Rectangle Quadtree on Disk

Assignment:

This project serves as a capstone for the semester, in that it builds on the earlier projects that you completed. You will re-implement Project 2 (with the same input commands), except that the entire database (rectangle records, skip list, and quadtree) will be stored in a file on disk. As with Project 3, all accesses to the disk file will be mediated by the buffer pool. Unlike Project 3, where the heap determined where on file all data would go, there is no simple mechanism for determining where any given piece of data (such as a rectangle record or skip list node) should go in the disk file. Thus, you will add a memory manager to determine where each piece of the database should be placed when it is stored.

Input and Command Line Parameters:

The input commands to the program will be read from standard input, and the input commands will be identical to Project 2. Your program will take three command-line parameters. The first parameter is the name of the file to be used to store the database while the program is running. Note that this will be your binary file (mediated by the buffer pool) and it will be initialized to be empty/zero length at the beginning of the program. The second parameter is the number of buffers in the buffer pool. The third parameter is the size (in bytes) for the buffers.

Implementation:

Just as in Project 3 you had to modify a normal memory-based heap to request that data be read from or written to the file via the buffer pool, you will need to modify your implementations for the skip list and quadtree to request data access to the buffer pool rather than dereferencing pointers or allocating space for records and nodes. Only a fixed (constant) amount of buffer space should be needed by your program for (temporarily) holding the contents of various nodes or records. The database itself is maintained on disk. For the remainder of this discussion, I will refer to the data exchanged between the database and the buffer pool as *messages*. A given message might be a rectangle record, or a skip list node, or a quadtree node. For each message type that you will be exchanging with the buffer pool (i.e., a rectangle record or a quadtree node), you will need access functions to transfer the data to/from the buffer pool. This is one of the two major changes from what you implemented in Project 2.

The second major change is that you need another entity to manage *where* in the file that a given message should be stored. When implementing the heapsort, the heap determined where each data record belonged within the heap's array representation. For this project you will implement a *memory manager*, whose job is to keep track of what bytes in the disk file are available to write information, and which bytes already store messages. Before beginning this project, you should read Section 12.4 in the textbook on memory management. Note that the database will not actually make calls to the buffer pool. It will actually make all of its access requests to the memory manager, which in turn will interact with the buffer pool to get the message transfered to/from disk.

Your memory manager will maintain a linked list of free blocks in the disk file, and you will use a **best fit** allocation policy, as described in Section 12.4.1. Note that the memory manager (including its linked list of free blocks) will be implemented in main memory.

A key consideration for your implementation is how the memory manager's client (the database in this project) interacts with the memory manager to store and retrieve its messages. Your memory manager should take a message and its size as input, and give back a **handle**. When the client

wants the contents of a message, it hands the handle to the memory manager, along with a suitable buffer in which the memory manager can copy the contents of the message. To make your project easier to implement, the handle should be four bytes long. The handle will actually be the byte position in the file for the message (although only the memory manager actually knows this). The skip list and the quadtree will replace pointers to records or nodes with handles. To recover a message, the memory manager actually needs to know the starting position of the message and the length of the message. The memory manager will get the starting position from the handle. The memory manager should put the length of the message at the beginning of the message, storing this as a two-byte quantity. For example, if the message was 20 bytes long, then the memory manager will locate the smallest free block that is at least 22 bytes long. It will store the message length in the first two bytes, and then the message in the remaining 20 bytes. Any additional space in the free block beyond 22 bytes will be retained as a free block. For example, if the smallest free block is 40 bytes long, then 18 bytes will be returned to the free list.

If a request is made for a free block that cannot be serviced by the existing pool of free blocks, then new space is allocated at the end of the file, causing the file size to grow as necessary. It is acceptable to allocate an entire block (equal to a buffer pool's buffer size) or multiple blocks (when necessary) to service the request. Any additional space allocated beyond that needed to store the message would be added to the free block list. For example, if the block (buffer) size is 256 bytes and the message is 100 bytes long, then the remaining $256 - 102 = 154$ bytes should make up a new free block. Be careful that any existing free block at the end of the file is used as part of this allocation. For example, if the last 50 bytes of the file are free, the message is 100 bytes long, and blocks (buffers) are 256 bytes, then a new block will be allocated, yielding 306 total free bytes at the end of the file. 102 of those bytes are allocated to store the message, leaving a free block of 204 bytes.

Your memory manager must merge adjacent free blocks together whenever space for a message is freed by the client. You will need to work out some suitable approach for doing so, by examining the current free blocks in the linked list to determine if the newly freed block is adjacent to one or two existing free blocks.

There is one change to the output requirements from Project 2. For Project 4, the "dump" command will also print out a listing of the free blocks currently in the memory manager, indicating for each its position in the file and size of the free block.