CS520 FALL 2019

LAB 5

Description

Implement a facility for creating, using and deleting symbol tables. The symbol tables should be implemented as hash tables that use separate chaining.

Follow these steps to complete the lab:

1. On agate create a subdirectory for this lab and copy the files in *~cs520/public/lab5* to the new subdirectory.
2. The file *symtab.h* describes the seven functions that you should implement for this lab. Copy this file into *symtab.c*, where you will implement the functions. Edit each function to put a statement block (matching curly brackets) with an appropriate return statement so that each function will compile and do nothing if called.
3. Put the following #include lines at the top of the file (after a comment describing the purpose of the file):

    - #include <stdlib.h>
    - #include <string.h>
    - #include "symtab.h"

4. The file *Makefile* describes how to build the test program using the files *main.c*, *symtab.h* and your *symtab.c*. To build the test program, simply type "make". Do this now to be sure your *symtab.c* will compile with your initial "stub" implementations.
5. Define a struct to be linked together at one level of the hash table. This struct needs to have a *char** member for the symbol, a *void** member for the data pointer, and a member to point to the next struct in the linked list. I recommend using a *typedef* to define a name for the struct type.
6. Define a struct to hold the control information for one symbol table. This struct needs to have a member to hold the length of the hash table and a member to point to the array of that length that contains pointers to the structs that are linked together. This latter member is a bit tricky, as it is a pointer to a pointer to the linked struct. Again I recommend using a *typedef* to define a name for the struct type.
7. Define a hash function, such as this one:

```
// modified FNV hash (see
http://en.wikipedia.org/wiki/Fowler_Noll_Vo_hash)
static unsigned int hash(const char *str)
{
  const unsigned int p = 16777619;
  unsigned int hash = 2166136261u;
  while (*str)
  {
    hash = (hash ^ *str) * p;
    str += 1;
  }
  hash += hash << 13;
  hash ^= hash >> 7;
  hash += hash << 3;
  hash ^= hash >> 17;
  hash += hash << 5;
  return hash;
}
```

Note the *u* suffix that makes the decimal constant 2166136261 unsigned.

8. Implement a helper function to do an internal lookup of a symbol in a given table, returning a pointer to the linked struct containing the symbol, if one is found. The function should be passed the control struct for a hash table and a *const char** pointer to the symbol. (The *const* keyword indicates that the function should not change the characters in the string representing the symbol.) The function should use the hash function to "hash" the symbol to an integer value and then compute the modulus of that integer and the table's length. The symbol is then searched for on the linked list at that level in the table. Use the *strcmp* function to compare symbol strings. (Be careful, *strcmp* returns zero when the strings are identical.) If the symbol is found, then return a pointer to the struct that contains the symbol. If the symbol is not found, then return NULL.
9. Use the *static* keyword to indicate that both the hash function and the lookup helper function should not be called from outside this file. (That is, they are not part of the public interface of this file.)
10. Implement *symtabCreate*.

    - Use *malloc* to allocate space for a control struct for the new symbol table. Use *sizeof* with the typedef name for the control struct to calculate the size in bytes of the struct, which is the value that should be passed to *malloc*.
    - Check the return value of *malloc* and, if NULL, return NULL.
    - Initialize the length member of the new member with the *sizeHint* value passed in to *symtabCreate*.
    - Use *malloc* to allocate a array of pointers to the linked struct. The size in bytes of this array is computed by multiplying the size of a pointer to a linked struct times the length member. Store the returned pointer in the second member of the control struct.
    - Check the return value of *malloc* and, if NULL, call *free* to deallocate memory for the control struct and then return NULL.
    - Return a pointer to the control struct.

11. Implement *symtabInstall*.

    - The *symtabHandle* parameter is actually a pointer to a control struct so assign it to a local variable of this type.
    - Call the lookup helper function to check if the symbol is already in the table.
    - If the symbol is found in the table, then simply update the data member of the struct returned by the lookup helper function.
    - If the symbol is not found, then use *malloc* to allocate a new linked struct. Also use *malloc* to allocate space for a copy of the new symbol. Use *strlen* to compute the length of the string representing the symbol. (When calling *malloc* don't forget to add one for the NULL byte on the end of C strings. *strlen* does not count this NULL byte but you need to allocate space for it.) Use *strcpy* to copy the symbol string to the newly allocated space. Store a pointer to the copy of the symbol in the symbol member of the new linked struct. Store the data value in the data member of the struct. Use the hash function applied to the symbol modulus the table length to compute the level in the table where the new linked struct should be inserted. It is probably easiest to link the new struct on the front of this list.
    - If a call to *malloc* fails, then return 0; otherwise return 1.

