1472. Design Browser History Medium Stack Design Array **Linked List Data Stream Doubly-Linked List**

Problem Description

steps in your browser history, or move forward a certain number of steps in the history. This needs to be done through implementing a BrowserHistory class with specific functionalities: • BrowserHistory(string homepage): Initializes a BrowserHistory object with a given homepage URL.

In this LeetCode problem, we're asked to simulate a browser history where you can visit new URLs, move back a certain number of

- string back(int steps): Moves steps back in the history, but only up to how many URLs are behind the current one. It returns
- the URL you land on.

void visit(string url): Simulates visiting a new URL from the current page, which erases all forward history.

- string forward(int steps): Moves steps forward in the history, but only up to how many URLs are ahead of the current one. Similarly, it returns the URL you land on.
- Some limitations have been placed on the inputs to keep the problem bounded: URL and homepage strings will only contain lowercase English letters and the period character, and their lengths will range

previous current page (this reflects real browser behavior).

Constructor: BrowserHistory(string homepage)

URL, and stk2 will be empty as there's no forward or backward history.

- Intuition
- To simulate browser history, we need a way to keep track of visited URLs and navigate back and forth. We can use two stacks to model the back and forward browser functionality. The main stack (stk1) keeps track of the browsing history where the top of the

• The steps parameter in the back and forward methods will be between 1 and 100.

• The visit, back, and forward methods will be called at most 5000 times in total.

between 1 and 20 characters.

stack is the current page being viewed. Another stack (stk2) is used to keep track of the pages that have been gone back from, enabling us to navigate forward. • Visit: When visiting a new URL, we push it onto stk1, representing that it is now the current page. We also clear stk2, because

• Back: When moving back, we pop URLs from stk1 and push them onto stk2, decrementing the steps each time until we either reach the desired number of steps or can't go back further (because we've reached the homepage). The top of stk1 after these operations gives us the current page.

• Forward: The forward action is similar to back but in reverse — we pop from stk2 and push onto stk1, moving the current page

visiting a new page after going back in history means that you can no longer go forward to the pages that were ahead of your

- forward in history. Again, we do this until we've completed the desired number of steps or we've reached the most recent page (when stk2 is empty). This approach uses stacks to effectively traverse through the history, ensuring that the order of navigation is maintained and that we have an efficient way to move back and forward through the pages visited.
- Solution Approach The solution provided uses two stacks (stk1 and stk2) for tracking the browser history. Let's walk through the implementation of each method and how they manipulate these stacks to satisfy the requirements.

When a new BrowserHistory object is created, it is initialized with the homepage. This is done by calling the visit method with the homepage URL as the argument, establishing the starting point of our browser history. At this point, stk1 will contain the homepage

Visit Method: visit(string url)

Back Method: back(int steps)

Forward Method: forward(int steps)

the top of stk1, which is now the current page.

Clearing stk2 is essential because any forward history is erased as per the real-life browser behavior whenever you visit a new page after navigating back. The current URL is now at the top of stk1.

This method simulates the action of the 'Back' button in a browser. When invoked, it checks if we can move back the given number

of steps. As long as there are more than one URLs in stk1 (we're not at the homepage), it transfers the top URL to stk2 (simulating

movement back in history). It reduces steps by 1 each time this transfer happens. Finally, when steps reach zero or cannot go back

This method simulates pressing the 'Forward' button in a browser, the reverse of the back method. While there are items in stk2 (i.e.,

there's forward history to traverse) and steps is not zero, it moves URLs from stk2 back to stk1. This process effectively steps back

through the history forward. Once steps have been depleted or there is no more forward history (stk2 is empty), it returns the URL at

When visiting a new URL, we push the URL onto stk1 (the stack representing our browse history) and clear stk2 (forward history).

further, it returns the current URL, which is at the top of stk1.

The overall pattern this solution follows is a classic use of stack data structures for managing history in such a way that the order of movement back and forth is naturally maintained. Stacks are ideal because they follow a last-in-first-out (LIFO) principle that directly aligns with the action of navigating backward and forward through a linear browsing history. **Example Walkthrough**

2. We visit a new URL "google.com": 1 browserHistory.visit("google.com") stk1 is now ["leetcode.com", "google.com"], and stk2 remains empty, as visiting a new URL doesn't involve moving back or

This transfers "facebook.com" from stk1 to stk2, leaving us with stk1 = ["leetcode.com", "google.com"] and stk2 =

We can only move back by 1 step because we're at "google.com" and can't go further back than "leetcode.com", so stk1

becomes ["leetcode.com"] and stk2 transforms to ["google.com", "facebook.com"]. The current_url is "leetcode.com".

