SVKM's NMIMS

Mukesh Patel School of Technology Management & Engineering (Mumbai Campus)

Computer Engineering Department (B.Tech Integrated Sem VII)

Data Structures and Algorithm

Mini Project

PART A

(Part A: TO BE REFFERED BY STUDENTS)

A.1 AIM:

Identify a problem statement and design a solution using Non-Linear Data Structure

A.2 Pre requisite:

Basic Knowledge of Data Structures

A.3 Outcome:

After successful completion of this experiment students will be able to:

Develop a solution for a problem using non-linear data structure

A.4 Theory:

A mini project is desirable to be completed by a group of three or four students

A.5 Procedure/Task:

1. Identify a real life problem and design a solution using non-linear data structure
2. Prepare the document. Save and close the file and name it as RollNo._MiniProject.

SVKM's NMIMS

Mukesh Patel School of Technology Management & Engineering (Mumbai Campus)

Computer Engineering Department (B.Tech Integrated Sem VII)

Data Structures and Algorithm

Mini Project

PART B

(PART B: TO BE COMPLETED BY STUDENTS)

(Students must submit the soft copy as per following segments within two hours of the practical. The soft copy must be uploaded on the Blackboard or emailed to the concerned lab in charge faculties at the end of the practical in case the there is no Black board access available)

Roll No.: C044	Name: Amishi Desai
Class: B	Batch: B2
Date of Experiment: 31/10/23	Date/Time of Submission: 31/10/23
Grade :	

B.1 Project Details

Problem Statement: Family Tree Generator

Develop a C++ program for building and visualizing a hierarchical family tree. The program should allow users to interactively add family members, specify their parent-child relationships, and visualize the family tree's structure. The program must handle scenarios where users attempt to add a family member to a specified parent who may or may not exist within the tree.

Constraints:

- Each individual in the family tree is represented by their name, birthdate, and a list of their children.
- The family tree begins with a single root individual.
- The program should support adding and visualizing multiple generations in the family tree.
- Visualization should use ASCII-based hierarchy representation for clarity.
- The program should provide user-friendly input validation and error messages for invalid operations

Motivation to take the proposed problem statement

The motivation behind creating a family tree generator in C++ stems from several factors:

1. Educational Purposes: Developing a family tree generator serves as a valuable learning exercise, gives understanding of data structures and their practical implementations. It provides an opportunity to apply concepts of tree data structures in a real-world context, thereby enhancing the understanding of data organization and management.

2. Problem-Solving and Algorithmic Thinking: Tackling the challenges involved in designing a family tree generator encourages the development of problem-solving skills and the cultivation of algorithmic thinking. It necessitates the consideration of various scenarios, such as handling user input, managing data relationships, and effectively visualizing hierarchical structures.

3. User-Friendly Interface: Emphasizing a user-friendly interface allows individuals with limited technical knowledge to easily utilize the program. By providing clear instructions and feedback, the program becomes accessible to a wider audience, including those with minimal programming experience or familiarity with complex data structures.

4. Personal and Cultural Significance: Family trees hold great significance for many individuals and communities, as they provide a means to preserve and document familial relationships across generations. Creating a program to generate and visualize family trees can help individuals understand their ancestry better and appreciate the importance of preserving family history.

5. Practical Application in Genealogy: The program has practical applications in genealogy and historical research. By facilitating the creation and visualization of family trees, the program can aid genealogists and historians in organizing and analyzing complex familial relationships, thus contributing to the preservation of historical records and the study of family histories.

Overall, the creation of a family tree generator combines elements of technical education, personal heritage preservation, historical research, and user-centric design, making it a meaningful and engaging project for both developers and users alike.

Solution description

Algorithm:

Step 1: Define the `Individual` class with private attributes for name, birthdate, and a vector of children. Include methods for adding a child, getting the name, birthdate, and children.

Step 2: Define the `FamilyTree` class with a private attribute for the root individual. Include methods for adding a child, visualizing the family tree, and getting the root.

Step 3: In the `main` function, create an instance of the `FamilyTree` class with a root name and birthdate.

Step 4: Create a vector of `Individual*` called `individuals` and add the root individual to it.

Step 5: Enter a loop to display the main menu and handle user input. Repeat until the user chooses to exit.

Step 6: Inside the loop, display the main menu options:

- Add Family Member
- Visualize Family Tree
- Exit

Step 7: Prompt the user to enter their choice.

Step 8: If the user selects "Add Family Member":

- Prompt the user for the parent's name, child's name, and child's birthdate.
- Search for the parent in the `individuals` vector to find the individual to whom the child will be added.
- Create a new `Individual` for the child and add it as a child of the parent.
- Add the child to the `individuals` vector.
- Display a success message.

