CS 201, Fall 2025

Homework Assignment 4

Due: 23:59, x, 2025

1 Introduction

In this homework, you will implement a system called **TimeTravelEngine**, which controls multiple parallel timelines and allows a traveler to move along the arrow of time — both into the *past* and into the *future*. In this fictional multiverse, time does not flow in a single direction: it can be rewound, fast-forwarded, and even snapped back to specific moments using special **time stones**.

Each timeline is represented by a **TimeBranch** entity and is identified by a unique branch ID managed by the **TimeTravelEngine**. Every TimeBranch maintains its own independent Past stack, Current state, and Future stack, so that time navigation within a branch is performed strictly using **stack-based** operations.

Your implementation will manage multiple TimeBranch objects through the **TimeTravelEngine** wrapper class. While operations that modify the temporal state of a branch's past stack, future stack and time stone stack must rely on **stack-based data structures** implemented by you, the `TimeTravelEngine` may store and manage TimeBranch objects using dynamic arrays. The use of STL container classes is strictly forbidden.

The `TimeTravelEngine` class you will implement must support creating new time branches, performing time travel operations on individual branches, printing branch states, and merging compatible branches into a single combined timeline. All operations must obey the exact behaviors, constraints, and output formats specified in this assignment.

1.1 Time Branches

Each **TimeBranch** entity represents a single universe in which the traveler moves along the arrow of time. A TimeBranch object must contain a **current location**, a **stack of past locations**, a **stack of future locations**, and a **stack of time stones** that serve as temporal checkpoints. You must design and implement the TimeBranch class yourself according to the specifications in this document.

Each TimeBranch is initialized with maximum capacities for its Past and Future stacks. These stacks must always remain within their capacity limits. If an operation would cause the Past or Future stack to exceed its capacity, that operation must be rejected and the branch must remain unchanged. The internal implementation of the `TimeBranch` class is left to you to design. Each TimeBranch must support the following operations:

- `fastForward`: Moves the traveler forward in time by up to k steps.

  At each step, the current location must be pushed onto the Past stack, and the nearest location from the Future stack must become the new current location. The **TimeStone stack must remain unchanged** throughout this operation.

  If the requested number of steps k is greater than the number of locations currently stored in the Future stack, the traveler must move forward only until the **end of the known future** is reached.

  Before performing any movement, the system must first check whether pushing the required number of locations onto the Past stack would exceed the **Past stack capacity**. If this capacity would be exceeded, the operation must **fail completely** and the TimeBranch must remain unchanged.

  For every successful operation, the **exact number of steps actually taken** must be reported in the output log.

  Example log messages:

  - Fast forwarded 2 steps in branch 3.
  - Cannot fast forward 12 steps in branch 4. Past stack capacity exceeded.

- `rewind`: Moves the traveler backward in time by up to k steps.

  At each step, the current location must be pushed onto the Future stack, and the most recent location from the Past stack must become the new current location. The **TimeStone stack must remain unchanged** throughout this operation.

  If the requested number of steps k is greater than the number of locations currently stored in the Past stack, the traveler must rewind only until the **earliest reachable past** is reached.

  Before performing any movement, the system must first check whether pushing the required number of locations onto the Future stack would exceed the **Future stack capacity**. If this capacity would be exceeded, the operation must **fail completely** and the TimeBranch must remain unchanged.

  For every successful operation, the **exact number of steps actually taken** must be reported in the output log.

  Example log messages:

  - Rewound 2 steps in branch 5.
  - Cannot rewind 8 steps in branch 3. Future stack capacity exceeded.

- `travelTo`: Moves the traveler in the specified TimeBranch to a **new location** that extends the current timeline.

Before performing the operation, the system must verify that the given branch ID exists. If the branch does not exist, the operation must be rejected and the TimeBranch must remain unchanged.

If the branch exists, the system must first check whether pushing the current location onto the Past stack would exceed the **Past stack capacity**. If the Past stack is already full, the operation must **fail completely** and no changes must be made.

If the operation is valid, the current location must be pushed onto the Past stack, the current location must be updated to the given new location, and the entire Future stack must be cleared, since a new timeline is being created from this point onward. The TimeStone stack must remain unchanged.

Example log messages:

- Traveled to Lab in branch 3.
- Cannot travel. There is no branch with ID 4.
- Cannot travel to Tower in branch 2. Past stack capacity exceeded.

