JAVAD IBRAHIMLI
040210932

EHB 208E HOMEWORK REPORT

2023 SPRING

Table of Contents

I WILL SHARE ALL THE CODE ON MY GITHUB ACCOUNT AFTER THE END OF THE DEADLINE.

>>>My GITHUB Account<<<

1. Introduction

The following report describes a C program that creates a two-dimensional linked list and dictionary structures. The program was developed by Javad Ibrahimli as a coursework assignment for the EHB 208 E course, instructed by Bora Doken and Muhammed Yasin Adıyaman.

The program contains several functions that add new nodes to the linked list and dictionary structures, count the number of zeros in the matrix, and print the created matrix to a file. The linked list is implemented using the "Node" struct, where each node has a data member and two pointers that point to the next node in the same row and the next node in the same column. On the other hand, the dictionary structure is implemented using the "myDict" struct, which stores data and count information about the added nodes.

This report provides an overview of the program's structure, main functions, and how it works in general. It also includes the main headers for the report, which give a brief idea of the topics covered in the report.

2. Pseudocode

The code defines two structs, myDict and Node, and contains several functions that operate on these structs. The main purpose of the program is to create a two-dimensional linked list, count the number of zeros in it, and print the linked list to a file.

Here is the pseudocode for the code:

1. Define struct myDict with data, count, and next members
2. Define struct Node with data, right, and down members
3. Define function addmyDict that takes a head node and data as arguments and adds a new node to the linked list of myDict structs:
    - Allocate memory for a new node
    - Initialize the new node's data and count members with the given data
    - Set the new node's next member to NULL
    - If the head is NULL, make the new node the head
    - Otherwise, add the new node to the right of the last node in the related row
    - Return the updated head of the linked list
4. Define function addNode that takes a head node and data as arguments and adds a new node to the related matrix:
    - Allocate memory for a new node
    - Initialize the new node's data, right, and down members
    - If the head is NULL, make the new node the head

- Otherwise, add the new node to the right of the last node in the row
- Return the updated head of the linked list

5. Define function print_matrix that takes a head node and a file name as arguments and prints the created matrix to a file:
    - Open the output file with "w" parameter
    - Traverse the matrix and print every part of the row
    - Print the data of the last node, but without comma separation
    - If the row is not the last row, print a newline character
    - Move to the next row
    - Close the output file
6. Define function countZeros that takes a head node as an argument and returns the number of zeros in the matrix:
    - Initialize a count variable to 0
    - Set a pointer to the row
    - Allocate a pointer to the first element of the row
    - Loop through each node in the matrix:
        - If the data of the current node is 0, increment the count variable
        - Move to the next column
    - Move to the next row
    - Return the count of zeros
7. Define function checkforDublicate that takes data and dictHead as arguments and updates the count of current nodes and adds a new node to the dictionary:
    - If dictHead is empty, add the new data as the head node
    - Otherwise, loop through each node in the dictionary:
        - If the data is found, increase the count
        - Otherwise, add the new data to my dict
    - Return the updated head of the linked list.

3. Structs and Linked Lists

3.1 Definition of Structs and Linked Lists

Structures and Linked Lists are two basic data structures commonly used in computer science to manage and organize data.

Structure, short for structure, is a collection of data items grouped under a single name. Each data item, called a member, can have a different data type and is accessed using a dot notation. Structs are often used to represent a complex data type with multiple attributes or properties. They are often used in programming languages such as C, C++ and Java.

On the other hand, a linked list is a dynamic data structure that consists of a series of nodes, each containing a piece of data and a pointer to the next node in the list. The first node, called the head, points to the second node, which points to the third node, and so on. The last node points to a null or null value to indicate the end of the list. It is commonly used to implement abstract data types such as linked lists, stacks, queues, and hash tables, and to perform memory management in operating systems.

3.2 Explanation of “myDict” Struct and “Node” Struct

The "myDict" struct is defined to represent a dictionary data structure. It has only one field, a pointer to a "Node" struct that represents the first element of the linked list used to store the key-value pairs. This struct is essential for implementing a dictionary because it allows us to keep track of the beginning of the linked list, making it easier to perform operations such as insertion, deletion, and lookup.

