Assignment 5

Due Monday, November 30, 2015

Designing a Generic Symbol Table ADT

Write a C module **`symTable.c`** that implements a symbol table ADT that stores <key, value> pairs. All keys in the table are distinct and are assumed to be character strings. The value associated with a key is an arbitrary object defined by the user that is passed to the ADT via a void pointer. In other words, the ADT is generic.

The header file **`symTable.h`** accompanying this assignment gives the symbol table ADT interface. You should **`#include`** this file *as is* in your implementation module and client module (described later). In other words, you should not make any changes to **`symTable.h`**. The interface includes the following declarations:

- **`typedef struct SymTable *SymTable_T;`**
  `SymTable`e is a symbol table object pointed to by a pointer of type **`SymTable_T`**. Its structure should be defined in the implementation module. A **`SymTable`** object should *make copies* of <key, value> pairs inserted into it; these copies should be destroyed when the keys are deleted from the table or when the **`SymTable`** object itself is destroyed.

- **`SymTable_T symTable_create(void);`**
  Return a new **`SymTable`** object or **`NULL`** if the object could not be created due to insufficient memory.

- **`void symTable_destroy(SymTable_T symTable);`**
  Destroy **`symTable`** by freeing all the space allocated to it.

- **`int symTable_size(SymTable_T symTable);`**
  Return the number of <key, value> pairs stored in **`symTable`**.

- **`int symTable_insert(SymTable_T symTable, const char *key,`**
  `const void *value, size_t valuesize);`
  If **`key`** is not in the symbol table, insert it in the table and copy its value from the object pointed to by **`value`**. The size of the object is **`valuesize`**. Return 1 if the key is successfully inserted. Return 0 if key is already in the table or it is not in the table but it could not be inserted because of insufficient memory.

- **`int symTable_search(SymTable_T symTable, const char *key,`**
  `void *value, size_t valuesize);`
  Search for **`key`** in the symbol table. If it is in the table, copy its value to the object pointed to by **`value`** and return 1. If key is not in the table, return 0.

- **`int symTable_delete(SymTable_T symTable, const char *key);`**

Delete **`key`** and its value from the table and free the space allocated to these objects. Return 1 if **`key`** is in the table and was successfully deleted; otherwise, return 0.

Your implementation should employ a ***hash table*** that uses chaining to resolve collisions. That is, keys that hash to the same "bucket" should be stored as a singly linked list of nodes for that bucket. The hash table implementation that you developed for Assignment 4 should prove useful for this part of the assignment.

For uniformity, you should use the hash function given below. Additionally, you should use a default hash table size of 61.

```
#define DEFAULT_TABLE_SIZE 61
#define HASH_MULTIPLIER 65599

/* Hash key to a hash table bucket in the range [0, htsize-1] */

static unsigned int hash(const char *key, const int htsize)
{
   int i;
   unsigned int h = 0U;

   for (i = 0; key[i] != '\0'; i++)
      h = h * HASH_MULTIPLIER + (unsigned char) key[i];

   return h % htsize;
```

Accompanying this assignment is a client program **`symTable_client.c`** that creates multiple instances of the **`SymTable`** object to store keys having values of various types – character, integer, float, and a double array. The client program, when compiled with your implementation module, should produce the output shown in **`symTable_client.output`** (also accompanying this assignment).

Create a **makefile** to automatically build the executable for modules **`symTable.c`** and **`symTable_client.c`**. Your makefile should include rules to separately compile the source files to their respective object files, and to link the object files into a single executable file. Additionally, include a rule with a target named "**`clean`**" that will remove all object files of the build when the command "**`make clean`**" is issued.

Submitting your Assignment

Before submitting your assignment, test your program exhaustively. Apply the techniques that we discussed in Lecture 7 – Debugging and Testing. Here are some things you should do:

- Check function return values to determine the status of the function call (e.g., whether the call to **`malloc`** has succeeded).
- Use the **`assert`** facility to validate pre- and post-conditions (e.g., that a pointer is not **`NULL`** before referencing the object that it points to).

- Use **gdb** and/or debug macros to debug your program.
- Use **valgrind** to check for memory leaks.

Include a **README** file documenting your submission. In particular, describe your solution approach (e.g., in the event of a bucket collision, do you insert new nodes at the beginning or at the end of the linked list?) and your rationale for the approach. Additionally, describe whether your programs compile and run correctly, or produce compile-time and/or run-time errors. If the latter, describe your attempts to correct the errors, including whether you have narrowed down the source of errors. (I will take this into account when grading your assignment.) Finally, state any assumptions that you made in your programs and why you had to make them.

- Submit your assignment by uploading the following files to the sakai website. Write your full name clearly in comments at the top of your programs.

File	Description	Remark
symTable.h	header file for symbol table ADT interface	Provided
symTable.c	implementation module for symbol table ADT	your own
symTable_client.c	test module for **symTable.c**	Provided
makefile	makefile to build executable	your own
README.txt	README file documenting your submission	your own

- Be sure your code has no compilation errors. Compiler errors will prevent your submission from being graded.

- This assignment is due by 11:55 pm on Monday, November 30, 2015. **Absolutely no late assignments will be accepted.**