Middle East Technical University | Department of Computer Engineering

CENG 140

C Programming

Spring' 2023-2024
Take-Home Exam 2

Due date: May 20, 2024, Monday, 23:59

ARRAYLINK

DYNAMIC ARRAYS

1 Overview

In this assignment, the goal is to store **name codes** in a 2D dynamic array structure.

In a nutshell, a name code C is a number that is obtained by applying a transformation function to a person's name. To store these name codes, we will create the structure as follows:

1. There will be an **IndexArray** of size N, where each element is a pointer to a **DataArray** of integers.
2. Given a name code C, we first calculate its index I (as C modulo N).
3. Then, store C in the data array pointed by the pointer at the corresponding element in the IndexArray, i.e., **IndexArray[I]**.

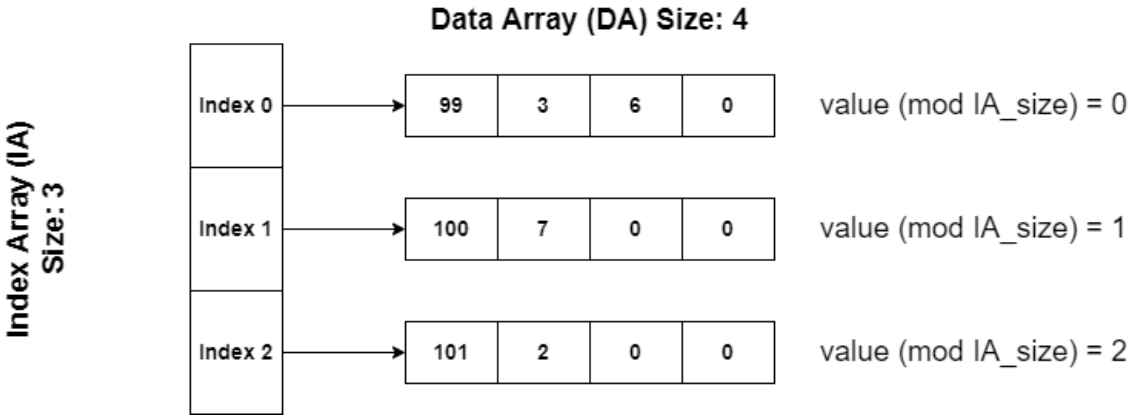

Figure 1: 2D dynamic array representation with initial sizes of 3 and 4 for IndexArray and DataArray, respectively. Zeros represents empty cells.

In this dynamic 2D array structure, each element in the IndexArray stores a pointer to an integer array, which is called DataArray. The 2D dynamic array structure will be called the **Storage**.

In order to initialize Storage, first you should allocate an IndexArray of size N and then, for each of its elements, you should allocate a DataArray of size M.

1.1 Storing Name Codes

Each `name` in a `char array` format will be inserted into the Storage, a 2D dynamic array structure, with the following steps:

- Transform each `name` into a value called a `name code` using a transformation function. This function returns an integer for a given character array. Details of the transformation operation are provided in Section 1.2.
- The obtained name code will be used to find the index I in the IndexArray. This index is found by taking the modulo of the obtained name vode with respect to N (the size of the IndexArray).
  - For example: $376 \equiv 1 \pmod 3$, so, the name code will be inserted into the DataArray that is pointed by the element at index 1 of the IndexArray.
- The name code will be added to the end of the DataArray associated with the determined index.
  - For example, if the code is 376, it will be placed in the next available slot in the DataArray at index 1 of the IndexArray. (In this case, the DataArray at index 1 of IndexArray will be $100 \rightarrow 7 \rightarrow 376 \rightarrow 0$ as seen in figure 2).
- Zeros represent empty cells in DataArrays.

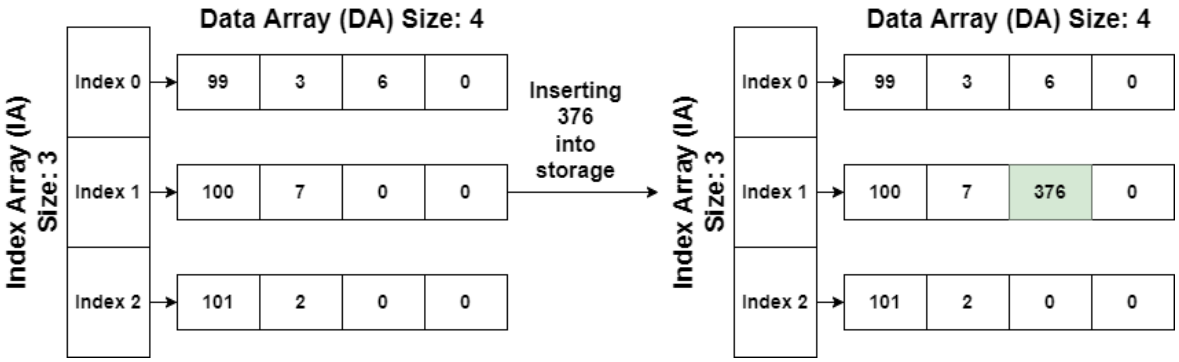

Figure 2: Inserting number 376

1.2 Transformation Function

Transformation function takes the person's name in a char array format, and generates the name code C for a given name. The formula for the transformation function is provided in the following formula 1.

