HCMC UNIVERSITY OF TECHNOLOGY AND EDUCATION
FALCUTY OF INTERNATIONAL EDUCATION

FINAL TERM PROJECT

Course name: Discrete Mathematics and Graph Theory

STUDY AND IMPLEMENT
A FAMILY TREE MANAGEMENT PROGRAM

Lecturer name: Assoc. Prof. Hoang Van Dung

List of members:

Student ID	Student name	Contribution (%)
22110010	Nguyen Huynh Quoc Bao	100
23110004	Vo Nguyen Ngoc Bich	100
21110058	Cao Khai Minh	100

Ho Chi Minh City, 11/2024

ACKNOWLEDGEMENT

First of all, we would like to express our deepest gratitude and sincere thanks to *Mr. Hoang Van Dung*, who has dedicatedly guided and supported us throughout the final project of the *Discrete Mathematics and Graph Theory* course. Through his guidance, we not only mastered essential concepts but also had the opportunity to explore practical applications of the subject. He has always been patient in answering our questions, inspiring creative and logical thinking, and helping us to develop analytical skills and solve complex problems.

Thanks to Mr. Dung's dedicated instruction, we were able to complete this project with newfound knowledge and insights. Although we have done our best, we understand that our project inevitably has shortcomings and limitations. We hope to receive feedback from Mr. Dung to further refine our knowledge and skills, which will serve us well in future academic projects and professional endeavors.

Once again, we would like to express our profound appreciation to Mr. Dung for his efforts, dedication, and passion for teaching, which he has generously shared with us and all his students. We wish him abundant health, happiness, and continued success in his career, inspiring and imparting knowledge to generations of students on their journey of learning and discovery.

Best regard,

List of students
Nguyen Huynh Quoc Bao
Vo Nguyen Ngoc Bich
Cao Khai Minh

TABLE OF CONTENTS

ABBREVIATION LIST

No	Abbreviation	Full text
1	ECMA	European Computer Manufacturers Association
2	ISO	International Standards Organization
3	COBOL	Common Business-Oriented Language
4	DFS	Depth-First Search
5	BFS	Breadth-First Search
6	UI	User Interface

Table 1: Abbreviation list

FIGURE LIST

TABLE LIST

TASK ASSIGNMENT

Student ID	Student Name	Contribute	Work
22110010	Nguyen Huynh Quoc Bao	100%	Function coding, advance method
23110004	Vo Nguyen Ngoc Bich	100%	Create report of the project Create the overlay of Family Tree Management program
21110058	Cao Khai Minh	100%	Function coding, method

Table 2: Work plan

A. INTRODUCTION

1. Rationale for Choosing the Project

Family history is an important aspect of heritage, often cherished and preserved across generations. However, in many families, maintaining an organized and accurate record of lineage and family connections can be challenging, especially as the family expands over time. Traditional methods, such as physical records or memory-based accounts, are prone to deterioration, loss, or inaccuracies. As these methods are neither sustainable nor efficient in the digital age, there is a pressing need for a modern solution that can effectively address these limitations.

The digital age provides us with tools and technologies that can store vast amounts of information, simplify retrieval, and allow easy access to family records with just a few clicks. Recognizing the need to move beyond traditional record-keeping, we chose the project ***"Family Tree Management"*** to create a system that simplifies the process of managing family information. This project aims to offer a reliable, user-friendly platform that makes it convenient for families to digitally store, retrieve, and organize information about family members and relationships. By developing a digital family tree management program, we hope to contribute to the preservation of family histories for current and future generations, allowing families to appreciate and connect with their ancestry in a structured and easily accessible way.

2. Project Objectives

- Create a system that allows users to add, modify, and delete family members.
- Develop functionality for displaying family trees visually and querying specific information.
- Ensure the application is user-friendly and scalable for future expansions.

B. CONTENT

1. Background Knowledge

Tree Data Structure in Programming:

Tree data structure is a hierarchical structure that is used to represent and organize data in a way that is easy to navigate and search. It is a collection of nodes that are connected by edges and has a hierarchical relationship between the nodes.

The topmost node of the tree is called the ***root***, and the nodes below it are called the child nodes. Each node can have multiple child nodes, and these child nodes can also have their own child nodes, forming a recursive structure.

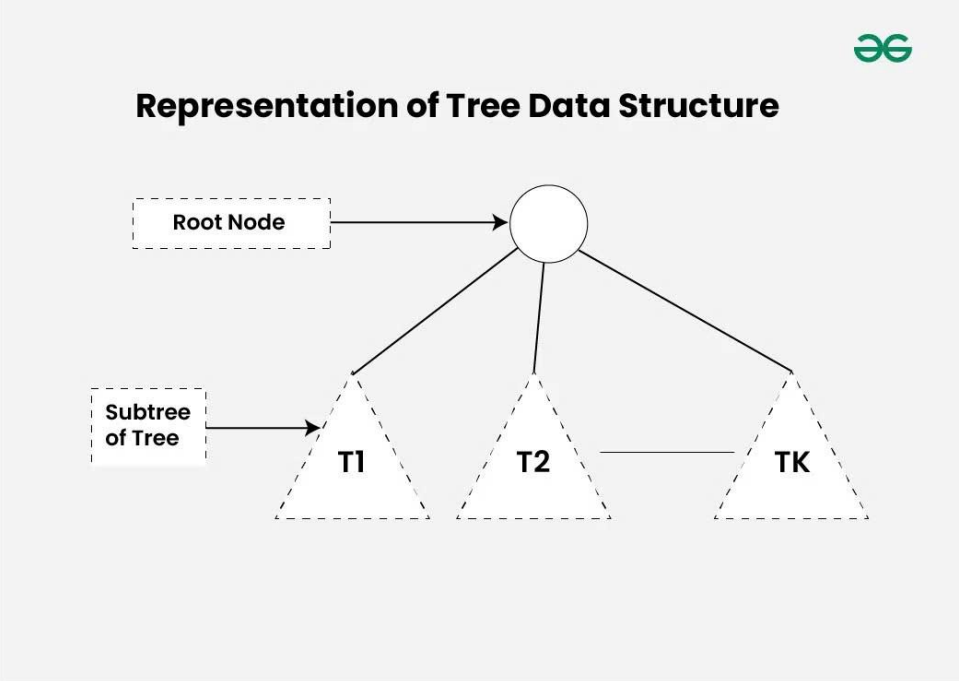

Figure 1: Representation of Tree Data Structure

Family Relationship Theory:

In ***Family Systems Theory***, the family is viewed as a system composed of interrelated parts, such that a problem for any member of a family has an effect on all others, and changes in any member result in adaptations among all other family members.

Family systems theory views an individual person as a complex being operating within a system. Epstein and Bishop have summarized the major assumptions of family systems theory as follows:

- The parts of the family are interrelated.
- One part of the family cannot be understood by simply understanding each of the parts.
- Family functioning cannot be fully understood by simply understanding each of the parts.
- A family's structure and organization are important factors determining the behavior of family members.

- Transactional patterns of the family system shape the behavior of family members.

This project will explore essential family relations (e.g., parent-child, spouse) and extended relations (e.g., siblings), which will inform the database and logic model designs.

Technology Overview:

- ***Programming Language:*** The project will utilize C# for core development.

C# is a general-purpose, modern and object-oriented programming language pronounced as ***"C sharp"***. It was developed by Microsoft led by Anders Hejlsberg and his team within the .Net initiative and was approved by the European Computer Manufacturers Association ***(ECMA)*** and International Standards Organization ***(ISO)***. C# is among the languages for Common Language Infrastructure and the current version of C# is version 7.2. C# is a lot similar to Java syntactically and is easy for the users who have knowledge of C, C++ or Java, a bit about .Net Framework .Net applications are multi-platform applications and framework can be used from languages like C++, C#, Visual Basic, COBOL etc. It is designed in a manner so that other languages can use it, know more about .Net Framework**.**

Why C#? C# has many other reasons for being popular and in demand. Few of the reasons are mentioned below:

1. **Easy to start:** C# is a high-level language so it is closer to other popular programming languages like C, C++, and Java, thus, becomes easy to learn for anyone.
2. **Widely used for developing Desktop and Web Application:** C# is widely used for developing web applications and Desktop applications. It is one of the most popular languages that is used in professional desktop. If anyone wants to create Microsoft apps, C# is their first choice.
3. **Community:** The larger the community the better it is as new tools and software will be developing to make it better. C# has a large community, so the developments are done to make it exist in the system and not become extinct.
4. **Game Development:** C# is widely used in game development and will continue to dominate. C# integrates with Microsoft, thus, has a large target audience. The C# features such as Automatic Garbage Collection, interfaces, object-oriented, etc, which make C# a popular game developing language.

