CS3114/5040 (Fall 2023)
PROGRAMMING ASSIGNMENT #2

Due Wednesday, October 11 @ 11:00 PM for 100 points
Due Tuesday, October 10 @ 11:00 PM for 10 point bonus
Note: This project also has three intermediate milestones. See the Piazza forum for details.

Assignment:

This project will provide better search support for the seminar database from Project 1. This project replaces the hash table with two types of search tree, and the trees will store references to the actual Seminar objects. There will be no memory manager in this project.

The Trees:

To search for seminar records that match a given ID or keyword, or that fall within a given date range or cost range, you will use a collection of Binary Search Trees (BST). There will be a BST for each of: IDs, costs, dates, and keywords. You may use the code from OpenDSA to help you write your BST, though it will require a fair amount of revision to satisify the needs of this project. You may use BST code that you have previously written, such as for another course — but make sure that it generates the same shape of tree as the OpenDSA version would. You may not use BST code taken from another source in your project. Note that your BST must insert equal values to the **left**. On deletion, if the deleted node has two non-null children, then it is replaced by the node with **maximum** value from the **left** subtree.

To support searches by 2D locations, you will use a Bintree. A binary search tree gives expected O(log n) performance for insert, delete, and search operations (if you ignore the possibility that it is unbalanced). This would allow you to insert and delete records, and locate them by key value or within a key range. However, the BST does not help when doing a coordinate search. You could combine the (x, y) coordinates into a single key and store cities using this key in a second BST. That would allow search by coordinate, but would not allow for efficient **range queries** – searching for cities within a given distance of a point. The problem is that the BST only works well for one-dimensional keys, while a coordinate is a two-dimensional key.

The Bintree (see Module 15.5 of OpenDSA) is one of many **hierarchical data structures** commonly used to store **spatial** data such as 2D locations. It allows for efficient insertion, deletion and search queries.

Invocation and I/O Files:

The program would be invoked from the command-line as:

```
java SemSearch {world-size} {command-file}
```

The name of the program is `SemSearch`. Parameter `{world-size}` is an integer that is a power of two, and it specifies the size of the world for the seminar location field. The world is assumed to have x and y coordinates from 0 to `{world-size - 1}`. Your program will read a series of commands from text file `{command-file}` You do not need to check for syntax errors in the command lines (although you **do** need to check for logical errors such as illegal duplicate insertions or deletions of records with non-existent keys). The program should terminate after reading the end of the file. The formats for the commands are identical to those used for Project 1, with the additions shown below.

All commands should generate a suitable output message. All output should be written to standard output. Every command that is processed will generate some sort of output message to indicate whether the command was successful or not. Complete details for the proper output from commands will be found in the example output file to be posted with this project description.

Commands for Project 2 and their syntax are as follows.

insert

The insert command syntax is identical to Project 1. The new Seminar object is inserted into each of the trees.

search

There are multiple forms of the search command, as follows. In all cases, do not visit more nodes in the tree than necessary. Vist BSTs using a (modified) inorder traversal so that the records found will be printed in ascending key order.

search ID <*ID*>

Print the record that matches the given ID (if one exists).

search cost <*low*> <*high*>

Print all records that fall within the range (inclusive).

search date <*date/time*> <*date/time*>

Print all records that fall within a range of date/times (inclusive).

search keyword <*keyword*>

Print all records that match the keyword.

search location <*x*> <*y*> <*radius*>

Print all records that fall within *radius* distance of the search point (inclusive). Do not visit a node if it is not required. To determine if a node should be visited, check if the bounding box of the search circle intersects the node. Count the number of nodes visited during the search. If you had to determine the type of the node (internal, empty, or full leaf) then it counts as visited.

delete <*ID*>

Delete the record with the given *ID* from all of the search trees.

print

There are multiple forms of the print command, as follows. Note that there are no longer "print hashtable" or "print blocks" commands, as there is no longer a hash table or memory manager!

print date
print keyword
print location
print cost
print ID

Depending on the modifier to the **print** command, print out a traversal of the corresponding tree. Node values when printed are indented by two spaces times their level (the root is at level 0).

BST nodes should be printed in reverse inorder (meaning, print the right subtree, then the root, then the left subtree). Bintrees should be printed using a preorder traversal. For each seminar record, print out the relevent key (for the tree being dumped).

Design Considerations:

Your main design concern for this project will be how to implement the BSTs. You need to make one implementation that can support the various demands of the different search keys. Some keys are integers. Some are strings. The ID tree does not permit duplicates, but the other search keys do permit duplicates. Some searches are range queries, others are exact match. Use of Java Generics is recommended, but not mandatory.

