CS 5040: Intermediate Data Structures and Algorithms
Virginia Tech
Fall 2020
Dr. Edmison

Programming Assignment 1

Last Updated: 08-18-2020 9:49:46-04:00

Due Friday, September 18 @ 11:00 PM ET for 125 points
Due Wednesday, September 16 @ 11:00 PM ET for a 10 point bonus
Note: This project also has three intermediate milestones.
See the Piazza forum for details.

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 modules on memory managers (Chapter 14) in the OpenDSA class textbook. Your memory manager will use the **buddy method** for allocating space from the memory pool, as described in Module 14.09.

The records that you will store for this project contain a series of one or more strings. 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. When a string 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 to merge adjacent free blocks whenever a block is released, as shown in the Buddy Method visualization at OpenDSA.

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 13.06) for storing strings. For information on hash tables, see the chapter on Hashing in the OpenDSA textbook (Chapter 13). You will use the second string hash function described in the book (the code is provided to you in the starter files), and you will use simple quadratic probing for your collision resolution method (the i'th probe step will be i^2 slots from the home slot). 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). 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

100 slots. Inserting 50 records is OK. When you try to insert the 51st record, you would first re-hash all of the original 50 records into a table of 200 slots. Likewise, if the hash table started with 101 slots, you would also double it (to 202) just before inserting the 51st record.

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 the strings are stored in a memory manager. One possible design is to store the name string 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.

What Gets Stored

The records to be stored for this project contain a name string, and might contain one or more key-value pairs (where the key and the value are each strings). The records will be "serialized" as a single string. Each record begins with the canonical form of the name string. Each field name-value pair will be added (in canonical form) with the characters `<SEP>` separating them.

We will use the following terms to help describe exactly what to store.

- **Separator**: The literal characters `<SEP>`.
- **White Space**: Any series of intermixed spaces and tabs.
- **Name**: A record name, field name, or field value may each contain any series of upper/lower case letters, numbers, whitespace (spaces or tabs), and normal punctuation, except that they are prohibited from containing the Separator.
- **Canonical Name**: A Name where all leading and trailing whitespace has been removed, and any internal whitespace is replaced with a single space character. See the sample input and output for examples.

Invocation and I/O Files

The program will be invoked from the command-line as:
`java MemMan <initial-memory-size> <initial-hash-size> <command-file>`

where:

- `MemMan` is then name of the program. The file where you have your main() method must be called MemMan.java
- `initial-memory-size` is an integer that is a power of two, and it specifies the initial size of the memory pool.

- `initial-hash-size` is an integer that specifies the initial size of the hash table (in terms of slots).
- `command-file` is the name of the command file to read.

Your program will read from text file `command-file` a series of commands, with one command per line. You are guaranteed that the commands in the file will be syntactically correct in all graded test cases. The program should terminate after reading the end of the file. 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 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.
The command file may contain any mix of the following commands. In the following description, terms in `{}` are parameters to the command.

- `add {name}`

  Insert [name] into the database (in canonical form) if it is not already there.

- `delete {name}`

  Remove [name] (in its canonical form) if it is in the database.

- `update add {name}<SEP>{field_name}<SEP>{field_value}`

  In this version of the **update** command, we will either add a field and value (if that field is not already part of the record), or else we will replace an existing value.

- `update delete {name}<SEP>{field_name}`

  In this version of the update command, we will remove that field name and its value from the record (if it exists).

- `print hashtable`

  Print out either a complete listing of the contents of the hash table. For printing the hash table, simply move sequentially through it retrieving the records and printing them 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.

- `print blocks`

  Print a list of the free blocks in the memory pool. 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.

Also, you should design your command file parser to be as agnostic as possible. It should not be tied to any particular command input format. This will allow you to reuse the command file parser in later projects with minimal changes. **You will have to process a command file in a later project.**

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 meet-

ing 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.

Allowed Java Data Structure 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 use the standard array operators. You may use typical classes for string processing, byte array manipulation, parsing, etc.

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 should implement your project using Eclipse, and you should 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***. 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 a 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***. Make sure that you declare your partner in Web-CAT when submitting. 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.

Honor Code Pledge

Your project submission must include this 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 must include:

```
// 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.