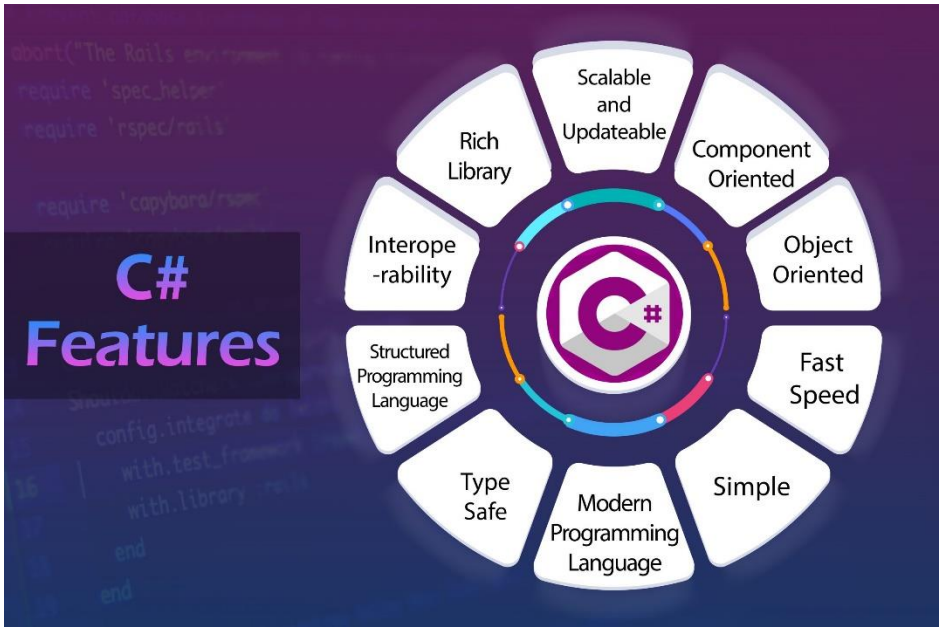

Figure 2: Features of C Sharp

- ***Text file:*** Used for storing purposes.

A text file is a type of computer file that stores plain text information without any embedded formatting options such as bold, italics, or images. The content in a text file is composed entirely of letters, digits, and special characters that can be easily read and edited using basic text editors.

Text files are distinguished by their simplicity and the lack of features for rich formatting, which makes them highly versatile and compatible across different platforms and applications. They are often used for writing and storing code, configuration directives, and documentation due to their straightforward nature which allows them to be processed by many software tools without the need for complex parsing or formatting.

2. Methods and Techniques

2.1. Methods

Tree Structure Representation: The project will implement tree data structures to represent family relationships. This will enable clear and organized family hierarchy representation, with distinct relationships like parent-child and sibling connections.

Family Relationship Modeling: The program will define core family relationships (such as parent-child and spouse) and extended relations (like siblings). These relationships will guide the design of the data model and logic, allowing users to add, edit, delete and view family connections.

Data Privacy and Accessibility: The system will prioritize user data security, ensuring that information about family members and their relationships is stored and accessed with respect to privacy.

2.2. Techniques

2.2.1. Tree Traversal Algorithms

Depth-First Search (DFS): DFS will be used to explore family connections deeply, allowing efficient navigation from a member to their extended family.

Breadth-First Search (BFS): BFS will be used to examine relationships at the same level within the family tree, which is especially useful for finding siblings or close relatives.

Parameters	BFS	DFS
Stands for	BFS stands for Breadth First Search.	DFS stands for Depth First Search.
Data Structure	BFS(Breadth First Search) uses Queue data structure for finding the shortest path.	DFS(Depth First Search) uses Stack data structure.
Definition	BFS is a traversal approach in which we first walk through all nodes on the same level before moving on to the next level.	DFS is also a traversal approach in which the traverse begins at the root node and proceeds through the nodes as far as possible until we reach the node with no unvisited nearby nodes.
Conceptual Difference	BFS builds the tree level by level.	DFS builds the tree sub-tree by sub-tree.
Approach used	It works on the concept of FIFO (First In First Out).	It works on the concept of LIFO (Last In First Out).
Suitable for	BFS is more suitable for searching vertices closer to the given source.	DFS is more suitable when there are solutions away from source.
Applications	BFS is used in various applications such as bipartite graphs, shortest paths, etc. If weight of every edge is same, then BFS gives shortest pat from source to every other vertex.	DFS is used in various applications such as acyclic graphs and finding strongly connected components etc. There are many applications where both BFS and DFS can be used like Topological Sorting, Cycle Detection, etc.

Figure 3: Difference between BFS and DFS

2.2.2. Data Management Techniques

- ***Relational Design:*** Text file will be designed to store individual family members and their relationships in an organized manner, reducing redundancy and ensuring data integrity.

Family Tree File: This file will store key information about each individual, such as:

+ Name: Full name of the invidual.
+ Date of Birth: The individual's birth date.
+ Partner: The individual's spouse.
+ Child: The individual's children.

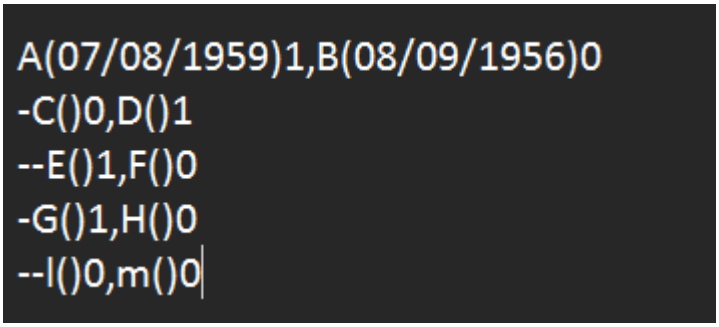

Figure 4: Text file storing Family Tree

- ***Modify:*** Adding, editing, deleting information will be done in two ways:
  + *Modifying in text file:* Directly editing the text file allows users to make changes to the Family Tree without needing to run the program.
  + *Modifying in program:* This method uses the program itself to read, modify, and writes back changes to the text file.

Aspect	Modifying in Text File	Modifying in Program
Ease of Use	Requires familiarity with file structure.	User-friendly, intuitive interface.
Validation	Manual, prone to errors.	Automated, ensures consistency.
Flexibility	Direct and immediate.	Program-guided, safer.
Error Handling	No built-in error prevention.	Error handling built into the program.

Figure 5: Summary of Modification Methods

3. System Design

System Requirements Analysis:

- ***Functional Requirements:*** Include the ability to add, edit, and delete family members, manage relationships, and visualize family trees.

- ***Non-functional requirements:*** The system should prioritize user data privacy, reliability, and user-friendly navigation.

System Architecture: The program's architecture will involve component:

- ***Person Class:*** Manage individual member details, and relationships.

- ***Modifying Methods:*** Add, edit, and delete family members.

- ***Functional Methods:*** Read, print text file, visualize family trees, check the relationship between two members.

- ***Interface:*** Simple interface with Console commands.

- ***Main program:*** Initialize variables and create switch cases for the program.

