Name: ______________________________ NetID: ___________________________

Linked list traversal code travels though the existing list finding information about the list. Consider the listLength( ) function that counts the number of items in the list as found in lab7.c

1. Re-write the listLength( ) function so that it uses RECURSION instead of an iterative traversal.
   (Step 1: If the pointer to the list is NULL, return 0)
   (Step 2: Otherwise, recursively call with pointer->next.
   Add 1 to returned value. Return this updated value.)

```
int listLength (linked* hd) {
```

2. Write the RECURSIVE linked list function that would return TRUE if the linked list contains the integer value given by a second parameter. It should return FALSE if the list does not contain the integer value.
   (Step 1: If the pointer to the list is NULL, return FALSE)
   (Step 2: If the element at the pointer is the target, return TRUE)
   (Step 3: Otherwise, recursively call with pointer->next.
   Return the value returned by the recursive call.)

```
int findR ( linked* hd , int target ) {
```

3. How many lines of code are in the recursive function insertInOrderR( ) in lab7.c?
   (include the lines of code in the function newLinkedNode( ))

4. How many lines of code are in the iterative function insertInOrder2( ) in lab7.c?

Name: ___________________________________ NetID: _____________________________

5. Which function do you like better from lab7.c: insertInOrderR( ) or insertInOrder2( ) ? Why?

6. Explain what occurs when the –c flag is used with the gcc command.

7. Explain what occurs when the –o flag is used with the gcc command.

8. Explain what the **make utility** does when the following rule is invoked. Note that the answer should not be an explanation of what the LINUX commands do, but what make will do with the rule (i.e discuss what happens depending on the time stamps associated with the various files).

```
logMaster.txt: logTransactions.txt logError.txt
        date >> logMaster.txt
        cat logTransaction.txt logError.txt >> logMaster.txt
        rm logTransaction.txt logError.txt
        touch logTransaction.txt logError.txt
        touch logMaster.txt
```

9. The rest of this Lab Exercise is more on-line than previous ones and will require the submission of files via blackboard. You are to submit the five files for the program for this lab via the Assignments Page in Blackboard. The five files are the 3 .c files, 1 .h file and 1 makefile. If possible, please zip these files together in a single zip file for submission in Blackboard.

This lab uses the file: **bjOriginal.c**

For this assignment take the blackjack program what was written in a single source code file and break it into 3 source code files.

The first source code file must contain only the following functions:
- playHand()
- main()

The second source code file must only contain the following functions:
- printDeck ()
- shuffle ()
- initDeck ()

The third source code file must only contain the following functions:
- evaluateHand ()
- dealerBlackJack ()

The source code files may include any "non-function" code as needed.

You must also create a header file and a makefile for this assignment. You must submit all five of these files for grading. Missing any of these five files will impact the grade for this assignment.

The header file (.h file) should contain all #include statements, the function prototypes, and any struct and/or typedef statements. The job of the header file is to contain the information so the source code files can talk to each other. Please review the .h file in the example below.

Notes and Discussion on Makefiles and C Compilation

The makefile MUST seperately compile each source code file into a ".o" file and separately link the ".o" files together into an executable file. Review the makefile in the example below to see how this is done.

The command to create the .o file is:

```
gcc –c program1.c
```

The command to link the files program1.o, program2.o and program3.o into an executable file is:

```
gcc program1.o program2.o program3.o
```

The above command will just name the executable file using the default name, most often the –o option is given to provide a specific name for the executable file.

```
gcc program1.o program2.o program3.o –o program.exe
```

Example of Multiple Source Code Files

Consider the program contained in the following files:

- max1.c
- max2.c
- max.h
- makefile

This example shows how to set up this simplest of multiple source code file program. Note that max1.c and max2.c just contain functions and a #include of max.h. The file max.h contains the prototypes (or forward declarations) for all of the functions that are called from outside its source code file and any "globally" needed information.

The makefile is a special file that helps in the compilation of the source code into the object files into the executable file. A makefile is executed by the use of the **make** command. The syntax of a makefile can be strange. I have always found that it is easiest to modify an existing makefile rather than trying to create one from scratch. The makefile will contain multiple rules. Each rule has the following syntax:

```
target:  dependencyList
         commandLine
```

The multiple rules in the make file are separated by a blank line. Also note (this is VERY IMPORTANT) the commandLine must use a TAB for its indentation! An example of a rule is:

```
max1.o: max1.c max.h
	gcc -c max1.c
```

The **commandLine** is **gcc -c max1.c**, which will compile the source code file of max1.c into the object code file of max1.o.

The **target** in the above example is the file **max1.o**, which is also the name of the file created when the commandLine is executed. This relationship is what causes makefiles to work.

The **dependencyList** in the above example is the two files: **max1.c** and **max.h**, which are the files needed for the commandLine to properly run. Again, this relationship is what causes makefiles to work.

The make command uses the timestamps of the files in the target and the dependencyList. If any file in the dependencyList has a more recent timestamp than the target file, the commandLine is executed. The idea is that if the user has recently changed either **max1.c** or **max.h**, then the object file **max1.o** needs to be re-compiled. Make is designed to help the programmer keep track of what needs to be compiled next.

Make and makefile tutorials can be found at:

- http://mrbook.org/blog/tutorials/make/
- http://www.gnu.org/software/make/manual/make.html
- http://www.opussoftware.com/tutorial/TutMakefile.htm