- `placeTimeStone`:A **TimeStone** represents a special temporal checkpoint placed by the traveler at a specific moment in time. Each time stone records the traveler's *current location* and it should be pushed to a separate stone stack. Later, activating a time stone forces the traveler back or forward to that exact recorded moment and erases all alternative futures beyond it. Time stones therefore act as **irreversible anchors** in the flow of time.

  This method places a new time stone at the traveler's current location. The current location must be pushed onto the TimeStone stack. There is no capacity limit on the TimeStone stack in this assignment; any number of time stones may be placed over the lifetime of a TimeBranch. It is allowed to place multiple time stones at the same location at different moments in the timeline.

  Example log messages:

  - Time stone placed at Forest.
  - Time stone placed at Tower.

- `activateTimeStone`: Activates the most recently placed time stone. A **TimeStone** represents a fixed anchor in the timeline and stores the traveler's location at the moment it was placed. Activating a time stone attempts to return the traveler to that recorded moment and collapses all alternative futures beyond it.

  The activation must follow the rules below:

  - If the TimeStone stack is empty, the operation must fail and the TimeBranch must remain unchanged.
  - Otherwise, let L be the location stored in the most recent time stone. The system must attempt to move the traveler to the **closest reachable occurrence** of location L relative to the current position in the timeline.

* If the current location is already L, the traveler remains at the same position. In this case, the **Future stack must be completely cleared**, while the Past stack remains unchanged.
* If L appears in both the Past and the Future stacks, the system must **prefer the Past**. The traveler must rewind step by step until the current location becomes L.
* If L appears only in the Past stack, the traveler must rewind step by step until reaching L.
* If L appears only in the Future stack, the traveler must fast-forward step by step until reaching L. Before performing this movement, system must first check if this movement exceeds past stack capacity. If it does, then the time stone must be popped and corresponding output must be logged.
* If L does **not appear** in the Past stack, the Future stack, or as the current location, the time stone is considered **unreachable**. In this case, the system must discard the time stone and leave the TimeBranch unchanged.

- The time stone must be popped from the TimeStone stack **regardless of whether it was successfully activated or found to be unreachable**.

Example log messages:

- Time stone activated. The traveler is now at Forest.
- Time stone discarded. The recorded moment is no longer reachable.
- Time stone discarded. Past stack capacity exceeded.
- Cannot activate time stone. No time stones available.

- `printTimeBranch`: Prints the contents of the Past stack, the Current location, the Future stack, and the TimeStone stack.

  For all stacks, elements must be printed **from bottom to top**, where:

  - the **leftmost** element represents the **bottom** of the stack,
  - the **rightmost** element represents the **top** of the stack.

  The output must follow the format below:

  - Past : [Home, Forest, Castle]
    Current: (Tower)
    Future : [Dungeon, Lab]
    Stones : [Forest, Castle]

1.2 TimeTravelEngine

The **TimeTravelEngine** is the core wrapper class of the system. It is responsible for creating, deleting, managing, printing, and merging multiple TimeBranch objects. Each TimeBranch is identified by a unique integer branch ID assigned by the TimeTravelEngine.

You will be given the header file of the `TimeTravelEngine` class. However, you must implement all of its member functions in accordance with the specifications below.

The required member functions of the `TimeTravelEngine` class are as follows:

- `createTimeBranch`: Creates a new TimeBranch with a given Past stack capacity, Future stack capacity, an initial location and TimeBranch ID. The system must assign the **ID** to the newly created TimeBranch and return this ID. If the system attempts to assign an ID that already exists, the operation must be rejected and the branch must not be created.

  Example log messages:

  - Created time branch with ID 3.
  - Cannot create time branch. A branch with ID 3 already exists.

- `deleteTimeBranch`: Deletes the TimeBranch with the specified branch ID and frees all dynamically allocated memory associated with it. If the branch does not exist, a warning message must be printed. Example log messages:

  - Deleted time branch 4.
  - Cannot delete branch. There is no branch with ID 4.

- `printAllBranches`: Prints the complete state of all existing TimeBranch objects in **ascending order of their branch IDs**. For each branch, its full time state (Past, Current, Future, and TimeStones) must be printed under a clear branch header. If there are no branches in the system, a warning message must be printed.