For a name of size n:

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

1.2.1 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 name code C is calculated using the transformation function formula as $74 + 444 + 936 + 1760 = 3214$. When inserting this name code C into the 2D array, Storage:

1. Index I at IndexArray in which the name code C will be stored is calculated by taking the modulo 3 (IndexArray size) of name code C.
   - $3214 \equiv 1 \pmod 3$, $I = 1$
2. The name code C will be inserted at the end of the DataArray at index 1 of the IndexArray.

2 Use Case

3 names "John", "Bob", and "Alice" will be inserted, respectively, into an empty 2D dynamic array, Storage.

- John $\rightarrow 3214 \rightarrow 3214 \equiv 1 \pmod 3 \rightarrow$ Insert into the DataArray at index 1 of the IndexArray
- Bob $\rightarrow 1392 \rightarrow 1392 \equiv 0 \pmod 3 \rightarrow$ Insert into the DataArray at index 0 of the IndexArray
- John $\rightarrow 3214 \rightarrow 5551 \equiv 1 \pmod 3 \rightarrow$ Insert into the DataArray at index 1 of the IndexArray

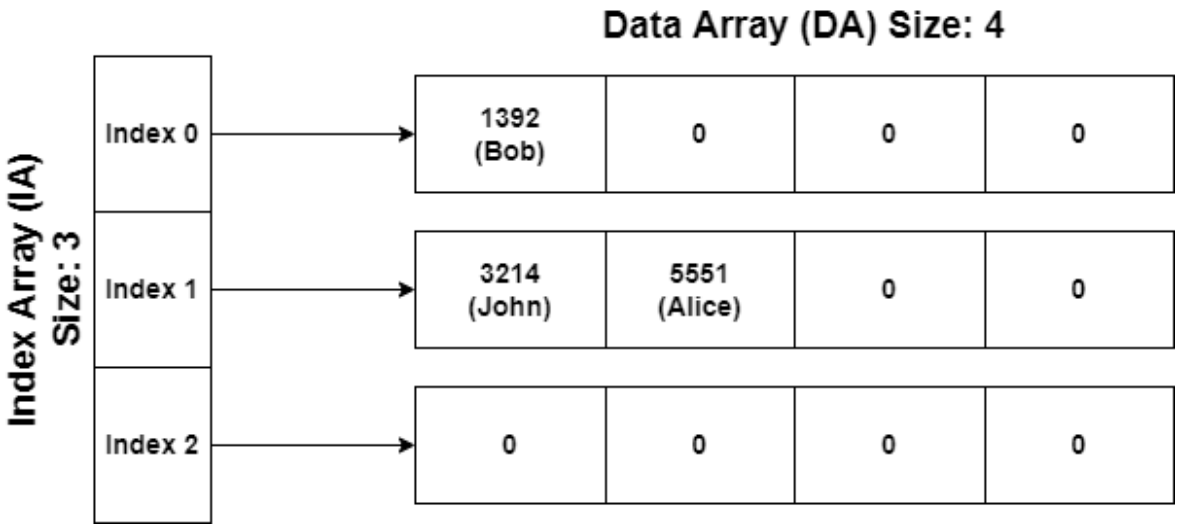

Figure 3: Inserting 3 name codes into the 2D array, Storage

2.1 Data Array Extension

After adding a few names into the Storage, some DataArrays will be full. When adding a new name code, if the corresponding DataArray is full, that DataArray will double in size.

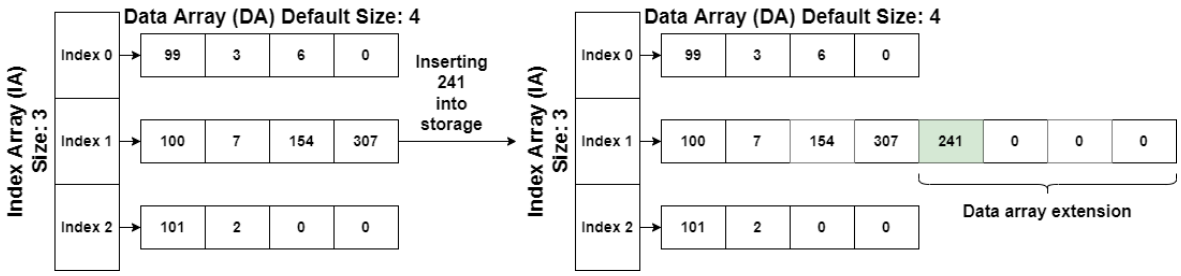

Figure 4: DataArray extension

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 DataArrays. To reduce this complexity, we will define a metric called Storage load factor. If that Storage load factor exceeds a threshold, we extend the IndexArray with a new size.

