CS3114 (Fall 2011)
PROGRAMMING ASSIGNMENT #4

Due Tuesday, December 6 @ 11:00 PM for 150 points
Due Monday, December 5 @ 11:00 PM for 10 point bonus
Initial Schedule due Tuesday, November 15 @ 11:00 PM
Second Schedule due Tuesday, November 29 @ 11:00 PM

Revised 11/17

Please note that this assignment is worth more than the others. It is not intended that this assignment be that much more difficult than the others. Rather, this should be an opportunity to make up for any past problems.

In this project, you will re-implement the Geographic Information System for storing point data from Project 2. However, this time the PR Quadtree and the city records will reside on disk. A buffer pool (using the LRU replacement strategy) will mediate access to the disk file, and a memory manager (similar to the one implemented for Project 1) will decide where to store the PR Quadtree nodes and city records. Note that the nodes of the BST from Project 2 will still be stored in memory in this project. However, it they will each reference a city record on disk (using a handle) and the keys (the city names) will be also be stored on disk (accessed through a handle).

Input and Output:

Your program will be named "Bindisk", and it will take three command-line arguments. The first is the name of the command input file. The second is the number of buffers in the buffer pool, and will be in the range 1–20. The third is the size for a block in the file (which therefore determines the buffer size as well).

The input for this project will consist of a series of commands (some with associated parameters, separated by spaces), one command for each line. The commands will be read from the file given in command line parameter 1, and the output from the command will be written to standard output. The format and interpretation for the commands will be identical to Project 2, with the following exceptions.

In addition to listing the nodes of the PR Quadtree, the "debug" command will also list the following: (1) Block ID's of the blocks currently contained in the bufferpool in order from most recently to least least recently used; and (2) a listing of the memory manager's free blocks, in order of their occurrence in the freeblock list.

The output for the PR quadtree in the "debug" command will be a little different from that used in Project 2. An internal node should be represented by parentheses (and) around the contents of that node. A leaf node should print its contents, then the "bar" or "pipe symbol" `|`. An empty leaf node should print as its contents an asterisk `*`. A non-empty leaf node should print as its contents the records it contains. Each record should be printed as "X,Y,NAME:" The Project 2 sample data file at a certain point produces the following debug output:

```
I0,0,Floyd5001,5012,Blacksburg5001,6213,Blacksburg|E5001,8414,Christiansburg|16383,
16383,Virginia_Beach|
```

In Project 4, that same tree would look like this:

```
(0,0,Floyd:5001,5012,Blacksburg:5001,6213,Blacksburg:|*|5001,8414,Christiansburg:|
16383,16383,Virginia_Beach:|)
```

You will need to create and maintain a disk file which the buffer pool is acting as the intermediary for. The name of this disk file must be "p4bin.dat". After completing all commands in the input

file, all unwritten blocks in the buffer pool should be written to disk, and the disk file should be closed, **not deleted**.

Note that "Block ID" simply refers to the block number, starting with 0. Thus, if the block size is 1024 bytes, then bytes 0-1023 are in Block 0, bytes 1024-2047 are in Block 1, and so on.

Implementation:

The implementation rules for the PR Quadtree from Project 2 are still in place. That is, all operations that traverse or descend the PR Quadtree structure **MUST** be implemented recursively, and the PR Quadtree nodes **MUST** be implemented with separate classes for the internal nodes and the leaf nodes, both of which inherit from some base node type. A flyweight **MUST** be used for empty leaf notes.

The PR Quadtree itself will be stored on disk, not in main memory. This is the primary difference from Project 2. The nodes will be of variable length, and where a node is stored on disk will be determined by a memory manager. When implementing the PR Quadtree nodes, access to a node (perhaps you did this by calling the node's "getchild" method) will now mean a request to the memory manager to return the node contents from the memory pool, and creation or alteration of a node will require writing to the memory pool. From the point of view of the memory manager and the buffer pool, communications are in the form of variable-length "messages" that must be stored.

The memory manager should follow the same definition as in Project 1. In particular, the location for placing the next message within the memory pool should be determined using best fit; a list of the free blocks may be maintained in main memory, and adjacent free blocks should be merged together.

