CS5040 (Fall 2023)
PROGRAMMING ASSIGNMENT #1

Due Thursday, September 14 @ 11:00 PM for 100 points

Due Wednesday, September 13 @ 11:00 PM for 10 point bonus

Note: This project also has three intermediate milestones. See the Piazza forum for details.

Assignment:

The projects for this semester will build upon each other to form a scheduling system for training sessions. Eventually, it will be possible to enter records for training sessions, delete them, and search by keyword, time, cost, or location.

For your first project, you will write a memory management package for storing **variable-length records** in a large memory space, and you will use a hash table to access the records by a simple key value. (The records will be a serialized version of the seminar information.) For background on memory managers, see Chapter 11 in OpenDSA. Your memory manager will use the **buddy method** for allocating space from the memory pool, as described in Module 11.9.1.1. Note that you will use the memory manager again in a later project.

Your **memory pool** will consist of a large array of bytes. Initially, this array will be some power of two specified by a command line parameter. If a request comes to store a record that there is no room to store, then you will replace the current byte array with a new one that is twice as long, and copy all of the contents of the old byte array to the new one before attempting again to satisfy the request. (You will also output a suitable message.) When a record is to be removed from the database, its associated block of bytes is given back to the freeblock list maintained by the memory manager. Be sure when a block is released to merge it with its buddy if the buddy is free, as shown in the Buddy Method visualization at OpenDSA. Potentially, this could cause a cascade of buddy merges.

Aside from the memory manager's memory pool and freeblock list, the other major data structure for your project will be a **closed hash table** (Module 10.6) for storing an integer as the search key (this will be the ID field for the records being stored), and a memory manager handle as the payload. For information on hash tables, see the chapter on Hashing in the OpenDSA textbook (Chapter 10). You will use double hashing as described in Module 10.7.4. In particular, you will use the approach shown in the second slideshow of that section. The hash table size will always be a power of two. When hashing key value k (the ID value) $h_1(k) = k \bmod M$ which is simply the ID mod the table size, and $h_2(k) = (((k/M) \bmod (M/2)) * 2) + 1$. The key difference from what OpenDSA describes is that your hash tables must be **extensible**. That is, you will start with a hash table of a certain size (defined when the program starts). This must be a power of two, or the program should immediately terminate with an error message. If the hash table exceeds 50% full, then you will replace the array with another that is twice the size, and rehash all of the records from the old array For example, say that the hash table has 32 slots. Inserting 16 records is OK. When you try to insert the 17th record, you would first re-hash all of the original 16 records into a table of 64 slots.

The hash table will somehow need to distinguish each record's key from the rest of that record's value. Ideally, it is shielded from the fact that records are are stored in a memory manager. One possible design is to store the key (one field of the record) and some sort of reference to the rest of the record, Another is to store the memory manager's "Handle" to the data record. (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.) Another design is to hide all of

these implementation details behind a Record object, and let the Hash system get its necessary information through the Record's methods.

Seminar Records

A seminar record contains the following fields:

- Title: A string
- Date/Time: A string in the format YYMMDDHHmm where YY is the last two digits of the year, MM is the month, DD is the date, HH is the hour (24-hour clock) and mm is the minutes.
- Length: An integer (minutes).
- Keywords: A list of keywords. Keywords are strings.
- Location: Two unsigned short integers representing the X and Y coordinates (short integers are two bytes each).
- Description: A string.
- Cost: An integer (whole dollar amount).
- ID: An integer that uniquely identifies the seminar.

In order for us to properly test your program, it is critical that Web-CAT and your program agree on the serialized form of the records as stored in the memory manager (since the messages stored have to be exactly the size that we expect). Therefore, the starter code for this project will include a class definition for the seminar records, along with methods to serialize and deserialize the seminar record to/from a byte array.

Invocation and I/O Files:

The program would be invoked from the command-line as:
`java SemManager {initial-memory-size} {initial-hash-size} {command-file}`

The name of the program is `SemManager`. Parameter `{initial-memory-size}` is an integer that is a power of two, and it specifies the initial size of the memory pool. Parameter `{initial-hash-size}` is an integer that is a power of two, and it specifies the initial size of the hash table (in terms of slots). Your program will read from text file `{command-file}` a series of commands. You do not need to check for syntax errors in the command lines (although you **do** need to check for logical errors such as illegal duplicate insertions or deletions of records with non-existent keys). The program should terminate after reading the end of the file. The formats for the commands are as follows. No line of the command file will require more than 80 characters. The commands are free-format in that any number of spaces may come before, between, or after the command name and its parameters.

