CS 2413 – Data Structures – Spring 2025 – Project One
Due 11:59 PM, January 30, 2025

Description: A **time-traveling scheduler** is a tool that maintains a chronological list of events, allowing users to interact with events in the past, present, and future. Each event has a specific timestamp and associated data. The concept of "time travel" involves the ability to revert the scheduler to a prior state, making it useful for applications such as scheduling systems, historical data analysis, and simulations. This project will help you design such a scheduler using **object-oriented principles** by encapsulating the scheduler in a class, managing data with dynamic arrays, and maintaining a sorted list of events.

Consider the following example:

Events:

- A meeting scheduled on January 1st, 2025, at 9:00 AM.
- A reminder to pay bills on February 1st, 2025, at 5:00 PM.
- A doctor's appointment scheduled on March 3rd, 2025, at 10:30 AM.

In the scheduler, these events are stored in chronological order:

Timestamp	Event
01/01/2025 09:00	Meeting: Team Sync
02/01/2025 17:00	Reminder: Pay Bills
03/03/2025 10:30	Appointment: Doctor Visit

Many scheduling systems only focus on future events. However, a **time-traveling scheduler** allows users to go back in time to view, modify, or restore events to a previous state. To implement this, the scheduler maintains a sorted list of events and takes snapshots of its state whenever an operation modifies the schedule.

In this project, you will create the data structure for this scheduler, encapsulated within a class, and implement operations for managing and querying events. Additionally, you will incorporate a basic versioning mechanism to enable "time travel."

Implementation details: You will implement two classes, field names, and necessary methods. You will have an Event class. SnapShot class, and a Scheduler class. You may add additional fields and methods if necessary. In the class definitions, we have provided the method prototypes that you must implement at the maximum. You are allowed to use the following libraries ONLY: string and iostream. If you use any other library, your submission will receive an automatic 0.

```
class Event {
protected:
    string timestamp;  // Timestamp of the event
    string eventData;  // Description or data associated with the event

public:
    // Constructor
    Event(const string& timestamp, const string& eventData);

    // Getters
    string getTimestamp() const;
    string getEventData() const;

    // Setters
    void setTimestamp(const string& timestamp);
    void setEventData(const string& eventData);

    // Comparison Operators (for sorting and searching)
    bool operator< (const Event& other) const;
    bool operator> (const Event& other) const;
    bool operator== (const Event& other) const;

    // Display Method
    void display() const;
};
```

```
class Snapshot {
protected:
    Event* events;        // Pointer to a dynamic array of events
    int numEvents;        // Number of events in the snapshot

public:
    // Constructor
    Snapshot(Event* eventsArray, int size); //you need to copy from eventsArray
                                            //to events one by one with a loop

    // Destructor
    ~Snapshot();

    // Accessors
    Event* getEvents() const;
    int getNumEvents() const;

    // Utility
    void displaySnapshot() const;
};
```

```
class Scheduler {
protected:
    Event* events;            // Dynamic array to store current events in sorted
order
    int eventCount;           // Number of events currently stored
    int eventCapacity;        // Capacity of the dynamic array assume 100

    Snapshot* history;        // Dynamic array of Snapshot objects assume 50 max
    int historyCount;         // Number of snapshots saved
    int historyCapacity;      // Capacity of the history array

public:
    // Constructor and Destructor
    Scheduler(int initialCapacity = 100);  // Initialize with a default capacity
    ~Scheduler();                          // Clean up memory

    // Event Operations
    void addEvent(const string& timestamp, const string& eventData);
    void removeEvent(const string& timestamp);
    Event* queryEvent(const string& timestamp);
    Event* queryRange(const string& start, const string& end, int& count);

    // Time Travel
    void timeTravel(int snapshotIndex);// Restore scheduler to a specific snapshot
    void addSnapshot();                // Save the current state to history

    // Utility Methods
    void displayEvents() const;        // Display all events in sorted order
    void displayHistory() const;       // Display all snapshots for debugging

    //It returns the index position where it found the Event, 0 otherwise
    //This is useful for removeEvent, queryEvent, and queryRange
    int binarySearch(const string& timestamp)

    //It returns the index position where the event needs to be inserted
    int binarySearchForAdd(const string& timestamp);
};
```

Note that events are always added in the timestamp order. The events array in the Scheduler always keeps a set of events sorted by event timestamp. The structure for your main program is also provided. You will use that for your project. When you remove an event, we need to move all the other events to the left in the array by one. Once you identify the location where to insert the new event, then you will push all the events to the right by one and then put the new event in the location.