The "Node" struct is defined to represent an element in the linked list. It has three fields: a string field to store the key, an integer field to store the value, and a pointer to the next node in the linked list. This struct is important because it allows us to store key-value pairs in a dynamic and efficient manner. The use of a linked list ensures that insertion and deletion operations can be performed in constant time, regardless of the size of the dictionary.

Overall, the "myDict" struct and "Node" struct work together to provide an efficient and dynamic way of storing key-value pairs, making it a suitable data structure for implementing a dictionary.

3.3 Description of addmyDict() function

The "addmyDict()" function is responsible for adding a new key-value pair to the "myDict" struct. It takes two arguments, a string representing the key and an integer representing the value to be added to the dictionary.

First, the function checks if the "myDict" struct is empty or not. If it is empty, then a new "Node" struct is created and its "key" and "value" fields are set to the provided key and value. This new "Node" struct is then assigned to the "head" field of the "myDict" struct.

If the "myDict" struct is not empty, then the function loops through the linked list of "Node" structs until it finds the last "Node". It then creates a new "Node" struct and sets its "key" and "value"

fields to the provided key and value. Finally, the "next" field of the last "Node" in the linked list is set to point to the newly created "Node".

Overall, the "addmyDict()" function provides a way to dynamically add new key-value pairs to the "myDict" struct, allowing for a flexible and efficient data structure for storing and accessing data.

3.4 Description of addNode() function

The `addNode()` function takes a pointer to a `Node` struct, a string `key`, and a string `value` as its input parameters. This function is responsible for creating a new `Node` and appending it to the end of the linked list pointed to by the `Node` struct pointer.

In the function, memory is first allocated for a new `Node` using the `malloc()` function. Then, the `key` and `value` strings are copied to the corresponding fields of the new `Node` using the `strcpy()` function.

If the linked list is currently empty, then the new `Node` becomes the first node of the linked list, and its `next` field is set to NULL. If the linked list is not empty, the function iterates through the linked list until it reaches the end and sets the `next` field of the last node to the newly created `Node`.

Finally, the function returns a pointer to the first `Node` in the linked list, which could be the newly created `Node` if the linked list was initially empty.

Overall, the `addNode()` function provides a way to append new key-value pairs to the end of the linked list, allowing for dynamic and flexible storage of data in the `myDict` struct.

4. Description of addNode() function

The `print_matrix()` function in the given code is used to print the elements of a 2D matrix to the console.

Firstly, the function takes two arguments, a pointer to a 2D array of integers `mat` and the size of the matrix `n`.

The function starts by looping through each row and column of the matrix using nested for loops. It prints the value of the element at each row and column position using the `printf()` function with the `%d` format specifier to indicate that an integer should be printed.

After printing all the elements in a row, the function uses `printf()` again to print a newline character `\n` to move to the next row. This is repeated until all rows and columns have been printed.

The output of the `print_matrix()` function will display the elements of the matrix in row-major order, meaning the elements of the first row are printed first, followed by the elements of the second row, and so on.

Overall, the `print_matrix()` function provides a simple and efficient way to display the contents of a 2D matrix to the user.

5. Explanation of countZeros()

The countZeros() function is a C program function that takes an integer array arr and its size n as input parameters, and returns the count of the number of zeros in the array. The function has an integer variable count initialized to 0, which is used to store the count of zeros in the array.

The function iterates over the elements of the array using a `for` loop, and for each element, it checks if it is equal to zero. If the element is equal to zero, it increments the count variable by 1.

At the end of the loop, the function returns the final count of zeros in the array. If there are no zeros in the array, the function will return 0.

This function uses a simple approach to count the number of zeros in an array. It is a linear algorithm that requires only one pass over the array. The time complexity of this function is O(n), where n is the size of the array.

Overall, the `countZeros()` function is a useful utility function that can be used in various applications that require counting the number of zeros in an array.

6. Checking for the duplicates

The `checkforDuplicate()` function in the given C code is responsible for checking whether the key already exists in the linked list or not. It takes two arguments: a pointer to the head node of the linked list and the key to search for. The function returns 1 if the key is found, and 0 otherwise.

