Middle East Technical University | Department of Computer Engineering

CENG 140

C Programming

Fall' 2021-2022
Take-Home Exam 2

Emre Külah
kulah@ceng.metu.edu.tr
Due date: Jan 10, 2022, Monday, 23:59

NAMEBOOK

1 Overview

In this assignment, our goal is storing **name codes** in a 2D dynamic array structure. In a nutshell, a name code C is a number which is obtained by applying a transformation function to a person name. To store these name codes, we will create a structure as follows. There will be an **IndexArray** of size N, where each element is a pointer to a **data array** of integers. Given a name code C, we will first compute its index I (as C modulo N), and then store C in the data array pointed by the pointer at the corresponding element in the index array, i.e., **IndexArray**[**I**].

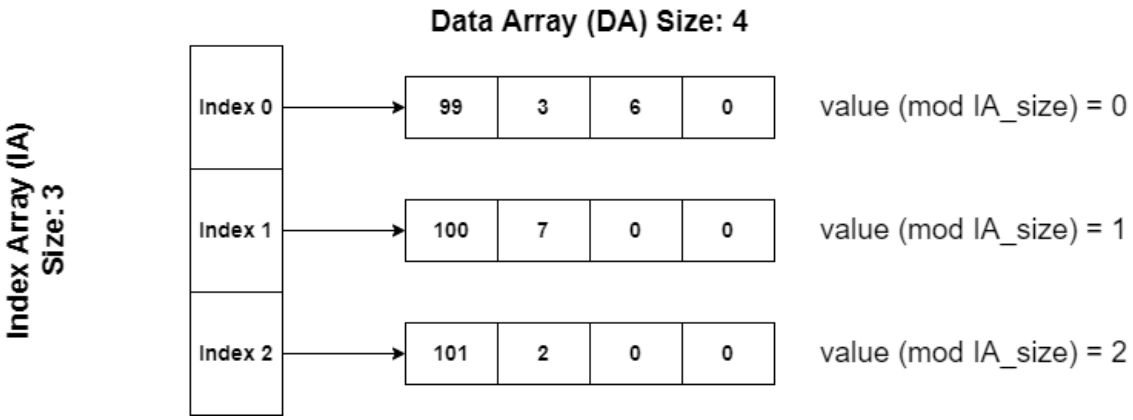

Figure 1: 2D dynamic array representation. Zeros represents empty cells.

In this dynamic 2D structure, each element in the IndexArray stores a pointer to an integer array, which is called a Data Array. Therefore, you should first allocate an IndexArray of size N, and then, for each of its elements, you should allocate a data array of size M. In Fig. 1, initial sizes of IndexArray and Data Arrays are selected as 3 and 4, respectively

Note: Data array sizes are not fixed, so do not think it as a 3x4 array.

Each person name in a char array format will be inserted into our 2D array by following steps:

- Transform name to a value called **name code** using a transform function. The transform function returns an integer for given char array. Transform operation will be described later.
- Obtained name code will be used to find the index in the IndexArray. The index is found by taking the modulo of the obtained name code according to N (size of IndexArray). For example: $376 \equiv 1 \pmod 3$, so the name code will be inserted to the data array pointed by the element (at index 1) of IndexArray.
- The name code will be inserted at the end of the data array of the found index. For example: 376 will be placed into the first empty cell in the data array of index 1 (Data array of index 1 will be $100 \rightarrow 7 \rightarrow 376 \rightarrow 0$)
- Zeros represents empty cells in chains.