Step 9: If the user selects "Visualize Family Tree":

- Call the `visualizeTree` method on the root individual to display the family tree hierarchy.

Step 10: If the user selects "Exit":

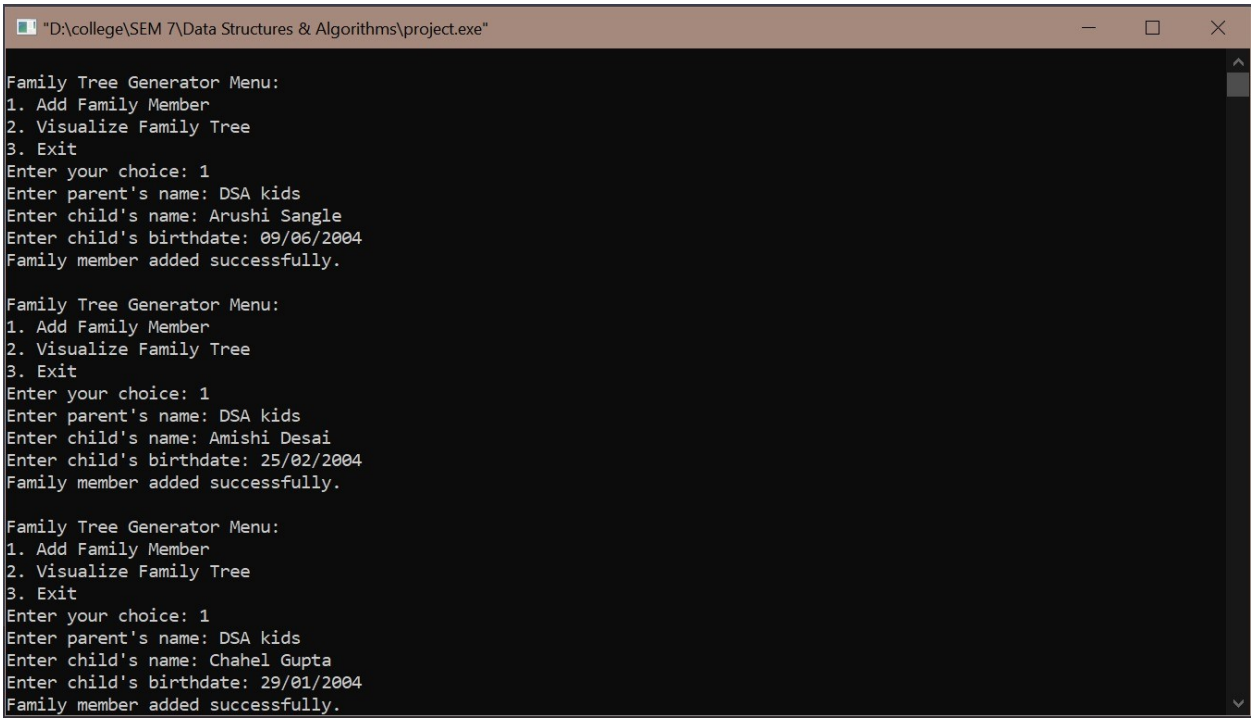
- Display a goodbye message.
- Break out of the loop to exit the program.

