Lab 1: Linked List

1 What is Node?Linked List?

The **node** is the basic building block of many data structures. Node has 2 functions:

- Its first function is that it holds a piece of data, also known as the **Value** node.
- The second function is its connectivity between another node and itself, using an object reference pointer, also known as the **Next** pointer.

Linked list is a linear data structure, which consists of a group of nodes in a sequence.

1.1 Advantages and Disadvantages

Advantages	Disadvantages
Dynamic Nature	More memory usage due to address pointer
Optimal insertion & deletion	Slow traversal compared to arrays
Stacks & queues can be easily implemented	No reverse traversal in singly linked list
No memory wastage	No random access

1.2 Real-life Applications

- Previous - n - next page in browser
- Image Viewer
- Music Player

2 Types of Linked List

There are 4 types of linked lists, but in general, we use 3 types only:

- **Singly-linked list:** Each node points to the next node, and the last node points to null.
- **Doubly-linked list:** Each node has two pointers, one pointing to the previous node and the other to the next node; the last node's pointer points to null.
- **Circular-linked list:** The last node points back to the first node.

2.1 Time Complexity

- Access: $O(n)$
- Search: $O(n)$
- Insert: $O(1)$
- Remove: $O(1)$

3 Warm-up Code Exercise

3.1 Creating a Linked List

Code:

```
#include <iostream>

class Node {
public:
    int data;
    Node* next;

    Node(int data) {
        this->data = data;
        this->next = nullptr;
    }
};

int main() {
    Node* n1 = new Node(10);
    Node* n2 = new Node(20);
    Node* n3 = new Node(30);

    Node* head = n1;
    head->next = n2;
    n2->next = n3;
    n3->next = nullptr;

    // Optional: Print the list to verify
    Node* current = head;
    while (current != nullptr) {
        std::cout << current->data << " ";
        current = current->next;
    }

    return 0;
}
```

Output:

```
 -------- -------- --------
| 10 | --> | 20 | --> | 30 |
 -------- -------- --------
```

3.2 Traversing a Linked List

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

template <typename TreeNode>
class Node {
public:
```

```
    TreeNode data;
    Node* next;

    Node(TreeNode data) {
        this->data = data;
        this->next = nullptr;
    }
};

void traverse(Node<int>* head) {
    Node<int>* curr = head;
    while (curr != nullptr) {
        cout << curr->data << " ";
        curr = curr->next;
    }
}

int main() {
    Node<int>* head = new Node<int>(1);
    head->next = new Node<int>(2);
    head->next->next = new Node<int>(3);

    traverse(head);

    // Clean up memory
    while (head != nullptr) {
        Node<int>* temp = head;
        head = head->next;
        delete temp;
    }

    return 0;
}
```

Output:

```
 ------   ------   ------
| 10 | --> | 20 | --> | 30 | --> X
 ------   ------   ------
curr^

print:= 10
                 ***********************************************

 ------   ------   ------
| 10 | --> | 20 | --> | 30 | --> X
 ------   ------   ------
        curr^

print:= 10, 20

                 ***********************************************
```

```
 ------   ------   ------
| 10 | --> | 20 | --> | 30 | --> X
 ------   ------   ------
              curr^

print:= 10, 20, 30
```

3.3 Inserting an Element in Linked List

```
#include <iostream>

void insert(int data, Node* head, int pos) {
    Node* toAdd = new Node(data);

    // Base Condition
    if (pos == 0) {
        toAdd->next = head;
        head = toAdd;
        return;
    }

    Node* prev = head;
    for (int i = 0; i < pos - 1; i++) {
        if (prev == nullptr) return; // Prevents accessing a null pointer
        prev = prev->next;
    }

    toAdd->next = prev->next;
    prev->next = toAdd;
}

int main() {
    Node* head = nullptr;
    insert(30, head, 3);

    // Code to print the list or further manipulate it can be added here

    return 0;
}
```

Output:

```
                    Given:
  -------- -------- -------- -------- --------
| 5 | --> | 10 | --> | 5 | --> | 24 | --> | 40 |
  -------- -------- -------- -------- --------
              Insert 30 at index 3
```

3.4 Deleting an Element from Linked List

```
void deleteNode(Node* head, int pos) {
    if (head == nullptr) return;
```

```
    // Base Condition
    if (pos == 0) {
        Node* temp = head;
        head = head->next;
        delete temp;
        return;
    }

    Node* prev = head;
    for (int i = 0; i < pos - 1 && prev->next != nullptr; i++) {
        prev = prev->next;
    }

    if (prev->next == nullptr) return;