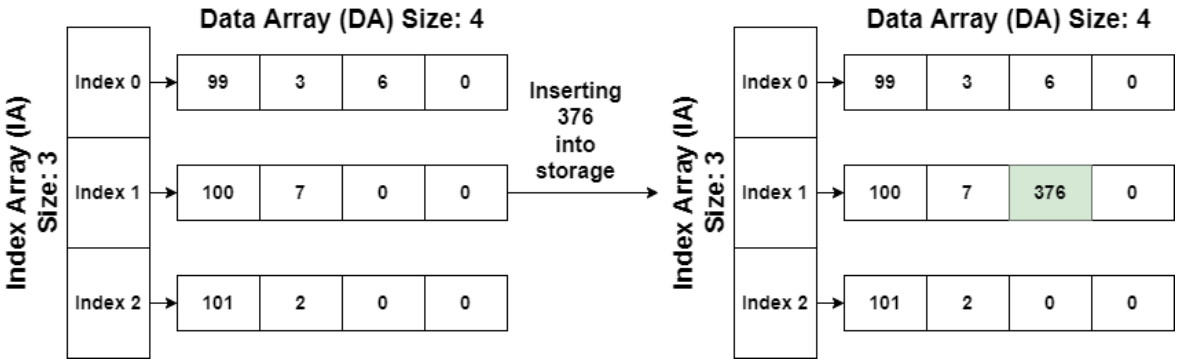

Figure 2: Inserting number 376

1.1 Transform Function

Transform function takes a person name in a char array format and generates a number for given name.

For a name of size n:

$$x = \sum_{i=0}^{n-1} (i+1)^2 \times name[i] \text{ (ASCII code of char)} \tag{1}$$

1.1.1 Example

Name: "John"

- J $\rightarrow 1^2 \times 74 = 74$
- o $\rightarrow 2^2 \times 111 = 444$
- h $\rightarrow 3^2 \times 104 = 936$
- n $\rightarrow 4^2 \times 110 = 1760$

The transformation will be $74 + 444 + 936 + 1760 = 3214$. While inserting this name code into our 2D array, we will take the number modulo 3 and the number will be inserted at the end of the index 1 ($3214 \equiv 1 \pmod 3$).

2 Use Case

We will insert 3 names "John", "Bob", "Alice" respectively into an empty 2D dynamic array.

- John $\rightarrow 3214 \rightarrow 3214 \equiv 1 \pmod 3 \rightarrow$ Insert into index 1
- Bob $\rightarrow 1392 \rightarrow 1392 \equiv 0 \pmod 3 \rightarrow$ Insert into index 0
- John $\rightarrow 3214 \rightarrow 5551 \equiv 1 \pmod 3 \rightarrow$ Insert into index 1

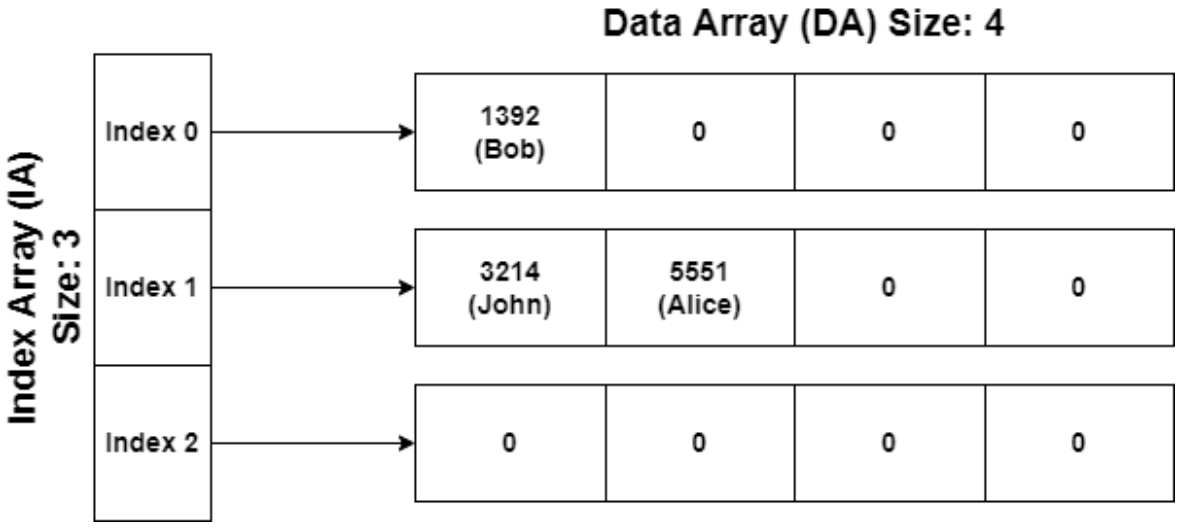

Figure 3: Inserting 3 person names into our 2D array

2.1 Data Array Extension

After adding a few names to the storage, some data arrays will be full. When adding a new name, if the data array it will be added to is full, that data array will be doubled in size.