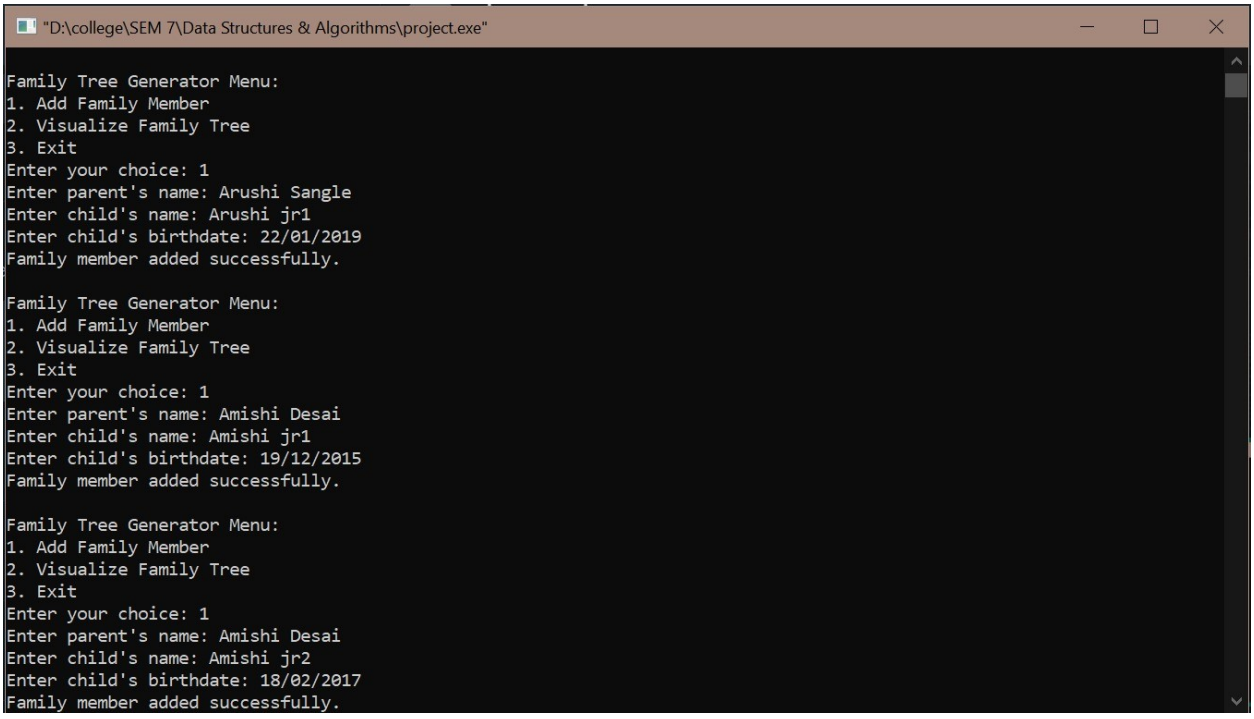
Step 11: If the user enters an invalid choice, display an error message.

Step 12: Repeat the loop until the user chooses to exit.

This algorithm outlines the steps to create and interact with a family tree using the provided code. Users can add family members, visualize the family tree, and exit the program as needed.

B.2 Output


"D:\college\SEM 7\Data Structures & Algorithms\project.exe"
Family Tree Generator Menu:
1. Add Family Member
2. Visualize Family Tree
3. Exit
Enter your choice: 1
Enter parent's name: DSA kids
Enter child's name: Arushi Sangle
Enter child's birthdate: 09/06/2004
Family member added successfully.
Family Tree Generator Menu:
1. Add Family Member
2. Visualize Family Tree
3. Exit
Enter your choice: 1
Enter parent's name: DSA kids
Enter child's name: Amishi Desai
Enter child's birthdate: 25/02/2004
Family member added successfully.
Family Tree Generator Menu:
1. Add Family Member
2. Visualize Family Tree
3. Exit
Enter your choice: 1
Enter parent's name: DSA kids
Enter child's name: Chahel Gupta
Enter child's birthdate: 29/01/2004
Family member added successfully.


"D:\college\SEM 7\Data Structures & Algorithms\project.exe"
Family Tree Generator Menu:
1. Add Family Member
2. Visualize Family Tree
3. Exit
Enter your choice: 1
Enter parent's name: Arushi Sangle
Enter child's name: Arushi jr1
Enter child's birthdate: 22/01/2019
Family member added successfully.
Family Tree Generator Menu:
1. Add Family Member
2. Visualize Family Tree
3. Exit
Enter your choice: 1
Enter parent's name: Amishi Desai
Enter child's name: Amishi jr1
Enter child's birthdate: 19/12/2015
Family member added successfully.
Family Tree Generator Menu:
1. Add Family Member
2. Visualize Family Tree
3. Exit
Enter your choice: 1
Enter parent's name: Amishi Desai
Enter child's name: Amishi jr2
Enter child's birthdate: 18/02/2017
Family member added successfully.

