CSc 352 (Fall 2025): Assignment 11

Due Date: 11:59PM Wed, Dec 10

The purpose of this assignment is to work with linked lists, memory allocation, command line arguments, reading from a file, using free(), representing graphs, and doing more complicated manipulation of pointers.

General Requirements

1. Your C code should adhere to the coding standards for this class as listed in the Documents section on the Resources tab for Piazza. This includes protecting against buffer overflows whenever you read strings.
2. Your programs should indicate if they executed without any problems via their *exit status*, i.e., the value returned by the program when it terminates:

Execution	**Exit Status**
Normal, no problems	**0**
Error or problem encountered	**1**

3. Under *bash* you can check the exit status of a command or program ***cmd*** by typing the command "**echo $?**" immediately after the execution of ***cmd***. A program can exit with status *n* by executing "`exit(n)`" anywhere in the program, or by having `main()` execute the statement "`return(n)`".
4. Remember your code will be graded on lectura using a grading script. You should test your code on lectura using the `diff` command to compare your output to that of the example executable.
5. Your code must show no errors when run with valgrind.
6. You must free all your allocated memory before your program exits.

Testing

Example executables of the programs will be made available. You should copy and run these programs on lectura to test your program's output and to answer questions you might have about how the program is supposed to operate. Our class has a home directory on lectura which is:

`/home/cs352/fall25`

You all have access to this directory. The example programs will always be in the appropriate **`assignments/assgN/probN`** subdirectory of this directory. They will have the same name as the assigned program with “ex” added to the start and the capitalization changed to maintain camelback. So, for example, if the assigned program is **`theBigProgram`**, then the example executable will be named **`exTheBigProgram`**. You should use the appropriate UNIX commands to copy these executables to your own directory.

Your programs will be graded by a script. This will include a timeout for all test cases. There must be a timeout or programs that don’t terminate will cause the grading script to never finish. This time out will never be less than 10 times the time it takes the example executable to complete with that test input and will usually be much longer than that. If your program takes an exceedingly long time to complete compared to the example code, you may want to think about how to clean up your implementation.

Makefiles

You will be required to include a Makefile with each program. Both commands:

`make`

`and`

`make` ***`progName`***

should create the executable file *progName*, where *progName* is the program name listed for the problem. The gcc commands in your Makefile that create the object files must include the -**`Wall`** flag. Other than that, the command may have any flags you desire.

Submission Instructions

Your solutions are to be turned in on the host **lectura.cs.arizona.edu**. Since the assignment will be graded by a script, it is important you have the directory structure and the names of the files exact. Remember that UNIX is case sensitive, so make sure the capitalization is also correct. For all our assignments the directory structure should be as follows: The root directory will be named **assg#**, where # is the number of the current assignment. Inside that directory should be a subdirectory for each problem. These directories will be named **prob#** where # is the number of the problem within the assignment. Inside these directories should be any files required by the problem descriptions.

To submit your solutions, go to the directory containing your assg11 directory and use the following command:

turnin cs352f25-assg11 **assg11**

prob1: bacon

Write a program called **`bacon`** which calculates the Bacon score (https://en.wikipedia.org/wiki/Six_Degrees_of_Kevin_Bacon) of various actors based on the information given in an input file. Kevin Bacon is a movie actor and the Bacon score for any movie actor is the number of movies it takes to connect Kevin Bacon with that actor. The definition is as follows:

- Kevin Bacon has a Bacon score of 0
- Any actor who was in a movie with Kevin Bacon has a Bacon score of 1
- For any actor X, if the lowest possible Bacon score of any actor X has been in a movie with is N, the X's Bacon score is N + 1

It may be hard to follow the outline above, but the concept is not that deep. Read through the wiki page linked to above if you are confused.

Basically, this is a graph problem. The actors are the vertices. The edges connecting the actors are movies. There is an edge between the vertices representing actors A and B if and only if A and B appeared in a movie together. An actor's Bacon score is then the smallest number of edges connecting the actor to Kevin Bacon.

- **Invocation:** Your program will be invoked with a required argument giving the file name of a text file containing movie names and cast lists and an optional flag to tell the program whether to print out just the Bacon score or to print out the score together with the path that generated it. The invocation will look like:

  ```
  bacon [-l] inFile
  ```

  where ***`inFile`*** is the name of a file which contains the movie list from which you will build your graph and –l (that's a lowercase L) is an optional flag. To make our program "UNIX like" we will allow the arguments to appear in any order and the option flag may be specified multiple times. Here are some examples of legal invocations:

  ```
  bacon myFile –l
  bacon –l –l myFile
  bacon myFile
  ```

  The following would be illegal invocations:

  ```
  bacon –l            //has no file specified
  bacon fileName fileName   //too many arguments
  bacon –ll myFile              //not a legal option
  ```

If the invocation is illegal, you should print an error message to **stderr** indicating how the program is used, and exit with a status of 1. If you are confused about what is legal and what is not, experiment with the example executable.