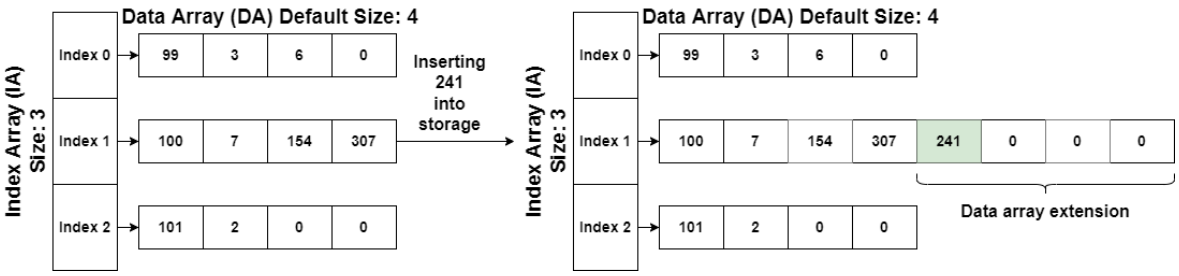

Figure 4: Data array extension

2.2 Index Array Extension

While searching an item in the storage, the time complexity of the operation depends on the average number of items in the data arrays. To reduce this complexity if the load factor of the storage exceeds a threshold, we extend the index array with a new size.

In this assignment, you are going to extend the IndexArray under following situation. If the total sizes of the data arrays is greater than or equal to 1.5 times the product of the size of the IndexArray and the default data array size, after a newly added name, we will initialize a new IndexArray twice the size of the old one. For each index of the new IndexArray, new data arrays will be allocated from scratch with default data array size (i.e. It is 4 for given example in the figure).

$$\sum_{i=0}^{C_c-1} |data\_array_i| \geq (C_c \times M) \times 1.5 \tag{2}$$

where C_c is the current size of the IndexArray and M is the default data array size.
Steps of index array extension:

- Create a new index array with new index array size $C_c \times 2$.
- Starting from the first row of the old index array add each name code into new 2D dynamic array properly.
- Free all old data arrays and old index array.

