Assignment Instructions :

- Use C++ only.
- **Do not use the STL library, you have to implement all data structures required!.**
- Ensure your code is efficient and releases all allocated memory properly.
- Any form of plagiarism will be detected and may result in a zero or negative score.
- You must understand every line of your code.
- All team members must be familiar with the assigned problems.
- Each team must consist of at least 4 and maximum 5 members.
- Team members can be from any group.
- The total score for the assignment is 100 points.
- **Deadline: [12-5-2025].**

Submission Instructions:

● Each problem must be submitted in a separate .cpp file.
Example: Problem1.cpp, Problem2.cpp, ...
● All .cpp files should be compressed into a single .zip file.
● The file name format should be as follows:
(A2_ID1_ID2_ID3_ID4_ID5.zip)
Example: A2_20220221_20220231_20220321_20220211.zip

Failure to follow instructions may lead to a reduction in your final grade.

● **Manual Test**
○ **Writing testcases consider as bonus (10 pt)**
○ **Use a file to read the input instead of "cin" for all problems.**

Problem 1:

Design and implement a simplified browser history system that allows users to:

1. **Visit a new URL:** This adds the new URL to the browsing history.
2. **Go back:** This navigates to the previously visited URL.
3. **Go forward:** This navigates to a URL that was visited after the current one (after a "back" operation).

Core Idea and Stack Usage:
We can use **two stacks** to manage the browser history:

- **backStack:** This stack will store the history of URLs visited in order. When a new URL is visited, it's pushed onto this stack. When the user goes "back," the current URL is popped from backStack and pushed onto the forwardStack.
- **forwardStack:** This stack will store the URLs that were navigated away from using the "back" button. When the user goes "forward," a URL is popped from forwardStack and pushed back onto the backStack.

Detailed Functionality and Implementation Steps:

1. **visit(url):**
    - If the forwardStack is not empty, it means the user had previously gone back. When

a new URL is visited, the current forward history becomes irrelevant, so the forwardStack should be cleared.
    - The current url is pushed onto the backStack.
    - The current URL should be tracked (let's say in a variable currentUrl).
2. **goBack():**
    - Check if the backStack has more than one URL. If it has only the initial page or is empty, going back is not possible.
    - Pop the current currentUrl from the backStack and push it onto the forwardStack.
    - The new currentUrl becomes the top element of the backStack (the previously visited URL).
    - Return the new currentUrl or indicate that going back is not possible.
3. **goForward():**
    - Check if the forwardStack is not empty. If it is, going forward is not possible.
    - Pop the top url from the forwardStack and push it onto the backStack.
    - The new currentUrl becomes the popped url.
    - Return the new currentUrl or indicate that going forward is not possible.

Example Scenario:
Let's say the user visits the following URLs in order: "https://www.google.com/url?sa=E&source=gmail&q=google.com", "wikipedia.org", "youtube.com".

1. visit("google.com"):
    - backStack: ["https://www.google.com/url?sa=E&source=gmail&q=google.com"]
    - forwardStack: []
    - currentUrl: "https://www.google.com/url?sa=E&source=gmail&q=google.com"
2. visit("wikipedia.org"):
    - backStack: ["https://www.google.com/url?sa=E&source=gmail&q=google.com", "wikipedia.org"]
    - forwardStack: []
    - currentUrl: "wikipedia.org"
3. visit("youtube.com"):
    - backStack: ["https://www.google.com/url?sa=E&source=gmail&q=google.com", "wikipedia.org", "youtube.com"]
    - forwardStack: []
    - currentUrl: "youtube.com"
4. goBack():
    - backStack: ["https://www.google.com/url?sa=E&source=gmail&q=google.com", "wikipedia.org"]
    - forwardStack: ["youtube.com"]
    - currentUrl: "wikipedia.org" (returned)
5. goBack():
    - backStack: ["https://www.google.com/url?sa=E&source=gmail&q=google.com"]
    - forwardStack: ["youtube.com", "wikipedia.org"]
    - currentUrl: "https://www.google.com/url?sa=E&source=gmail&q=google.com" (returned)
6. goForward():
    - backStack: ["https://www.google.com/url?sa=E&source=gmail&q=google.com", "wikipedia.org"]
    - forwardStack: ["youtube.com"]

