Basic Memory Manager

Assignment:

You will write a memory management package for storing **variable-length records** in a large memory space. For background on this project, view the tutorial on sequential fit memory managers available at `http://research.cs.vt.edu/algoviz/MMtutorial/tutorial.php`.

Your **memory pool** will consist of a large array of bytes. You will use a doubly linked list to keep track of the free blocks in the memory pool. This list will be referred to as the **freeblock list**. The freeblock list should store the free blocks in descending order by size of the free block. If two or more blocks are of the same length, then they should appear in ascending order of their position in the memory pool. You will use the worst fit rule for selecting which free block to use for a memory request. That is, the first free block in the linked list (which is the largest block) will be used to service the request if possible. If not all space of this block is needed, then the remaining space will make up a new free block and be returned to the free list.

Be sure to merge adjacent free blocks whenever a block is released. To do the merge, it will be necessary to search through the freeblock list, looking for blocks that are adjacent to either the beginning or the end of the block being returned. Do **not** merge the free blocks at the beginning and end of the memory pool. That is, the memory pool itself is not considered to be circular.

The records that you will store will contain the *xy*-coordinates and name for a city. Aside from the memory manager's memory pool and freeblock list, the other major data structure for your project will be the **record array**, an array that stores the "handles" to the data records that are currently stored in the memory pool. A handle is the value returned by the memory manager when a request is made to insert a new record into the memory pool. This handle is used to recover the record. (Note that the record array is something of an artificial construct that is being used to simplify testing the memory manager for this project. It will be replaced by something more appropriate in later projects. The idea is that the record array gives us an easy way to identify the records independent of their placement in the memory pool.)

Invocation and I/O Files:

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

```
java memman <pool-size> <num-recs> <command-file>
```

The name of the program is `memman`. Parameter `<pool-size>` is the size of the memory pool (in bytes) that is to be allocated. Parameter `<num-recs>` is the size of the record array that holds the handles to the records stored in the memory pool. Your program will read from text file `<command-file>` a series of commands, with one command per line. The program should terminate after reading the end of the file. No command line will require more than 80 characters. The formats for the commands are as follows. The commands are free-format in that any number of spaces may come before, between, or after the command name and its parameters. All coordinates will be signed values small enough to fit in an `int` variable.

insert *recnum x y name*

Parameter *recnum* specifies which slot in the record array will hold the handle for this record. An error should be reported if the value of *recnum* is outside of the range 0 to `num-recs` − 1. Parameters x and y are the *xy*-coordinates for the record, and *name* is the name of the city for this record. *name* may consist of upper and lower case letters and the underscore symbol. If there is already a record stored at position *recnum* in the record array, then that earlier record should first be removed from the memory pool. If there is no room in the memory pool to handle the request, print a suitable message and do not modify the memory pool in any way. If the insert command is to a *recnum* that is already used, then the first step will be to delete the old record, and the second step will be to attempt to insert the new record. Should this attempt to insert fail, then the old record will remain deleted.

remove *recnum*
Remove the record whose handle is stored in position *recnum* of the record array. If there is no record there, print a suitable message. An error should be reported if the value of *recnum* is outside of the range 0 to `num-recs` − 1.

print *recnum*
Print out the record (coordinates and name) whose handle is stored in position *recnum* of the record array. If there is no record there, print a suitable message. An error should be reported if the value of *recnum* is outside of the range 0 to `num-recs` − 1.

print
Dump out a complete listing of the contents of the memory pool. This listing should contain two parts. The first part is the listing of city records currently stored in the memory pool, in order of the record number. Print the value of the position handle along with the record. The second part is a listing of the free blocks, in order of their occurrence in the freeblock list.

Design Considerations:

Your main design concern for this project will be how to construct the interface for the memory manager class. While you are not required to do it exactly this way, we recommend that your memory manager class include something equivalent to the following methods.

```
// constructor
MemManager(int poolsize);

// Insert a record and return its position handle.
// space contains the record to be inserted, of length size.
Handle insert(byte[] space, int size);

// Free a block at posHandle. Merge adjacent blocks if appropriate.
void remove(Handle theHandle);

// Return the record with handle posHandle, up to size bytes.
// Place the record into space.
void get(byte[] space, Handle theHandle, int size);

// Dump a printout of the freeblock list
void dump();
```

Another design consideration is how to deal with the fact that the records are variable length. One option is to store the record's handle and length in the record array. An alternative is to store the record's length in the memory pool along with the record. Both implementations have advantages and disadvantages. We will adopt the second approach.

The records stored in the memory pool **must** have the following format. The first byte will be the length of the record, in bytes. Thus, the total length of a record may not be more than 256 bytes. The next four bytes will be the x-coordinate. The following four bytes will be the y-coordinate. Note that the coordinates are stored in binary format, not ASCII. The city name then follows the coordinates. You should **not** store a NULL terminator byte for the string in the memory pool.