  Example log messages:

  - There are no time branches to show.
  - Time branches in the system:
    Branch 1:
    Past : [Home, Forest]
    Current: (Castle)
    Future : [Tower, Dungeon]
    Stones : [Forest]

    Branch 3:
    Past : [ ]
    Current: (Lab)
    Future : [Observatory]

Stones : [ ]

Branch 7:
Past : [Town]
Current: (Bridge)
Future : [Cave, Ruins]
Stones : [Town, Bridge]

- `(BONUS) mergeBranches`: Merges two existing TimeBranch objects into a single new TimeBranch with a **user-specified branch ID**. The function takes three parameters: the IDs of the two branches to be merged and the ID to be assigned to the newly created merged branch.

  Before attempting the merge, the system must verify that **both input branch IDs exist** in the system. If either of the branch IDs does not exist, the operation must be rejected and no changes must be made.

  If both branches exist, they can only be merged if their **current locations are identical**. If the merge is successful:

  - Both original branches must be **deleted** from the system.
  - A new TimeBranch must be created with the specified new branch ID.
  - The **Past stack capacity** of the new branch must be the **sum** of the Past stack capacities of the two merged branches.
  - The **Future stack capacity** of the new branch must be the **sum** of the Future stack capacities of the two merged branches.
  - The new branch's **current location** must be set to the common current location of the two original branches.
  - The new branch's **Past stack** must be formed by first taking all locations from the Past stack of the first branch (from bottom to top) and then all locations from the Past stack of the second branch (from bottom to top), preserving the internal order within each original stack.
  - The new branch's **Future stack** must be formed in the same way: first all locations from the Future stack of the first branch (bottom to top), then all locations from the Future stack of the second branch (bottom to top).
  - The new branch's **TimeStone stack** must be formed by concatenating the TimeStone stacks of the two branches: all stones from the first branch (bottom to top), followed by all stones from the second branch (bottom to top).

  If the specified new branch ID is already in use, the operation must be rejected. If the current locations of the two branches are different, the operation must also be rejected. In all failure cases, the original branches must remain unchanged.

All possible log messages:

- Time branches 2 and 5 merged into new branch 8.
- Cannot merge branches. Current locations do not match.
- Cannot merge branches. A branch with ID 8 already exists.
- Cannot merge branches. Missing branch IDs.

Below is the required public part of the `TimeTravelEngine` class that you must write in this assignment. The name of the class must be `TimeTravelEngine`, and it must include the following public member functions.

The interface for the class must be written in the file called `TimeTravelEngine.h` and its implementation must be written in the file called `TimeTravelEngine.cpp`. You can define additional public and private member functions and data members in this class. You can also define additional classes in your solution and implement them in separate files.

```
class TimeTravelEngine
{
public:
    TimeTravelEngine();
    ~TimeTravelEngine();

    int createTimeBranch(const int pastCapacity,
                         const int futureCapacity,
                         const string startLocation,
                         const int branchId);

    void deleteTimeBranch(const int branchId);
    void printAllBranches() const;

    void travelTo(const int branchId, const string newLocation);

    void fastForward(const int branchId, const int k);
    void rewind(const int branchId, const int k);

    void placeTimeStone(const int branchId);
    void activateTimeStone(const int branchId);

    void printTimeBranch(const int branchId) const;

    int mergeBranches(const int branchId1,
                      const int branchId2,
                      const int newBranchId);
};
```

Here is an example test program that uses this class and the corresponding output. We will use a similar program to test your solution so make sure that the name of the class is `TimeTravelEngine`, its interface is in the file called `TimeTravelEngine.h`, and the required functions are defined as shown above. Your implementation **MUST** use exactly the same format given in the example output to display the messages expected as the result of the defined functions.