3.1. Person Class

Attributes	public string PersonID {get; set;} public Person[] parent = new Person[2]; public string name {get; set;} public DateTime bDay {get; set;} public Person partner {get; set;} public List<Person> child {get; set;} public int step {get; set;} public bool gender {get; set;}

Constructors	public Person(string name, DateTime bDay, Person partner, List<Person> child, Person[] parents, int step, bool gender)
	{
	this.step = step;
	this.parent = parents;
	this.name = name;
	this.bDay = bDay;
	this.partner = partner;
	this.child = child;
	this.PersonID = createID();
	this.gender = gender;
	}
	public Person(string name, int step, bool gender)
	{
	this.step = step;
	this.name = name;
	this.bDay = bDay;
	this.partner = new Person();
	this.child = new List<Person>();
	this.parent = new Person[2];
	this.PersonID = createID();
	this.gender = gender;
	}
	public Person(string name, DateTime bDay, int step, bool gender)
	{
	this.step = step;
	this.name = name;
	this.bDay = bDay;
	this.partner = new Person();
	this.child = new List<Person>();
	this.parent = new Person[2];
	this.PersonID = createID();

	this.gender = gender;
	}
	public Person()
	{
	this.step = -1;
	this.name = "";
	this.bDay = default(DateTime);
	this.partner = null;
	this.child = new List<Person>();
	this.parent = null;
	this.PersonID = createID();
	this.gender = true;
	}

3.2. Modifying Methods

Add	- *Add Parents*
	public static void AddParent(Person A, Person B)
	{
	if (A == null || B == null) return;
	
	// Initialize A's parent array if it is null
	if (A.parent == null) A.parent = new Person[2];
	
	// Add B as the first parent if not already set
	if (A.parent[0] == null)
	{
	A.parent[0] = B;
	A.parent[1] = B.partner;
	}
	else if (A.parent[1] == null && A.parent[0] != B) A.parent[1] =B;
	
	// Handle A's partner's parents
	if (A.partner != null && !string.IsNullOrEmpty(A.partner.name))
	{
	if (A.partner.parent == null) A.partner.parent = new Person[2];
	
	// Add B and B's partner as A's partner's parents
	if (A.partner.parent[0] == null)
	{
	A.partner.parent[0] = B;
	A.partner.parent[1] = B.partner;
	}

	else if (A.partner.parent[1] == null && A.partner.parent[0] !=B)
	A.partner.parent[1] = B.partner;
	}
	}
	
	- *Add Children*
	public void addChild(Person child)
	{
	if (this.partner == null) return;
	//if (child.bDay.Year - this.bDay.Year < 0)
	//{
	// return;
	//}
	this.child.Add(child);
	this.partner.child.Add(child);
	child.parent[0] = this;
	child.parent[1] = this.partner;
	child.step = this.step + 1;
	if (child.partner != null)
	{
	child.partner.parent= child.parent;
	child.partner.step = this.step + 1;
	}
	return;
	}
	public void addPersonConnection(Person A)
	{
	if (this.partner != null) return;
	this.partner = A;
	return;
	}
	
	- *Add Person Connection*
	public void addPersonConnection(Person A)
	{
	if (this.partner != null) return;
	this.partner = A;
	return;
	}

Remove	- *Remove Parents*
	public static void RemoveParent(Person A, Person B)
	{
	if (A == null || B == null) return;

```
            // Remove B from A's parent array
            if (A.parent != null)
            {
                // Clear the reference to the first parent
                if (A.parent[0] == B)   A.parent[0] = null;
                // Clear the reference to the second parent
                else if (A.parent[1] == B)  A.parent[1] = null;
            }

            // Handle A's partner's parents
            if (A.partner != null && !string.IsNullOrEmpty(A.partner.name) &&
A.partner.parent != null)
            {
                // Clear the reference to the first parent of the partner
                if (A.partner.parent[0] == B)   A.partner.parent[0] = null;
                // Clear the reference to the second parent of the partner
                else if (A.partner.parent[1] == B.partner)   A.partner.parent[1] =
null;
            }
        }
```

- *Remove Children*

```
public void removeChild(Person child)
        {
            if (child == null || this.child == null || this.partner == null)
                return;

            // Remove the child from the current parent's list of children
            if (this.child.Contains(child))
            {
                this.child.Remove(child);
            }
            // Remove the child from the partner's list of children
            if (this.partner.child.Contains(child))
            {
                this.partner.child.Remove(child);
            }
            // Clear the parent references in the child
            if (child.parent != null)
            {
                if (child.parent[0] == this) child.parent[0] = null; // Clear reference
to this parent
                if (child.parent[1] == this.partner) child.parent[1] = null; // Clear
reference to the partner
```

	}
	child.step = 0;
	// If the child has a partner, clear the partner's parent reference
	if (child.partner != null)
	{
	child.partner.parent = null; // Clear the parent's reference
	child.partner.step = 0; // Reset step if needed
	}
	}
	
	- *Remove Person Connection*
	public void removePersonConnection(Person A)
	{
	if (this.partner == A) this.partner = null;
	if (A.partner == this) A.partner = null;
	return;
	}

Create ID	public string createID()
	{
	DateTime dateTime = this.bDay;
	string tmp = dateTime.ToString("ddMMyy");
	string tmp_ = "";
	for (int i = 0; i < tmp.Length; i++)
	{
	if (tmp[i] == '/')
	{
	continue;
	}
	tmp_ += tmp[i];
	}
	tmp_ = this.name + tmp_ + this.step;
	return tmp_;
	}

3.3. Functional Methods

Read text file	public static List<string> readFile(string str) //Code for file txt reading
	{
	String line;
	string A = "";
	List<string> lines = new List<string>();
	int count = 0;
	try
	{

```
            // Pass the file path and file name to the StreamReader constructor
            using (StreamReader sr = new StreamReader(str))
            {
                // Continue to read until you reach the end of the file
                while ((line = sr.ReadLine()) != null)
                {
                    foreach (char c in line)
                    {
                        if (c == '-')
                        {
                            count++;
                        }
                        if (c == ',')
                        {
                            // Append A to lines and reset A
                            A = A + c + count;
                        }
                        else if (char.IsLetterOrDigit(c) || c == '(' || c == ')' || c=='/')
//Check for alphanumeric or parentheses
                        {
                            A += c;
                        }
                    }
                    lines.Add(count + A);
                    A = "";
                    count = 0;
                    //write the line to console window
                    Console.WriteLine(line);
                } while (line != null) ;
                //close the file
                sr.Close();
                Console.WriteLine("press any key to continue...\n");
                Console.ReadLine();
                return lines;
            }
        }
        catch (Exception e)
        {
            Console.WriteLine("Exception: " + e.Message);
            return null;
        }
```

	finally
	{
	Console.WriteLine("Executing finally block.");
	}
	}

Print to text file	public static void printToFile(List<Person> randomPerson, string pth)
	{
	List<Person> checkedList = new List<Person>();
	try
	{
	using (StreamWriter wt = new StreamWriter(pth))
	{
	foreach (Person person in randomPerson)
	{
	// Skip if the person's partner has already been checked
	if (person.partner != null && checkedList.Contains(person.partner))
	continue;
	checkedList.Add(person);
	int gender, gender_;
	gender = gender_ = 0;
	if (person.gender) gender = 1;
	if (person.partner.gender) gender_ = 1;
	if (person.bDay != default(DateTime)) wt.Write(new string('-', person.step) + person.name + "(" + person.bDay.Date.ToString("dd/MM/yyyy") + ")" + gender + ",");
	else wt.Write(new string('-', person.step) + person.name + "()" + gender + ",");
	if (person.partner != null)
	{
	if (person.partner.bDay != default(DateTime)) wt.WriteLine(person.partner.name + "(" + person.partner.bDay.Date.ToString("dd/MM/yyyy") + ")" + gender_);
	else wt.WriteLine(person.partner.name + "()" + gender_);
	}
	}
	}
	}
	catch (Exception e)
	{
	Console.WriteLine("Exception: " + e.Message);

	return;
	}
	finally
	{
	Console.WriteLine("Print Complete!!!.");
	Console.ReadKey();
	}
	}
	public static bool birthdayFormatChecking(string bDay) {
	string pattern = @"^(0[1-9]|[12][0-9]|3[01])/(0[1-9]|1[0-2])/\d{4}$";
	
	// Create a Regex object
	Regex regex = new Regex(pattern);
	
	// Check if the input matches the pattern
	if (regex.IsMatch(bDay))
	{
	return true;
	}
	else return false;
	}

Check	- *Birthday Format Checking*
	public static bool birthdayFormatChecking(string bDay) {
	string pattern = @"^(0[1-9]|[12][0-9]|3[01])/(0[1-9]|1[0-2])/\d{4}$";
	
	// Create a Regex object
	Regex regex = new Regex(pattern);
	
