CMPS 12B
Introduction to Data Structures
Programming Assignment 3

The goal of this project is to implement a Dictionary ADT in C based on a linked list data structure. A state in the Dictionary ADT is a finite set of pairs, the members of which are called *key* and *value* respectively. Thus the set $\mathcal{S}$ of states is

$$\mathcal{S} = \{ \text{finite sets of (key, value) pairs} \}$$

In general, the elements of a Dictionary can be pairs of any objects of any kind. In this implementation, a (key, value) pair will be a pair of C strings (i.e. pointers to null terminated char arrays.) Keys will be distinct, whereas values may be repeated. Thus any two (key, value) pairs must have different keys, but may possibly have identical values. You can think of a key as an account number, while a value is more like an account balance, both represented by strings. The five Dictionary ADT operations are:

1. $\text{size}(D)$:
   Return the number of (key, value) pairs in Dictionary D.
2. $\text{lookup}(D, k)$:
   If D contains a pair whose *key* matches argument k, then return the corresponding *value*, otherwise return NULL.
3. $\text{insert}(D, k, v)$:
   Insert the pair (k, v) into D.
   Pre: $\text{lookup}(D, k) ==$ NULL. (D does not contain a pair whose first member is k.)
4. $\text{delete}(D, k)$:
   Remove pair whose first member is the argument k from D.
   Pre: $\text{lookup}(D, k) \neq$ NULL. (D contains a pair whose first member is k.)
5. $\text{makeEmpty}(D)$:
   Reset D to the empty state, the empty set of pairs.

Implementation of the Dictionary ADT
The header file Dictionary.h, with prototypes for the above functions, is provided on the class webpage. This file also contains a typedef for the exported `Dictionary` reference type, and prototypes for a constructor `newDictionary()` and destructor `freeDictionary()`.

It also contains a prototype for another function called `DictionaryToString()`. This function will return a text representation of a Dictionary as a C string. Each (key, value) pair is represented by the string *key*, followed by a space, followed by the string *value*, followed by a newline character. The pairs will occur in the same order they were inserted into the Dictionary. The function will allocate a `char` array from heap memory of sufficient size to store the full string (with a terminating null `'\0'` character), then return a pointer to that array. It will be the responsibility of the client module to free this heap memory.

Your implementation of the Dictionary ADT will be contained in a file called Dictionary.c, and will utilize a linked list data structure. The linked list may be any of the variations discussed in class: singly linked with a head reference, optional tail reference, circular, doubly linked, with dummy Node(s), or any combination of these designs. It is recommended that you use the linked list implementation of the IntegerList ADT as a starting point for this project. Just rename the file IntegerList.c, and make the necessary changes. In particular, Dictionary.c should contain a private struct called `NodeObj`, and reference type `Node`. The struct `NodeObj` however, will not contain a data field of type `int`. Instead it should have

two fields of type `char*` respectively, to store a pair of C strings. You can give the fields any names you like, but I will refer to them by their obvious titles *key* and *value*.

It order to efficiently check preconditions and implement insert(), lookup() and delete(), it is highly recommended that you include a private helper function with prototype

```
Node findKey(Dictionary D, char* k)
```

This function should return a `Node` reference pointing to the Node object containing its argument *k* as *key*, or return `NULL` if no such Node exists. (See the function `find()` in the linked list implementation of the Integer List ADT for an analogous helper method.) Another useful helper function would be

```
int countChars(Dictionary D)
```

that counts the number of characters in the text representation of *D* described above. This can be used to calculate the length of the char array to be allocated by function DictionaryToString(). As always, helper functions can have any name you like. The above names are just suggestions.

Testing

Create another file called DictionaryTest.c whose purpose is to serve as a test client for the Dictionary ADT while it is under construction. The design philosophy we are promoting is that an ADT should be thoroughly tested in isolation before it is used in any application. Build your file Dictionary.c one function at a time, calling each operation from within DictionaryTest.c to wring out any bugs before going on to the next operation. The idea is to add functionality to the Dictionary in small bits, compiling and testing as you go. In this way, you are never very far from having something that compiles and runs. Errors that do arise are likely to be confined to recently written code. You will submit the file DictionaryTest.c with this project. It is expected that it will change significantly during your testing phases. As that happens, comment out the old code as you insert tests of more recently written operations. The final version of DictionaryTest.c should contain enough test code (possibly all in comments) to convince the grader that you did in fact test your Dictionary in isolation before proceeding.

The webpage link Examples/Programs/pa3/ contains the header Dictionary.h, three more test clients for the Dictionary ADT called DictionaryClient1.c, DictionaryClient2.c and DictionaryClient3.c respectively, as well as model output files for those clients. It also contains a Makefile for the project. If your Dictionary.c is written according to specs, then your output from those programs should match the model output exactly. For instance, after doing `make` at the command line, do

```
DictionaryClient2 > MyClient2-out
```

then

```
diff MyClient2-out DictionaryClient2-out
```

The result of the diff command should be no output at all, indicating no difference in the two files. Alter the Makefile as necessary to run the other test clients. If you are unfamiliar with the above commands, note that the Unix operator `>` redirects program output to a file, associating the data stream `stdout` with the file on its right hand side, instead of the console window. Similarly, the Unix operator `<` associates the file on its right hand side with the data stream `stdin`, instead of the keyboard. See the man pages or a good Unix reference for a description of the `diff` command. The Makefile also contains a `memcheck` target that invokes valgrind for detecting memory errors.

What to turn in

You may alter the provided Makefile to include submit and test utilities, or alter it in any other way you see fit, as long as it creates an executable binary file called DictionaryClient2, and includes `clean` and `memcheck` utilities. Do not alter the file Dictionary.h however, not even to insert the customary identifying comment block. Submit the following 5 files to pa3 before the due date.

README	table of contents, notes to grader
Dictionary.h	unchanged
Dictionary.c	created by you
DictionaryTest.c	created by you
Makefile	alter at your discretion

As always, start early and ask plenty of questions.