    - currentUrl: "wikipedia.org" (returned)
7. visit("facebook.com"):
    - backStack: ["https://www.google.com/url?sa=E&source=gmail&q=google.com", "wikipedia.org", "facebook.com"]
    - forwardStack: [] (forward history is cleared)
    - currentUrl: "facebook.com"

Problem 2 :
Interactive AVL Tree Address Book with Menu:

Your task is to design and implement an interactive Address Book application that utilizes an AVL tree to store and manage contact information.
Each contact will be stored as a key-value pair in the AVL tree:
 * **Key**: A unique integer ID for each contact.
 * **Value**: A dictionary containing the contact's details with the following keys:
   * "name": The full name (string).
   * "phone": The phone number (string).
   * "email": The email address (string).

Your Address Book application should present the following menu to the user repeatedly until they choose to exit (though an exit option isn't explicitly required for this problem, it's good practice for interactive applications):

```
Address Book Application
------------------------
1. Add New Contact
2. Search for Contact
3. Delete Contact (Optional)
4. List All Contacts (Sorted by ID)
5. Display Current Tree Structure
------------------------
Enter operation (1-5):
```

The user will then input a number corresponding to the desired operation. Let's detail the expected interaction for each option:
 * Add New Contact:

```
Enter operation (1-5): 1
Enter unique ID (integer): 310
Enter name: Omar Hassan
Enter phone: 010-876-5432
Enter email: omar.hassan@example.com
Contact added successfully.
```

   (If the ID already exists, you should output an error message like: "Error: Contact with ID 310 already exists.")
 * Search for Contact:

```
  Enter operation (1-5): 2
Enter ID to search: 310
```

```
Contact found:
ID: 310
Name: Omar Hassan
Phone: 010-876-5432
Email: omar.hassan@example.com
```

(If the ID is not found:)

```
Enter operation (1-5): 2
Enter ID to search: 400
Contact not found.
```

* Delete Contact (Optional):

```
Enter operation (1-5): 3
Enter ID to delete: 310
Contact deleted successfully.
```

(If the ID is not found for deletion:)

```
Enter operation (1-5): 3
Enter ID to delete: 500
Contact not found.
```

* List All Contacts (Sorted by ID):

```
Enter operation (1-5): 4
Contacts in Address Book (sorted by ID):
ID: 150, Name: John Doe, Phone: 012-345-6789, Email: john.doe@example.com
ID: 210, Name: Peter Jones, Phone: 011-123-7890, Email: peter.jones@example.org
ID: 310, Name: Omar Hassan, Phone: 010-876-5432, Email: omar.hassan@example.com
```

* (If the address book is empty:)

```
Enter operation (1-5): 4
Address Book is empty.
```

* Display Current Tree Structure:

```
Enter operation (1-5): 5
Current AVL Tree:
     210
    /   \
  150    310
```

Implementation Guidelines:
* Implement the AVL tree with insert, search, delete (optional), and inorderTraversal operations, including balancing rotations.
* Your application should present the menu of options to the user.
* The user should be able to choose an operation by entering the corresponding number (1-5).
* Handle the input and output as shown in the examples above.
* Ensure the uniqueness of contact IDs is enforced during the addition of new contacts.

Problem 3:

You are given a binary array called arr (each element is either 0 or 1), and an integer k.

You can perform an operation called a **segment flip**, where you choose any **contiguous subarray** of length k and flip every bit inside it:

- Turn 0 into 1
- Turn 1 into 0

You can perform this operation as many times as needed.

Your task:
Determine the **minimum number of segment flips** required to turn all elements in arr into 1.
If it's **not possible**, return -1.

Notes 🙂 don't use any built in Data structure.

Example 1:

Input:
arr = [0, 1, 0], k = 1
Output:
2
Explanation:

- Flip at index 0 → [1, 1, 0]
- Flip at index 2 → [1, 1, 1]
  Two flips total.

Example 2:

Input:
arr = [1, 1, 0], k = 2
Output:
-1
Explanation:
There's no way to flip segments of size 2 and make the entire array all 1s.