Initially, the memory pool (and the file) should have length 0. Whenever a request is made to the memory manager that it cannot fulfill with existing free blocks, the size of the memory pool should grow by one (or more if necessary) disk blocks to meet the request.

The memory manager will be managing data residing in the memory pool, and this memory pool will reside on disk, but the memory manager does not actually have direct access to the disk. All disk access is through the buffer pool. Thus, the flow of control for a node access will be that the PR Quadtree will request a "message" from the memory manager via a handle, the memory manager will ask the buffer pool for the data at a physical location, the buffer pool will hand the contents of the "message" to the memory manager, which will in turn hand the contents of the "message" back to the PR Quadtree.

The layout of the PR Quadtree node messages sent to the memory manager **MUST** be as follows. (Note that, as in Project 1, the memory manager will actually add to the message the length of the message as stored in the memory pool.) Internal nodes will store 17 bytes: a one-byte field used to distinguish internal from leaf nodes, followed by four 4-byte fields to store handles for the four children.

Empty nodes will be represented by storing in the empty node's parent a handle value that is recognized by the node class "getchild" method as representing an empty leaf node (the flyweight). The flyweight may be actually represented as a physical node on disk, or you may use a special handle value that is simply recognized as the flywight.

Leaf nodes that contain one or more city records will require 14 bytes, storing the following fields: a one-byte field used to distinguish internal from leaf nodes; a one-byte field to indicate the number of city records stored; and then three 4-byte fields for the three handles to the city records.

Each city record will be stored as a separate message in the memory pool. The city record will store two 4-byte fields for the city x- and y-coordinates, respectively, and a 4-byte handle for the city name. The city name will be sent to the memory manager as a separate message. This message will contain the length of the name in the first byte, followed by the characters for the name, one byte per character. The null character at the end of the city name string should not be included in the message, nor stored on disk.

Programming Standards:

You must conform to good programming/documentation standards. Our most important rule of thumb is that the program must be easy to understand (which makes it easier for the TAs to grade). While we do not require a specific standard, here is a set of reasonable guidelines that are good to follow.

- You should always include a header comment, preceding main(), specifying things like the author(s), compiler and operating system used, and date completed.
- Your header comment should describe what your program does; don't just plagiarize language from this spec.
- You should include a comment explaining the purpose of every variable or named constant you use in your program. This is not just for the TAs – if you ever plan to look at the code again for a future project, you need this for your own sake.
- You should use meaningful identifier names that suggest the meaning or purpose of the constant, variable, function, etc. This is one of the most important ways to create readable code.
- Always use named constants or enumerated types instead of literal constants in the code. Doing so will save you from making many errors.
- Precede every major block of your code with a comment explaining its purpose. You don't have to describe in detail how it works unless you do something so sneaky it deserves special recognition.
- Proper indentation and blank lines 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 required pre-conditions that it assumes.

Do not expect the GTAs or instructors to help debug an implementation unless it is properly documented and exhibits good programming style. Its hard enough to debug good code. Be sure to begin your internal documentation right from the start, since that will save you time in the long run.

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 data file will be posted to the website to help you test your program. 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 the data structures required by the program. Thus, while the test data provided should be useful, you should also do testing on your own test data to ensure that your program works correctly.

Deliverables:

When structuring the source files of your project (be it in Eclipse as a "Managed Java Project," or in another environment), use a flat directory structure; that is, your source files will all be contained in the project root. Any subdirectories in the project will be ignored. If you used a makefile to compile your code, or otherwise did something that won't automatically compile in Eclipse, be sure to include any necessary files or instructions so that the TAs can compile it.

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 all of the source code for the project, along with any files or instructions necessary to compile the code. If you need to explain any pertinent information to aid the TA in the grading of your project, you may include an optional "readme" file in your submitted archive.

You will submit your project through the automated Web-CAT server. Links to the Web-CAT client and instructions for those students who are not developing in Eclipse are posted at the class website. If you make multiple submissions, only your last submission will be evaluated.

You are permitted (and ecouraged) to work with a partner on this project. 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.

Scheduling:

In addition to the project submission, you are also required to submit an initial project schedule, an intermediate schedule, and a final schedule with your project submission. The schedule sheet template for this project is posted at the course website. You won't receive direct credit for submitting the schedule as required, but each instance of failing to submit scheduling information as required will lose points from the project grade.

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 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.