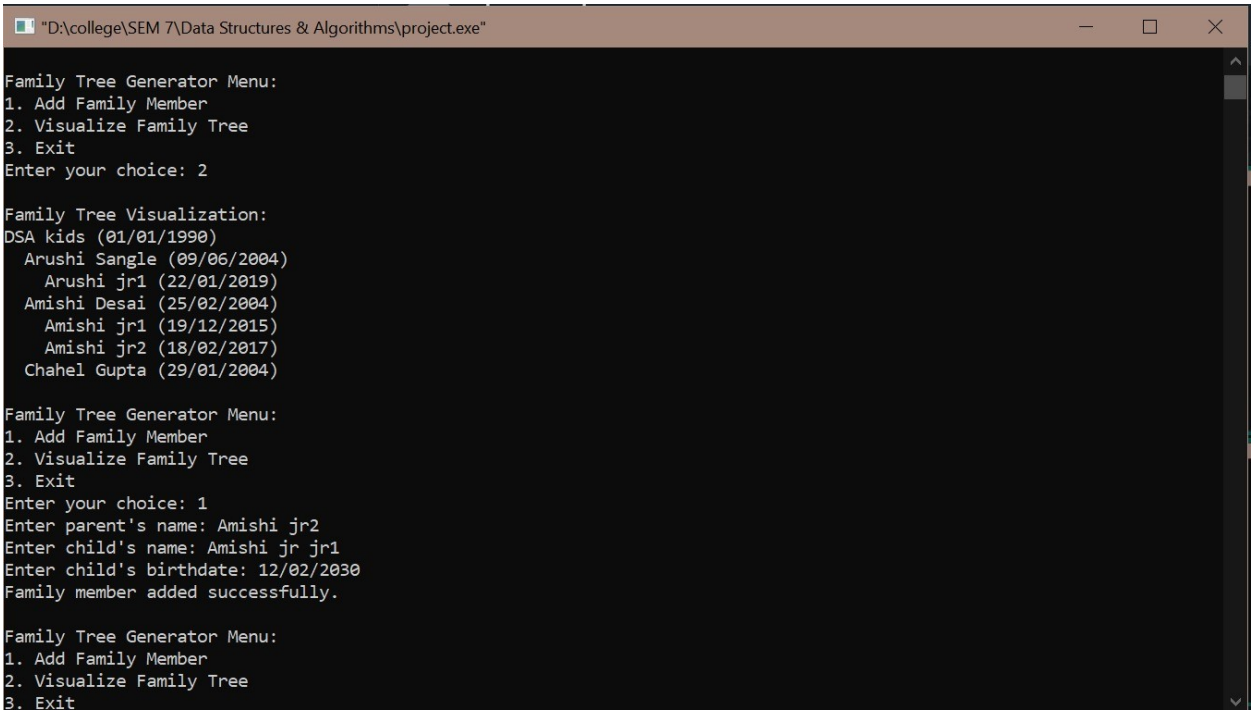
SVKM's NMIMS

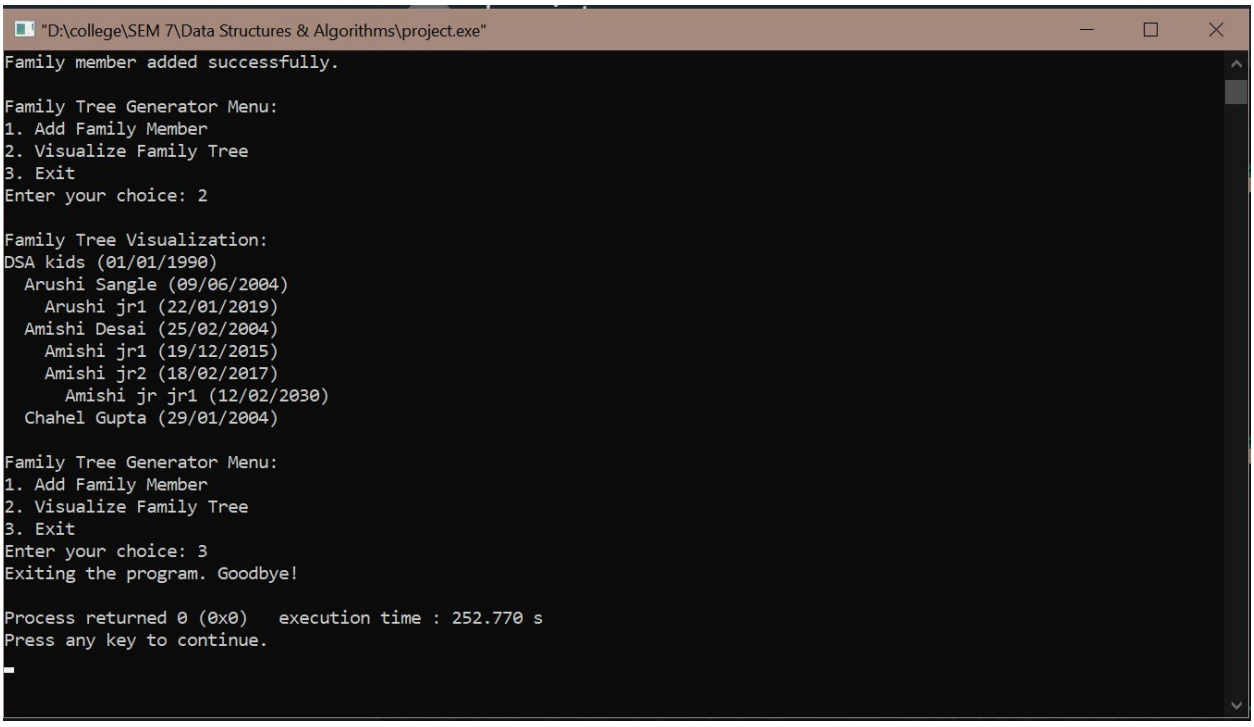
Mukesh Patel School of Technology Management & Engineering (Mumbai Campus)