Example test code:

```cpp
#include <iostream>
#include "TimeTravelEngine.h"

using namespace std;

int main()
{
    TimeTravelEngine engine;

    engine.printAllBranches();
    cout << endl;

    engine.deleteTimeBranch(1);
    cout << endl;

    const int b1 = 1;
    const int b2 = 2;
    const int b3 = 3;

    engine.createTimeBranch(3, 2, "Home", b1);
    engine.createTimeBranch(2, 2, "Town", b2);
    engine.createTimeBranch(2, 3, "Village", b3);
    engine.createTimeBranch(5, 5, "Nowhere", b1);
    cout << endl;

    engine.printAllBranches();
    cout << endl;

    engine.travelTo(b1, "Forest");
    engine.travelTo(b1, "Castle");
    engine.travelTo(42, "Lab");
    engine.travelTo(b2, "Ruins");
    engine.travelTo(b2, "Gate");
    engine.travelTo(b2, "Tower");
    cout << endl;

    engine.placeTimeStone(b1);
    engine.activateTimeStone(b3);
    cout << endl;

    engine.travelTo(b3, "Field");
    engine.travelTo(b3, "Forest");
    engine.placeTimeStone(b3);
    engine.rewind(b3, 1);
```

```
    engine.placeTimeStone(b3);
    engine.activateTimeStone(b3);
    engine.activateTimeStone(b3);
    cout << endl;

    engine.rewind(b1, 1);
    engine.rewind(b1, 3);
    cout << endl;

    engine.printTimeBranch(b1);
    cout << endl;

    engine.fastForward(b1, 5);
    cout << endl;

    engine.printTimeBranch(b1);
    cout << endl;

    engine.mergeBranches(10, 11, 8);
    engine.mergeBranches(b1, b3, 8);

    const int m1 = 4;
    const int m2 = 5;
    const int mergedId = 8;

    engine.createTimeBranch(2, 2, "Home", m1);
    engine.createTimeBranch(2, 2, "Home", m2);

    engine.travelTo(m1, "Forest");
    engine.travelTo(m2, "Forest");

    engine.mergeBranches(m1, m2, b1);
    engine.mergeBranches(m1, m2, mergedId);
    cout << endl;

    engine.printAllBranches();
    cout << endl;

    engine.deleteTimeBranch(mergedId);
    engine.deleteTimeBranch(99);
    cout << endl;

    engine.travelTo(b2, "Portal");
    cout << endl;

    engine.rewind(b2, 1);
```

```
    engine.rewind(b2, 5);
    cout << endl;

    engine.printAllBranches();
    cout << endl;

    return 0;
}
```

Output of the example test code:

```
There are no time branches to show.

Cannot delete branch. There is no branch with ID 1.

Created time branch with ID 1.
Created time branch with ID 2.
Created time branch with ID 3.
Cannot create time branch. A branch with ID 1 already exists.

Time branches in the system:
Branch 1:
Past   : [ ]
Current: ( Home )
Future : [ ]
Stones : [ ]

Branch 2:
Past   : [ ]
Current: ( Town )
Future : [ ]
Stones : [ ]

Branch 3:
Past   : [ ]
Current: ( Village )
Future : [ ]
Stones : [ ]

Traveled to Forest in branch 1.
Traveled to Castle in branch 1.
Cannot travel. There is no branch with ID 42.
Traveled to Ruins in branch 2.
Traveled to Gate in branch 2.
Cannot travel to Tower in branch 2. Past stack capacity exceeded.

```

```
Time stone placed at Castle.
Cannot activate time stone. No time stones available.

Traveled to Field in branch 3.
Traveled to Forest in branch 3.
Time stone placed at Forest.
Rewound 1 steps in branch 3.
Time stone placed at Field.
Time stone activated. The traveler is now at Field.
Time stone discarded. The recorded moment is no longer reachable.

Rewound 1 steps in branch 1.
Rewound 1 steps in branch 1.

Past   : [ ]
Current: ( Home )
Future : [ Castle, Forest ]
Stones : [ Castle ]

Fast forwarded 2 steps in branch 1.

Past   : [ Home, Forest ]
Current: ( Castle )
Future : [ ]
Stones : [ Castle ]

Cannot merge branches. Missing branch IDs.
Cannot merge branches. Current locations do not match.
Created time branch with ID 4.
Created time branch with ID 5.
Traveled to Forest in branch 4.
Traveled to Forest in branch 5.
Cannot merge branches. A branch with ID 1 already exists.
Time branches 4 and 5 merged into new branch 8.

Time branches in the system:
Branch 1:
Past   : [ Home, Forest ]
Current: ( Castle )
Future : [ ]
Stones : [ Castle ]

Branch 2:
Past   : [ Town, Ruins ]
Current: ( Gate )
Future : [ ]
```

```
Stones : [ ]

Branch 3:
Past   : [ Village ]
Current: ( Field )
Future : [ ]
Stones : [ ]

Branch 8:
Past   : [ Home, Home ]
Current: ( Forest )
Future : [ ]
Stones : [ ]

Deleted time branch 8.
Cannot delete branch. There is no branch with ID 99.

Cannot travel to Portal in branch 2. Past stack capacity exceeded.

Rewound 1 steps in branch 2.
Rewound 1 steps in branch 2.

Time branches in the system:
Branch 1:
Past   : [ Home, Forest ]
Current: ( Castle )
Future : [ ]
Stones : [ Castle ]

Branch 2:
Past   : [ ]
Current: ( Town )
Future : [ Gate, Ruins ]
Stones : [ ]

Branch 3:
Past   : [ Village ]
Current: ( Field )
Future : [ ]
Stones : [ ]
```