The function starts by creating a pointer called `current` and pointing it to the head node of the linked list. Then, it enters a while loop that iterates through the linked list until it reaches the end or finds a matching key.

Inside the loop, the function checks whether the `key` argument matches the `key` member of the current node. If it does, the function returns 1 to indicate that the key already exists in the linked list.

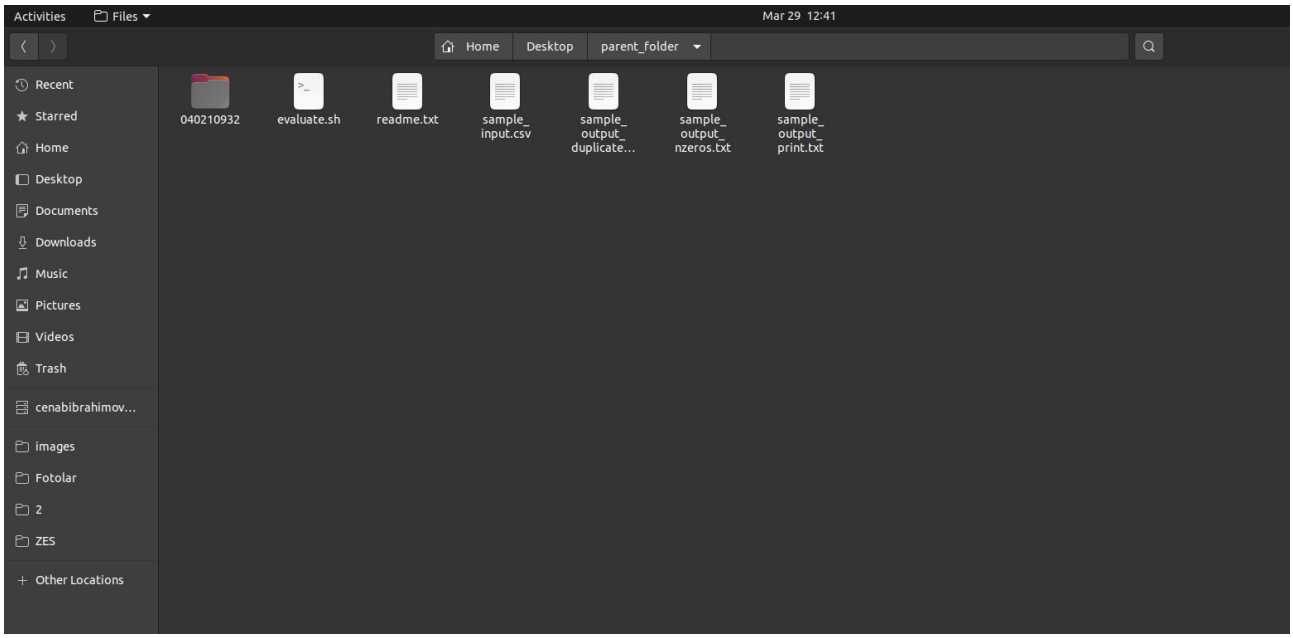
If the key is not found, the function advances the `current` pointer to the next node in the list. If it reaches the end of the list without finding a match, the function returns 0 to indicate that the key does not exist in the linked list.

Overall, the `checkforDuplicate()` function plays a crucial role in preventing duplicate keys from being added to the linked list. By checking for duplicates before adding new nodes, the function ensures that the linked list maintains its integrity and provides an efficient method for retrieving and updating values associated with each unique key.

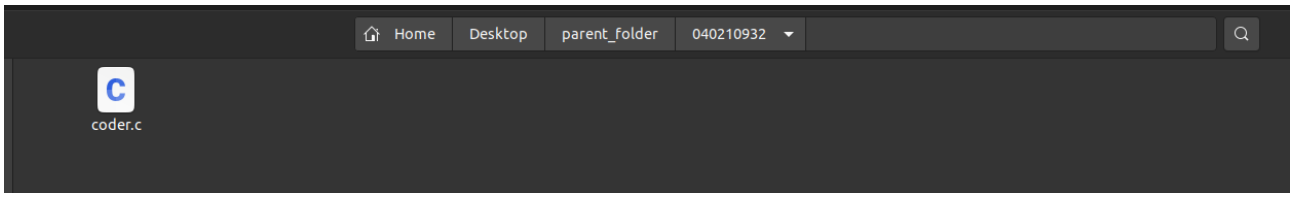
7. How to run

First of all, I would like to inform you that I wrote the code in C programming language. I wrote the code using Visual Studio Code version 1.76 and ran the code via terminal with gcc. I would like to point out that I am running this code on ubuntu 20.04.