	// Check if the input matches the pattern
	if (regex.IsMatch(bDay))
	{
	return true;
	}
	else return false;
	}
	
	public static void dfsTraverse(Person person)
	{
	if (person == null)
	{
	Console.Write($")");

```
            return;
        }
        // Display current person's details
        Console.Write($"( {person.name},");

        // Display partner's details if they exist
        if (person.partner != null)
        {
            Console.Write($" {person.partner.name})");
        } else Console.Write(")");

        // Traverse children
        foreach (var child in person.child)
        {
            Console.Write("->");
            dfsTraverse(child); // Recursive call for each child
        }
    }

    public static void bfsTraverse(Person person)
    {
        if (person == null)
        {
            Console.WriteLine(")");
            return;
        }

        Queue<Person> queue = new Queue<Person>();
        HashSet<Person> visited = new HashSet<Person>(); // Track visited
persons and partners

        queue.Enqueue(person);
        visited.Add(person);

        Console.WriteLine("("); // Start traversal

        while (queue.Count > 0)
        {
            Person current = queue.Dequeue();

            // Display current person's details
```

```
            Console.Write($"( {current.name}");

            // Display partner's details if they exist and haven't been visited
            if (current.partner != null && !visited.Contains(current.partner))
            {
                Console.Write($", {current.partner.name}");
                visited.Add(current.partner); // Mark partner as visited
            }
            Console.Write(")->");

            // Enqueue each child to process in the next level
            foreach (var child in current.child)
            {
                if (!visited.Contains(child)) // Process only if not visited
                {
                    queue.Enqueue(child);
                    visited.Add(child); // Mark child as visited
                }
            }
        }
        Console.WriteLine(")"); // End traversal
    }
```

- Check the relationship between two members

```
public static void DetermineRelationship(Person person1, Person person2)
    {
        if (person1.child.Contains(person2))
        {
            Console.WriteLine($"{person1.name} is a parent of
{person2.name}.");
        }
        else if (person2.child.Contains(person1))
        {
            Console.WriteLine($"{person1.name} is a child of
{person2.name}.");
        }
        else if ((person1.parent != null && (person1.parent[0] == person2 ||
person1.parent[1] == person2)) ||
                 (person2.parent != null && (person2.parent[0] == person1 ||
person2.parent[1] == person1)))
        {
```

	Console.WriteLine($"{person1.name } and {person2.name} are siblings.");
	}
	else
	{
	Console.WriteLine($"{person1.name} and {person2.name} have no direct relationship.");
	}
	}

Family tree	- *Create information of members*
	public static List<Person> fromStringCreateInfo(List<String> lines)
	{
	Person tmp = new Person();
	Person tmp_ = new Person();
	List<Person> list = new List<Person>();
	int step;
	string name, dob;
	bool gender;
	int flag = 0;
	foreach (string line in lines)
	{
	name = "";
	dob = "";
	flag = 0;
	for (int i = 0; i < line.Length; i++)
	{
	if (line[i] == ',')
	{
	for (int j = 1; j < i; j++)
	{
	if (line[j] == ')') break;
	if (flag != 0) dob += line[j];
	if (line[j] != '(')
	{
	if (flag == 0) name += line[j];
	}
	else flag = j;
	}
	if (line[i - 1] - '0' == 1) gender = true; else gender = false;
	if (dob != "")
	{

```
                    DateTime birthDay = DateTime.Parse(dob, new
CultureInfo("en-GB"));
                    tmp = new Person(name, birthDay, line[0] - '0', gender);
                }
                else tmp = new Person(name, line[0] - '0', gender);
                //reset for next person
                if (line[i + 1] == null) break;
                name = "";
                dob = "";
                flag = 0;
                gender = false;
                for (int m = line.Length - 3; m > i + 1; m--)
                {
                    if (flag != 0) name = line[m] + name;
                    if (line[m] != '(')
                    {
                        if (flag == 0) dob = line[m] + dob;
                    }
                    else flag = m;
                }
                if (line[line.Length - 1] - '0' == 1) gender = true;
                if (dob != "")
                {
                    DateTime birthDay = DateTime.Parse(dob, new
CultureInfo("en-GB"));
                    tmp_ = new Person(name, birthDay, line[i + 1] - '0', gender);
                }
                else tmp_ = new Person(name, line[i + 1] - '0', gender);
                tmp_.partner = tmp;
                tmp.partner = tmp_;
                list.Add(tmp);
                list.Add(tmp_);

                //add Partner for each line


                break;
            }
        }
    }
    return list;
```

```
    }

- Print list of members
public static void printList(List<Person> randomPerson)
    {
        if (randomPerson.Count == 0) return;

Console.WriteLine("=====================================");
        foreach (var person in randomPerson) {
            Console.WriteLine("Name: " + person.name + "| | ID: " +
person.PersonID);
        }
    }

public static void printListPerson(List<Person> randomPerson)
{
   foreach (Person person in randomPerson)
   {
      Console.WriteLine(person.name + " " +
person.bDay.ToString("dd/MM/yyyy") + " " + person.PersonID);
   }
}
- Create Family Tree
public static List<Person> CreateTreeFromList(List<Person> list)
    {
        List<Person> checkedList = new List<Person>();
        Person tmp = new Person();
        foreach (var person in list)
        {
            // Skip if the person's partner has already been checked
            if (person.partner != null && checkedList.Contains(person.partner))
                continue;

            checkedList.Add(person);

            // Loop through checkedList to find and set parents
            foreach (var checkedPerson in checkedList)
            {
                if (checkedPerson.step == person.step - 1)
                {
                    AddParent(person, checkedPerson);
```

	RemoveParent(person, checkedPerson);
	tmp = checkedPerson;
	//Person[] parent_ = new Person[2];
	//parent_[0] = checkedPerson;
	//parent_[1] = checkedPerson.partner;
	//person.parent = parent_;
	//person.partner.parent = parent_;
	}
	}
	tmp.addChild(person);
	tmp.addChild(person.partner);
	}
	return list;
	}

Tree traverse	- *DFS Traverse*
	public static void dfsTraverse(Person person)
	{
	if (person == null)
	{
	Console.Write($")");
	return;
	}
	// Display current person's details
	Console.Write($"({person.name},");
	
	// Display partner's details if they exist
	if (person.partner != null)
	{
	Console.Write($" {person.partner.name})");
	} else Console.Write(")");
	
	// Traverse children
	foreach (var child in person.child)
	{
	Console.Write("->");
	dfsTraverse(child); // Recursive call for each child
	}
	}
	
	- *BFS Traverse*
	public static void bfsTraverse(Person person)

```
    {
        if (person == null)
        {
            Console.WriteLine(")");
            return;
        }

        Queue<Person> queue = new Queue<Person>();
        HashSet<Person> visited = new HashSet<Person>(); // Track visited
persons and partners

        queue.Enqueue(person);
        visited.Add(person);

        Console.WriteLine("("); // Start traversal

        while (queue.Count > 0)
        {
            Person current = queue.Dequeue();

            // Display current person's details
            Console.Write($"( {current.name}");

            // Display partner's details if they exist and haven't been visited
            if (current.partner != null && !visited.Contains(current.partner))
            {
                Console.Write($", {current.partner.name}");
                visited.Add(current.partner); // Mark partner as visited
            }
            Console.Write(")->");

            // Enqueue each child to process in the next level
            foreach (var child in current.child)
            {
                if (!visited.Contains(child)) // Process only if not visited
                {
                    queue.Enqueue(child);
                    visited.Add(child); // Mark child as visited
                }
            }
        }
```

	Console.WriteLine(")"); // End traversal
	}

3.4. Interface

Main UI	public static int selectUI()
	{
	int mode;
	string tmp;
	do
	{
	Console.Clear();
	