Example 3:

Input:
arr = [0, 0, 0, 1, 0, 1, 1, 0], k = 3
Output:
3
Explanation:

- Flip from index 0 → [1, 1, 1, 1, 0, 1, 1, 0]
- Flip from index 4 → [1, 1, 1, 1, 1, 0, 0, 0]

- Flip from index 5 → [1, 1, 1, 1, 1, 1, 1, 1]

Problem 4:

Emergency Room Priority Queue

You are tasked with designing a custom priority queue system for a hospital emergency room using a binary heap data structure. Each patient has a name, severity (an integer from 1 to 100, where 100 is the highest severity), and arrival_time (a timestamp or a simple increasing ID, with lower values indicating earlier arrival).

The system must meet the following requirements:

- High severity patients are treated first.
- If two patients have the same severity, earlier arrivals are treated first.

You must implement the system using a max-heap, prioritizing patients based on a combination of severity and arrival time. Complete the following tasks:

1. Heap Implementation:

Implement a MaxHeap class. You can represent the heap internally using an array.

Functions to implement:

- insert(patient): Inserts a new patient into the priority queue. A patient should be represented as a dictionary or object with name, severity, and arrival_time attributes.
- extract_max(): Returns and removes the patient with the highest priority (highest severity, earliest arrival if severities are equal). Returns None if the heap is empty.
- peek(): Returns, but does not remove, the patient with the highest priority. Returns None if the heap is empty.
- print_heap(): Prints the heap as an array, showing the order of patients in the underlying array representation. For simplicity, you can print just the patient names in the array.

2. Simulation and Testing:

Simulate the arrival of 10 patients with random severity and arrival times. You can generate this data randomly or use a predefined set.

- Insert them into your heap-based queue one by one.
- Print the heap (using print_heap()) after each insertion to show how the heap structure changes.
- Extract patients one by one (using extract_max()) and print the name of each patient as they are "treated" to demonstrate the treatment order.

Example Input/Output:

Patient Data (Example):

```
[

  {"name": "Alice", "severity": 80, "arrival_time": 1},
  {"name": "Bob", "severity": 90, "arrival_time": 2},
  {"name": "Charlie", "severity": 70, "arrival_time": 3},
  {"name": "David", "severity": 85, "arrival_time": 4},
  {"name": "Eve", "severity": 90, "arrival_time": 5},
  {"name": "Frank", "severity": 75, "arrival_time": 6},
  {"name": "Grace", "severity": 95, "arrival_time": 7},
  {"name": "Henry", "severity": 80, "arrival_time": 8},
  {"name": "Ivy", "severity": 70, "arrival_time": 9},
  {"name": "Jack", "severity": 100, "arrival_time": 10}
]
```

Heap After Insertions:

```
Inserting: Alice
Heap: ['Alice']

Inserting: Bob
Heap: ['Bob', 'Alice']

Inserting: Charlie
Heap: ['Bob', 'Alice', 'Charlie']

Inserting: David
Heap: ['Bob', 'David', 'Charlie', 'Alice']

Inserting: Eve
Heap: ['Eve', 'David', 'Charlie', 'Alice', 'Bob']

Inserting: Frank
Heap: ['Eve', 'David', 'Charlie', 'Alice', 'Bob', 'Frank']

Inserting: Grace
Heap: ['Grace', 'David', 'Eve', 'Alice', 'Bob', 'Frank', 'Charlie']
```

```
Inserting: Henry
Heap: ['Grace', 'David', 'Eve', 'Henry', 'Bob', 'Frank', 'Charlie', 'Alice']

Inserting: Ivy
Heap: ['Grace', 'David', 'Eve', 'Henry', 'Bob', 'Frank', 'Charlie', 'Alice', 'Ivy']

Inserting: Jack
Heap: ['Jack', 'David', 'Eve', 'Henry', 'Bob', 'Frank', 'Charlie', 'Alice', 'Ivy', 'Grace']
```

Treatment Order :

```
Treating: Jack
Treating: Grace
Treating: Eve
Treating: Bob
Treating: David
Treating: Alice
Treating: Henry
Treating: Frank
Treating: Charlie
Treating: Ivy
```