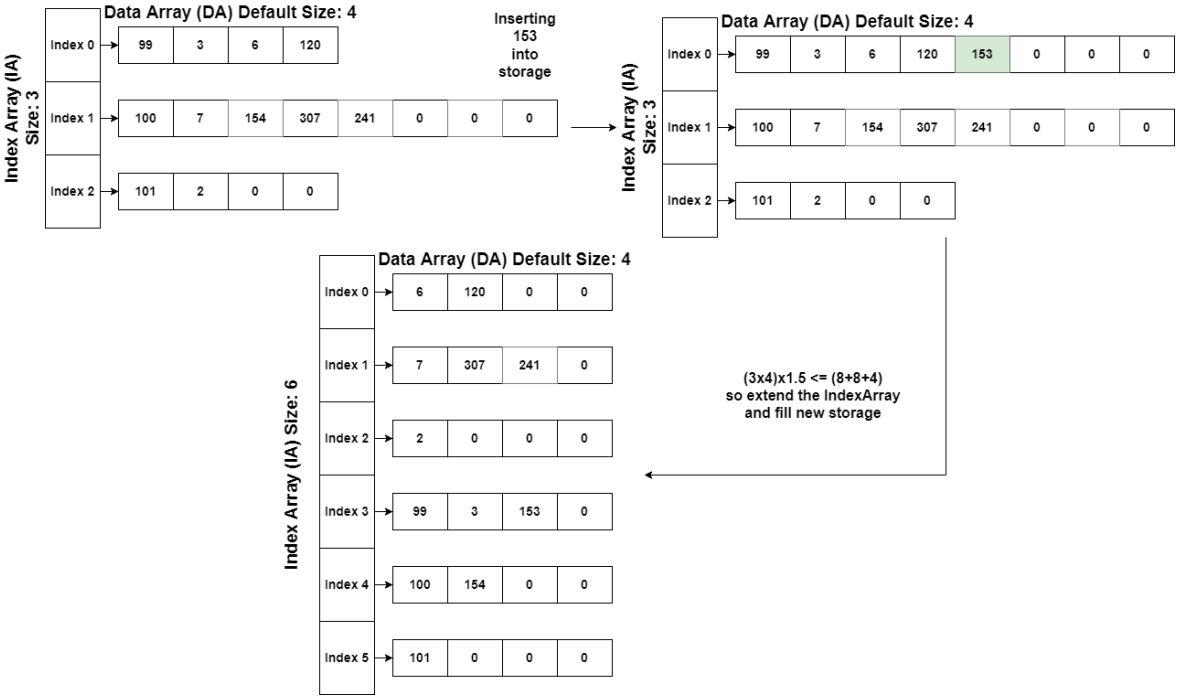

Figure 5: Index array extension

3 Tasks

In this assignment, we will create and initialize 2D dynamic arrays, insert new name codes and check whether given 2D dynamic array contains a name code or not. There will be several helper functions as well. To be able to implement these functions, we need some variables keeping some statistics of the 2D dynamic array.

- **Index array size**: This variable is kept in the main function. It is updated when the index array extends.
- **Default data array size**: This variable is kept in the main function. It keeps the default data array size.
- **Data array sizes**: An array of the same size as IndexArray size. It keeps the data array size of each index. At first they are all initialized as given default data array size value. Then, when the data arrays are full, this array is also updated according to the changed size.

Following abbreviations will be used in the code template:

- IA $\rightarrow$ *IndexArray*
- DA $\rightarrow$ *DataArray*

You are going to implement 7 functions.

- **void print_storage(int** storage, int IA_size, int* DA_sizes)**: function gets a 2D dynamic array, IndexArray size and the array keeping data array sizes. It prints the content of the storage in the following format:

```
0 -> 99 0 0 0
1 -> 100 7 154 307 241 0 0 0
2 -> 101 0 0 0
```

  Note: There will be **NO** extra whitespace at the end of the rows but there will be an extra newline character at the end.

- **int** initialize_storage(int IA_size, int DA_size, int** DA_sizes)**: function gets IndexArray size, default data array size and reference to a pointer which will keep data array sizes.
  Steps:
  - **storage:** Allocate a 2D array with given IndexArray size and data array size.
  - **DA_sizes:** Allocate an array of the same size as IndexArray size and initialize each element with default data array size. Assign it to the value of DA_sizes variable to be able to access data array sizes in main function.
- **int transform_name(char* name)**: function gets a person name in a char array format. It generates a name code for given person name using given formula in Section 1.1.
- **void insert(int** storage, int IA_size, char* name, int* DA_sizes)**: function takes a 2D dynamic array, IndexArray size, a person name and DA_sizes array. It first calls transform function to find corresponding name code. It checks whether the data array of the index to which name code should be inserted is full. If it is full, first the data array is extended, the DA_sizes array is updated, then the insertion is done.
  Steps:
  - Transform person name into a name code.
  - Find corresponding index by taking modulo.
  - Check if corresponding data array is full or not. If the data array is full, double the data array size.
  - Insert name code at the end of the data array.
- **void resize_IA(int*** storage, int* IA_size, int DA_size, int** DA_sizes)**: This function will be called after each insertion operation. It takes storage, IndexArray size, default data array size and DA_sizes array. First of all, it checks if the extension rule is violated. If it is violated, then it initialize a new IndexArray with doubled size and a new DA_sizes array with the same size as new IndexArray. You can use **initialize_storage** function here. Then it transfers the content of the old storage to the new one. You can use **fill_new_storage** function here. At the end, it updates IA_size in the main function and frees old data arrays, old IndexArray and old DA_sizes array.
- **void fill_new_storage(int** storage, int IA_size, int** new_storage, int* new_DA_sizes)**: function takes a storage, old IndexArray size, a new storage and new chain_sizes array. This function transfers all the content of the old storage to the new one. While doing that, it updates new DA_sizes array. **insert2** function (defined below) will be used in this function. You can directly insert name codes into the new storage with **insert2** function.
  Since insert functions can update DA_sizes array's content, this function will be very simple.
- **void insert2(int** storage, int IA_size, int name_code, int* DA_sizes)**: function takes storage, IndexArray size, name code and DA_sizes array. The only difference than the first insert function is that it gets code name directly instead of the person name as a char array.