Computer Engineering Department (B.Tech Integrated Sem VII)

Data Structures and Algorithm

Mini Project





B.3 Project Code:

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

class Individual {
private:
    string name;
    string birthdate;
    vector<Individual*> children;

public:
    Individual(string n, string b) : name(n), birthdate(b) {}

    void addChild(Individual* child) {
        children.push_back(child);
    }

    string getName() const {
        return name;
    }

    string getBirthdate() const {
```

```
    return birthdate;
  }

  const vector<Individual*>& getChildren() const {
    return children;
  }
};

class FamilyTree {
private:
  Individual* root;

public:
  FamilyTree(string rootName, string rootBirthdate) {
    root = new Individual(rootName, rootBirthdate);
  }

  void addChild(Individual* parent, Individual* child) {
    parent->addChild(child);
  }

  void visualizeTree(const Individual* node, int depth) {
    if (node == nullptr) return;
```

```
        for (int i = 0; i < depth; i++) {
            cout << "  ";  // Indentation for a visual hierarchy
        }
        cout << node->getName() << " (" << node->getBirthdate() << ")" << endl;

        for (const Individual* child : node->getChildren()) {
            visualizeTree(child, depth + 1);
        }
    }

    Individual* getRoot() const {
        return root;
    }
};

int main() {
    FamilyTree family("DSA kids", "01/01/1990");
    vector<Individual*> individuals;

    individuals.push_back(family.getRoot());

    while (true) {
        cout << "\nFamily Tree Generator Menu:" << endl;
        cout << "1. Add Family Member" << endl;
```

```
cout << "2. Visualize Family Tree" << endl;
cout << "3. Exit" << endl;

int choice;
cout << "Enter your choice: ";
cin >> choice;

if (choice == 1) {
    string parentName, childName, childBirthdate;

    cout << "Enter parent's name: ";
    cin.ignore();
    getline(cin, parentName);
    cout << "Enter child's name: ";
    getline(cin, childName);
    cout << "Enter child's birthdate: ";
    getline(cin, childBirthdate);

    bool parentFound = false;
    for (Individual* parent : individuals) {
        if (parent->getName() == parentName) {
            Individual* child = new Individual(childName, childBirthdate);
            family.addChild(parent, child);
            individuals.push_back(child);
```

```
                parentFound = true;
                cout << "Family member added successfully." << endl;
                break;
            }
        }

        if (!parentFound) {
            cout << "Parent not found in the family tree. Please check the name." << endl;
        }
    } else if (choice == 2) {
        cout << "\nFamily Tree Visualization:" << endl;
        family.visualizeTree(family.getRoot(), 0);
    } else if (choice == 3) {
        cout << "Exiting the program. Goodbye!" << endl;
        break;
    } else {
        cout << "Invalid choice. Please select a valid option." << endl;
    }
}

    return 0;
}
```

B.4 Conclusion:

This project provides a practical example of how to implement a family tree generator in C++. It showcases the power of object-oriented programming and data structures to represent and manipulate hierarchical data structures. This code can serve as a foundation for more complex genealogy applications and offers a useful reference for those looking to build similar family tree-related software.

B.5 Observations and Learning:

This project demonstrates the use of object-oriented programming concepts such as classes, member functions, and dynamic data structures (vectors) to create and interact with a family tree. It also highlights the importance of data encapsulation and modularity in program design. The use of recursive visualization in the `visualizeTree` function allows for a clear and hierarchical display of the family tree structure, making it easy to understand.

Throughout the development of this project, several key observations and learnings were made:

- Understanding of Hierarchical Data: The project reinforced the concept of representing hierarchical data structures, which are prevalent in various applications beyond family trees.
- Recursion for Tree Traversal: The use of recursion to visualize the family tree structure highlighted the importance of recursion in processing hierarchical data.
- Error Handling: The project emphasized the significance of error handling to provide clear and informative feedback to users, enhancing the user experience.
- Data Modeling: The project demonstrated how to model real-world relationships and entities using object-oriented programming, specifically through classes and objects.