

Extra Credit Lab

*April 18, 2006**Due Date: May 5th by 4pm*

Here are three extra credit lab options – If you submit all of your required labs by the original grading date then you can do ONE extra credit lab. 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 minus the 10% late penalty. If you have a different extra credit project you'd like to do (related to what we've been covering), you are welcome to ask Dr. Goldman about it.

Option 1 (25 points)

Replace your skiplist implementation for Lab 3 by a red-black tree. You should submit the output required for Lab 3 (for both 20 events and all events). There is no need to show the search path through the red-black tree. For the 20events, output the red-black tree in some form that shows the structure (including the node colors). You are not allowed to look at or make use of any red-black tree implementations you find (on the web or elsewhere).

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. Again, you are not allowed to look at or use any Fibonacci heap implementations.

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

For this lab you are going to modify your Lab 2 to to implement a Mapping ADT that will be used for a “toy” inventory program for the 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 is the name (String), quantity (integer), and price (double). 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. The hash table itself should be in main memory the entire time the program is executing.

In order to avoid unnecessary memory allocation and garbage collection of Record objects, the following interface must be supported.

`void put(String key, Object data)` associates the specified data with the given key. The data itself must be stored on disk (in a random access file). If the mapping previously contained a mapping for this key, the old data (on disk) should be replaced by the specified data.

`boolean contains(String key)` returns true iff there is data associated with the given key.

`boolean get(String key, Record data)` retrieves the the data (if any) associated with the given key and stores it in the provided Record object `data`. It returns true iff there is a value associated with the given key.

`boolean remove(String key, Record data)` removes the mapping (if one exists) from the given key and puts the associated data (retrieved from disk) into the provided Record object `data`. It returns true iff there was a value associated with the given key (which was then removed).

`void save()` Saves the associations held in the mapping to disk.

`void load()` Loads the associations that were saved from the last session.

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 `SampleGUI.java` that is called by the driver `SampleIO.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`. The class `Record` is also provided (with comments that should make everything clear, if not 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 the Mapping class that you are to provide. Note that you must include the `load` and `save` method since the inventory GUI calls them. For Part 1 just have them do nothing.

Part 2 (5 points)

Implement a load and save methods 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. You should reuse such wasted 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 for Option 3

Included in the zip file `ec-lab-provided.zip` are a list of operations to perform, `part1-operations` and `part2-operations`.

If you just do Part 1, then perform the operations given in `part1-operations` and submit the output created by the Inventory program when doing the requested sequence of operations should be submitted. (It keeps a history of the inputs and return values so that can check that your Mapping was working correctly).

For Part 2, you should do what is listed above and then exit the GUI and then restart it. Then you should perform the operations given in `part2-operations`. Submit the output created by the Inventory program for both `part1-operations` and `part2-operations`.

If you do Part 3, then every time an insert (i.e. `put`) is performed, you are to print 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 output 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 output from the Inventory program for both `part1-operations` and `part2-operations` and we can check everything from that.