4 Example

I added some extra prints to the example to give details about the steps. Do not put that kind of prints in your submissions.

Let's add 10 names into a newly initialized bucket: James, Sophia, Evelyn, Michael, Emily, David, April, Joseph, Eva, Jesse. Please trace that output well.

We start with the example main function:

```
#include <stdio.h>
#include "lab2.h"

int main(){
    char names[][10] = {"John", "Sophia", "Evelyn", "Aaron", "Emily",
                        "Russell", "April", "Patrick", "Eva", "Jesse"};  /* names to be added into storage */
    int i, j;           /* iterators */
    int IA_size = 3;    /* initial IndexArray size */
    int DA_size = 3;    /* default data array size */
    int** storage;      /* storage variable */
    int* DA_sizes;      /* data array sizes */

    storage = initialize_storage(IA_size, DA_size, &DA_sizes);  /* initialize the storage */

    for (i = 0; i < 10; i++)  /* for each person name in the list */
    {
        insert(storage, IA_size, names[i], DA_sizes);       /* insert a person name into storage */
        print_storage(storage, IA_size, DA_sizes);          /* print storage */

        /* Check if need IndexArray extension. */
        /* Sent references of storage, current IndexArray size and data array sizes */
        /* to not lost their references while updating them in the implemented functions */
        resize_IA(&storage, &IA_size, DA_size, &DA_sizes);
    }
    return 0;
}
```

```
Inserting John: 3214
0 - 0 0 0
1 - 3214 0 0
2 - 0 0 0
IA size: 3
Data array sizes: 3 3 3  --> Capacity 9
Extension rule: 3x3x1.5 = 13.50 <= 9 (capacity) --> 0
---------------------------
Inserting Sophia: 9316
0 - 0 0 0
1 - 3214 9316 0
2 - 0 0 0
IA size: 3
Data array sizes: 3 3 3  --> Capacity 9
Extension rule: 3x3x1.5 = 13.50 <= 9 (capacity) --> 0
---------------------------
Inserting Evelyn: 10163
0 - 0 0 0
1 - 3214 9316 0
2 - 10163 0 0
IA size: 3
Data array sizes: 3 3 3  --> Capacity 9
Extension rule: 3x3x1.5 = 13.50 <= 9 (capacity) --> 0
---------------------------
Inserting Aaron: 6005
0 - 0 0 0
1 - 3214 9316 0
2 - 10163 6005 0
IA size: 3
Data array sizes: 3 3 3  --> Capacity 9
Extension rule: 3x3x1.5 = 13.50 <= 9 (capacity) --> 0
---------------------------
Inserting Emily: 6203
0 - 0 0 0
```

```
1 - 3214 9316 0
2 - 10163 6005 6203
IA size: 3
Data array sizes: 3 3 3  --> Capacity 9
Extension rule: 3x3x1.5 = 13.50 <= 9 (capacity) --> 0
---------------------------
Inserting Russell: 15130
0 - 0 0 0
1 - 3214 9316 15130
2 - 10163 6005 6203
IA size: 3
Data array sizes: 3 3 3  --> Capacity 9
Extension rule: 3x3x1.5 = 13.50 <= 9 (capacity) --> 0
---------------------------
Inserting April: 5919
0 - 5919 0 0
1 - 3214 9316 15130
2 - 10163 6005 6203
IA size: 3
Data array sizes: 3 3 3  --> Capacity 9
Extension rule: 3x3x1.5 = 13.50 <= 9 (capacity) --> 0
---------------------------
Inserting Patrick: 14768
0 - 5919 0 0
1 - 3214 9316 15130
2 - 10163 6005 6203 14768 0 0
IA size: 3
Data array sizes: 3 3 6  --> Capacity 12
Extension rule: 3x3x1.5 = 13.50 <= 12 (capacity) --> 0
---------------------------
Inserting Eva: 1414
0 - 5919 0 0
1 - 3214 9316 15130 1414 0 0
2 - 10163 6005 6203 14768 0 0
IA size: 3
Data array sizes: 3 6 6  --> Capacity 15
Extension rule: 3x3x1.5 = 13.50 <= 15 (capacity) --> 1
########## EXTENDING STORAGE ##########
---------------------------
Inserting Jesse: 5878
0 - 0 0 0
1 - 0 0 0
2 - 14768 0 0
3 - 5919 0 0
4 - 3214 9316 15130 1414 5878 0
5 - 10163 6005 6203
IA size: 6
Data array sizes: 3 3 3 3 6 3  --> Capacity 21
Extension rule: 6x3x1.5 = 27.00 <= 21 (capacity) --> 0
---------------------------
```