In this assignment, you are going to extend the IndexArray in the following situation holds:

1. After adding a new name code to the Storage, we check the following condition: if it's met, we will create a new IndexArray twice the size of the previous one.
2. If the total size of the DataArrays is 1.5 times or more than the product of the IndexArray size and the default DataArray size as shown in the following formula 2, then we need to extend the IndexArray.
3. For each index of the new IndexArray, new DataArrays will be allocated from scratch with the default DataArray size (It is 4 for the given example in figure 5).

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

where C_c is the current size of the IndexArray and M is the default DataArray size.
Steps of the IndexArray extension:

- Create a new IndexArray with the new size: $C_c \times 2$.
- Starting from the first row (index 0) of the old IndexArray, add each name code C to the new 2D dynamic array, Storage, properly calculating their new indices.
- Free all old DataArrays and the old IndexArray.

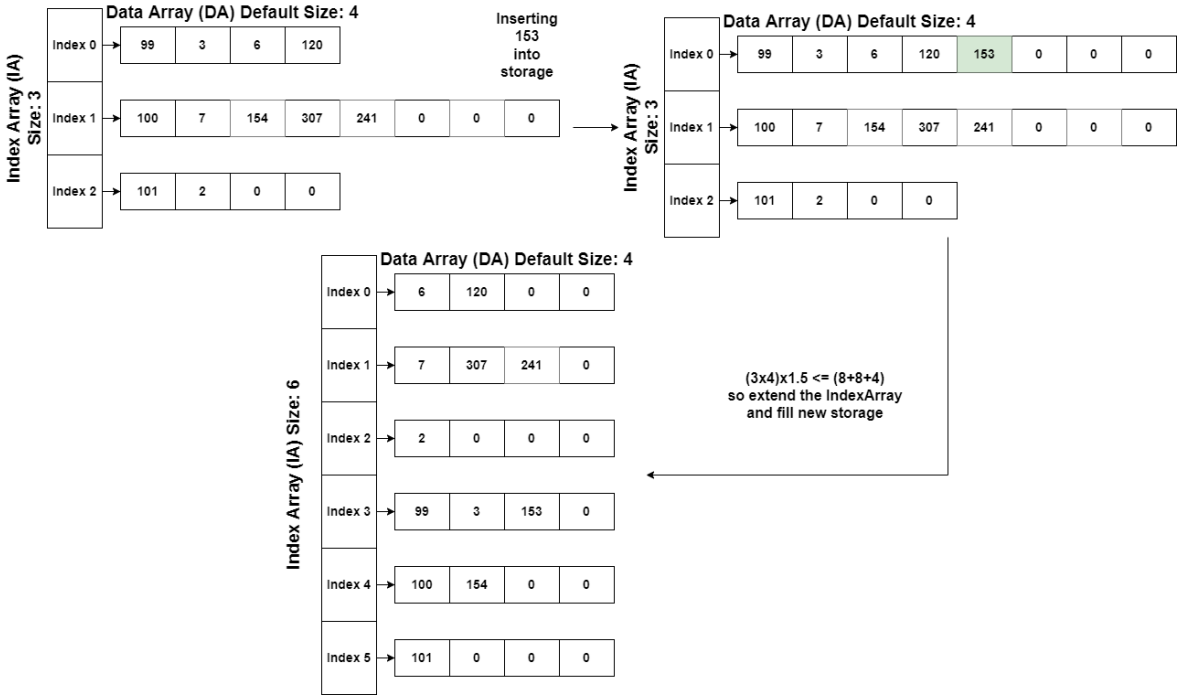

Figure 5: Index array extension

3 Tasks

In this assignment,

- We will create and initialize 2D dynamic arrays
- Insert new name codes
- Check whether given 2D dynamic array, Storage, contains a name code or not

There will also be several helper functions. To be able to implement these functions, we need some variables keeping some statistics of the 2D dynamic array, Storage.

- **Index array size**: This variable is maintained in the main function and gets updated when the IndexArray is extended.
- **Default data array size**: This variable is also maintained in the main function, keeping track of the default size of the DataArray.
- **Data array sizes**: An array with the same size as the IndexArray, it stores the size of each DataArray corresponding to each index. Initially, all elements are initialized to the default DataArray size value. When the DataArrays become full, this array is updated to reflect any changes in size.

Following abbreviations will be used in the code template:

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

You are going to implement 7 functions.

