

CMSC 212
Project #5
Due 11/29/2005 11:59 PM

1. Introduction

Often software is written for one purpose and then it is discovered that it is useful in a variety of other situations. In this case, the code is frequently generalized and put into a library so that many different applications can use the library.

One of the tricks of making libraries generic in C is how to represent the items to be stored. The way this is done in C is to use a generic pointer type, `void *`, to refer to items of an un-known type.

A second way to make software re-usable is to provide the same API, but to use different algorithms and data structures to implement them. For example, an API might be to store and lookup items, but the underlying implementation might be a linked list, a hash table, or a tree.

In this assignment, you will be given an implementation of a program (`intCount`) that reads through a file consisting of multiple lines each containing one or more numbers. It counts the number of times each number appears in the file. After reading the entire file, `intCount()` prints each number found in the file, in ascending order, together with the number of times that each number appeared.

The key bit of software to be re-used is the representation that stores the numbers and their frequency. The data is stored in an AVL tree. AVL trees are a type of binary tree that is balanced so that the depth of any two leaf nodes differ by at most one. You will extract the AVL implementation and transform it into an abstract table implementation that can store any data type, not just integers. The implementation will then be compiled into a shared library. You will implement a second shared library that performs the same functionality using chained hashing. Finally, you will implement an interface that allows multiple library implementations to be linked and used in a single program.

2. Table Interface Routines

There will be conceptually three parts to an application that uses one of your libraries: the application itself, the *table interface routines* (the stuff you will write in `table.c`), and the library implementation (the stuff that ends up in `libavl.so` or `libhashTable.so`).

The goal of the table interface routines is to allow dynamically linked table abstractions to register themselves with the application, and for the application to then call routines in the library implementation. This works as follows.

Upon linking in to a program (either when the program is first loaded, or later via `dlopen()`), the library's *constructor* is called. The constructor is specified via the

`__attribute__((constructor))` syntax you learned/will learn in lab. The constructor calls `defineImplementation()` with a pointer to a structure containing function pointers to relevant routines in the library, together with a string identifying the library. You store the name and the pointers in a linked list of registered implementations.

It's important to understand that when a library is loaded via `dlopen()`, the library's unresolved symbols are resolved against symbols defined in the application (and `table.c`). This is how the library's initialization routine can call `defineImplementation()`. However, the converse is not true. None of the library's symbols are visible to the application. There are then two approaches to calling routines in the library: having the library notify the application of function addresses, or having the application call `dlsym()` to resolve well-known function names. We are taking the former approach in this project.

The rest of the table interface routines are called from the application. The job of the interface routines is to receive these calls and vector them to the proper routine in a registered library.

The routines are used by the application to create and manipulate tables. These are rather straightforward as long as you understand how the table interface routines maintain per-table state (i.e. in a structure whose address is passed back to the application for safekeeping).

The one exception is `createTable()`. Two things to note. First, the parameters passed into the routine from the application include a pointer to a comparison function, and an additional, undefined pointer (both discussed more below). Second, the return value is a "tableADT". To the application, this is an opaque pointer (note the incomplete definition in `table.h`), but the real definition (in `tableInternals.h`) shows a pointer to a structure that keeps four pieces of state:

- 1) a set of pointers to functions provided by the implementation chosen for that table,
- 2) a pointer to the comparison function provided by the application via the `createTable()` call for that table
- 3) the extra, undefined pointer, and
- 4) a void *data* pointer.

The data pointer points to a library-specific structure containing all the data for that particular table. The application saves the `tableADT` pointer, including it on all subsequent calls into the table interface routines. Again, since the application should not include `tableInternals.h`, this pointer is opaque, and therefore the application never calls library routines directly. All calls to library routines must go through the API implemented in `table.c`.

The table interface routines are defined as follows:

```
void defineImplementation(tableImplementation *funcs, char *name)
```

Define an implementation of the table abstraction. The `funcs` parameter is a structure containing the addresses of the functions that implement the various features of the table API. The `name` parameter is the name of the implementation (i.e. hash table). Return is 0 for success and -1 for error (including re-defining the

name of an implementation). This is called by the library's initialization routine to register an implementation with the main program.

`tableADT createTable(char *implementationName, compareFunc, extraPtr)`

Create a new instance of the table using the desired implementation (`implementationName`). The caller supplies a comparison function to be used when the implementations need to compare two values. Returns `NULL` if the table can not be created, such as due to malloc errors, an undefined implementation name, or null `compareFunc`.

The integer-valued compare function takes as parameters two pointers to keys: **a** and **b**. It should return 0 if the keys pointed to by the parameters are equivalent (however the application defines this), -1 if **a** is less than **b**, and +1 if **a** is greater than **b**.

extraPtr is an optional parameter that can be used for any implementation-specific purpose. The hash table implementation will use this parameter to pass a pointer to a function that returns a hash value, given a pointer to a key. The AVL implementation will not use it.