All commands should generate a suitable output message. All output should be written to standard output. Every command that is processed will generate some sort of output message to indicate whether the command was successful or not. Complete details for the proper output from commands will be found in the example output file to be posted with this project description.

Commands and their syntax are as follows.

insert *ID*
<title>
<date/time> <length> <x> <y> <cost>
<keyword list>
<description>

An insert command spans five lines. There will be no blank lines within an insert command. No line will require more than 80 characters. The ID is an integer The title appears on a single line. You should trim any leading or trailing whitespace, but preserve any whitespace within the title. The date/time is a string (with no internal whitespace), x and y are short integers, while length and cost are integers. There can be multiple keywords on the keyword line (separated by spaces), but there is only one line of keywords. The description is on a single line. You should trim any leading or trailing whitespace, but preserve any whitespace within the description. It is an error to attempt to insert a record whose ID duplicates that of an existing record in the database. Such inserts should be ignored (with a suitable message printed).

delete `{ID}`
Remove the record with ID value `{ID}` if it is in the database.

search `{ID}`
Print the record with ID value `{ID}` if it is in the database.

print `hashtable`
print `blocks`
Depending on the parameter value, you will print out either a complete listing of the contents of the hash table, or else you will list the free blocks in the memory pool. For printing the hash table, simply move sequentially through it retrieving the records and printing their ID in the order encountered (along with the slot number where it appears in the hash table). Then print the total number of records that are stored. If the parameter is `blocks`, then print a listing of the freeblocks. The format is shown in the sample output file.

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. poolsize defines the size of the memory pool in bytes
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);

// Return the length of the record associated with theHandle
int length(Handle theHandle);
```

```
// Free a block at the position specified by theHandle.
// Merge adjacent free blocks.
void remove(Handle theHandle);

// Return the record with handle posHandle, up to size bytes, by
// copying it into space.
// Return the number of bytes actually copied into space.
int 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. That is, each record stores strings, and so different records need different amounts of space. One option is to encode the length of the record in that record's handle. 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 first approach.** Therefore, the handle class will need to store both the start position of the record in the memory pool, and the length of the record.

The trickiest design aspect is likely to be the relationship between the hash table and the memory manager. A simple approach is to make the memory manager belong to the hash table. But this is not a good idea for the futue since it makes the hash table implementation unnecessarily ideosyncratic to this project. Future projects are unlikely to pair the hash table and memory manager together. So it is better to find a way to let the class that processes the command deal directly with the memory manager and hide this from the hash table.

There should be good separation between the main SemManager, your command processing code, and your code that does inserts/deletes/searches in the database. In particular, SemManager should not be reading commands from the command file, and the class reads the commands should not even know about the hash table or memory manager.

Programming Standards:

You must conform to good programming/documentation standards. Web-CAT will provide feedback on its evaluation of your coding style, and be used for style grading. Beyond meeting Web-CAT's checkstyle requirements, here are some additional requirements regarding programming standards.

- You should include a comment explaining the purpose of every variable or named constant you use in your program.
- You should use meaningful identifier names that suggest the meaning or purpose of the constant, variable, function, etc. Use a consistent convention for how identifier names appear, such as "camel casing".
- Always use named constants or enumerated types instead of literal constants in the code.
- Source files should be under 600 lines.
- There should be a single class in each source file. You can make an exception for small inner classes (less than 100 lines including comments) if the total file length is less than 600 lines.

We can't help you with your code unless we can understand it. Therefore, you should no bring your code to the GTAs or the instructors for debugging help 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 provided by the instructor. Note that the OpenDSA 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.

Java Data Structures Classes:

You are not permitted to use Java classes that implement complex data structures. This includes `ArrayList`, `HashMap`, `Vector`, or any other classes that implement lists, hash tables, or extensible arrays. (You may of course use the standard array operators.) You may use typical classes for string processing, byte array manipulation, parsing, etc. **Exception:** You may use ArrayList within your parser to help simplify processing of the keywords list. No Arraylist should be used by the hash table or memory manager classes.

If in doubt about which classes are permitted and which are not, you should ask. There will be penalties for using classes that are considered off limits.

Deliverables:

You will implement your project using Eclipse, and you will submit your project using the Eclipse plugin to Web-CAT. Links to the Web-CAT client are posted at the class website. If you make multiple submissions, only your last submission will be evaluated unless you arrange otherwise with the GTA. 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 these 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.

When structuring the source files of your project, use a flat directory structure; that is, your source files will all be contained in the project "src" directory. Any subdirectories in the project will be ignored.

You are permitted to work with a partner on this project. While the partner need not be the same as who you worked with on any other projects this semester, you may only work with a single partner during the course of one project unless you get special permission from the course instructor. 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.

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 current or
//   former 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.