5 Specifications

- You can initialize data array values as zero directly. Empty string will not be added into the storage. So zeros will be empty cells.
- In print_bucket function, there will be **NO** extra whitespace at the end of the rows but there will be an extra newline character at the end.
- Do not forget to free old storage and DA_sizes array after extending the storage.
- You do not need math.h to calculate square in transform function. You can multiply the index by itself.
- Horizontal dashed separators printed in outputs of tests in VPL are printed in test main functions, do not add your own prints for that separators.
- **initialize_storage** and **print_storage** functions are base functions to be able to get a non zero grade (test1 and test2). Implement these 2 functions first.

6 Regulations

- **Programming Language:** C

- **Libraries and Language Elements:**
  You should not use any library other than *"stdio.h"*, *"stdlib.h"*. You can use conditional clauses (switch/if/else if/else), loops (for/while), allocation methods (malloc, calloc, realloc). **You can NOT use any further elements beyond that (this is for students who repeat the course).** You can define your own helper functions.
- **Compiling and running:**
  DO NOT FORGET! YOU WILL USE ANSI-C STANDARTS. You should be able to compile your codes and run your program with given **Makefile**:

  ```
  >_ make the2
  >_ ./the
  ```

  You can run test cases using **Makefile** as well. First, you need to copy the2.h file to the tests folder and run following commands:

  ```
  >_ make testN
  >_ ./the
  ```

  testN can be test1, ..., test8.
- **Submission:**
  You will use OdtuClass system for the homework just like Lab Exams. You can use the system as an editor or work locally and upload the source files. Late submission IS NOT allowed, it is not possible to extend the deadline and **please do not ask for any deadline extensions**.
- **Evaluation:** Your codes will be evaluated based on several input files including, but not limited to the test cases given to you as an example. You can check your grade with sample test cases via CengClass system but do not forget it is not your final grade. Your output must give the exact output of the expected outputs. It is your responsibility to check the correctness of the output with the invisible characters. Otherwise, you can not get a grade from that case. If your program gives correct outputs for all cases, you will get 100 points.
- **Cheating: We have zero-tolerance policy for cheating**. People involved in cheating will be punished according to the university regulations and will get 0. Sharing code between each other or using third party code is strictly forbidden. Even if you take a "part" of the code from somewhere/somebody else - this is also cheating. Please be aware that there are "very advanced tools" that detect if two codes are similar. So please do not think you can get away with by changing a code obtained from another source.