- **Movie File:** The movie file whose name is given as a command line argument will contain a list of movies and their cast lists. The format of the file is as follows:

```
Movie: <name of movie>
<actor 1>
<actor 2>
    .
    .
    .
<actor n>
Movie: <name of movie>
<actor 1>
   . . .
```

  where the actors listed (one per line) after the movie name are the actors in that movie.

  An example input file is in the subdirectory for this project of the class home directory on lectura. The file may contain blank lines (lines containing only white space) and these should be ignored. A single space will follow the keyword "Movie:". The name of the movie is considered to be the string starting at the character beyond that space and continuing until the end of the line but NOT including the '\n'. The actor's name is everything on the line except for the possible newline ('\n') at the end. To simplify the program, do not worry about trimming white space from the ends of the lines (other than the '\n') or capitalization. In other words the actor's "John Wayne", "John Wayne ", "john wayne", and "   John Wayne" can all be considered to be **different** by your program.

- **Behavior:** After reading the file of cast lists and creating your graph, your program will do the following:

  1. read in a line from **stdin** containing an actor's name
  2. compute the Bacon score for that actor (using a breadth first search)
  3. print the Bacon score (and the connecting actors and movies if –l option invoked) to **stdout** according to the format specified below.

  until no further input can be read.

- **Assumptions:** You can assume the following:

1. The movie file, if it exists, is in correct format. In other words you may assume the first nonblank line contains a movie name. You may also assume that if a line starts with "Movie: ", then the line continues with some name. Note that since blank lines are legal, and empty file is in fact a legal movie file.

- **Output:** Output should be printed to **stdout**. To print out the score, use the format

  ```
  printf("Score: %d\n", score)
  ```

  where score is the calculated Bacon score. If there is no path between the actor and Kevin Bacon, your program should print "`Score: No Bacon!`" on a single line.

  If the actor queried is not in the graph, print a message to stderr and continue reading queries. As always, whenever you print to stderr, when your program finally exits it should exit with a status of 1.

  The –l option

  If the –l option is specified when the program is invoked, your program should print the list of movies and actors connecting the queried actor to Kevin Bacon. The format for this output should be

  ```
  <Queried Actor>
  was in <Movie Name> with
  <Actor>
  was in <Movie Name> with
  <Actor>
  . . .
  was in <Movie Name> with
  Kevin Bacon
  ```

  Keeping track of this list is more difficult and only 10% of the test cases will involve this option. In other words you can still get 90/100 (90%) on the assignment even if you fail to implement the –l option (you must still recognize an invocation with the –l option as legal though), or 100/100 if you correctly implement the –l option. Also note that while the Bacon score is unique, the list printed out may not be. This means that if you are testing this option you may end up with a different output than the example executable and still be correct. Don't worry about getting the same list as the example as long as your list is correct. When I test this part of the program I will use test cases that have only one path to generate the Bacon score. You may want to create such test cases yourself. (The example movie file should do.)

- **Data Structures:**

  Once again you will use linked lists to represent the graph. The vertices of the graph will be the actors. The edges will be the movies. If you are implementing the –l option the edges will need to contain the movie name.

There are certainly variations on this. I used a linked list of actors. Each actor contains a linked list of movies (nodes which point to a movie). Each movie contains a linked list of actors (nodes pointing to an actor).

- **The Algorithm:**

A depth first search will not work here. A depth first search finds if there is a path between two vertices. We want to find the shortest path between two vertices. To accomplish this, we will use a breadth first search. A depth first search recursively searches all the children of each vertex. A breadth first search puts all the children on a queue. This way all the children are search before the grandchildren, etc. Here is the algorithm for a breadth first search:

```
BFS(Start, Target)
if Start == Target return 0
mark Start as queued
set level of Start to 0
add Start to the queue
while queue is not empty do
   Take A from top of queue
   For all children C of A do
      if C == Target
         return level of A + 1  //found, score A.level + 1
      if C not queued
         mark C as queued
         set level of C to level of A + 1
         add C to queue
return Target not found //If you get through loop without
                        //finding Target, there is no
                        //path from Start to Target
```

Note that in C you will have to implement this queue with a linked list. Don't forget to free it again after you're done using it.

Here's a hint for implementing the –l option. It requires you be able to trace the path from Start to Target. If some actor A is put on the queue when looking at the children of some actor B, the path to A comes from B. If your structure for nodes of the queue contains a link to this "parent" node, then you can following these links from the queue node for the Target back to the queue node for the Start. (i.e. It records your path.)

- **Error Conditions:**

    1. *Fatal errors*: Bad invocation of program; input file cannot be opened for reading.
    2. *Non-fatal errors*: Queried actor is not in the graph.