    Node* toDelete = prev->next;
    prev->next = prev->next->next;
    delete toDelete;
}
```

Output:

```
                    Given:
        -------- -------- -------- -------- --------
| 5 | --> | 10 | --> | 15 | --> | 12 | --> | 14 | --> X |
        -------- -------- -------- -------- --------
          delete 3rd element from linked list
```

4 In-Class Exercise - Part 1

1. **Find the Middle Node of a Linked List** *Easy*
2. **Detect a Cycle in a Linked List** *Easy*
3. **Combine Two Sorted Linked Lists** *Easy*
4. **Find the Intersection of Two Linked Lists** *Easy*
5. **Reverse a Linked List** *Easy*
6. **Eliminate Duplicates from a Sorted Linked List** *Easy*
7. **Check if a Linked List is a Palindrome** *Easy*
8. **Write a function to search for nodes with the value X in the list. If found, return the addresses of the nodes; if not found, return NULL.** *Easy*

5 In-Class Exercise - Part 2

1. **Add Two Numbers** *Medium*
2. **Copy List with Random Pointers** *Medium*
3. **Swap Nodes in a Linked List** *Medium*

4. **Remove the N-th Node from the End of a List** *Medium*
5. **Separate Odd and Even Nodes in a Linked List** *Medium*
6. **Divide a Linked List into Parts** *Medium*
7. **Remove Zero-Sum Consecutive Nodes from a Linked List** *Medium*
8. **Write a function to input values for a list using the automatic input method, with values selected from the range [-99; 99]. The number of entries is randomly chosen from the range [39; 59] (using a function to insert at the end of the list)** *Medium*

6 Homework

6.1 Question 1: Music player

You are tasked with designing a simple music player using a linked list. The player should be able to perform the following operations:

1. **Add a Song**: Add a song to the end of the playlist.
2. **Play Next**: Move to the next song in the playlist. If at the end, loop back to the first song.
3. **Play Previous**: Move to the previous song in the playlist. If at the beginning, loop to the last song.
4. **Remove a Song**: Remove a song by its name from the playlist.
5. **Display Playlist**: Output the current playlist in order.

Input

- The first line contains an integer n, the number of operations.
- The next n lines contain operations in the format:
  - `ADD <song_name>` to add a song.
  - `NEXT` to play the next song.
  - `PREV` to play the previous song.
  - `REMOVE <song_name>` to remove a song.
  - `DISPLAY` to display the current playlist.

Output

For each `DISPLAY` operation, output the current playlist as a space-separated list of song names.

Constraints

- $1 \leq n \leq 10^5$
- Song names are unique and consist of lowercase English letters.

Example

```
Input:
6
ADD song1
ADD song2
NEXT
DISPLAY
REMOVE song1
DISPLAY

Output:
song2 song1
song2
```

1. How would you handle edge cases, such as trying to remove a song that doesn't exist?
2. How can you efficiently loop back to the start or end of the list?
3. What data structure(s) would you use to implement this playlist, and why?
4. How would you ensure that operations like `NEXT` and `PREV` are efficient, given the constraints?

6.2 Question 2: Web Browser back and next

In a web browser that operates with a single tab, the user starts on the homepage and can navigate to various URLs, retrace their steps in the browsing history, or advance through it.

To implement this functionality, the 'BrowserHistory' class will be designed with the following specifications:

- **Initialization**: The constructor 'BrowserHistory(string homepage)' initializes the instance with the designated homepage URL.
- **Visiting URLs**: The method 'void visit(string url)' allows the user to navigate to a new URL from the current page, while simultaneously clearing any forward history.
- **Navigating Back**: The method 'string back(int steps)' enables the user to move backward through the history by a specified number of steps. If the number of steps exceeds the available history, the user will return only as far back as possible. This method returns the current URL after the backward movement.
- **Navigating Forward**: The method 'string forward(int steps)' allows the user to advance through the history by a specified number of steps. Similar to the back method, if the number of steps exceeds the available forward history, the user will only move forward as far as possible. The current URL is returned after this operation.

Example Usage

Consider the following sequence of operations:

```
Input:
["BrowserHistory", "visit", "visit", "visit", "back", "back", "forward",

"visit", "forward", "back", "back"] [["uit.edu.vn"], ["google.com"], ["facebook.com"],
```

```
["youtube.com"], [1], [1], [1], ["linkedin.com"], [2], [2], [7]]

Output:
[null, null, null, null, "facebook.com", "google.com", "facebook.com", null, "linkedin.com",