"google.com" is popped from stk2 and pushed onto stk1, resulting in stk1 = ["leetcode.com", "google.com"] and stk2 =

stk1 becomes ["leetcode.com", "google.com", "youtube.com"] and stk2 is cleared out, signifying that the forward history is

Now, stk1 has ["leetcode.com", "google.com", "facebook.com"], and stk2 is still empty.

forward.

["facebook.com"]. The current_url is "google.com". 5. We move back again, this time by 2 steps:

Let's take an example to illustrate how the BrowserHistory class and its methods work in practice:

1. Suppose we initialize the browser history with the homepage:

After this step, stk1 contains ["leetcode.com"] and stk2 is empty.

1 browserHistory = BrowserHistory("leetcode.com")

3. Let's visit another URL "facebook.com":

1 browserHistory.visit("facebook.com")

1 current_url = browserHistory.back(1)

1 current_url = browserHistory.back(2)

6. Now we move forward by 1 step:

4. We decide to move back by 1 step in the history:

1 current_url = browserHistory.forward(1)

We can't move forward because stk2 is empty, hence the current_url remains "youtube.com".

Visit the homepage, which clears forward history and records the provided homepage.

7. If we visit a new site "youtube.com": 1 browserHistory.visit("youtube.com")

erased due to this new visit. stk2 = [].

8. Finally, if we try to go forward by 2 steps:

1 current_url = browserHistory.forward(2)

def __init__(self, homepage: str):

self.backward_history = []

def visit(self, url: str) -> None:

self.forward_history.clear()

self.backward_history.append(url)

return self.backward_history[-1]

return self.backward_history[-1]

def forward(self, steps: int) -> str:

self.forward_history = []

self.visit(homepage)

Initialize stack for backward history and forward history

Stack 1 keeps all backward history from the current page.

Move the current URL to the forward history.

Decrease the steps we need to go back.

Decrease the steps we need to go forward.

38 # Your BrowserHistory object will be instantiated and called as such:

Return the current URL after moving forward.

Return the current URL after moving back.

moving URLs back to backward history.

while steps and self.forward_history:

self.forward_history.append(self.backward_history.pop())

Move one URL from forward history back to backward history.

self.backward_history.append(self.forward_history.pop())

Python Solution

1 class BrowserHistory:

6

9

10

11

12

13

14

20

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

43

47 */

C++ Solution

#include <stack>

private:

10 public:

13

14

15

16

17

19

21

22

23

24

25

26

27

28

29

31

39

30 }

steps--;

33 visit("homepage");

37 currentUrl = forward(1);

34 visit("url1");

35 visit("url2");

32 // Usage of the global functions

36 let currentUrl: string = back(1);

by the number of pages in stk2.

#include <string>

class BrowserHistory {

// The BrowserHistory class keeps track of browser navigation history.

// Constructor initializes the browser history with the homepage.

// Clear the forward stack since we visited a new URL.

forwardHistory = std::stack<std::string>();

// It returns the current URL after going back.

BrowserHistory(std::string homepage) {

visit(homepage);

void visit(std::string url) {

std::string back(int steps) {

backHistory.push(url);

std::stack<std::string> backHistory; // Stack to manage back navigation.

std::stack<std::string> forwardHistory; // Stack to manage forward navigation.

// The back method moves back 'steps' times in the history unless we are at the first page.

// Go back at most steps times and make sure not to go past the initial page.

// The visit method navigates to a new URL, clears the forward history, and pushes the URL onto the back stack.

48

["facebook.com"]. The current_url becomes "google.com".

So our final stacks look like this: stk1 = ["leetcode.com", "google.com", "youtube.com"] and stk2 = []. This series of actions demonstrates how the methods work together to simulate real-world browser history navigation.

Stack 2 is cleared every time a new page is visited and keeps pages for potential forward navigation.

Record the visited URL and clear the forward history, because new navigation breaks the forward path.

Go forward in the history the given number of steps if there are enough pages in forward history,

15 16 def back(self, steps: int) -> str: 17 # Move back in the history the given number of steps, if possible, # moving URLs from backward to forward history. 18 19 while steps and len(self.backward_history) > 1:

steps -= 1

steps -= 1

39 # obj = BrowserHistory(homepage)

41 # param_2 = obj.back(steps)

42 # param_3 = obj.forward(steps)

