

Homework #1: Circular Doubly Linked List

Name: 여승엽

Student ID: 20215414

1. Introduction

1.1. What is circular linked list?

▶ A circular linked list is a variation of a linked list in which the last node points back to the first node instead of null. This creates a circular structure where traversal can continue indefinitely. Circular linked lists can be either singly or doubly linked and are often used in applications that require a continuous loop, such as scheduling, buffering, or round-robin algorithms.

1.2. What is doubly linked list?

▶ A doubly linked list is a type of linked list in which each node contains two pointers: one pointing to the next node and the other pointing to the previous node. This bidirectional structure allows traversal in both forward and backward directions, offering more flexibility than singly linked lists. Doubly linked lists are useful in scenarios where quick backward navigation is required, such as in web browsers or undo functionality.

1.3. What is circular doubly linked list?

▶ A circular doubly linked list combines the properties of both circular and doubly linked lists. In this structure, each node has two pointers (next and previous), and the last node's next pointer links to the first node, while the first node's previous pointer links back to the last node. This enables continuous traversal in both directions without encountering a null reference, making it especially useful for applications like media playlists, memory management, or navigation systems.

2. Implementation (Explain for core functions)

: What have you modified from the code in the lecture notes?

► <Focused Area>

Based on the code provided in the lecture notes, the core functions such as the constructor, destructor, empty, forward, backward, add, and remove were almost left unchanged. Instead, the focus was on modifying the operator<< function to match the required output format exactly as shown in the executable results.

► <Modifications : operator>

A significant portion of the modifications focused on the `operator<<` function to ensure that the printed output of the program matches the required format provided in the executable results.

The function was rewritten to traverse the list in both forward and backward directions, starting from the node next to the cursor and ending at the cursor itself. During traversal, arrows (`->`) were inserted appropriately between elements, and the current cursor position was marked using an asterisk (`*`). This modification allows for intuitive visualization of the list's structure while ensuring strict conformity with the expected output format.

► <code>

```
ostream& operator<<(ostream& out, const CDlist& c) {
    if (c.empty()) {
        out << "ERROR: cannot print. The list is empty" << std::endl << std::endl;
        return out;
    }

    // === Forward hopping ===
    out << "Forward hopping: ";
    CDNode* v = c.cursor->next;
    do {
        if (v == c.cursor) out << v->elem << "*";
        else out << v->elem;
        v = v->next;
        if (v != c.cursor->next) out << "->";
    } while (v != c.cursor->next);
    out << std::endl;

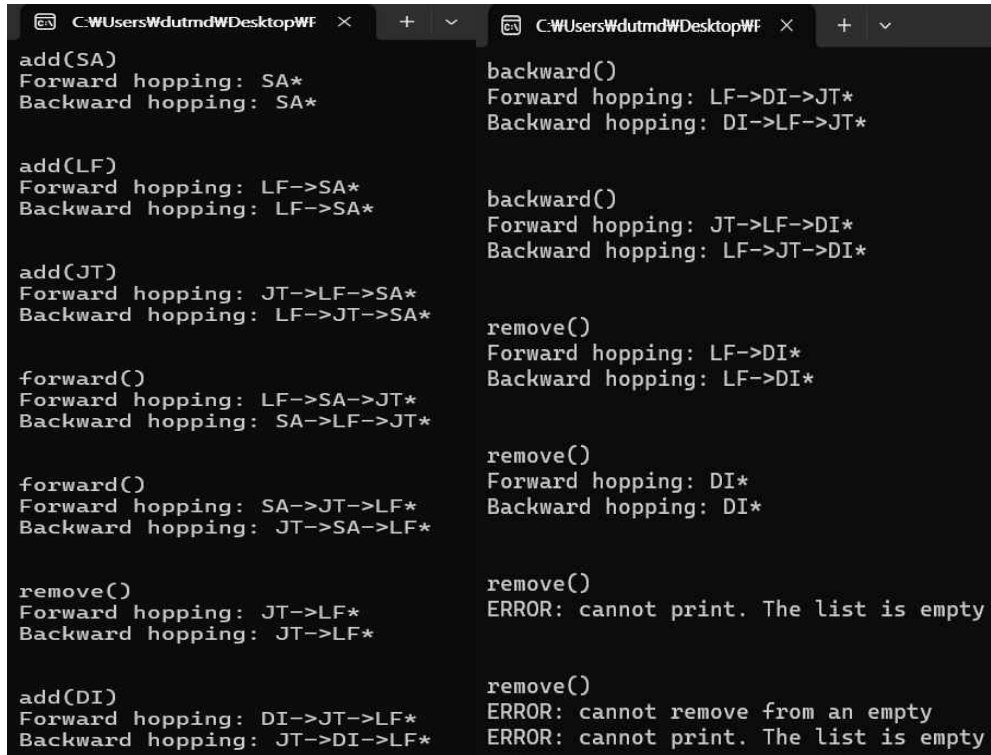
    // === Backward hopping ===
    out << "Backward hopping: ";
    v = c.cursor->prev;
    CDNode* stop = v;
    do {
        if (v == c.cursor) out << v->elem << "*";
        else out << v->elem;
        v = v->prev;
        if (v != stop) out << "->";
    } while (v != stop);
    out << std::endl << std::endl;

    return out;
}
```