	Console.WriteLine("\n==========");
	Console.Write("Mode Selection:" +
	"\n1. Add Person" +
	"\n2. Update Person" +
	"\n3. Delete Person" +
	"\n4. Print Tree" +
	"\n5. Print File" +
	"\n6. Read File" +
	"\n7. Create Tree" +
	"\n8. Check Relationship" +
	"\n\nSelect: ");
	tmp = Console.ReadLine();
	if (tmp.Length < 2 && tmp != "")
	{
	if (Convert.ToChar(tmp) > 47 && Convert.ToChar(tmp) < 57)
	{
	mode = Convert.ToInt32(tmp);
	continue;
	}
	}
	Console.WriteLine("Please Select Again!" +
	"\nPress any key to try again!");
	Console.ReadKey();
	mode = -1;
	} while (mode < 0 || mode > 8);
	return mode;
	}

Add Person UI	public static Person addPersonUI() { Console.WriteLine("\n==========" +

	"\n/To Exit Type Exit in name/");
	Person person = new Person();
	Console.Write("Add member:\n" +
	"Name: ");
	person.name = Console.ReadLine();
	Console.Write("bDay: ");
	string bDay = "dd/mm/yyyy";
	do
	{
	bDay = Console.ReadLine();
	} while(!Functions.birthdayFormatChecking(bDay));
	person.bDay = DateTime.Parse(bDay, new CultureInfo("en-GB"));
	return person;
	}

Remove Person UI	public static Person removePersonUI()
	{
	Console.WriteLine("\n====================" +
	"\n/To Exit Type Exit in name/");
	Console.Write("Remove member:\n" +
	"Name: ");
	string nameToRemove = Console.ReadLine();
	if(nameToRemove.Equals("Exit",
	StringComparison.OrdinalIgnoreCase))
	return null;
	
	Person personToRemove = person.FirstOrDefault(p =>
	p.name.Equals(nameToRemove, StringComparison.OrdinalIgnoreCase));
	if (personToRemove != null)
	{
	person.Remove(personToRemove);
	return personToRemove; // Return the removed person
	}
	else
	{
	Console.WriteLine("Person not found.");
	return null; // Return null if no person was found
	}
	}

Select Person UI	public static Person selectPersonUI(List<Person> list) {

	Console.WriteLine("\n==========");
	if (list.Count <= 0) return null;
	Console.Write("Select Person: " +
	"\nPersonID:");
	string pID = Console.ReadLine();
	foreach (Person person in list)
	{
	if (person.PersonID == pID) return person;
	}
	Console.WriteLine("No Person Was Found!");
	return null;
	}

Confirm User	public static bool confirmationUser(string str)
	{
	string txt = "";
	do
	{
	Console.WriteLine("Want to " + str + " (y or n)");
	txt = Console.ReadLine();
	txt.ToLower();
	} while (txt != "y" && txt != "n");
	if (txt == "n") return false;
	return true;
	}

Selection part	public static string partSelection()
	{
	string tmp = "";
	do
	{
	tmp = Console.ReadLine();
	tmp = tmp.ToLower().Trim();
	if (tmp == "dob") break;
	if (tmp == "name") break;
	} while (true);
	return tmp;
	}

3.5. Main Program

Initializing	string str = "<Your_Tree.txt_Path>"; UI program = new UI();

	List<Person> randomPerson = new List<Person>(); List<String> list = new List<String>(); bool state = true; Person p1 = new Person(); Person p2 = new Person();

Options	while (state) {
	switch (UI.selectUI())
	{
	case 1: // add a person
	Functions.printList(randomPerson);
	int mode_;
	p1 = UI.addPersonUI();
	if (randomPerson.Count == 0)
	{
	Console.WriteLine("Add Completed!\n");
	break;
	}
	
	Console.WriteLine("==========" +
	"\nWhat is the role of this person?\n");
	string role = "";
	do
	{
	role = Console.ReadLine();
	role = role.Trim().ToLower();
	if (UI.confirmationUser(role))
	{
	if (role == "partner") // add connection
	{
	Console.WriteLine("select This Person Partner:\n");
	p2 = UI.selectPersonUI(randomPerson);
	if (p1 == null || p2 == null) break;
	p1.addPersonConnection(p2);
	p2.addPersonConnection(p1);
	break;
	}
	else if (role == "child") // add child
	{
	mode_ = 0;
	do
	{
	Console.WriteLine("PARENT\n");
	p2 = UI.selectPersonUI(randomPerson);
	if (p2 == p1)

```
                    {
                        Console.WriteLine("Can not be the same Person!\n");
                        mode_ = 1;
                    }
                    else if (p2.partner == p1)
                    {
                        Console.WriteLine("Can not be both role!\n");
                        mode_ = 1;
                    }
                } while (mode_ == 1);

                p2.addChild(p1);
            }
        }
    } while (role != "child" || role != "partner");
    randomPerson.Add(p1);
    break;
case 2: // update a Person
    Console.WriteLine("/Update Person Information/");
    Functions.printList(randomPerson);
    p1 = UI.selectPersonUI(randomPerson);
    Console.WriteLine("Choose The Part you want to update: ");
    if (UI.partSelection() == "name") {
        string newName = Console.ReadLine();
        p1.name = newName.Trim();
    }else if (UI.partSelection() == "dob")
    {
        string newDob = Console.ReadLine();
        DateTime birthDay = DateTime.Parse(newDob, new
CultureInfo("en-GB"));
        p1.bDay = birthDay;
    }
    break;
case 3: // remove a person
    Functions.printList(randomPerson);
    p1 = UI.removePersonUI();
    if (randomPerson.Count == 0)
    {
        Console.WriteLine("Remove Completed!\n");
        break;
    }

Console.WriteLine("==================================" +
        "\nWhat is the role of this person?\n");
```

```
            role = " ";
            do
            {
               role = Console.ReadLine();
               role = role.Trim().ToLower();
               if (UI.confirmationUser(role))
               {
                  if (role == "partner") // remove connection
                  {
                     Console.WriteLine("select This Person Partner:\n");
                     p2 = UI.selectPersonUI(randomPerson);
                     if (p1 == null || p2 == null) break;
                     p1.removePersonConnection(p2);
                     p2.removePersonConnection(p1);
                     break;
                  }
                  else if (role == "child") // add child
                  {
                     mode_ = 0;
                     do
                     {
                        Console.WriteLine("PARENT\n");
                        p2 = UI.selectPersonUI(randomPerson);
                        if (p2 == p1)
                        {
                           Console.WriteLine("Can not be the same Person!\n");
                           mode_ = 1;
                        }
                        else if (p2.partner == p1)
                        {
                           Console.WriteLine("Can not be both role!\n");
                           mode_ = 1;
                        }
                     } while (mode_ == 1);

                     p2.removeChild(p1);
                  }
               }
            } while (role != "child" || role != "partner");
            randomPerson.Remove(p1);
            break;
         case 4: // print tree
            Functions.printListPerson(randomPerson);
            if (randomPerson.Count == 0)
```

```
            {
                Console.WriteLine("No Person in list yet!\n");
                Console.ReadKey();
                break;
            }
            p1 = UI.selectPersonUI(randomPerson);
            do
            {
                Console.Write("Select dfs or bfs:");
                string typeTraverse = Console.ReadLine();
                typeTraverse = typeTraverse.Trim().ToLower();
                if (typeTraverse == "dfs")
                {
                    Functions.dfsTraverse(p1);
                    Console.ReadKey();
                    break;
                }
                else if (typeTraverse == "bfs")
                {
                    Functions.bfsTraverse(p1);
                    Console.ReadKey();
                    break;
                }
            } while (true);
            break;
        case 5: // print file
            Functions.printToFile(randomPerson,
"C:\\Users\\admin\\Downloads\\Học tập\\Discrete\\discrete\\tree.txt");
            break;
        case 6: //read file
            list = Functions.readFile(str);
            randomPerson = Functions.fromStringCreateInfo(list);
            foreach(Person person in randomPerson)
            {
                Console.WriteLine(person.name + " " +
person.bDay.ToString("dd/MM/yyyy") + " " + person.step + " " +
person.partner.name);
            }
            Console.ReadKey();
            break;
        case 7: //create tree
            randomPerson = Functions.CreateTreeFromList(randomPerson);
            Console.WriteLine("complete!");
            Console.ReadKey();
```

```
            break;
        case 8: //check relationshop
            Functions.printList(randomPerson);
            p1 = UI.selectPersonUI(randomPerson);
            p2 = UI.selectPersonUI(randomPerson);
            if(p1 == p2)
            {
                Console.WriteLine("Is the same Person");
                Console.ReadKey();
                break;
            }
            Functions.DetermineRelationship(p1, p2);
            Console.ReadKey ();
            break;
        case 0:
            state = false;
            break;
        default:
            Console.WriteLine("please enter a mode!\n");
            continue;
            break;
    }
}
return;
```

4. Implementation, Test Cases, Results, and Discussion

The project will proceed in phases, beginning with developing the family tree's core features, followed by constructing the user interface and optimizing the system's data-handling capabilities.