2 Specifications

1. You **ARE NOT ALLOWED** to use STL container classes such as `stack`, `queue`, `vector`, `list`, or any other STL data structure. You must implement your **own stack-based data**

structures to manage the Past, Future, and TimeStone stacks of each `TimeBranch`. Any submission using STL containers for these structures will receive **no points** for the relevant parts.

2. You MUST implement all stack operations (`push`, `pop`, `peek`, overflow handling, capacity control, etc.) manually. You may use the stack implementations discussed during the lectures as a reference, but you must write your own code. The use of any external container or library for stack-like behavior is strictly forbidden.

3. You **ARE NOT ALLOWED** to use any global variables or any global functions in your implementation. All data must be encapsulated properly inside classes.

4. The output message printed for **each operation** MUST match the format shown in the example outputs of this assignment exactly.

5. Your code **MUST NOT** have any memory leaks. You will lose points if your program contains memory leaks even if all output messages are correct. To detect memory leaks, you may use `Valgrind`, which is available at `http://valgrind.org`.

6. All dynamically allocated memory related to `TimeBranch` objects, their internal stacks, and all auxiliary structures must be properly released when a branch is deleted or when the program terminates.

7. You are free to define additional helper classes (e.g., a custom `Stack` class), but all time travel logic, memory management, and data structure operations must be implemented by you.

3 Submission

1. In this assignment, you must have separate interface and implementation files (i.e., separate `.h` and `.cpp` files) for your class. Your class name MUST BE `TimeTravelEngine` and your file names MUST BE `TimeTravelEngine.h` and `TimeTravelEngine.cpp`. Note that you may write additional class(es) in your solution.

2. The code (`main` function) given above is just an example. We will test your implementation using different scenarios, which will contain different function calls. Thus, do not test your implementation only by using this example code. We recommend you to write your own driver files to make extra tests. However, you MUST NOT submit these test codes (we will use our own test code). In other words, do not submit a file that contains a function called `main`.

3. You should put all of your `.h` and `.cpp` files into a folder and zip the folder (in this zip file, there should not be any file containing a `main` function). The name of this zip file should conform to the following name convention: secX-Firstname-Lastname-StudentID.zip where X is your section number. The submissions that do not obey these rules will not be graded.

4. Make sure that each file that you submit (each and every file in the archive) contains your name, section, and student number at the top as comments.

5. You are free to write your programs in any environment (you may use Linux, Windows, MacOS, etc.). On the other hand, we will test your programs on "dijkstra.ug.bcc.bilkent.edu.tr" and we will expect your programs to compile and run on the dijkstra machine. Your code will be tested by using an automated test suite that includes multiple test cases where each case corresponds to a specific number of points in the overall grade. We will provide you with example test cases by email. Thus, we strongly recommend you to make sure that your program successfully compiles and correctly works on dijkstra.ug.bcc.bilkent.edu.tr before submitting your assignment. If your current code does not fully compile on dijkstra before submission, you can try to comment out the faulty parts so that the remaining code can be compiled and tested during evaluation.

6. This assignment is due by 23:59 on x, x x, 2025. You should upload your work to Moodle before the deadline. No hardcopy submission is needed. Late submissions will not be accepted (if you can upload to Moodle, then you are fine). There will be no extension to this deadline.

7. We use an automated tool as well as manual inspection to check your submissions against plagiarism. For questions regarding academic integrity and use of external tools (including generative AI tools), please refer to the course home page and the Honor Code for Introductory Programming Courses (CS 101/102/201/202) at `https://docs.google.com/document/d/1v_3ltpV_lClLsROXrMbojyuv4KrFQAm1uoz3SdC-7es/edit?usp=sharing`.

8. This homework will be graded by your TAs **Sude Önder** (sude.onder@bilkent.edu.tr. Thus, you may ask your homework related questions directly to her. There will also be a forum on Moodle for questions.