40 # obj.visit(url)

```
Java Solution
   class BrowserHistory {
       // Deques for navigating backward and forward
       private Deque<String> historyStack = new ArrayDeque<>();
       private Deque<String> forwardStack = new ArrayDeque<>();
       // Constructor stores the homepage and clears the forward history
       public BrowserHistory(String homepage) {
           visit(homepage);
8
9
10
       // Method to visit a new URL: Pushes the current URL to the history stack
11
12
       // and clears the forward history
       public void visit(String url) {
13
           historyStack.push(url);
14
           forwardStack.clear(); // Clear forward navigation because a new page is visited
15
16
17
       // Method to move 'steps' back in the browser history
18
       public String back(int steps) {
19
20
           // Pop from history stack and push into forward stack while more than 1 page is in history
21
           // and allowed steps are remaining
           while (steps > 0 && historyStack.size() > 1) {
               forwardStack.push(historyStack.pop());
23
24
               steps--;
26
           return historyStack.peek(); // Return the current URL
27
28
29
       // Method to move 'steps' forward in the browser history
       public String forward(int steps) {
30
           // Pop from forward stack and push into history stack while there are pages in forward history
31
32
           // and allowed steps are remaining
33
           while (steps > 0 && !forwardStack.isEmpty()) {
               historyStack.push(forwardStack.pop());
34
               steps--;
36
37
           return historyStack.peek(); // Return the current URL
38
39 }
40
   // Example on how to use the BrowserHistory class:
42 /*
  BrowserHistory browserHistory = new BrowserHistory("www.homepage.com");
  browserHistory.visit("www.page1.com");
45 String backPage = browserHistory.back(1); // Returns to 'www.homepage.com'
46 String forwardPage = browserHistory.forward(1); // Goes forward to 'www.pagel.com'
```

36 37 38 39 40 41

```
27
           while (steps && backHistory.size() > 1) {
28
               forwardHistory.push(backHistory.top());
               backHistory.pop();
29
30
               --steps;
31
32
           return backHistory.top(); // Return the current URL.
33
34
35
       // The forward method moves forward 'steps' times in the history if possible.
       // It returns the current URL after going forward.
       std::string forward(int steps) {
           // Go forward at most steps times and only if there is forward history available.
           while (steps && !forwardHistory.empty()) {
               backHistory.push(forwardHistory.top());
               forwardHistory.pop();
               --steps;
43
           return backHistory.top(); // Return the current URL.
44
45
46 };
47
   /* Usage of the BrowserHistory class
   BrowserHistory* browserHistory = new BrowserHistory("homepage");
  browserHistory->visit("url1");
51 browserHistory->visit("url2");
52 std::string currentUrl = browserHistory->back(1);
53 currentUrl = browserHistory->forward(1);
   delete browserHistory; // Don't forget to delete the object to avoid memory leak.
55 */
56
Typescript Solution
1 let backHistory: string[] = []; // Array to manage back navigation.
   let forwardHistory: string[] = []; // Array to manage forward navigation.
   // The visit function navigates to a new URL, clears the forward history, and pushes the URL onto the back stack.
   function visit(url: string): void {
       backHistory.push(url);
       forwardHistory = []; // Clear the forward history since we visited a new URL.
8 }
 9
   // The back function moves back 'steps' times in the history unless we are at the first page.
11 // It returns the current URL after going back.
12 function back(steps: number): string {
       // Go back at most 'steps' times and make sure not to go past the initial page.
13
       while (steps > 0 && backHistory.length > 1) {
14
           forwardHistory.push(backHistory.pop()!); // Assert non-null as we know there's more than one item.
16
           steps--;
17
       return backHistory[backHistory.length - 1]; // Return the current URL.
18
19 }
20
21 // The forward function moves forward 'steps' times in the history if possible.
22 // It returns the current URL after going forward.
   function forward(steps: number): string {
       // Go forward at most 'steps' times and only if there is forward history available.
24
25
       while (steps > 0 && forwardHistory.length > 0) {
           backHistory.push(forwardHistory.pop()!); // Assert non-null as the length check ensures the item exists.
26
```

Time and Space Complexity The time complexity for each operation in the BrowserHistory class is as follows:

// No need for a delete operation since there's no object allocation.

return backHistory[backHistory.length - 1]; // Return the current URL.

• back method: The worst-case time complexity is O(steps), which occurs when you move steps times back in the history. However, it cannot exceed n - 1 where n is the number of elements in stk1. This is because you can only move back as many

The space complexity is the sum of the space taken by stk1 and stk2:

times as there are pages in stk1 minus the current page. • forward method: The time complexity is O(steps) similarly to the back method, with the maximum number of steps being limited

visit method: The time complexity is 0(1) for appending a URL to stk1 and 0(n) for clearing stk2, where n is the number of

elements in stk2. However, since the clear operation assigns stk2 a new empty list, it can be considered 0(1) in practice.

- Space complexity: 0(m + k) where m is the number of total pages ever visited and k is the number of pages that have been moved back from using the back method. Note that $k \ll m$. Therefore, we can simplify the space complexity to O(m) which is
- dictated by the total number of unique pages visited throughout the use of the browser history.