Assignment 01:

Recursive Algorithms and Sparse Matrix ADT

Time Management

Average expected estimated time to complete this assignment is **12 hours**.

- **[1 HOUR] Software Requirement Understanding**
  - Reading the assignment document.
  - Understanding the problem statement.
  - Identifying software inputs and outputs.
- **[2 HOURS] Software Design**
  - Identification of user-software interaction sequences:
    - Software menus.
    - User inputs.
    - Software flows.
  - Identification of required variables and data structures to be used:
    - For input, processing, and output.
    - Storage class specifications of required variables.
  - Modularization of software into required functions:
    - Proper naming of functions.
    - Identification of tasks to be performed within functions.
    - Identification of parameters and return types of functions.
  - Identification of user-defined header files:
    - Names and functions to be placed in the files.
  - Flow sequence of the main function and calls to various user-defined functions from it.
- **[6 HOURS] Software Coding**
  - Coding of user-defined functions.
  - Placement of user-defined functions into header file(s).
  - Coding of the main function.
- **[2 HOURS] Software Testing**
  - Running software on various inputs.
  - Identifying software errors and bugs.
  - Removal of software errors and bugs.
- **[1 HOUR] Software Documentation**
  - Proper indentation of code.
  - Use of proper naming conventions.
  - Commenting on the code.

Objectives

- Understand and implement recursive algorithms.
- Work with Abstract Data Types (ADT) and apply recursion for problem-solving.
- Implement sparse matrix operations and optimize for space efficiency.

Prerequisite Skills (C and C++)

- Recursion
- Control Structures
- Pointers and Dynamic Memory Management
- Classes and Objects
- Matrix Representation and Operations

Overview

Recursive Algorithm

A technique where a function calls itself to solve smaller instances of the same problem. The goal is to understand and implement solutions using recursion.

Sparse Matrix ADT

A sparse matrix is a matrix in which most of the elements are zero. This ADT optimizes memory usage and allows efficient storage and operations for matrices with a large number of zero elements.

Problems to Solve

Task 01 Problem Statement: Sparse Matrix ADT

Instructions:

The objective of this assignment is to define a class/data structure that represents a Sparse Matrix, which efficiently stores and performs operations on sparse matrices, i.e., matrices predominantly filled with zero values. The Sparse Matrix ADT will support the following operations:

1. **Recursive Sum of Diagonal Elements**
   Implement a *recursive* method within the MatrixADT class to calculate the sum of the

diagonal elements of the matrix. The diagonal elements are defined as those where the row index equals the column index.

Method Signature:

int RecursiveDiagonalSum(int index);

- **Constraints:**
  - The recursion base case occurs when the index exceeds the matrix size.
  - The matrix size (n) should be provided as an argument, and the recursion should stop when `index == n`.

- **Transpose**

A function to find the transpose of a sparse matrix using recursion.

- **Simple Multiplication**

A function to multiply two sparse matrices (without recursion).

Data Structure:

A suitable data structure to store non-zero elements will be utilized, and the operations will be implemented using both recursive and iterative approaches where appropriate.

Class Declaration:

```
template <class T>

class SparseMatrixADT {

private:

   // A suitable structure (e.g., a vector of pairs) to store non-zero elements

   std::vector<std::pair<std::pair<int, int>, T>> elements;

        int rows;
        int cols;

public:

   // Constructor

   SparseMatrixADT(int r, int c);
```

```
    // Method to set a value in the sparse matrix
    void SetValue(int row, int col, T value);

    // Method to get a value from the sparse matrix
    T GetValue(int row, int col) const;

    // Recursive sum of diagonal elements
    int RecursiveDiagonalSum(int index);

    // Transpose the sparse matrix using recursion
    SparseMatrixADT<T> Transpose() const;

    // Multiply two sparse matrices
    SparseMatrixADT<T> Multiply(const SparseMatrixADT<T>& other) const;

    // Additional utility functions can be added as needed
};
```

Sample Runs:

Sample Run 1: Recursive Sum of Diagonal Elements

```
int main() {
    SparseMatrixADT<int> matrix(4, 4); // 4x4 sparse matrix
    matrix.SetValue(0, 0, 5);
    matrix.SetValue(1, 1, 10);
```

```
    matrix.SetValue(2, 2, 15);
    matrix.SetValue(3, 3, 20);

    int diagonalSum = matrix.RecursiveDiagonalSum(0);
    std::cout << "Sum of diagonal elements: " << diagonalSum << std::endl;
    return 0;
}
```

Output:

Sum of diagonal elements: 50

Sample Run 2: Transpose of Sparse Matrix

```
int main() {
    SparseMatrixADT<int> matrix(3, 3);
    matrix.SetValue(0, 1, 1);
    matrix.SetValue(1, 2, 2);
    matrix.SetValue(2, 0, 3);

    SparseMatrixADT<int> transposedMatrix = matrix.Transpose();
    std::cout << "Transposed Matrix:" << std::endl;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            std::cout << transposedMatrix.GetValue(i, j) << " ";
        }
        std::cout << std::endl;
    }
```

```
    return 0;

}
```

output

Transposed Matrix:

0 3 0
0 0 1
0 0 2

Sample Run 3: Multiplication of Sparse Matrices

```
int main() {

    SparseMatrixADT<int> matrix1(2, 2);

    SparseMatrixADT<int> matrix2(2, 2);



    matrix1.SetValue(0, 0, 1);

    matrix1.SetValue(0, 1, 2);

    matrix2.SetValue(0, 0, 3);

    matrix2.SetValue(1, 0, 4);



    SparseMatrixADT<int> result = matrix1.Multiply(matrix2);

    std::cout << "Resulting Matrix after Multiplication:" << std::endl;

    for (int i = 0; i < 2; i++) {

        for (int j = 0; j < 2; j++) {

            std::cout << result.GetValue(i, j) << " ";

        }

        std::cout << std::endl;

    }
```

```
    return 0;

}
```

output:

Resulting Matrix after Multiplication:

```
11  0
0   0
```

Task 02 Problem Statement: N-Queen Problem (Recursive Solution)

Instructions: The N-Queen problem is a classic combinatorial problem that requires placing N queens on an NxN chessboard so that no two queens threaten each other. This means that no two queens can share the same row, column, or diagonal. Our objective is to define a class that implements a recursive solution to this problem, allowing for efficient exploration using backtracking. The class will provide functionalities to initialize the board, place queens, and print solutions.

Requirements:

- User will be able to specify the size of the chessboard (N) at runtime.
- The algorithm should efficiently explore potential placements of queens using recursion and backtracking.
- A method to display the chessboard configuration when a valid solution is found.
- Implement appropriate checks to ensure that placing a queen does not lead to threats from other queens.

Class Declaration:

```
class NQueen {

private:

   int N;                // Size of the chessboard (NxN)

   int* board;           // Pointer to hold the positions of queens


   bool isSafe(int row, int col); // Method to check if placing a queen is safe

   bool solve(int row);      // Recursive method to solve the N-Queen problem
```

public:

NQueen(int n); // Constructor to initialize the board size

~NQueen(); // Destructor to free allocated memory

void solveNQueens(); // Public method to initiate solving the N-Queen problem

void printSolution(); // Method to print the current board configuration

};

Method Signatures:

- `NQueen(int n)` – Constructor to initialize the board with size `n`.
- `~NQueen()` – Destructor to clean up resources.
- `void solveNQueens()` – Public method to start the recursive solution.
- `bool isSafe(int row, int col)` – Returns true if placing a queen at `(row, col)` is safe.
- `bool solve(int row)` – Recursively attempts to place queens row by row.
- `void printSolution()` – Displays the current configuration of the chessboard.

Sample Runs:

Sample Run 1:

```
int main() {

    int n;

    cout << "Enter the number of queens (N): ";

    cin >> n;

    NQueen nQueen(n);

    nQueen.solveNQueens();  // Initiate solving process

    return 0;

}
```

Output:

```
Enter the number of queens (N): 4
```

Solution 1:

. Q . .
. . . Q
Q . . .
. . Q .

Solution 2:

. . Q .
Q . . .
. . . Q
. Q . .

Sample Run 2:

```
int main() {
    int n;
    cout << "Enter the number of queens (N): ";
    cin >> n;
    NQueen nQueen(n);
    nQueen.solveNQueens();  // Initiate solving process
    return 0;
}
```

Output:

Enter the number of queens (N): 8

Solution 1:

. . . . Q . . . .
Q . . . .
. . . . Q
. . Q . . . . .
. . . . . Q . .

. . . . Q
. Q . . . . . .
. . . Q . . . .

...

Constraints:

- The value of N should be greater than or equal to 4, as it is impossible to place 2 or 3 queens on a board without them threatening each other.
- The algorithm should efficiently backtrack to explore possible solutions and print all valid configurations

---

Key Concepts to Apply

- **Software and Algorithmic Maintenance:** Ensure that your solution is maintainable and easily modifiable.
- **Reusability:** Implement your ADTs and functions in a reusable manner for potential future extensions.
- **Encapsulation and Abstraction:** Design the Sparse Matrix ADT such that its internal implementation details are hidden from the user.
- **Principle of Least Privilege:** Only expose necessary functions and data members in your classes.
- **Time and Space Complexity:** Analyze and optimize the time and space complexity for each operation, particularly for matrix operations where space optimization is crucial.

Files to be Submitted

Submit the following files in a zip archive:

1. **NQueen**
    - `NQueen.doc`: Contains problem explanation and analysis of the N-Queen problem.
    - `NQueen.cpp`: Contains the recursive solution and driver program.
2. **SparseMatrix**
    - `SparseMatrix.h`: Contains the class definition and declarations of operations (sum, transpose, multiplication).
    - `SparseMatrix.cpp`: Contains the implementation of Sparse Matrix operations and a driver program to demonstrate its functionality.

Deadline

The assignment is due on **Sunday, October 20, 2024 11:59 pm** Please make sure to submit before the deadline.