Programming Assignment I

1 Regulations

Due Date	: *Check Blackboard*
Late Submission Policy	: *The assignment can be submitted at most 2 days past the due date. Each late submission will be subjected to a 10 point per day penalty.*
Submission Method	: *The assignment will be submitted via Blackboard Learn/Gradescope*
Collaboration Policy	: *The assignment must be completed individually.*
Cheating Policy	: *Do not use code from sources except lecture slides. "Borrowing" code from sources such as friends, web sites, AI tools for any reason, including "to understand better" is condiered cheating. All parties involved in cheating get a 0 for the assignment and will be reported to the university.*

2 Goal of the assignment

The goal of this assignment is threefold.

- To enhance (or build) your C programming skills. Particularly, this assignment (and this course, as a whole) will focus on C programs that manage memory dynamically. Therefore, you will be exposed to various characteristics of C including, but not limited, to the definition and use of *structs* as well as functions, the use of pointers for dynamic memory management, and reading/writing data from/to files as well as the *stdin*.
- To help you understand the underlying time and space complexity for an elementary data structure such as arrays. This is intended to help you develop rudiments in the field of algorithmic complexity analysis.
- To prepare you for upcoming programming assignments where the primary goal will be to solve a real-life puzzle modeled as a computational problem. In those assignments, you will need to implement efficient data structures to achieve the desired goal while also implementing algorithms that make use of those data structures in a clever way.

Given all of that, we hope you realize that this assignment will play a crucial role in your success in this course.

3 Practical problem to be solved, in a nutshell

First, let's talk about what you will need to do at the application level. Then, we will dive into technical details.

In this assignment, you will be given a text file that contains certain commands. For example, take a look at the sample input text file below which contains one command per line.

insertToHead john brown 1.76 35
insertToTail jonathan green 1.68 27
printList
printListInfo
insertToPosition 1 carolyn fusch 1.72 21
findPosition carolyn
deleteFromPosition 2
deleteFromHead
deleteList

Your program will take the path to such a file from the *command line*, read it line by line, and execute the commands one at a time.

For example, consider the input text file above. By default, your program will start with an empty list. Then, you will first insert an entry to the list for *<john,brown,1.76,35>* where the elements of the entry signify name, lastname, height, and age, respectively. You, then, insert the entry *<jonathan,green,1.68,27>* but this time to the end of the list (i.e., append it to the end of the list). Then, you will print the contents of the list to the *stdout* (i.e., screen). The fourth line means that your program has to print the properties of the list. And so on ...

Further details for each command that you are expected to implement are provided below. Now that you have an idea about what you will be programming for this assignment, let's start talking about how you would actually do it.

4 Array-based implementation of lists

A **List** is an abstract data type that represents a finite number of ordered values. It can store duplicate values where each value will be considered a distinct item.

Lists are commonly implemented using either a linked list or an array. In this assignment, you will implement a list data structure using arrays.

4.1 What to store in the list:

NOTE: *You have been provided a starter code C file.*

Although one can implement lists to contain any structure, we have to be more specific. For the purposes of this assignment, you will implement a list that will have the following fields.

- **size:** An *int* variable, keeping a count of the number of data entries present in the list.
- **capacity:** An *int* variable, denoting the maximum number of data entries that can be stored in the list.
- **entries:** An array (I.e., a contiguous sequence of memory blocks) of data entries, where each entry stores the following information :
  - **name**: A *char* array to store name. Noe that since the C programming language does not have a built-in *"string"* data type, you need to store strings as *char* arrays. You may not assume that names can have a maximum length. You need to allocate sufficient memory for each input. See the *intializeEntry* function in the starter code for an example.
  - **lastname:** A *char* array to store the last name.
  - **height:** A *float* variable for the height attribute.
  - **age:** An *int* variable for the age attribute.

In C terminology, you will need to implement two *structs*: One that defines an *"entry"* with four fields (name, lastname, height, age), and a second that defines the *"list"* with three fields (size, capacity, entries).