`void *insertTable(tableADT t, void *key, void *item)`

Insert a new item with key value `key` into the table. On success the function should return `item`. On failure it should return null.

`void *lookupTable(tableADT t, void *key)`

Look for the item `key` in the passed table. If found, return the corresponding item.

`int visitTableItems(tableADT t, visitorFunc)`

Invoke the passed `visitorFunc` on every instance of the items stored in the table. Returns -1 if the implementation does not define this function, and 0 on success. Note that the order that in which items are visited is defined for the avl library, but not for the hash library.

`int deleteTableItem(tableADT t, void *key)`

Delete the item with the passed key from the table. Return 0 on success and -1 on failure.

`int deleteTable(tableADT t, visitorFunc cleanupItem)`

Delete the passed table. Return 0 on success. `cleanupItem` is a visitor function that is called on every node of the table just before it is deleted. If `NULL` is passed for this parameter, your implementation should not call a `cleanupItem` function for that table deletion.

3. Shared libraries

You will build two shared-library implementations of the table abstraction. Each must provide the functions defined in the `tableImplementation` structure in `table.h`. The first will be the AVL implementation, which is mostly written for you, and the second will be a hash implementation.

AVL Table Implementation

Almost all of this code is already provided for you in the `intCount` implementation. Mostly you will need to refactor it to operate on `void *` pointers and to put it into a separate file (`avl.c`). Note that this is still a bit of work, but it is relatively straightforward.

The `visit` function will need to traverse the AVL tree in the order left sub-tree, current node, right sub-tree. This will ensure that the `intCount` program will produce output in the same order as the original program.

`extraPtr` is not needed for the AVL implementation.

This library should register itself (when calling `defineImplementation()`) under the name “`avl`”.

Hash Table Implementation

You will also create a chained hash table implementation of the table abstraction. A chained hash table has a fixed number of elements in the hash array. To handle collisions, a linked list of items is maintained for each hash location. Your implementation includes all of the functions of the table abstraction.

The application will need to pass a pointer to a hash function in `extraPtr`. This function will be of type “`hashFunc`”, will have a single pointer to a `void` as a parameter, and will return an unsigned 32-bit value. Your hash table implementation will use this value to index into the hash table, possibly using modulo arithmetic to convert it to a smaller number that corresponds to the current size of the hash table.

As an example, the hash function for your integer count program might simply dereference its parameter, which is a pointer to an integer, and return this integer as the hash value. Note that the result of this function should never be used directly as an index, as your hash array will presumably be much smaller than 2^{32} . Also use the result of the `hashfunc` modulo the size of your hash array.

This library should register itself (when calling `defineImplementation()`) under the name “`hash`”.

Library Interfaces

Both the AVL and hash table implementations of the table abstraction will be stored in their own shared libraries. Each library will also contain an initialization function, identified by the attribute constructor preprocessor command (as you have used in the labs), that calls the `defineImplementation()` table API function to register each implementation when it is loaded into the program. Note that the constructors for the two libraries must have different names.

Your AVL tree implementation of the table API should be in a file called `avl.c`, and the hash table implementation in a file called `hashTable.c`. The shared libraries should be called `libavl.so` and `libhash.so`

Since the implementations of the table abstraction are stored in shared libraries, there are two ways they can be linked into the application. First, they can be loaded via the `-l<library>` option to the linker. Second, they can be loaded (like a plugin) at runtime using the `dlopen()` system call.

The interface routines (`table.c`) should not be in a shared library, but should be compiled into `table.o` and linked into each application that uses the table abstraction. Note that the routines in `table.o` should accommodate loading multiple libraries during a single execution of a program.

4. What You Need To Do:

- 1) Extract the AVL implementation from `intCount.c` and put into `avl.c`. *Refactor* the code so that it accepts any pointer type.
- 2) Write a few funcs, like the initialization routines, and the `visitTableItems()` call. The initialization routine should be specified via the attribute constructor method, and should perform all necessary initialization, including calling `defineImplementation()` in order to register the table implementation with the main program.
- 3) Turn `avl.c` into a library, `libavl.so`.
- 4) Write another implementation of the API, using chained hashing, from scratch. Build this into another library, `libhash.so`.
- 5) Re-write `intCount.c` so that it uses a shared library. The library should be linked via `dlopen()`, but which library is used is up to you. Update the makefile to test.
- 6) Write `table.c`, which keeps track of which implementations are registered, and vectors function calls for a table to the correct library implementation. `table.o` is linked into programs that wish to use one of our table libraries, not the libraries themselves.

Our tests will use your `intCount.c`, `table.c`, `avl.c`, and `hashTable.c`. All other files will be ignored.

Final notes:

First, the libraries have no way of knowing whether the incoming keys and values are allocated dynamically or statically, or even if the passed values are pointers at all. Therefore, *you should never free any passed keys or values*. You should just assume they are pointers to data that remains static over the course of the program.

Second, the environmental variable `LD_LIBRARY_PATH` must include “.” or dynamic libraries will not be found.