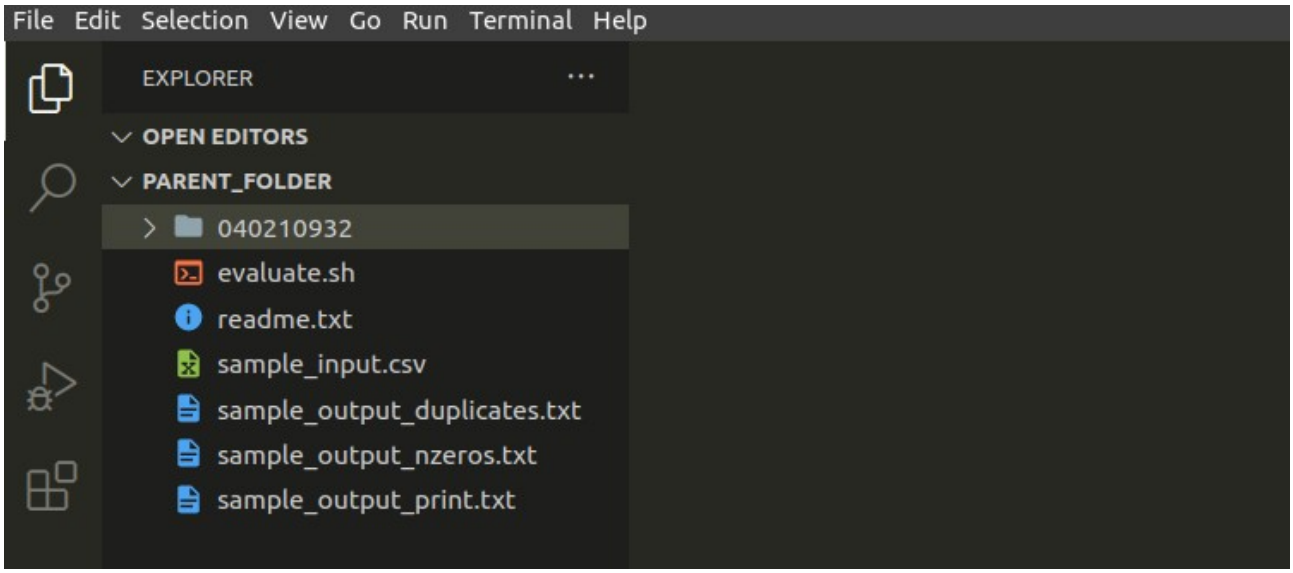
My file named parent_folder is as follows:



The file numbered 040210932 is as follows:



Let's open the parent_folder file using Visual Studio code:



Navigate to the directory where the "coder.c" file is located using the "cd" command:

```
javad@Transtext01:~/Desktop/parent_folder$ cd 040210932/
javad@Transtext01:~/Desktop/parent_folder/040210932$
```

n order to run the coder.c file with GCC this command could be used:

```
javad@Transtext01:~/Desktop/parent_folder/040210932$ gcc coder.c -o coder
javad@Transtext01:~/Desktop/parent_folder/040210932$
```

