

Lab 2

September 23, 1999

Due Date: October 7

CAREFULLY READ THIS ENTIRE ASSIGNMENT BEFORE BEGINNING!

“B” Portion:

As we discussed in class, we are going to build a simple inventory system. My task has been to implement a very simple front end that interacts with the user and makes calls to the CS241Dictionary class. In addition, I have implemented the DataRecord class that holds the data required for the inventory application. All of the provided code can be found on the course web page.

Your assignment is to implement the CS241Dictionary class. While I will use your class for the backend of the inventory system, your dictionary class could be used by any application as long as it is provided with a DataRecord class that has the public methods described below. You must implement the CS241Dictionary class by a hash table with open addressing using double hashing. You are *required* to use the provided hash functions and a hash table with $m = 101$ slots. More specifically, for key k you should consider the slots of the hash table in the order $s_0, s_1, s_2, \dots, s_{m-1}$ where $s_i = (\text{hash1}(k) + i \cdot \text{hash2}(k)) \bmod m$. While, you would expect very few collisions based on the hash table size and the number of objects that will be inserted in the dictionary, the data is carefully selected to really test your code well and hence was selected to force collisions for the provided hash functions.

Provided DataRecord Public Methods. Here’s the specification for the public methods of the DataRecord class that have been provided for your use. The other public methods are just used by the inventory application (which is also provided).

`clone` takes no arguments and returns a reference to a copy of the object’s DataRecord.

Note: In the Java implementation this returns a reference to an `Object` that can then be cast to a reference to a `DataRecord`.

`copyInto` takes as input a reference to a DataRecord and copies into it the object’s DataRecord.

The return type is `void`.

`write` takes as input a DataOutput Object (in C++ it takes a `fstream`) and it writes the DataRecord referenced by the object to it.

`read` takes as input a DataInput Object (in C++ it takes a `fstream`) (as written by `write`) and places it into the object’s DataRecord.

Required CS241Dictionary Public Methods. Here’s the specification for the public methods for the CS241Dictionary class. In completing your lab, you will create other classes and most likely some private methods for your CS241Dictionary class. You should be careful to never store in your dictionary a DataRecord reference passed as a parameter nor return a reference from your dictionary.

`insert` takes two parameters, first the key (an integer) and second the data (a reference to a DataRecord) and returns a boolean. If the key is already used then no change should be made to the dictionary and `false` should be returned. Otherwise, the key with the

reference of a COPY of the data should be inserted into the database and **true** should be returned. Remember, you do not want to expose the representation and hence you don't want to directly insert the provided `DataRecord` reference into your dictionary.

search takes two parameters, first the key (an integer) and second the `retrievedData` (a reference to a `DataRecord`) and returns a boolean. If the key is not in use then **false** should be returned and `retrievedData` is not changed. Otherwise, the the `DataRecord` stored with the key should be copied into `retrievedData` and **true** should be returned.

Note: If **search** returned a reference to a copy of the `DataRecord` then for every call to **search**, a new `DataRecord` would need to be allocated which would be a very inefficient use of space. And if **search** just returned a reference to the `DataRecord` stored in the dictionary then we would be exposing the representation. This interface was selected since it allows the user to reuse the same `DataRecord` (if desired) for many searches.

remove takes one parameter, the key (an integer) and returns a boolean. If the key is not in use then **false** should be returned. Otherwise, the key and its associated `DataRecord` should be removed from the dictionary returning **true**.

updateData takes two parameters, the first is the key (an integer) and the second is the new data (a reference to a `DataRecord`). If the key is not in use then **false** should be returned and no update is made. Otherwise, **true** is returned and the and the data should be updated with a COPY of the new data.

Make this method as efficient as you can (i.e. don't just implement it by a **remove** followed by an **insert** though some of the same private methods may be used).

updateKey takes two parameters, the first is the old key (an integer) and the second is the new value for the key (an integer). It returns an integer which is either **SUCCESSFUL** (0), **NOTFOUND** (1) or **INUSE** (2). Please define and use these names versus using 0, 1 and 2.

If the key is not in use then no change should be made to the dictionary and **NOTFOUND** should be returned. If the key is in use but the new key is also already used then no change should be made to the dictionary and **INUSE** should be returned.

Otherwise, the key should be updated (and whatever other changes are required because of this should also be made). Make this method as efficient as you can.

load takes no arguments and has a return type of void. If you are just doing the "B" portion then just have this procedure do nothing (i.e. **void load()**). If you choose to implement save and load, the load function will reload the hashtable into main memory.

save takes no arguments and has a return type of void. If you are just doing the "B" portion then just have this procedure do nothing (i.e. **void save()**). If you choose to implement save and load, the save function will save the hashtable to secondary storage.

You are only to submit any files that you wrote or modified and the output when your run your executable on the provided file `hash-test`. If you are using Java the provided inventory program is set up so that you can give two command line arguments (the first is the name of the input file and the second is the name of the file for the transcript). If you are having trouble using the command line arguments, you can comment out (and/or modify) lines that directly do this. If you are using C++ on Unix the easiest way to create your final output is with `inventory.out <hash-test >my-output`.

While you are debugging you may find it easiest to answer the questions asked by typing responses from the keyboard. However, if you just pick "random" bar codes then the probability that you will have a collision is very small. I have provided the hash function so that I

can force collisions while still just inserting a small amount of data. Hence, you may want to do a similar thing while you are doing your preliminary testing.

You are expected to very carefully check that the output is correct. There will be a 2/3 of a letter grade deduction for wrong output that is not clearly annotated as wrong. (This deduction is in addition to whatever deduction you receive because of the error that caused the output to be wrong). So go through your output carefully and *think* about whether or not the output is correct.

“A” portion:

In reality, it is likely that the data record would be stored in an external file in secondary storage versus in main memory. In order to receive an additional 2/3 of a letter grade for this assignment, revise your implementation so that the data (i.e. the price, quantity and name for a record) is stored in secondary storage. This information should only be brought into main memory when it is being read/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.

NOTE: You are *not* expected to already know about file I/O. The only task you need to handle is determining where in the file (i.e. at what offset) each `dataRecord` is stored. While, this takes some thought, it is independent of the actually code needed to read or write in a file. See the course web page for sample code demonstrating how to open a file and select a location/offset so that you can then use the provided `read` (respectively, `write`) methods for the `DataRecord` class to read from (respectively, write to) the file.

For an additional 1/3 of a letter grade keep a “free list” to reuse secondary storage space when an item is deleted.

For an additional 1/3 of a letter grade, implement the load and save functions. The save function should save the hash table and the dictionary records to a file. The load function should use the file created by save to rebuild the dictionary. The inventory program already calls load and save as appropriate – You do not need to change it. Note that you can do this part even if you do not do the other additional features. However, you should make this the last additional feature you implement since it will change depending on whether or not the data is kept in memory or secondary storage while the inventory program is running.

You have a good amount of flexibility in the design for this portion of the assignment. For example, you can just use one data file, or instead use two data files – one for data and another to actually save the hash table between sessions. However, be sure that you **DO NOT CHANGE** the provided `DataRecord` class or inventory program. Hence, you are not to change the calling structure of any of the publicly provided methods for the `CS241Dictionary`, just the representation and implementation of the methods.

As in the basic portion, all you need to submit here is any code that your wrote or modified. If you implemented the load/save functions you should first run your executable on `hash-test` (with an empty dictionary). Then run it on `hash-test-part2`. For this second run it should be reloading the data from the file (do NOT re-enter the data from `hash-test`).