- **int transform_name(const char *name)**: The function receives a person's name in a char array format. It generates a name code for the given person's name using a specified formula from Section 1.2.
- **int** initialize_storage(int IA_size, int DA_size, int** DA_sizes)**: The function receives the IndexArray size, default DataArray size, and a reference to a pointer that will store DataArray sizes.

  Steps:

  - **storage:** Allocate a 2D array of size: (*IndexArray size*) x (*DataArray size*), and assign it to the storage variable that will be returned.
  - **DA_sizes:** Allocate an array of the same size as the IndexArray and initialize each element with the default DataArray size. Assign this array to the reference of the integer pointer DA_sizes, enabling access to the DataArray sizes in the main function.
- **void print_storage(int** storage, int IA_size, int* DA_sizes)**: The function receives a 2D dynamic array, Storage, the size of the IndexArray, and an array containing DataArray 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.

- **void insert(int** storage, int IA_size, const char* name, int* DA_sizes)**: The function takes a 2D dynamic array, IndexArray size, a person's name, and the DataArray sizes array. It begins by calling the transformation function to determine the corresponding name code. Then, it checks whether the DataArray of the index in the IndexArray where the name code should be inserted is full. If the DataArray is full, it extends the DataArray, updates the DataArray sizes array, and then proceeds with the insertion.

  Steps:

  - Transform the person name into a name code.
  - Find the corresponding index by taking modulo.
  - Check if the corresponding DataArray is full or not. If the DataArray is full, double the size of the DataArray.
  - Insert the name code at the end of the DataArray.

- **void insert2(int** storage, int IA_size, int name_code, int* DA_sizes)**: The function takes storage, IndexArray size, name code, and DataArray sizes array. The only difference from the first insert function is that it receives the name code directly instead of the person's name as a char array.
- **void fill_new_storage(int** storage, int IA_size, int** new_storage, int* DA_sizes, int* new_DA_sizes)**: This function takes a storage, old IndexArray size, a new storage, and a DataArray sizes array, as well as a new DataArray sizes array.

  - It transfers all the content from the old storage to the new one, updating the new DA sizes array in the process.
  - The **insert2** function (defined below) will be used in this function to directly insert name codes into the new storage.

  Since the insert functions can update the content of the DataArray sizes array, this function will be straightforward.

- **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 DataArray size, and DA_sizes array.

  - First, it checks if the extension rule that is explained in the section 2.2 is violated.
  - If it is violated, then it initializes a new IndexArray with doubled size and a new DA_sizes array with the same size as the new IndexArray. You can use **initialize_storage** function here.
  - Then, it transfers the content of the old storage to the new one. You can call **fill_new_storage** function here.
  - In the end, it updates IA_size in the main function and frees the old storage, and old DA_sizes array.

4 Example

Note: Please do not add extra prints like those in the example, as they are meant for detailing the steps.

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

Example main function:

```
#include <stdio.h>
#include "the2.h"

int main(){
    char names[][10] = {"John", "Sophia", "Evelyn", "Aaron", "Emily",
                        "Russell", "April", "Patrick", "Eva", "Jesse"};  /* names to be added into storage */
    int i;              /* 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 there is a need for the IndexArray extension. */
        /* Sent references of storage, current IndexArray size and data array sizes */
        /* to not lose their references while updating them in the implemented functions */
        resize_IA(&storage, &IA_size, DA_size, &DA_sizes);
    }

    /* Free up allocated memory for storage and DA_sizes */
    for (i =  0; i < IA_size; i++)
    {
        free(storage[i]);
    }
    free(storage);
    free(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 storage values as zero directly. An empty string will not be added to the storage. So, zeros will be empty cells.
- Do not forget to free old storage and DA_sizes array after extending the storage.
- You do not need math.h to calculate the square in a transformation 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 those separators.
- **initialize_storage** and **print_storage** functions are base functions to be able to get a nonzero 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.
- **Submission:**
  You will use OdtuClass system for the Take Home Exam 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 request for any deadline extensions**.
- **Evaluation:** Your codes will be evaluated based on several input files different than the test cases given to you as an example. You can check your grade with sample test cases via the OdtuClass system but do not forget that it is not your final grade. Your output must give the exact output of the expected output. It is your responsibility to check the accuracy of the output with invisible characters. Otherwise, you cannot get a grade in that case. If your program produces the correct output for all cases, does not have any memory leaks, and utilizes dynamic memory allocation properly, you will receive 100 points. Since THE2 evaluates your understanding of dynamic memory allocation, your code will be inspected for memory leaks.
- **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 with each other or using third-party code is strictly forbidden. Even if you take a "part" of the code from somewhere/someone else, this is also cheating. Please be aware that there are "very advanced tools" that detect if two codes are similar. So, do not think you can get away with changing a code obtained from another source.