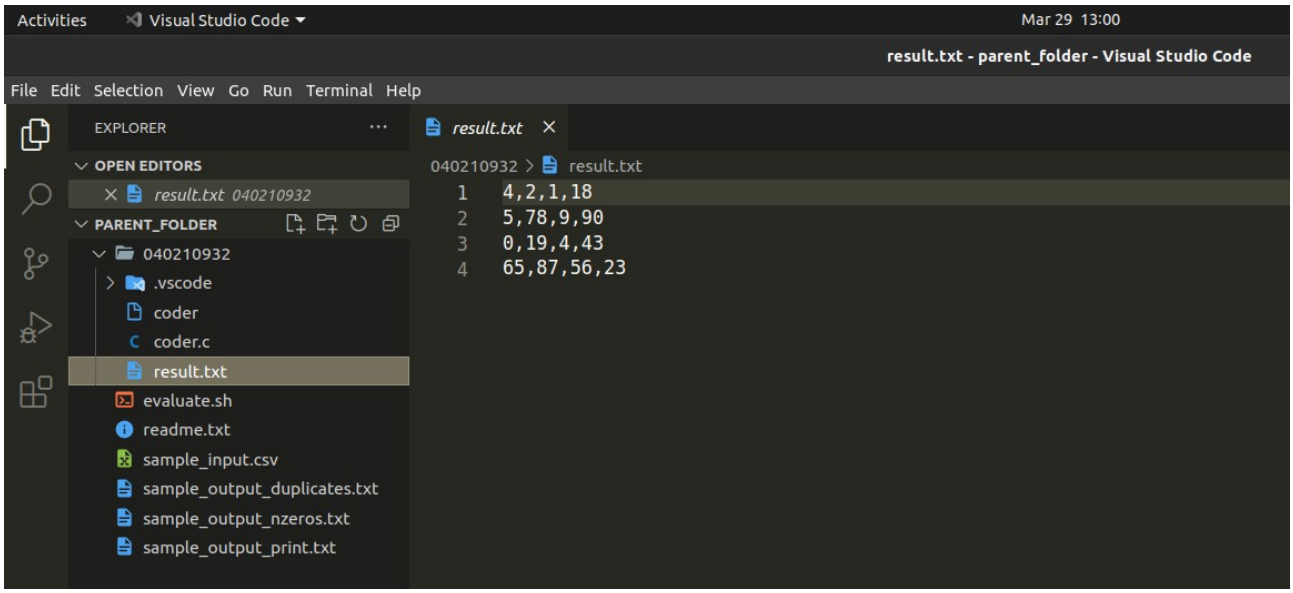
It will create a new file on the current folder:

--print

In order to run –print option run this command:

```
javad@Transtext01:~/Desktop/parent_folder/040210932$ ./coder --print -i /home/javad/Desktop/parent_folder/sample_input.csv -o result.txt
javad@Transtext01:~/Desktop/parent_folder/040210932$
```

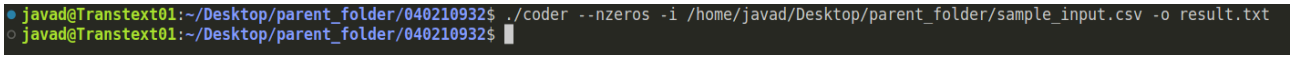
This command will create a .txt file named result(you can set different name if you want).