"google.com", "uit.edu.vn"]
```

Explanation

1. An instance of ‘BrowserHistory‘ is created with the homepage set to ”leetcode.com”.
2. The user visits ”google.com”, then ”facebook.com”, and subsequently ”youtube.com”.
3. The user navigates back once, returning to ”facebook.com”, and then again to ”google.com”.
4. The user moves forward one step to ”facebook.com”.
5. The user visits ”linkedin.com”, clearing the forward history.
6. The user attempts to move forward two steps but cannot, as there is no forward history.
7. The user navigates back two steps, returning to ”facebook.com” and then ”google.com”.
8. Finally, the user attempts to move back seven steps but can only return to ”leetcode.com”.

Constraints

- The length of the homepage string is between 1 and 20 characters.
- The length of any URL is also between 1 and 20 characters.
- The number of steps for navigation is limited to a maximum of 100.
- Both the homepage and URLs consist of lowercase English letters and periods.
- The total number of method calls to ‘visit‘, ‘back‘, and ‘forward‘ will not exceed 5000.

An authentication system utilizes authentication tokens to manage user sessions. For each session, a unique authentication token is issued to the user, which is set to expire after a specified duration, known as `timeToLive`, measured in seconds from the current time. If the token is renewed, the expiration time is adjusted to extend `timeToLive` seconds from the new current time, which may differ from the previous one.

6.3 Question 3: AuthenticationManager Class

The `AuthenticationManager` class should be implemented with the following functionalities:

- **Constructor**: The constructor `AuthenticationManager(int timeToLive)` initializes the authentication manager with the defined `timeToLive` duration.
- **Token Generation**: The method `generate(string tokenId, int currentTime)` creates a new token associated with the specified `tokenId` at the given `currentTime` in seconds.
- **Token Renewal**: The method `renew(string tokenId, int currentTime)` allows for the renewal of an unexpired token identified by `tokenId` at the specified `currentTime`. If no unexpired token exists for the provided `tokenId`, the request is disregarded, resulting in no action taken.

- **Counting Unexpired Tokens**: The method `countUnexpiredTokens(int currentTime)` returns the total number of tokens that remain unexpired at the specified `currentTime`. It is crucial to note that if a token's expiration occurs at time t, any subsequent actions, such as renewal or counting unexpired tokens, will consider the expiration to have taken place prior to those actions.

Example Usage

Consider the following sequence of operations:

```
Input:
["AuthenticationManager", "renew", "generate", "countUnexpiredTokens", "generate", "renew",

"renew", "countUnexpiredTokens"]
[[5], ["aaa", 1], ["aaa", 2], [6], ["bbb", 7], ["aaa", 8], ["bbb", 10], [15]]
Output:
[null, null, null, 1, null, null, null, 0]
```

Explanation

1. An instance of `AuthenticationManager` is created with a `timeToLive` of 5 seconds.
2. The `renew` method is called for `tokenId "aaa"` at time 1, but no token exists, so no action is taken.
3. A new token with `tokenId "aaa"` is generated at time 2.
4. At time 6, the `countUnexpiredTokens` method returns 1, as the token with `tokenId "aaa"` is still valid.
5. A new token with `tokenId "bbb"` is generated at time 7.
6. The `renew` method for `tokenId "aaa"` is called at time 8, but the token has expired at time 7, so the request is ignored.
7. The `renew` method for `tokenId "bbb"` is executed at time 10, successfully renewing the token, which will now expire at time 15.
8. Finally, at time 15, the `countUnexpiredTokens` method is invoked, which returns 0, as both tokens have expired.

Constraints

- The value of `timeToLive` must be between 1 and 10^8.
- The `currentTime` must also fall within the range of 1 to 10^8.
- The length of `tokenId` is restricted to a maximum of 5 characters, consisting solely of lowercase letters.
- Each call to `generate` will utilize unique `tokenId` values.
- The values of `currentTime` across all function calls will be strictly increasing.
- The total number of calls to all functions combined will not exceed 2000.

Notice

- Use C++ for practice.
- In the programming file, the student should include the following complete information:

```
//STT: 39 (Example)
//Full Name: X, With X is you, don't need to find X anywhere else.
//Session 01 - Exercise 01
//Notes or Remarks: ......
```

References :

[1]. Skiena, S. S. (1998). The algorithm design manual (Vol. 2). New York: springer.
[2]. Pai, G. V. (2023). A Textbook of Data Structures and Algorithms, Volume 3: Mastering Advanced Data Structures and Algorithm Design Strategies. John Wiley & Sons.
[3]. Anggoro, W. (2018). C++ Data Structures and Algorithms: Learn how to write efficient code to build scalable and robust applications in C++. Packt Publishing Ltd.
[3].Leetcode
[4].Codeforce