CS-3114 Spring 2014 – Project 1 – BST Database **Updated Feb 3, 2014**

Due: Feb 14, 5pm.
Late Policy as in the syllabus: 10% per day, maximum of 3 days.

Assignment:

For this project, you will implement a simple database system with multiple key indices. Specifically, the records in the database will contain information about cities, including name and population. The records will be organized for efficient search by two Binary Search Trees (BSTs), one for *name* and the other for *population*. You will design an augmented BST data structure to support the findKth operation.

Design & Implementation Requirements:

Records in the city database each contain three fields. The *population* field is integer valued. The *name* field is a variable length string beginning with a letter and containing any number of letters (upper and lower case) and digits. Names are case sensitive. You should compare names for lexical order using the standard String.compareTo() function. The third field is called *payload*, is a string, and represents other information about the city, but is not indexed. (Hint: in the future, we may use this information for something else.)

Your database will contain two separate BSTs: one is used to index cities by the *name* field, and the other is used to index by the *population* field. However, you must use good OO design (using Java Generics) and implement only a single BST class. Furthermore, only one single copy of each city record may be stored – you are not allowed to replicate any of the city records' data between the two trees. Instead, each tree node must contain only a pointer to the appropriate city data record. Therefore, you **must** use the concept of *comparator function classes* (OpenDSA 6.3) to enable each tree to extract and order based on different fields in the city records. Thus, the same BST class can be used for both tree instances, and no data is replicated. There will be a significant deduction for replicated tree code or replicated city data values in the BSTs.

All BST operations must be implemented recursively. That is, any function that traverses or otherwise moves through the tree must be recursive. There will be a deduction if your node class uses parent pointers.

Duplicate values are allowed in the input data (e.g. two cities with the same name or same population are allowed). Your BST *insert* operation **must** use the rule that duplicates are stored to the right (OpenDSA 7.8.1 Note). Your BST *delete* operation **must** be implemented using the "promote the minimum value of the right subtree" rule. It is important that you follow these rules so that your output will match the expected grading output. Also, since duplicate values are allowed, you will have to be very careful about performing *delete* operations. You need to make sure that you are deleting the same actual record from both trees, so matching only by key values will not be sufficient.

The *findKth* operation must run in $O(\log n)$ time in the average case, and will require you to slightly augment the design of the BST data structure. You will need to store some additional information in each node to enable *log n* performance. Part of your assignment is to design and implement this new capability for BSTs. The *insert*, *find*, and *delete* operations must also run in $O(\log n)$ average case time. There will be a significant deduction if your *findKth* implementation relies on an $O(n)$ tree traversal, or if it sacrifices the $O(\log n)$ time of the *insert* or *delete*

operations.

Additional design requirements: You are expected to use good object-oriented design in your solution. Imagine that your database will include additional information and indices of different types in the future, or that you might switch your BSTs to a different data structure in the future. Thus, your BSTs should be encapsulated and insulated from the database or I/O command processor functionality by a mediating abstract interface for *index* functionality that specifies typical index operations as below.

Input and Output:

The input to this program should be read from standard input and the output should be directed to standard output. The name of the program should be "bst". Your program must be runnable using:

```
% java bst  < input.txt  > output.txt
```

Note that "< input.txt" and "> output.txt" are not command line arguments to your program, but are standard shell i/o redirects from/to files.

The input for this project will consist of a series of commands (some with associated parameters, separated by spaces), one command for each line. Commands are free format in that an arbitrary number of additional spaces may be interspersed between parameters. You do not need to check for syntax errors in the command lines, however you do need to check for logical errors (such as deleting a value that is not in the database). Commands that have a *field* parameter can be either "name" or "population". Blank input lines are ignored, and lines beginning with "#" character are ignored as comments.

In the output, each command should be echoed to output with its parameters on one line, followed on the next line by any additional required output for that command. Find and delete commands that find no matching records should output "Not found". See sample output for detailed syntax.

When printing a city record, output it on a single line in the following format:

name population payload

When printing multiple city records, print one per line.

Eight input commands and their syntax are as follows:

insert *name population payload*

Insert a city record into the database. You will create a new record object, and insert it into both BST indices. No additional output.

find *field value*

Find and print all city records with the given value for the given field (name or population).

findKth *field k*

Find and print the city record that would appear at position *k* in the listing of the cities sorted by the given field in ascending order. *k* is an integer value, where *k=0* indicates the first city in the sorted list.

findRange *field minvalue maxvalue*

Find and print the city records (1 per line) that are within the specified range (inclusive) for the given field. This operation should execute as efficiently as possible, avoiding a complete traversal whenever possible.

delete *field value*

Find and remove all city records with the given value for the given field from the database, if any exist. Be sure to remove the same record(s) from both BST indices.

sort *field*

Print out the city records in the database, sorted by the given field in ascending order. The listing should be processed by an in-order traversal of the appropriate BST index. Print each city record on a separate line. If the database is empty, output "Database empty".

tree *field*

Identical to the *sort* command, except reveal the *field* BST structure in an indented format as follows: If the city record appears in a node at depth *i* in the BST (depth of the tree root = 0), print *4i* space characters before the city record in the output.

makenull

Reinitialize the database to be empty. No additional output.

Programming Standards:

You must conform to good programming/documentation standards. Some specifics:

- You must include a header comment, preceding main(), specifying the compiler and operating system used and the date completed.
- Your header comment must describe what your program does; don't just plagiarize language from this spec.
- You must include a comment explaining the purpose of every variable or named constant you use in your program.
- You must use meaningful identifier names that suggest the meaning or purpose of the constant, variable, function, etc.
- Always use named constants or enumerated types instead of literal constants in the code.
- Precede every major block of your code with a comment explaining its purpose. You don't have to describe how it works unless you do something so sneaky it deserves special recognition.
- You must use indentation and blank lines to make control structures more readable.
- Precede each function and/or class method with a header comment describing what the function does, the logical significance of each parameter (if any), and pre- and post-conditions.
- Decompose your design logically, identifying which components should be objects and

what operations should be encapsulated for each. Neither the GTAs nor the instructors will help any student debug an implementation 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 taken from the textbook. Note that the textbook 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.

Testing:

A sample input and output file will be posted. This is not the data file that will be used in grading your program. The test data provided to you will attempt to exercise the various syntactic elements of the command specifications. It makes no effort to be comprehensive in terms of testing all of the edge cases in the data structures required by the program. Thus, while the test data provided should be useful, you must also create your own test cases to ensure that your program works correctly. It is acceptable to share input and output files on the class forum.

Deliverables:

1. Report

Submit a PDF file containing a short report that presents:

(1) Your overall object-oriented design.

(2) How you modified the BST data structure design to support the *findKth* method.

(3) The big-O average-case running times of all the listed operations, with justification.

You should submit your report PDF along with your code on WebCAT as below, by including it in your Eclipse project or your submitted zip file.

2. Code:

You will submit your project code through the automated Web-CAT server. Links to the Web-CAT client are posted at the class website. If you make multiple submissions, only your last submission will be evaluated. 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 grader's 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.

If submitting through Eclipse, the format of the submitted archive will be managed for you. If you choose not to develop in Eclipse, you will submit either a ZIP-compressed archive (compatible with Windows ZIP tools or the Unix zip command) or else a tar'ed and gzip'ed archive. Either

way, your archive should contain the source code for the project only (no .class files or binary resources).

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

```
// On my honor:
//
// - I have not used source code obtained from another 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 the 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.
```