4.1. Main menu

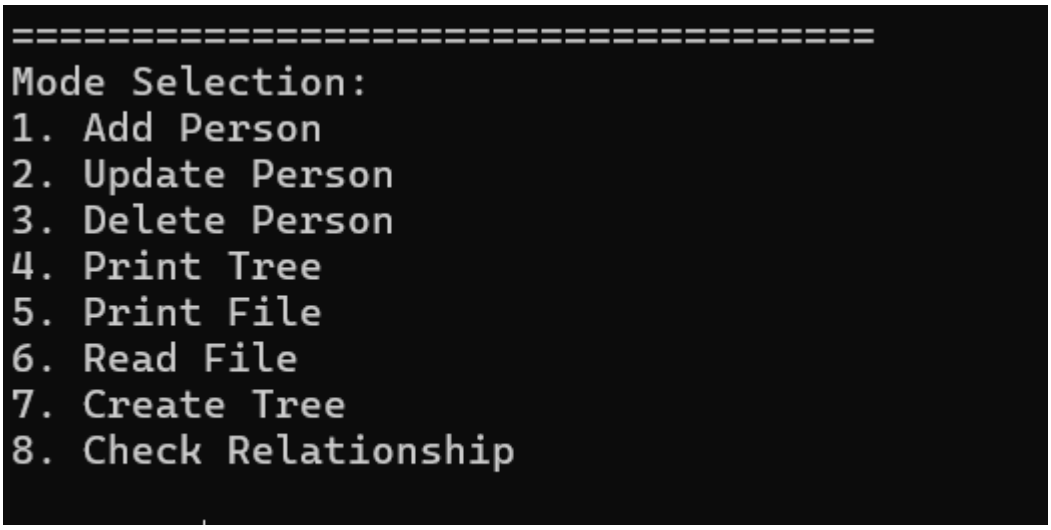

Figure 6: Interface when first running the program

4.2. Add Person

When selecting the add person function, the program will ask to enter the information of that member. After finishing the data entry, the program will print the information back into the text file and create a new tree.

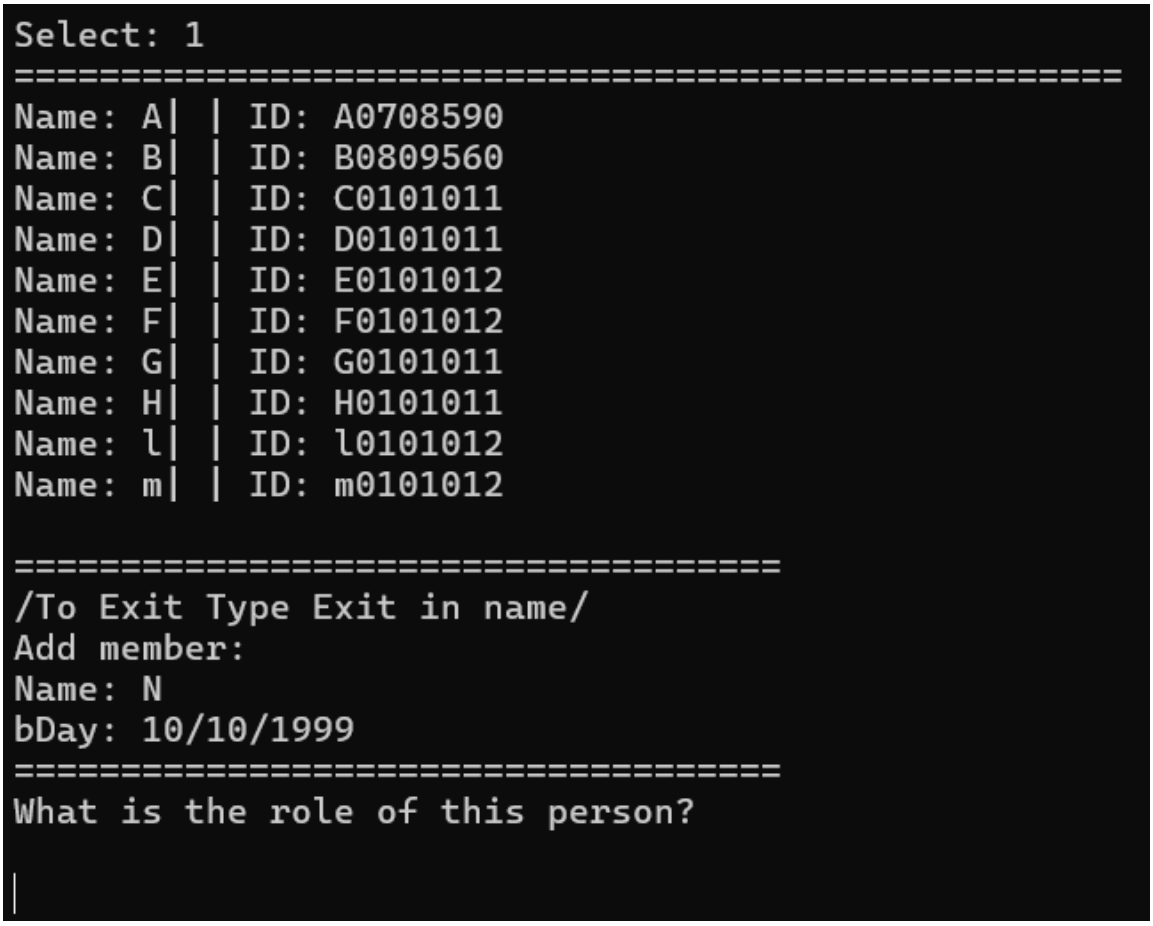

Figure 7: Interface of Add Person function

4.3. Update Person

When selecting the update person function, the program will ask you to enter the part that needs to be updated. After filling in the information that needs to be updated, the program will print the information back into the text file.

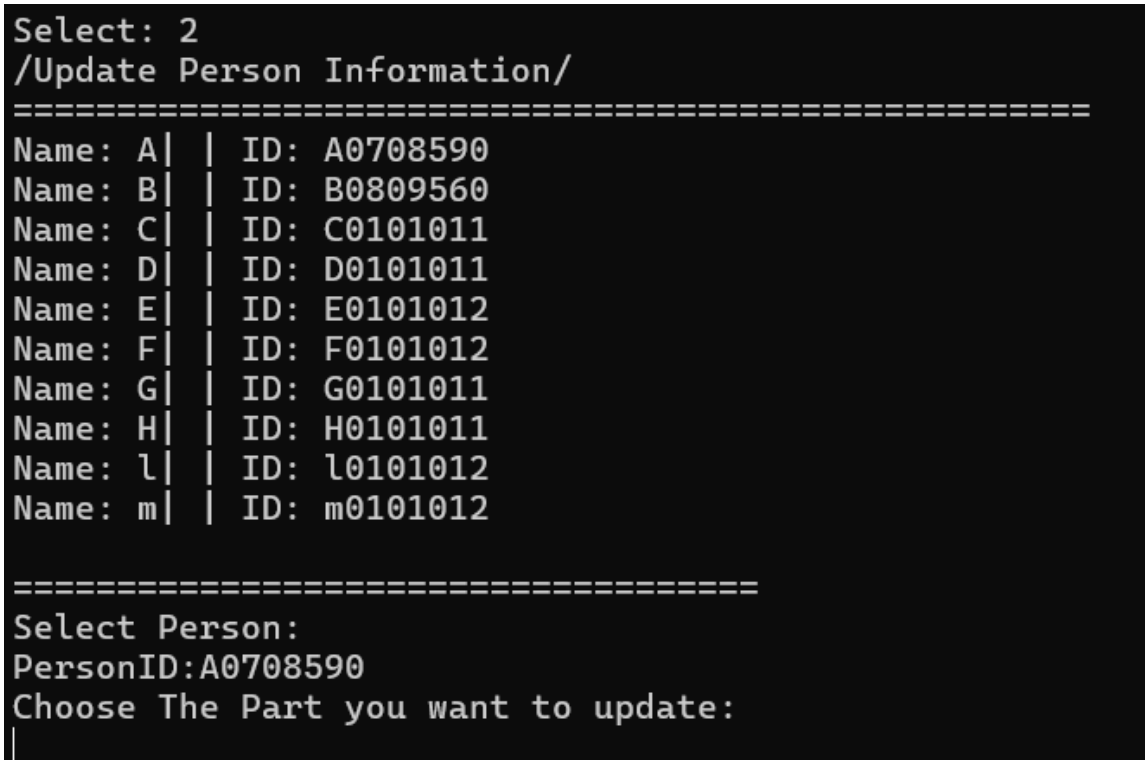

Figure 8: Interface of Update Person function

4.4. Delete Person

When selecting delete person function, the program will remove that member from the list and recreate the family tree.