12. Implement *symtabLookup*. This can easily be done using the lookup helper function.
13. Define a struct to hold an iterator. This struct needs to have a member that points to the control struct for the hash table being iterated over, a unsigned integer member that is the index of the

level in the symbol table that contains the next (symbol,data) pair to be returned, and a pointer to the struct containing that (symbol,data) pair. Again I recommend using a *typedef* to define a name for the struct type.

14. Implement *symtabCreateIterator*.

    - The *symtabHandle* parameter is actually a pointer to a control struct so assign it to a local variable of this type.
    - Use *malloc* to allocate space for an iterator struct. If *malloc* returns NULL, then return NULL.
    - Save the *symtabHandle* parameter in the control struct pointer member of the new iterator struct.
    - Now search from the beginning of the table to find the first level that contains a symbol. Save the index of this level in the iterator struct. Save the address of the linked struct that contains the first symbol of this level in the iterator struct.
    - If there are no symbols in the table, then set the linked struct pointer member of the iterator struct to NULL.
    - Return a pointer to the iterator struct.

15. Implement *symtabNext*.

    - The *iteratorHandle* parameter is actually a pointer to a iterator struct so assign it to a local variable of this type
    - If the linked struct pointer member of the iterator struct is NULL, then return NULL.
    - Otherwise, initialize a local variable to contain the pointer currently stored in the symbol member of the linked struct pointed to by the iterator struct, return the data member of that linked struct through the *returnData* parameter, which is an output parameter (meaning it points to where you should store the data value), and then find the next symbol that should be returned by the iterator and update the iterator struct members accordingly. (Note that the index member might not change, if there is another symbol at the current level in the table that has not yet been returned.) If all symbols in the table have now been processed, then set the linked struct pointer member of the iterator struct to NULL. (Note that this would happen when your search for the next symbol to return hits the end of the table without finding a symbol.)
    - Return the symbol pointer that you stashed in a local variable.

16. Test your code using the provided *main.c*. There are some large text files available in *~cs520/public/books*. Here are the correct results for two of these files:

    alice.txt

```
o     little 128
o     gutenberg 93
o     project 87
o     herself 83
o     thought 74
o     turtle 59
o     hatter 56
o     gryphon 55
o     rabbit 51
o     looked 45
o     duchess 42
o     dormouse 40
o     before 40
o     without 34
o     nothing 34
o     things 33
```

- `looking 32`
- `moment 31`
- `should 29`
- `replied 29`

calais.txt

- `colonel 121`
- `should 103`
- `blackadder 95`
- `myself 95`
- `gutenberg 93`
- `project 88`
- `little 70`
- `before 68`
- `henriette 68`
- `thought 64`
- `without 60`
- `falfani 59`
- `station 57`
- `course 53`
- `nothing 52`
- `echelle 50`
- `through 48`
- `claire 48`
- `enough 48`
- `moment 46`

17. Implement *symtabDelete*. This requires freeing the strings for all symbols in the table, freeing all linked structs in the table, freeing the array of pointers to linked structs, and freeing the control struct. Note: do not call *free* for the data member of the linked structs.
18. Implement *symtabDeleteIterator*. This simply requires a call to *free* to deallocate the memory for the iterator struct.
19. Re-test your code to be sure the delete functions are working. Use the valgrind memcheck facility to validate that you properly free everything.
20. Read over my *main.c*. It illustrates some useful C features that you might not yet know, such as *isalpha*, *tolower* and the *printf %s* format for printing strings. It also illustrates some things you need to do in this lab, such as using *malloc*, *strlen* and *strcpy* to make copies of strings. It also illustrates the dubious technique of storing an integer value in a *void**.
21. After you complete the lab, submit your file **symtab.c** and nothing else.