Department of Computer Science

CS260
Data Structures

Winter 2022

Programming Assignment 1

1 Regulations

Due Date : 01/12/2021 - 11:59pm
Late Submission: 10% off for each late day, with at most 2 day late submission
Submission : via Gradescope.
Team : The homework is to be done and turned in individually. No teaming up.
Cheating : All parties involved in cheating get zero from assignment and will be reported to the University.

2 Goal of the assignment

The goal of this assignment is threefold:

- to refresh your coding skills in C, including but not limited to: definition and muse of structs as well as functions, use of pointers to do dynamic memory management, reading/writing data through files as well as standard input, compiling code using makefiles
- to make you realize the underlying space and time complexity for a data structure as simple as array based lists, thus motivating you to start thinking about algorithmic complexity
- to warm you up for the upcoming programming assignments, in which you will be asked to write a program that solves a given real life problem/application. In those assignments, you will need to implement the most efficient data structures to achieve the desired goal while also implementing an algorithm that solves the problem over those data structures.

Thus, this assignment has a crucial role for your overall success in this course as your better performance in this assignment will give a you a head start for the rest of the programming assignments throughout the term.

3 Practical problem to be solved, in a nutshell

First, let's talk about what you will need to do at application level. We will then dive into technical details. In this assignments, you will be given a text file, that contains certain commands. Example explains the phenomenon best, so look at the sample input file below, which contains a command at each 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 command line, read it line by line, and execute the commands one at a time. On this sample input, for example, by default you will start with an empty list, and then you will first insert an entry to the list for <john,brown,1.76,35> where elements of the entry signifies name, lastname, height, and age, respectively. You then insert the entry for jonathan, 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 standard output (i.e., screen). The fourth line is asking you to print the properties of the list. And so on... Further detail for each command that you are expected to implement is provided below. Now that you have an idea about what you will be doing, let's start talking about how to do it. (Note that, you are provided a starter code along with this pdf document)

4 Array based implementation of lists

A list is an abstract data type that represents a finite number of ordered values. A list can contain the same value more than once, where each entry 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:

Although one can implement the lists to contain any structure, for the purposes of this assignment, you will implement a list that will have the following fields:

- **size:** An integer variable, keeping a count of number of data entries present in the list.
- **capacity:** An integer variable, denoting the maximum number of data entries that can be stored in the list.
- **entries:** An array (I.e., contagious set of memory locations) of data entries, where each entry stores the following information:
  - **name**: A char array to store name. (Note: Since C does not have a built in "string" data type, you need to store strings as char arrays!Do not assume that names can have a maximum length!! You need to allocate sufficient memory for each input. No more, no less!!)
  - **lastname:** A string to store last name. Same idea as the variable "name" above.
  - **height:** A floating point variable to keep height.
  - **age:** An integer variable to keep the age.

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

IMPORTANT NOTE: In C, the following two are equally valid ways of obtaining an integer array:

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 coding (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 provided by the user at runtime. In a nutshell, 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 allows inserting/deleting/locating data entries. The operations that you are expected to support are listed below, along with their explanation:

- **initialize a list:** Although there will not be an explicit command for initializing a list, you are expected to generate an empty list (i.e., size=0) with capacity 2. (These constraints are important!!) when the program is run. Thus, you should have internal function(s) written to maintain this function.
- **delete a list:** You will be expected to delete the list when the command *deleteList* is given in the input file. This operation should free all the memory allocated in the list, including the char arrays that are allocated for each entry, and the array of entries themselves. If another insert command comes after a *deleteList* command, your program would give 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 will be asked to insert them to beginning of the list, end of the list, or to a certain location in the list, with the specifics detailed below:
  - *insertToHead* <name> <lastname> <height> <age>
  - *insertToTail* <name> <lastname> <height> <age>
  - *insertToPosition* <position> <name> <lastname> <height> <age>

  Note that, location to insert the entry is indexed starting from 0 (i.e., inserting to location 0 means 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 it is full, then you need to **increase the capacity by doubling** it. (remember: capacity-size is the number of empty locations). **Second,** when implementing *insertToHead*, keep in mind that, you need to shift all the entries (of course, 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 location to be inserted is within the already filled part of the list. 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 sorts <insertToPosition 3 ....> should print an error message and not do anything. (You will be provided with a running executable of the expected program, from where you can get the expected error messages to print)