3. Result (Results of the provided main.cpp)

3.1. Figures

► exe result(1, 2 screenshot)



```
C:\Users\Wdutmnd\Desktop\WF x + v C:\Users\Wdutmnd\Desktop\WF x + v
add(SA)
Forward hopping: SA*
Backward hopping: SA*

add(LF)
Forward hopping: LF->SA*
Backward hopping: LF->SA*

add(JT)
Forward hopping: JT->LF->SA*
Backward hopping: LF->JT->SA*

forward()
Forward hopping: LF->SA->JT*
Backward hopping: SA->LF->JT*

forward()
Forward hopping: SA->JT->LF*
Backward hopping: JT->SA->LF*

remove()
Forward hopping: JT->LF*
Backward hopping: JT->LF*

add(DI)
Forward hopping: DI->JT->LF*
Backward hopping: JT->DI->LF*

backward()
Forward hopping: LF->DI->JT*
Backward hopping: DI->LF->JT*

backward()
Forward hopping: JT->LF->DI*
Backward hopping: LF->JT->DI*

remove()
Forward hopping: LF->DI*
Backward hopping: LF->DI*

remove()
Forward hopping: DI*
Backward hopping: DI*

remove()
ERROR: cannot print. The list is empty

remove()
ERROR: cannot remove from an empty
ERROR: cannot print. The list is empty
```

3.2. Explanations

► The above execution result demonstrates the core functionality of a circular doubly linked list before any additional features were implemented.

- The **`add()`** function inserts a new node immediately after the current cursor. As new nodes are added, they are inserted between the cursor and its next node, while preserving the circular structure of the list. The output clearly shows the updated node order after each insertion.

- The **`forward()`** function moves the cursor one node forward in the list. This change is reflected in both the forward and backward hopping outputs, where the asterisk (`*`) indicates the current cursor position.

- The **`backward()`** function moves the cursor one node backward. Like the **`forward()`** function, it updates the cursor and affects the list traversal output accordingly.

- The `remove()` function deletes the node immediately following the cursor. When the list becomes empty, appropriate error messages are displayed to indicate that further operations are not possible.
- The `operator<<` function prints the current state of the list in both forward and backward directions, allowing a clear visualization of node connections and the cursor's location.

4. Additional Result (Results after modifying the main.cpp (your choice))

4.1. Figures

```
Enter an element to add: Q
Q has been added to the playlist.

Enter an element to add: W
W has been added to the playlist.

Enter an element to add: E
E has been added to the playlist.

Enter an element to check if it exists in the list: Q
Q is in the playlist.
Current playlist length: 3

After clear:
ERROR: cannot print. The list is empty

Press Enter to exit...
```

4.2. Explanations

- In the extended portion of `main.cpp`, a set of interactive features was added while preserving the original structure of the function. These additions enhance user interaction by allowing input-based control of the playlist. These additions demonstrate how interactive list management operations can be handled directly through class methods, keeping the `main()` function clean and minimal.
- `playlist.addByInput()` is called three times, prompting the user to enter three elements. Each input is immediately added to the circular doubly linked list using the internally defined `add()` function.
- `playlist.checkByInput()` allows the user to input a single value and checks whether that value exists in the current list. It then prints the result along with the current length of the list using the `contains()` and `length()` functions.
- `playlist.clear()` is called to empty the entire list, and the list is then printed to confirm that it has been cleared.
- Finally, to prevent the console from closing immediately after execution, a prompt is displayed asking the user to press Enter.