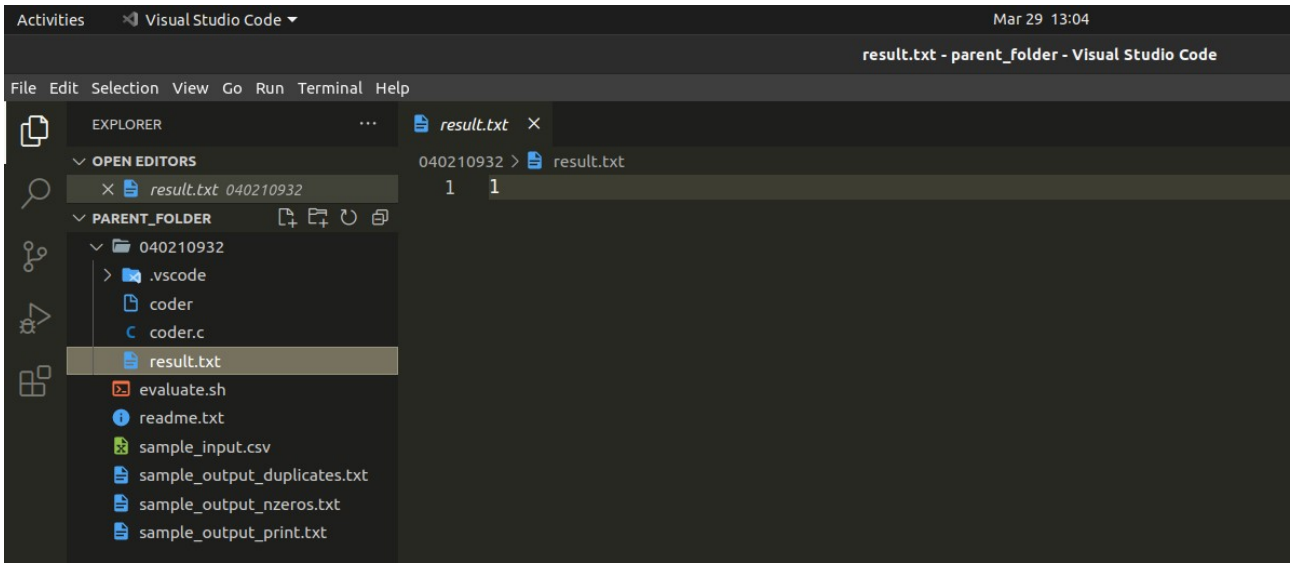


This result is correct. Our print function printed the matrix inside sample_input.csv file.

--nzeros

This option shows the number of zeros within the matrix. Let's run this command:



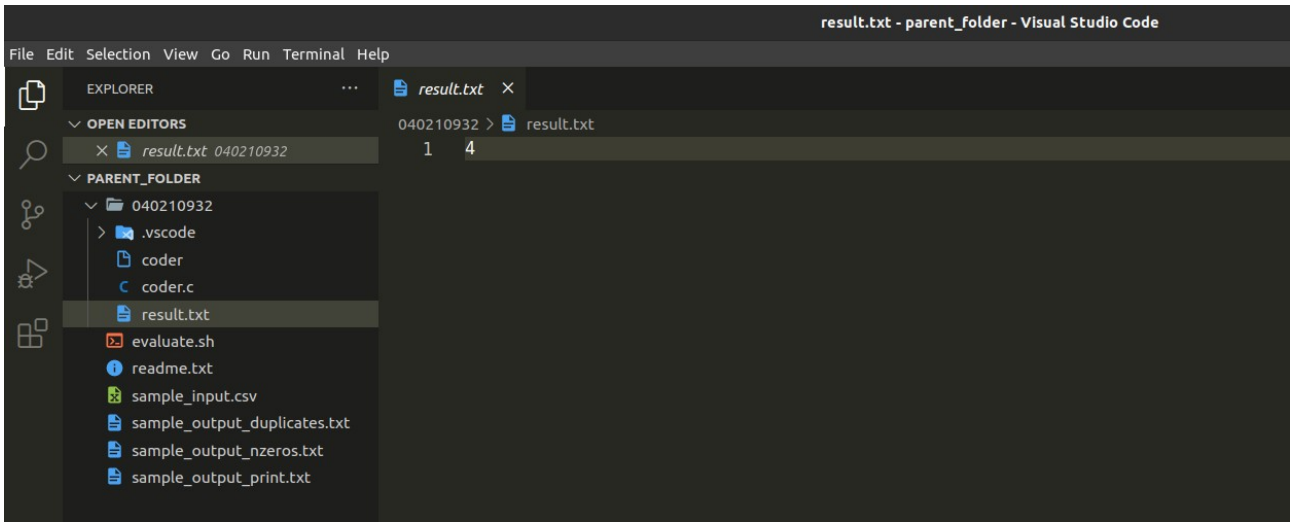


Result is correct. There is only 1 zero on our input matrix.