NOTE: *In C, both of the following commands are valid ways to obtain int arrays.*

int intArray1[10]; // *statically allocates space for storing 10 integers*
int* intArray2 = malloc(sizeof(int)*10); // *dynamically allocates space for storing 10 integers*

Since you do not know how many entries will be in your list while programming (i.e., at compile time), you need to define your data structure using pointers and allocate space (and increase/decrease capacity) that would be sufficient for the input at runtime. In essence, this is the motivation for dynamic memory management.

4.2 Operations to be supported by the list:

You need to support several functions in your data structure that allow insertion, deletion, and search for data entries. The operations that you are expected to support are listed below.

- **Initialize a list:** Your program is expected to generate an empty list (i.e., size = 0) with capacity 2. These constraints are important.
- **Delete a list:** You will be expected to delete the list when the command *deleteList* is provided as a command in the input file. This operation should free all the memory allocated for and by the list, including the *char* arrays that are allocated for each entry, and the array of entries themselves. If an insert command follows a *deleteList* command, your program should run into a segmentation fault (which is the expected behavior in this case) since you have already freed all the memory.
- **Insert entries to list:** You need to be able to insert new entries into the list. Keeping in mind that each entry will consist of four fields (name, lastname, height, age), you could be asked to insert them to the beginning of the list, the end of the list, or to a certain location in the middle of the list. For more detail, see below.
  - *insertToHead* <name> <lastname> <height> <age>
  - *insertToTail* <name> <lastname> <height> <age>
  - *insertToPosition* <position> <name> <lastname> <height> <age>

  Note that, the location argument given to *insertToPosition* is indexed starting from 0 (i.e., inserting to location 0 is the same thing as inserting to the beginning of the list).

  When implementing these commands, you need to think about what each command is doing (see 4.1).

  First of all, you need to check whether the list has empty space available for inserting new entries. If not, you need to **increase the capacity by doubling** it. Keep in mind that capacity - size is the number of empty locations.

  Second, when implementing *insertToHead*, you need to shift all the entries (if the list is not empty) to right, before writing the contents of the new entry to the first slot of the list.

  Third, when implementing *insertToPosition*, you need to make sure the position argument lies in the range $[0, size]$. That is to say, if the list has capacity 4, and contains 2 elements at this point (i.e., $size = 2$), then a command of the form *<insertToPosition 3 ....>* should print an error message and do nothing. (You have been provided with a running executable of the expected program from where you can get the expected error messages to print).

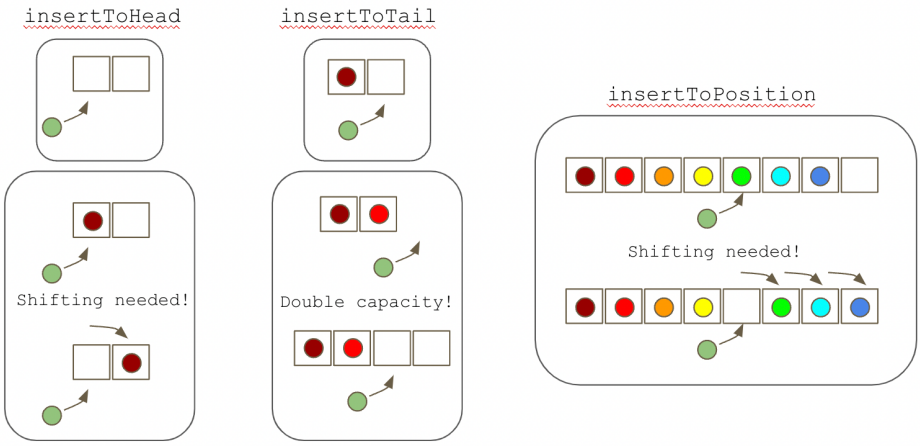

Figure 4.1: A schematic of insert functions. *insertToHead* and *insertToPosition* require shifting entries to the right if there exist entries to the right of the position we are inserting the new entry. All three insert functions might require increasing the size of the list (by doubling the capacity) if the list is already full (as exemplified in the figure in the case of *insertToTail*)

.