Your main program will have the following structure (changes may be required).

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

// Class Definitions Go Here; YOU may need to change this main program also

int main() {
    Scheduler scheduler; // Create an instance of the scheduler
    string command;

    while (true) {
        getline(cin, command); // Read an entire line of input

        if (command == "EXIT") break;

        // Parse the command and execute appropriate actions
        if (command.rfind("ADD", 0) == 0) {
            // Format: ADD <timestamp> <eventData>
            string timestamp = command.substr(4, 16);
            string eventData = command.substr(21);   // Extract the event data
            scheduler.addEvent(timestamp, eventData);
        } else if (command.rfind("REMOVE", 0) == 0) {
            // Parse and handle the REMOVE command
            // Format: REMOVE <timestamp>
            string timestamp = command.substr(7);
            scheduler.removeEvent(timestamp);
        } else if (command.rfind("QUERY", 0) == 0 &&
                           command.find("QUERYRANGE") == string::npos) {
            // Format: QUERY <timestamp>
            string timestamp = command.substr(6);
            Event* event = scheduler.queryEvent(timestamp);
            if (event) {
                event->display();
            } else {
                cout << "Event not found at timestamp " << timestamp << endl;
            }
        } else if (command.rfind("QUERYRANGE", 0) == 0) {
            // Format: QUERYRANGE <startTimestamp> <endTimestamp>
            size_t spacePos = command.find(' ', 11);
            string startTS = command.substr(11, spacePos - 11);
            string endTS = command.substr(spacePos + 1);
            int count = 0;
            Event* events = scheduler.queryRange(startTS, endTS, count);

            if (events) {
                for (int i = 0; i < count; ++i) {
                    events[i].display();
                }
                delete[] events; // Free the dynamically allocated memory
            } else {
                cout << "No events found in the range " << startTS << " to " <<
                          endTS << endl;
            }
        } elseif (command == "DISPLAY") {
            scheduler.displayEvents();
        } else if (command == "SNAPSHOT") {
            // Manually create a snapshot
            scheduler.addSnapshot();
            cout << "Snapshot created." << endl;
        } else if (command == "SHOWSNAPSHOTS") {
            // Display available snapshots
            scheduler.displaySnapshots();
        } else if (command.rfind("TIMETRAVEL", 0) == 0) {
            // Parse and handle the TIMETRAVEL command
            // Format: TIMETRAVEL <snapshotIndex>
            int snapshotIndex = stoi(command.substr(11));
            scheduler.timeTravel(snapshotIndex);
            cout << "Time traveled to snapshot " << snapshotIndex << "." << endl;
        }

    }

    cout << "Scheduler program exited." << endl;
    return 0;
}
```

You must ensure that your program outputs the correct results.

Sample Input:

```
DISPLAY
ADD 2025-01-02 04:14 Event-404
REMOVE 2025-01-02 04:14
ADD 2025-05-11 07:09 Event-801
QUERY 2025-05-11 07:09
QUERY 2025-05-11 07:09
REMOVE 2025-05-11 07:09
SNAPSHOT
DISPLAY
ADD 2025-06-25 08:12 Event-45
DISPLAY
TIMETRAVEL 0
QUERY 2025-06-25 08:12
QUERY 2025-06-25 08:12
EXIT
```

Redirected Input: Redirected input provides you a way to send a file to the standard input of a program without typing it using the keyboard. To use redirected input in Visual Studio environment, follow these steps: After you have opened or created a new project, on the menu go to project, project properties, expand configuration properties until you see Debugging, on the right you will see a set of options, and in the command arguments type "< **input filename**". The < sign is for redirected input and the **input filename** is the name of the input file (including the path if not in the working directory). A simple program that reads a matrix can be found below.