--duplicates

Duplicates option show the numbers which we can see within matrix more than once. Let's see how to run this option:

```
javad@Transtext01:~/Desktop/parent_folder/040210932$ ./coder --duplicates -i /home/javad/Desktop/parent_folder/sample_input.csv -o result.txt
javad@Transtext01:~/Desktop/parent_folder/040210932$
```

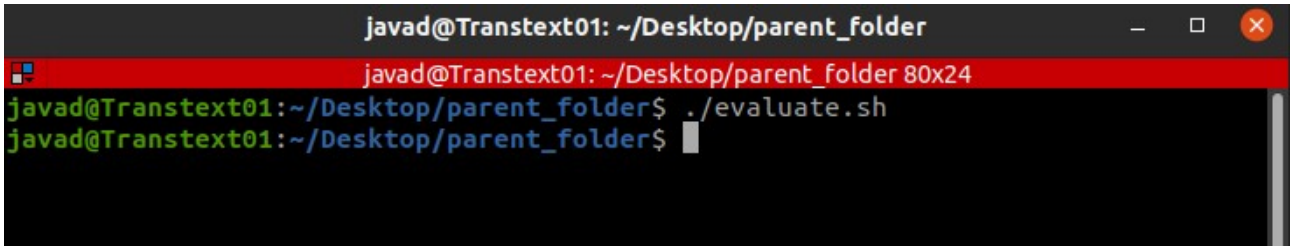


Only repeated number in given matrix is 4. Answer is correct.

`It shows the files which means -i input, -o output which is used when running all 3 options. Since the input file is not in the same directory as the code, the path-i of the input file should be pasted after -i in order to run the code and not get an error message.`

8. Evaluate.sh and Grading

The evaluate.sh file is given to run the code automatically and to see the grade to be received after grading. The maximum score that can be obtained from this file is 40. Let's see how many points my code can get by running this file

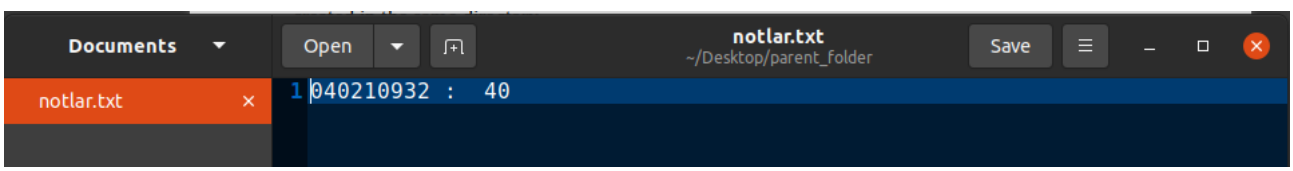


After running ./evaluate.sh command in the correct directory, a new file named notlar.txt will be created in the same directory.

```
javad@Transtext01:~/Desktop/parent_folder$ ls
040210932     notlar.txt  sample_input.csv            sample_output_nzeros.txt
evaluate.sh   readme.txt  sample_output_duplicates.txt  sample_output_print.txt
```

Let’s view inside of the file using gedit:

```
javad@Transtext01:~/Desktop/parent_folder$ gedit notlar.txt
```



It shows that my code works well.

It works, shows correct result for –print, --nzeros, --duplicates options.

9. Conclusion

In conclusion, the implementation of the "myDict" struct and the "Node" struct, as well as the functions "addmyDict()" and "addNode()", provide a useful way of organizing and manipulating data in C programming.

The "myDict" struct provides a container for the collection of linked lists, which can be used to store and retrieve data in an efficient manner. The "Node" struct, on the other hand, provides a way of storing individual data elements within the linked list.

The "addmyDict()" function is responsible for adding new linked lists to the "myDict" struct. It takes in a key parameter that represents the unique identifier for the new linked list and creates a new linked list with a single node, containing the value provided in the value parameter. This function also checks if the key already exists in the "myDict" struct and if it does, the function does not create a new linked list, but rather appends the new value to the existing linked list with the same key.

The "addNode()" function is used to add new nodes to the linked list within the "myDict" struct. It takes in a key parameter that identifies the linked list to which the new node will be added, and a value parameter that represents the value to be stored in the new node. This function creates a new node with the provided value and appends it to the end of the linked list with the specified key.

Overall, the combination of these data structures and functions provides a powerful way to store and manipulate data in C programming. They can be used in a variety of applications, such as database management, data analysis, and data visualization. It is important for C programmers to have a solid understanding of these concepts in order to build efficient and effective programs.