- **Delete entries from list:** Your program needs to be able to delete entries from the list. As in insertion, you will be asked to delete from the head, tail, or a certain location of the list. More details are provided below.
  - *deleteFromHead* <name> <lastname> <height> <age>
  - *deleteFromTail* <name> <lastname> <height> <age>
  - *deleteFromPosition* <position> <name> <lastname> <height> <age>

  Since deletion is the opposite of insertion (in some sense), your program needs to shift the elements to the left starting from the position to the right of the deleted element. Also keep in mind that, you will be expected to **reduce the capacity of the list by half** if it is less than half full.

- **Find the position of an element:** Your program needs to be able to search the entries by name (only name, not last name or anything else) and return the location of the entry in the list and -1 if the element is not in the list.
  - *findPosition* <name>

  You can assume that there will not be two entries with the same name.

- **Print contents of the list:** Your program should be able to print the entries of the list if the command *printList* is provided in the input file. The output should be printed in the following format, one line per existing entry.

  [<index>] <name> <lastname> <height> <age>

  Note that, the index starts from 0, height should be printed up to 2 digits after the decimal (which is achieved by a %0.2f parameter in the *printf* function), and each item is separated by a tab character (i.e., <name> and <lastname> variables are separated by a tab character).

  This function has been provided to you in the starter code.

- **Print information about the status of the list:** Your program should be able to print the size and capacity of the list if the command *printListInfo* is provided in the input file. The output should be printed in the following format (spaces are white space characters).

  size: <size>, capacity: <capacity>

Again, this function has been provided to you in the starter code.

- **Increase/Decrease capacity of the list:** Although explained above in insertion and deletion, it is worth repeating it again here. Your program should (1) double the capacity of the list if the size equals the capacity of the list and an insertion is attempted, and (2) halve its capacity if the size becomes less than half of the capacity. Although there will not be any explicit commands in the input file for increasing/decreasing the size of the list, you should be handling it via the insert/delete functions. We will check whether you maintain this property by printing the status of the list using *printListInfo* command.

Along with this PDF file and the starter C code, you will be given a sample input file and its expected output.

Also, you can check the expected behavior of the program by playing around with the executable called *"list"* provided to you in the starter package. Copy the executable to your local home folder and run it as *"./list input.txt"*, assuming that your input file is named as *input.txt*. Make sure to compare the output you get from your program with that you obtain from this executable before submission as we will use this implementation when grading your assignment.

5 Starter code

You are provided with a C file that reads the input file and contains the function declarations that you need to implement as well as 4 function definitions. Note that, it is possible to implement the functionalities described in this PDF with a different set of functions. You are welcome to do so, as long as your output matches the expectations described in this document (and matches what the provided executable is producing).

6 Submission

- Even if you develop it elsewhere, your code must run on **tux**. So, make sure that it compiles and runs on tux prior to submitting it.
- Your **program** should take a single argument from the command line, which will be the path to the input file containing commands for manipulating the list (as explained above) and printing the output to standard output.
- Your submission will consist of two separate files (do not compress/zip the files).
  1. A single C file named **main.c**, which includes your code for the assignment.
  2. A single filed named **self_grade**, without any file extension, which consists of two lines, such that:
     - first line consists of a single number in [0,30] range to indicate your self assessment for your effort/performance for the assignment.
     - second line is a short description of how much effort you put into the assignment, such as when you started, how many hours it took in total, whether you attended office hours etc.
- Submit your assignment by following the Gradescope link for the assignment through Blackboard Learn.

7 Grading

- Assignment will be graded out of 100 points.
- You will provide your self-assessment score for your effort as a 30-point portion of the assignment.
- The remaining 70 points will be awarded according to how many test cases your program is able to pass.