>recordNumber</code>.
 *
 * @author Mordechai Sadowsky
 * @version 25-mar-2014
 *
 */
public class DataStructure {

    private DatabaseRecord[] database;
    private Index firstNames, lastNames, ids;
    private int databasePointer;
    private final int SIZE = 100;
    private DBStack deletedRecords = new DBStack(SIZE);
    private final String PATH =
            "/Users/Mordechai/git/COSC311/Program3/src/data.txt";

    public DataStructure() {
        Scanner inputStream = null;
        try {
            inputStream = new Scanner(new FileInputStream(PATH));
        }
        catch (FileNotFoundException e) {
            System.out.println(e.getMessage());
            System.out.println("Don't forget to update file path name!");
            System.exit(1);
        }

        //initialize data members
        database = new DatabaseRecord[SIZE];
        firstNames = new Index();
        lastNames =  new Index();
        ids =  new Index();
        databasePointer = 0;

        //Read in database from external file and
        //  add records to main database and the indexes
        while (inputStream.hasNextLine()) {
            String first = inputStream.next();
            String last = inputStream.next();
            String id = inputStream.next();
            if (ids.find(id) != -1)
                continue;
```

```
            database[databasePointer] = new DatabaseRecord(first, last, id);
            firstNames.insert(first, databasePointer);
            lastNames.insert(last, databasePointer);
            ids.insert(id, databasePointer);
            databasePointer++;
        }
        if (inputStream.hasNextLine()) {
            System.out.println("File is too big! Increase database SIZE.");
            System.exit(1);
        }
    }

    /**
     * Searches through the <code>Index</code> of IDs because the database may
     *  contain deleted records.
     *
     * @param id number of a student to search for
     * @return The reference <code>recordNumber</code> of the student, i.e. the
     *  index of the student record's position in the <code>database</code>
     */
    public int search(String id) {
        return ids.find(id);
    }

    /**
     * Adds a record to the database and each <code>Index</code>.
     *  Records are inserted in lexicographical order into the indices,
     *  but are entered into the <code>database</code> at the site of a
     *  previously deleted record or the end of the <code>database</code>.
     * @param first First name of the new student.
     * @param last Last name of the new student.
     * @param id ID number of the new student.
     * @return true for successful insertion, false for failure
     */
    public boolean insert(String first, String last, String id) {
        int bookmark;

        if (isFull()) {
            System.out.println("Error, database full!");
            return false;
        }

        //check the stack to see if any lines in the middle of the database
        //  are free for insertion
        if (!deletedRecords.isEmpty()) {
            bookmark = databasePointer; //keep track of database end
            databasePointer = deletedRecords.pop(); //point to "open" space
        }
        else
            bookmark = databasePointer+1; //if no open spaces, (database end)++

        database[databasePointer] = new DatabaseRecord(first, last, id);

        //insert record pieces into their respective indices
        firstNames.insert(first, databasePointer);
        lastNames.insert(last, databasePointer);
```

```
        ids.insert(id, databasePointer);

        databasePointer = bookmark; //update pointer back to end or incremented
        return true;
    }

    /**
     * Removes a record from each index, and adds its location in the main
     *  <code>database</code> to the stack of <code>deletedRecords</code>
     * @param id ID number of student to delete
     */
    public void delete(String id) {
        int recordToDelete = ids.find(id); //finds reference recordNumber

        firstNames.delete(recordToDelete);
        lastNames.delete(recordToDelete);
        ids.delete(recordToDelete);

        deletedRecords.push(recordToDelete);
    }

    /**
     * Displays entire database in one of 6 different orders by reading through
     *  an Index in order to pull the reference numbers and print the associated
     *  records one by one.
     *
     * @param a determines which <code>Index</code> to sort by:
     *  1-ID number; 2-first name; 3-last name
     * @param b determines in which lexicographical order to display:
     *  1-ascending order; 2-descending order
     */
    public void listIt(int a, int b) {
        if (b == 1) { //ascending prints
            ids.setIteratorFront();
            firstNames.setIteratorFront();
            lastNames.setIteratorFront();

            if (a == 1)
                for (int i = 0; i < ids.getLength(); i++) {
                    print(ids.getIteratorRecNum());
                    ids.iterateForward();
                }
            else if (a == 2)
                for (int i = 0; i < firstNames.getLength(); i++) {
                    print(firstNames.getIteratorRecNum());
                    firstNames.iterateForward();
                }
            else if (a == 3)
                for (int i = 0; i < lastNames.getLength(); i++) {
                    print(lastNames.getIteratorRecNum());
                    lastNames.iterateForward();
                }
            else
                return;
        }
        else if (b == 2) { //descending prints
```

```
                ids.setIteratorBack();
                lastNames.setIteratorBack();
                firstNames.setIteratorBack();

                if (a == 1)
                    for (int i = 0; i < ids.getLength(); i++) {
                        print(ids.getIteratorRecNum());
                        ids.iterateBackward();
                    }
                else if (a == 2)
                    for (int i = 0; i < firstNames.getLength(); i++) {
                        print(firstNames.getIteratorRecNum());
                        firstNames.iterateBackward();
                    }
                else if (a == 3)
                    for (int i = 0; i < lastNames.getLength(); i++) {
                        print(lastNames.getIteratorRecNum());
                        lastNames.iterateBackward();
                    }
                else
                    return;
            }
           else
               return;
        }

        public boolean isFull() {
            return (databasePointer == SIZE-1) && (deletedRecords.isEmpty());
        }

        /**
         * Displays a single <code>DatabaseRecord</code>
         * @param recordNumber
         */
        public void print(int recordNumber) {
            System.out.println(database[recordNumber]);
        }
    }
```

```java
/**
 * COSC 311 - Project 3 (threaded)
 *
 * This file defines the DatabaseRecord type. Each record describes a student,
 *  with a first name, last name and ID number.
 *
 * @author Mordechai Sadowsky
 * @version 02-feb-2014
 *
 */
public class DatabaseRecord {

    private String firstName;
    private String lastName;
    private String idNumber;

    public DatabaseRecord(String f, String l, String i) {
        firstName = f;
        lastName = l;
        idNumber = i;
    }

    public String toString() {
        return firstName+" "+lastName+" "+idNumber;
    }
}
```

```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) {
```

```
            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;
    }
}
```

```java
/**
 * COSC311 - Program 3 (threaded)
 *
 * This file describes a stack data type used to store the location of deleted
 *  student records so those spaces can be filled by new student additions.
 *
 * @author Mordechai Sadowsky
 * @version 02-feb-2014
 *
 */
public class DBStack {

    private int[] stack;
    private int pointer, size;

    public DBStack(int x) {
        pointer = 0;
        size = x;
        stack = new int[size];
    }

    public boolean isFull() {
        return (pointer == size);
    }

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

    public void push(int x) {
        stack[pointer++] = x;
    }

    public int pop() {
        return stack[--pointer];
    }
}
```