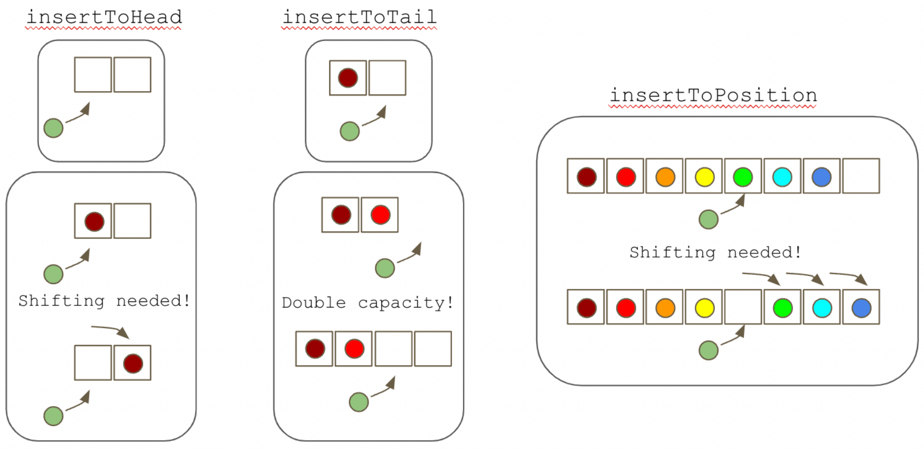

Figure 4.1: A schematic of insert functions. *insertToHead* and *insertToPosition* would require shifting entries to right, if there exists an entry to their right. All three insert functions might require increasing the size of the list (by doubling the capacity) if the list is already full (as examplified in the figure in the case of *insertToTail*)

.

- **delete entries from list:** You need to be able to delete entries from the list. As in inserting, you will be asked to delete from head, tail, or a certain location of the list. Specifics are provided below (while avoiding unnecessary repeat, as the logic is similar to insertion):
  - *deleteFromHead* <name> <lastname> <height> <age>
  - *deleteFromTail* <name> <lastname> <height> <age>
  - *deleteFromPosition* <position> <name> <lastname> <height> <age>

  Being the opposite of insertion, this time you need to make sure that you shift the elements 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 should search the entries by name (only name, not last name or anything else), and return the location of the entry if found in the list, and -1 if not found. The command for this purpose would be as follows:
  - *findPosition* <name>

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

- **print contents of the list:** Your program should 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 (i.e., you should only orint **size** number of lines, not *capacity*):

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

  Note that, index starts from 0, height should be printed up to 2 digits after the decimal (which can be achieved by %0.2f parameter in the printf function).

- **print information about the status of the list:** Your program should 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:

  size: <size>, capacity: <capacity>

- **increase/decrease capacity of the list:** Although explained above in insertion and deletion, it is worth repeating it again here. Your program should be doubling the capacity of the list if the size becomes equal to capacity, and halving 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 size of the list, you should be handling it internally as insert/delete commands come. We will check whether you maintain this property by printing the status of the list using *printListInfo* command.

Along with this pdf file, 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 provided in the started code 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 (or put whatever you name it here). Make sure to compare the output you get from your program with that you obtain from this test program before submission, as we will use this implementation when grading your assignment.

5 Starter code

You are provided with a starter code that reads the input file, and also contains the function declarations that you need to implement. 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). However, if you are not feeling that much self driven, then go ahead and build up your code on top of the provided starter code. You are also provided with a makefile that compiles the code. Make sure you know how it works (as you have already learned how it works in CS265 anyways) and don't forget to put it in the submission package.

6 Submission

- Your code must be running on tux. So, make sure that it compiles and runs on tux before you submit, even if you develop it elsewhere.
- Your submission package **must include** your **source code**, a **makefile**. Put all these material in a folder named with your userid, (example: abc123), compress it into a zip file named as your student id (example: abc123.zip). Submit this zip file through Gradescope.
- Your makefile needs to produce an executable named "list", once we type "make" in command line.
- Your **program** should take a single argument from 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.

7 Grading

- Assignment is going to be graded out of 100 points.
- You will provide your self assessment for your effort for the 50 point portion of the assignment. You will find a rubric in gradescope to give you an idea about how you can judge your self assessment.
- The remaining 50 points will be for the test cases of the assignment.