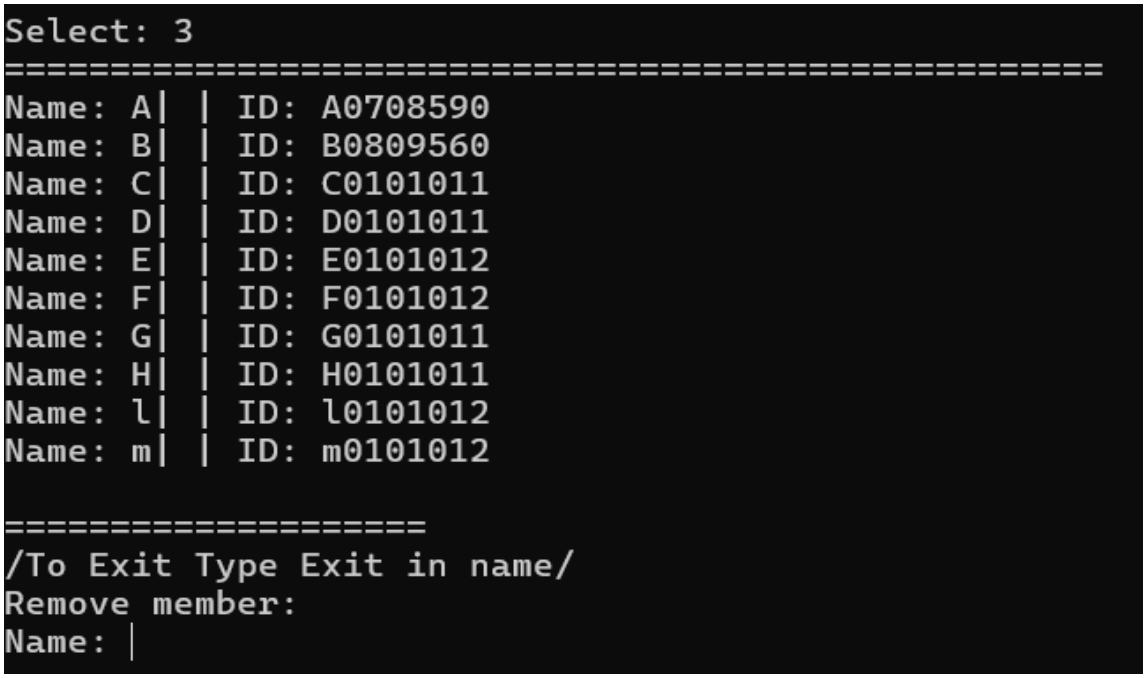

Figure 9: Interface of Remove Person function

4.5. Print Tree

- Using DFS Traverse

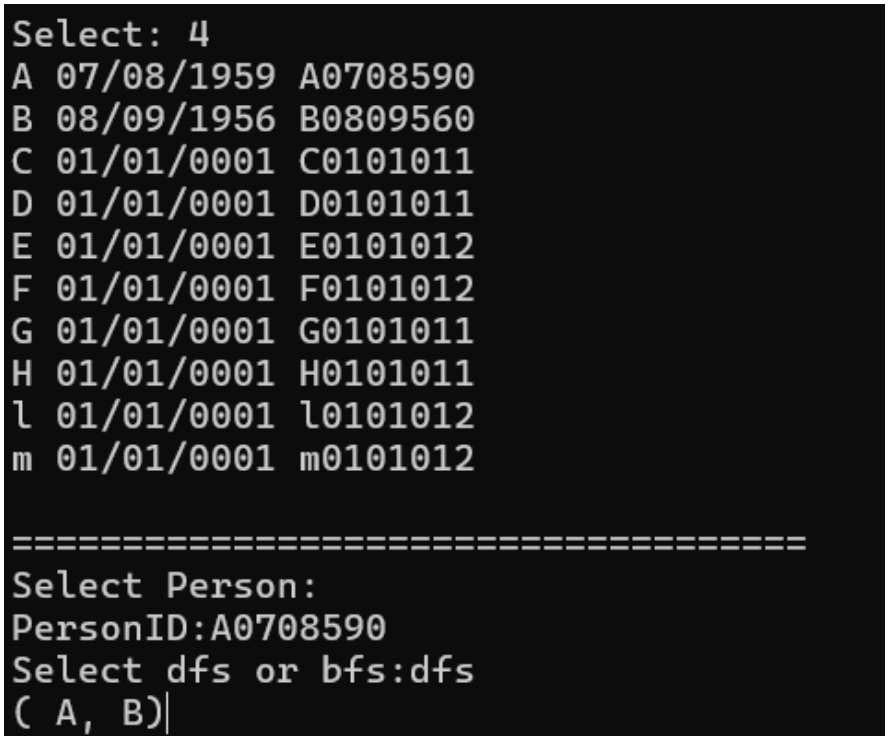

Figure 10: Interface of Print Tree function using DFS Traverse

- Using BFS Traverse

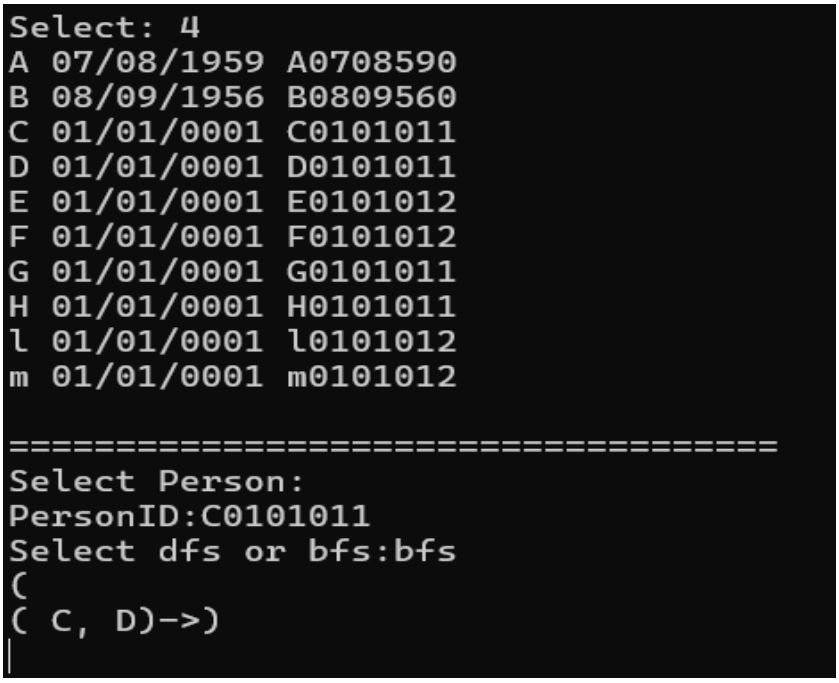

Figure 11: Interface of Print Tree function using BFS Traverse

4.6. Read File, Print File and Create Tree

The program will read, print the text file and create a tree based on it, only if the file is written in the correct structure.