```
#include <iostream>

using namespace std;

int main () {

        int r,c,nsv;
        cin >> r >> c >> cv;
          cout << r << c << cv << endl;
        for (int i=0; i < nsv; i++) {
           cin >> rn >> cn >> val;
           cout << rn << cn << val << endl;
        }
        return 0;
}
```

Constraints

1. In this project, the headers you will use is #include <iostream> using namespace std, and #include<string>
2. None of the projects is a group project. Consulting with other members of this class our seeking coding solutions from other sources including the web on programming projects is strictly not allowed and plagiarism charges will be imposed on students who do not follow this.

Project Submission Requirements: Time-Traveling Scheduler

1. **Code Development (75%)**: Implement the provided class structure for the **Time-Traveling Scheduler** by writing the necessary methods to manipulate events and snapshots. Your implementation must be fully compatible with the main program provided, which is designed to create and manipulate Event and Snapshot objects within a Scheduler. Specifically, your code should include methods for:
    - Adding events.
    - Removing events.
    - Querying events by timestamp.
    - Querying events within a range of timestamps.
    - Saving and restoring snapshots to enable time travel.

    Your code must successfully run with the main program and the corresponding input, demonstrating the correct functionality of the Event, Snapshot, and Scheduler classes. All methods must integrate seamlessly with the main program.

    - **LLM/AI Tool Usage**: You can use Large Language Models (LLMs) or AI tools, such as GitHub Copilot, to assist in writing and refining your classes and their methods. If you did not use LLM or AI tools to write your project, you still need to show for requirement (2) below how you would have used these tools to find a solution to the project. Ensure that you use the provided class structure as the basis of your implementation; failure to do so will result in zero points for this project.
2. **LLM and GitHub Copilot Usage Documentation (15%)**: If you choose to use LLM tools or GitHub Copilot, you must document your usage. This documentation (in **PDF format**) should include:
    - **Prompts and Suggestions**: Provide the specific prompts or suggestions you used, such as "Generate a method to add events in C++" or "How can I implement time travel in a scheduling system?"
    - **Rationale**: Explain why you chose these prompts or suggestions and how they contributed to the development of your classes. For instance, describe how a particular suggestion helped you handle dynamic memory allocation for the Snapshot class or efficiently manage events in sorted order.
    - **Incremental Development**: Detail how you used the tools to build and refine your classes and methods incrementally. For example, you might start by generating the basic structure for the Event class, then refine individual methods like addEvent or queryEvent, ensuring compatibility with the main program.
3. **Debugging and Testing Plan (10%)**: Submit a comprehensive debugging and testing plan. This should include:
    - **Specific Tests**: Describe the tests you conducted on your class methods, such as:
        - Checking that addEvent maintains the sorted order of events.
        - Verifying that timeTravel correctly restores snapshots.
        - Ensuring queryRange retrieves all events within the specified range.
    - **Issues and Resolutions**: Document any issues you encountered, such as managing memory for the dynamic events array or handling edge cases in the removeEvent method, and how you resolved them.
    - **Verification**: Explain how you verified that your classes work correctly with the provided main program. This could involve:
        - Running a series of test cases provided by the main program.
        - Creating additional test cases to ensure robustness, such as testing edge cases (e.g., empty snapshots or duplicate timestamps).

Example Submission Structure

1. **Code Implementation**:
    - Submit your .cpp file containing the implementation of the Event, Snapshot, and Scheduler classes and the main program
2. **LLM Documentation** (PDF):
    - Include screenshots of prompts and responses, along with written explanations for their use.
3. **Testing Plan**:
    - Provide a separate text or PDF file detailing your debugging and testing process, along with sample input and output files to verify the scheduler's functionality.