When implementing your Bintree, you may assume that it is storing Seminar records. Thus, the Seminar x(), y() and id() methods can be used directly as needed. Your Bintree design must meet the following requirements.

1. No Bintree node may store a pointer to its parent. (Likewise, BST nodes may not store pointers to the parent.)
2. Bintree nodes may not store their position or width information. That should be passed in during a traversal.
3. The internal nodes and the leaf nodes must implement a general "bintree node" interface.
4. The empty nodes might be a separate class, or just a special instance of the "leaf" class. Either way, it must implement the Flyweight design pattern, meaning that there is only a single empty node created, and all places in the tree that need an empty node point to the same object.

Programming Standards:

You must conform to good programming/documentation standards. Web-CAT will provide feedback on its evaluation of your coding style, and be used for style grading. Beyond meeting Web-CAT's checkstyle requirements, here are some additional requirements regarding programming standards.

- You should include a comment explaining the purpose of every variable or named constant you use in your program.
- You should use meaningful identifier names that suggest the meaning or purpose of the constant, variable, function, etc. Use a consistent convention for how identifier names appear, such as "camel casing".
- Always use named constants or enumerated types instead of literal constants in the code.
- Source files should be under 600 lines.
- There should be a single class in each source file. You can make an exception for small inner classes (less than 100 lines including comments) if the total file length is less than 600 lines.

We can't help you with your code unless we can understand it. Therefore, you should no bring your code to the GTAs or the instructors for debugging help unless it is properly documented and exhibits good programming style. Be sure to begin your internal documentation right from the start.

You may only use code you have written, either specifically for this project or for earlier programs, or code provided by the instructor. Note that the OpenDSA code is not designed for the specific purpose of this assignment, and is therefore likely to require modification. It may, however, provide a useful starting point.

Java Data Structures Classes:

You are not permitted to use Java classes that implement complex data structures. This includes `ArrayList`, `HashMap`, `Vector`, or any other classes that implement lists, hash tables, or extensible arrays. (You may of course use the standard array operators.) You may use typical classes for string processing, byte array manipulation, parsing, etc. **Exception:** You may use ArrayList within your parser to help simplify processing of the keywords list. No Arraylist should be used by any of the trees.

If in doubt about which classes are permitted and which are not, you should ask. There will be penalties for using classes that are considered off limits.

Deliverables:

You will implement your project using Eclipse, and you will submit your project using the Eclipse plugin to Web-CAT. Links to the Web-CAT client are posted at the class website. If you make multiple submissions, only your last submission will be evaluated unless you arrange otherwise with the GTA. There is no limit to the number of submissions that you may make.

You are required to submit your own test cases with your program, and part of your grade will be determined by how well your test cases test your program, as defined by Web-CAT's evaluation of code coverage. Of course, your program must pass your own test cases. Part of your grade will also be determined by test cases that are provided by the graders. Web-CAT will report to you which test files have passed correctly, and which have not. Note that you will **not** be given a copy of these test files, only a brief description of what each accomplished in order to guide your own testing process in case you did not pass one of our tests.

When structuring the source files of your project, use a flat directory structure; that is, your source files will all be contained in the project "src" directory. Any subdirectories in the project will be ignored.

You are permitted to work with a partner on this project. While the partner need not be the same as who you worked with on any other projects this semester, you may only work with a single partner during the course of one project unless you get special permission from the course instructor. When you work with a partner, then **only one member of the pair** will make a submission. Be sure both names are included in the documentation. Whatever is the final submission from either of the pair members is what we will grade unless you arrange otherwise with the GTA.

Pledge:

Your project submission must include a statement, pledging your conformance to the Honor Code requirements for this course. Specifically, you must include the following pledge statement near the beginning of the file containing the function main() in your program. The text of the pledge will also be posted online.

```
// On my honor:
//
// - I have not used source code obtained from another current or
//   former student, or any other unauthorized source, either
//   modified or unmodified.
//
// - All source code and documentation used in my program is
//   either my original work, or was derived by me from the
//   source code published in the textbook for this course.
//
// - I have not discussed coding details about this project with
//   anyone other than my partner (in the case of a joint
//   submission), instructor, ACM/UPE tutors or the TAs assigned
//   to this course. I understand that I may discuss the concepts
//   of this program with other students, and that another student
//   may help me debug my program so long as neither of us writes
//   anything during the discussion or modifies any computer file
//   during the discussion. I have violated neither the spirit nor
//   letter of this restriction.
```

Programs that do not contain this pledge will not be graded.