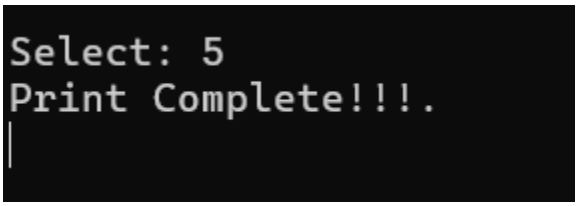

Figure 12: Interface of Print File funtion

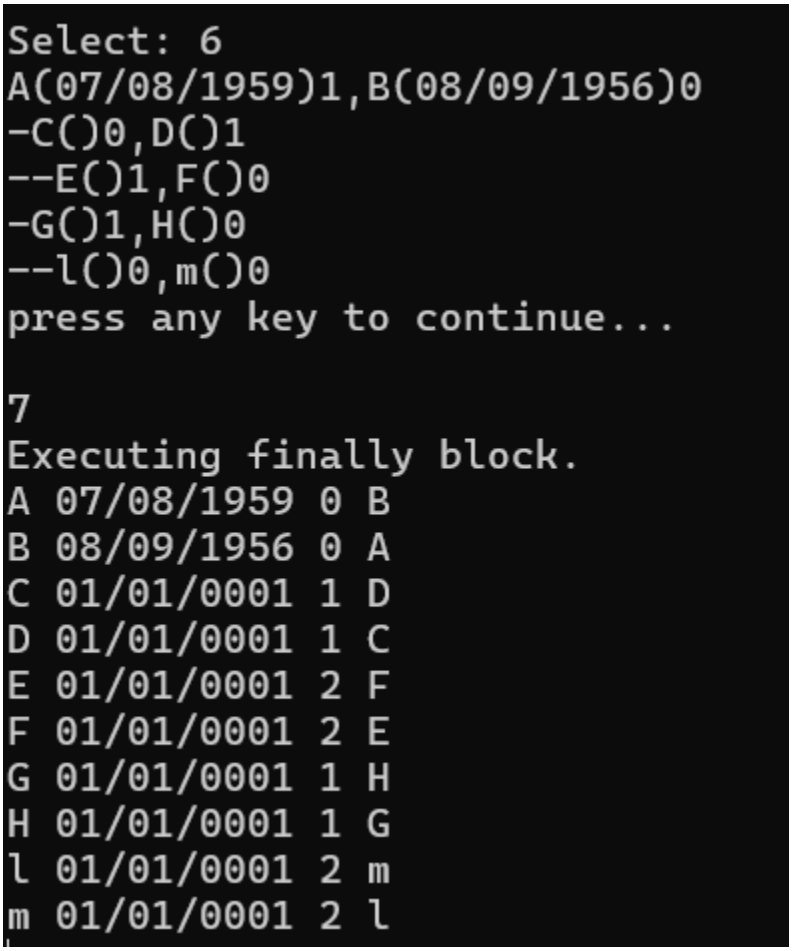

Figure 13: Interface of Read File and Create Tree funtion

4.7. Check Relationship between two members

When selecting Determine Relationship function, the program will check if two members have relationship or not and return the result on the interface.

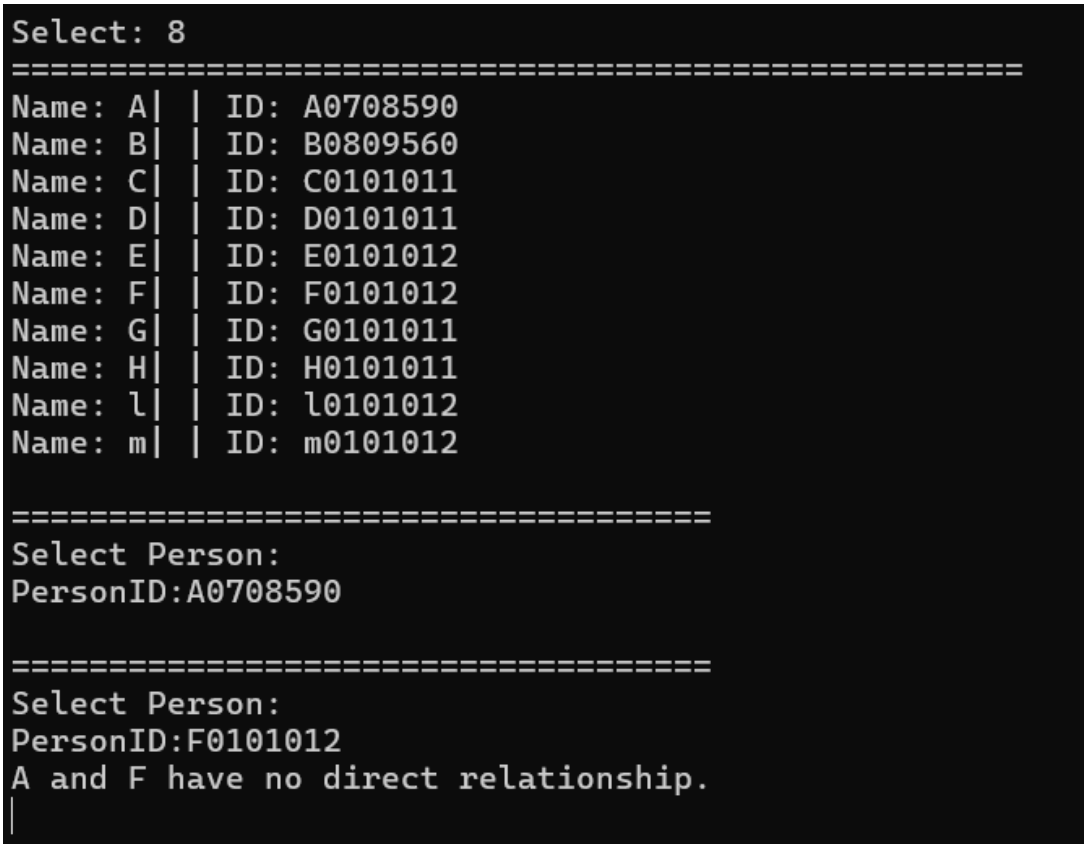

Figure 14: Interface of Determine Relationship function

C. CONCLUSION

1. Conclusion

Summary:

The project team successfully addressed the primary requirements set out for the Family Tree Management Program, while also integrating additional useful features such as conditional search, sorting family members by various criteria, and enabling personal information updates for each family member. The program offers users an organized, efficient way to store and explore their family history digitally.

Strengths:

- ***User-friendly Interface:*** The program has a streamlined, intuitive interface that is easy to navigate, making it accessible for users of all technical levels.

- ***Performance Stability:*** After extensive testing, the program demonstrated consistent accuracy, free from crashes, errors, or lag. This stability ensures the program reliably meets user expectations for quick and accurate data handling.

- ***Incorporation of Algorithms:*** The application leverages algorithms effectively, applying relevant techniques from the team's studies to optimize performance.

Limitations:

- ***Limited Search Options*:** The search functionality could benefit from additional filter options to enhance flexibility and user control.

- ***Sorting Options:*** While sorting is functional, there is room for improvement in expanding sorting criteria to further customize data views.

- ***Optimization for Large Data Sets:*** As the program handles increasing amounts of family information, there are opportunities to refine and optimize data processing for greater efficiency.

2. Future Development Directions

Future enhancements may include integrating cloud storage, allowing for remote access, adding multimedia support, and offering advanced data-sharing options among family members.

REFERENCES

1. Geeks for Geeks – *Introduction to Tree Data Structure*, last updated in 22 Oct, 2024. https://www.geeksforgeeks.org/introduction-to-tree-data-structure/
2. Science Direct – *Family System Theory*, last updated in 2018. https://www.sciencedirect.com/topics/medicine-and-dentistry/family-systems-theory#definition
3. Geeks for Geeks – *Introduction to C#*, last updated in 23 Mar, 2023. https://www.geeksforgeeks.org/introduction-to-c-sharp/
4. Anastazija Spasojevic – *What is a text file?* on Phoenix NAP website, last updated in 29 Apr, 2024. https://phoenixnap.com/glossary/what-is-a-text-file#:~:text=A%20text%20file%20is%20a,edited%20using%20basic%20text%20editors
5. Geeks for Geeks – *Difference between BFS and DFS*, last updated in 18 Oct, 2024. https://www.geeksforgeeks.org/difference-between-bfs-and-dfs/