P4 MemManage Specification

Link to the Project 4 assignment page (https://canvas.vt.edu/courses/156039/assignments/1568392)

Link to Project Approach Lecture sketches (https://canvas.vt.edu/courses/156039/files/folder/Project4/Images)

Assignment

You are tasked with storing and managing some basic data about a number of songs. This is normally quite simple, but these songs could be part of a sampling library for editing large audio projects. Software that is able to manage **this many** **(https://youtu.be/9d4-URyWEJQ?t=100)** songs allows a musician to **explore** **(https://youtu.be/VPLCk-FTVvw)** farther than ever before. For this software to be more usable (and not just an endless loading screen), adding and removing songs must happen as quickly as possible, and it will only need a single dedicated memory space.

In a general sense, you will write a memory management package for storing variable-length records in a large memory space. There will also be hash tables tasked with quickly finding the locations of the stored records. **For background on this project, refer to the modules that cover hashing, closed hash tables, and memory managers using sequential fit in the OpenDSA textbook.**

The **memory pool** of your memory manager will be a large array of bytes. You will use a doubly linked list to keep track of the locations of free blocks within the memory pool. This list will be referred to as the **freeblock list**. You will use the **best fit** rule for selecting which free block to use for a memory request. That is, the smallest free block in the linked list that is large enough to store the requested space will be used to service the request (if any such block exists). 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. If there is no free block large enough to service the request, then you will grow the memory pool, as explained below.

Be sure to merge adjacent free blocks whenever a block is released. To do the merge, whenever a block is released 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 released. Do not consider the first and last memory positions of the memory pool to be adjacent. That is, the memory pool *is not circular*.

Aside from the memory manager's memory pool and freeblock list, the other major data structure for your project will be **closed hash tables**. There will be two of them: one for locating data based on the artist name, and one for locationg data based on the song titles. These hashtables will hash the artist/song name and use that hash to store a MemoryHandle in the table's array. They do not need to store the strings, since that is the memory pool's job. Instead, the hashtable will use one a memory handle to retrieve the full record of data from the memory manager.

Both of these hash tables will have the following properties:

1. Uses the string folding hash function (described in OpenDSA and given below)
2. Use quadratic probing as the collision resolution mechanism (probe step `i` will attempt to use slot `homeSlot + i*i`)
3. Replace the slot with a tombstones during removal, and allow insertion to replace a tombstone on collision.
4. Support for **extending** the table to grow its size
    - The starting size of the hash table is given at program start (in the args)
    - If the hash table would is 50% full before an insertion, then you will replace the array with another that is twice the size, and rehash all of the records from the old array to put them in the new 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.

Invocation and I/O Files

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

```
java MemMan {initial-hash-size} {initial-block-size} {command-file}
```

where:

- `MemMan` is the name of the program. The file where you have your `main()` method must be called `MemMan.java`
- `initial-hash-size` is an integer that specifies the initial size of the hash table (in terms of slots).
- `initial-block-size` is an integer that is the initial size of the memory pool (in bytes).
- `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 commands are free-format in that any number of spaces may come before, between, or after the command name and its parameters. All output should be written to standard output. **Every command that is processed should generate a corresponding output message to indicate whether the command was successful or not.**

Whenever the memory pool has insufficient space to insert the next request, it will be replaced by a new array that adds an additional `{initial-block-size}` bytes. Note that this is not doubling the memory pool size each time to expand, so it is different than the hash table's growth. All data from the old array will be copied over to the new array, the freeblock list will be updated appropriately, and

then the new string will be added. Using `System.arraycopy(src, sPos, dest, dPos, length)` is a convienent way to move data to and from the memory pool to other byte arrays.

The command file may contain any mix of the following commands. In the following description, terms in `{ }` are parameters to the command. The sub-points after the command show an example of that command's expected output.

- `insert artist {artist-name}`
  - `|Franz Ferdinand| is added to the artist database`
- `insert song {song-name}`
  - `|Use Somebody| is added to the song database`
- `insert {artist-name}<SEP>{song-name}`
  - `|Franz Ferdinand| is added to the artist database`
  - `|No You Girls| is added to the song database`
  - `|Kings Of Leon| is added to the artist database`
  - `|Wicker Chair| is added to the song database`
  - `|Kings Of Leon| duplicates a record already in the artist database`
  - `|Use Somebody| duplicates a record already in the song database`

Note that the characters `<SEP>` are literally part of the string (this is how the raw data actually comes to us), and are used to separate the artist name from the song name. Check if `{artist-name}` appears in the artist hash table, and if it does not, add that artist name to the memory pool, and store the resulting handle in the appropriate slot of the artist hash table. Likewise, check if `{song-name}` appears in the song hash table, and if it does not, add that song name to the memory pool, and store the resulting handle in the appropriate slot of the song hash table. You should print a special message whenever an insert causes a hash table or the memory pool to expand in size (e.g. `Memory pool expanded to be 64 bytes` or `Artist hash table size doubled`).

- `remove song {name}`
- `remove artist {name}`
  - `|Mississippi Boweavil Blues| is removed from the song database`
  - `|(The Best Part Of) Breakin' Up| does not exist in the song database`
  - `|Johnny Cash| is removed from the artists database`

Remove the specified artist or song name from the appropriate hash table and the memory pool. Report the outcome (whether the name appears, and whether it was successfully removed). Be aware that you need to locate an exact match, since several different artists/songs might have the same homeSlot in the hash table.

- `print blocks`
  - `(44,11) -> (97,28) -> (239,17)`

Print out a listing of the current freeblock list in the memory pool. The order of printing should match the order in the freeblock list. For each block, print its start position and its length.

- `print artists`
- `print songs`
  - `|Charley Patton| 4`
  - `|Bukka White| 5`
  - `total artists: 2`

Print out a complete listing of the artists or songs contained in the database. Simply move sequentially through the associated hash table, and use the memory handles found to retrieve the data from the memory pool. Convert that data to a string, and print it along with the slot number of where it appears in the hash table. Then print the total number of artists or total number of songs.

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);

// 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. Some strings are very long, and some are short. One option is to encode the length in the record's handle, which would mean the hash table would be responsible for it. But there is an alternative that lets the memory pool have more control. We can 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 two bytes will be the number of bytes in the *encoded* record, in bytes. The value of that number is "k". The next k bytes after the first two will be the string encoded into bytes (Can convert back and forth using `aStr.getBytes(); <-and-> new String(byteArr, offsetB, lenB);`).

k will never be a negative number, so we will utilize the full range of 2 bytes: from zero to 2^{16}. However, Java shorts are signed so they have a range of -(2^{15}) to 2^{15}. We should find a way to make that last bit of data useful. java.nio.ByteBuffer's `getShort(pos)` and `putShort(pos,x)` methods may be helpful here. Also, there is `Short.toUnsignedInt(s)` and of course type casting if you know the outcome.

sFold hashing algorithm:

```
public int sFoldHash(String s) {
    int intLength = s.length() / 4;
    long sum = 0;
    for (int j = 0; j < intLength; j++) {
        char[] c = s.substring(j * 4, (j * 4) + 4).toCharArray();
        long mult = 1;
        for (int k = 0; k < c.length; k++) {
            sum += c[k] * mult;
            mult *= 256;
        }
    }
    char[] c = s.substring(intLength * 4).toCharArray();
    long mult = 1;
    for (int k = 0; k < c.length; k++) {
        sum += c[k] * mult;
        mult *= 256;
    }
    return (int)(Math.abs(sum); // don't forget to % table size
}
```