

CS2604 (Fall 2002)

PROGRAMMING ASSIGNMENT #4

Due Friday, November 22 @ 11:00 PM for 150 points

Early bonus date: Thursday, November 21 @ 11:00 PM for a 15 point bonus

Late date: Monday, December 2 at 11:00 PM for a 30 point penalty

Revised: 11/10/2002

Please note that this assignment is worth more than the others. It is not the intention 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 Bintree will reside on disk. A buffer pool 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 Bintree nodes.

Input and Output:

Your program will be named “bindisk”, and it will take two command-line arguments. The first is the number of buffers in the buffer pool. The second 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 standard input, 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 exception. In addition to listing the nodes of the Bintree, 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 recently used; and (2) a listing of the memory manager’s free blocks, in order of their occurrence in the freeblock list

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**.

Implementation:

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

The Bintree 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 Bintree nodes, access to a node (by calling the node’s “getchild” method) will 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 worst

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) 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 Bintree 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 Bintree.

The layout of the Bintree 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 9 bytes: a one-byte field used to distinguish internal from leaf nodes, followed by two 4-byte fields to store handles for two children. Leaf nodes that contain a city record will store the following fields: a one-byte field used to distinguish internal from leaf nodes, 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. There should be no null character at the end of the city name when stored on disk. 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.

Programming Standards:

You must conform to good programming/documentation standards, as described in the Elements of Programming Style. 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 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. You may not use code from STL, MFC, or a similar library in your program.

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. 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:

You will submit a tarred and gzipped file containing all the source code for the project, a makefile and an optional readme file. The file should end in a tar.gz extension. The makefile will allow the TA to simply type make at the command line and create your executable. The optional readme file will explain any pertinent information needed by the TA to aid them in the grading of your submission.

Once you have assembled the archive for submission, for your own protection, please move it to a location other than your development directory, unzip the contents, build an executable, and test that executable on at least one input. Failure to do this may result in delayed evaluation of your program and a loss of points.

You will submit your project to the automated Curator server. The instructions and necessary software are available at: <http://ei.cs.vt.edu/~eags/CuratorGuides.html>. If you make multiple submissions, only your last submission will be evaluated.

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 in the header comment preceding the function main() in your program. The text of the pledge will also be posted online.

```
// On my honor:  
//  
// - I have not used C++ language code obtained from another student,  
//   or any other unauthorized source, either modified or unmodified.  
//  
// - All C++ language 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 instructor, ACM/UPE tutors or the GTAs 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.