

Extra Credit Lab

*April 20, 2004**Due Date: May 10*

Here are three extra credit lab options – You can do only one of them. In order to submit an extra credit lab you must have submitted all 4 required labs in by the due date. If you lost more than 10 points on one of the required labs, then instead of doing the extra lab, you have the option of instead resubmitting one of the regular labs. Your lab grade will be replaced by the grade of your revision - 10 points (the late penalty).

For everything that is provided in Java on the webpage, I have comparable files for C++ that use a textual interface instead of a GUI. If anyone would like to do this in C++ just contact me and I can provide you with those files.

Option 1 (25 points)

Replace your skiplist implementation for Lab 3 by a red-black tree. (So you need just implement the red-black tree methods needed by Lab 3.) You should submit the output required for Lab 3 (for both 20 events and all events). However, there is no need to show the search path through the red-black tree. For the 20events, just output the red-black tree in some form that we can use to draw the red-black tree to be sure it is correct. (You can use an inorder traversal to do this). Also show the output for allEvents.

Option 2 (25 points)

Read about Fibonacci heaps in the text book and implement a Fibonacci heap class that has all of the public methods from the BinaryHeap class of Lab 4. You should submit the output from both the BinaryHeapTester and for the application of Part 3 exactly as required in Lab 4.

Option 3 (25 points)**Part 1 (10 points)**

For this lab you are going to modify your Lab 2 for a toy inventory program for the very common situation in which the hash table itself (i.e. the keys and references to the associated data) are stored in main memory, and the associated data (i.e. the data records) are stored in a file (i.e. on disk). For the toy inventory system the associated data will hold a quantity (integer), price (double) and name (String). Of course in a real system, the associated data would contain many other fields. The associated data should only be brought into main memory when it is being read or modified. If it is updated then that change should be reflected in the file holding the data. Note that hash table itself should be in main memory the entire time the program is executing.

You are *not* expected to already know about file I/O. A simple GUI that demonstrates the use of a random access file in java is provided in `SampleIO.java` which is the driver for `SampleGUI.java`. The data is saved in a file called `test.dat` and records inserted to it in one session remain there in later sessions. If you want to start from scratch again, just delete the file `test.dat`. When you are implementing this part, remember that when you have found an error and fixed it and are ready to start over again from scratch, you will want to remove the data file that you create.

I think the use of the GUI should be self explanatory. Also, the provided java files are commented extensively and from the comments everything should be clear – If it is not, just ask. In order to make Part 3 manageable, when storing the name to the data file it will always be stored as 20 characters (padded with blanks or truncated) as needed. The provided sample program shows you how to do this. If you are not doing Part 3 then you don't need to do this but you may. Finally, a simply GUI is in `GUI.java` that makes the needed calls to your `StringDictionary` of Lab 2.

Here's a list of the changes you need to make to `StringDictionary`.

- Change your constructor so the parameter passed is the size of the dictionary. Be sure to use the provided `baseHash` and `stepHash` functions from Lab 2. I want to be able to control the collisions created so I can create good test data. To do this I need to know your dictionary size. I will guarantee a load factor of no more than 1/2.
- You must add an `update(int key, Record newData)` method that replaces the associated data for `key` to `newData`.
- You need to include a `load` and `save` method that both do nothing and return `false`. You'll create these for Part 2 but since the inventory GUI calls them, they must be there.
- Finally, the main change is to modify `StringDictionary` so that the associated data is stored on disk and just brought into main memory when read and updated.

Part 2 (5 points)

Clearly when your program exits, you do not want to lose your hash table which is the mechanism by which you can access the data. Thus you should implement a load and save method. The `save` method will be called when the GUI is exited and should save the hash table itself to a file. The `load` method should be called within the constructor for the `StringDictionary` and rebuild the hashtable for the data file (unless it is the first time run in which case a new hash table needs to be created). Load and Save must be written without using any provided Java method that does this for you such as an object output stream or the java serializable functionality.

Part 3 (10 points)

Within Part 1, whenever a data item is removed, the data will still be in the file but it is garbage that is wasting space. What you really want to do is reuse the file space when later inserts are made. In this part you will keep a “free list” that will contain the offsets (or record numbers) in the file that are available for re-use. Furthermore you will keep this free list in the data file itself by just using the first 8 bytes as a “head pointer” to the free list. Then each “cell” in the free list will have the location of the next free cell. So in essence you have a linked list of the available cells that are actually stored in the free cells. This is a very short description so just come ask me to explain in more depth if it is not clear.

What to Submit

The provided `Inventory.java` and `GUI.java` keep a history of the inputs and return values from the `StringDictionary` that are printed out using `Terminal.println(...)`. On the web page under labs will be a list (`part1-operations`) of required operations that you should execute in the GUI to demonstrate that Part 1 works. The transcript created by the `Inventory` program when doing the requested sequence of operations should be submitted. Feel free to add the call to `startTranscript("filename")` if you want to `Inventory.java`. It is currently set up to take this filename in a command line argument.

For Part 2, you should exit the GUI and then restart it. On the course web page under labs there will be a clearly marked list (`part2-operations`) of operations that you should execute once you restart the GUI. Again, you should submit the transcript created by the `inventory` program.

If you do Part 3, then every time an insert (i.e. `put`) is performed, you are to print (using `Terminal.println(...)`) the disk offset (or record number) where the data is being placed followed by the free list. Likewise, whenever remove is performed, you are to use `Terminal.println(...)` to show the free list. This output will then be included in the traces that are being submitted with Parts 1 and 2. There is no need to submit separate output for Parts 1 and 2 that do not use the free list. Just submit the two transcript files from which we can check that Parts 1 and 2 work correctly and also see the information to check the free list.