Appendix A. CDList.cpp (origin)

```
#include "CDList.h"
```

```
// Constructor: Initializes the list as empty by setting cursor to nullptr
```

```
CDList::CDList() : cursor(nullptr) {}
```

```
// Destructor: Deletes all nodes in the list to prevent memory leaks
```

```
CDList::~~CDList() {
```

```
    while (!empty()) {
```

```
        CDNode* old = cursor->next; // Node to delete (the one after cursor)
```

```
        if (old == cursor) {           // Only one node exists
```

```
            delete old;
```

```
            cursor = nullptr;
```

```
        } else {
```

```
            // Re-link the list to bypass the node being deleted
```

```
            cursor->next = old->next;
```

```
            old->next->prev = cursor;
```

```
            delete old;
```

```
        }
```

```
    }
```

```
}
```

```
// Checks if the list is empty (cursor is null)
```

```
bool CDList::empty() const {
```

```
    return cursor == nullptr;
```

```
}
```

```
// Returns the element at the front (right after the cursor)
```

```
const Elem& CDList::front() const {
```

```
    return cursor->next->elem;
```

```
}
```

```
// Returns the element at the back (at the cursor)
```

```
const Elem& CDList::back() const {
```

```
    return cursor->elem;
```

```
}
```

```
// Moves the cursor one node forward (clockwise direction)
```

```
void CDList::forward() {  
    if (!empty()) cursor = cursor->next;  
}
```

```
// Moves the cursor one node backward (counterclockwise direction)
```

```
void CDList::backward() {  
    if (!empty()) cursor = cursor->prev;  
}
```

```
// Inserts a new node with element e immediately after the cursor
```

```
void CDList::add(const Elem& e) {  
    CDNode* v = new CDNode(e); // Create a new node with the given element  
    if (cursor == nullptr) {  
        // If the list is empty, the new node points to itself  
        v->next = v;  
        v->prev = v;  
        cursor = v; // Cursor points to the new node  
    } else {  
        // Link new node between cursor and cursor->next  
        v->next = cursor->next;  
        v->prev = cursor;  
        cursor->next->prev = v;  
        cursor->next = v;  
    }  
}
```

```
// Removes the node immediately after the cursor
```

```
void CDList::remove() {  
    if (empty()) {  
        std::cout << "ERROR: cannot remove from an empty" << std::endl;  
        return;  
    }  
}
```

```

CDNode* old = cursor->next; // Node to be deleted
if (old == cursor) {
    // If the list has only one node, delete it and empty the list
    delete old;
    cursor = nullptr;
} else {
    // Re-link to remove old node from the list
    cursor->next = old->next;
    old->next->prev = cursor;
    delete old;
}
}

// Overloaded << operator to print the list in both forward and backward directions
ostream& operator<<(ostream& out, const CDList& c) {
    if (c.empty()) {
        out << "ERROR: cannot print. The list is empty" << std::endl << std::endl;
        return out;
    }

    // === Forward hopping ===
    out << "Forward hopping: ";
    CDNode* v = c.cursor->next; // Start from the node after cursor
    do {
        if (v == c.cursor) out << v->elem << "*"; // Mark cursor position
        else out << v->elem;
        v = v->next;
        if (v != c.cursor->next) out << "->"; // Arrow between nodes
    } while (v != c.cursor->next);
    out << std::endl;

    // === Backward hopping ===
    out << "Backward hopping: ";
    v = c.cursor->prev; // Start from the node before cursor

```

```

CDNode* stop = v;           // Stopping point for backward traversal
do {
    if (v == c.cursor) out << v->elem << "*"; // Mark cursor position
    else out << v->elem;
    v = v->prev;
    if (v != stop) out << "->"; // Arrow between nodes
} while (v != stop);
out << std::endl << std::endl;

return out;
}

```

Appendix B. CDList.h (origin)

```

//-----<<Provided (head)>>-----
#ifndef CDLIST_H           // Prevents multiple inclusion of this header file
#define CDLIST_H

#include <iostream>        // For input/output streams
#include <string>          // For using the string type

using std::string;       // Allows using 'string' instead of 'std::string'
using std::ostream;      // Allows using 'ostream' instead of 'std::ostream'

typedef string Elem;      // Defines 'Elem' as an alias for string (easy to change type
                           later)

//-----<<Provided (tail)>>-----

// Structure definition for a node in the doubly linked list
struct CDNode {
    Elem elem;            // Data stored in the node
    CDNode* next;         // Pointer to the next node
    CDNode* prev;         // Pointer to the previous node
}

```



```

    // Constructor to initialize a node with a given element
    CDNode(const Elem& e) : elem(e), next(nullptr), prev(nullptr) {}
};

// Class definition for a Circular Doubly Linked List
class CDList {
private:
    CDNode* cursor;    // Pointer to the current position in the list

public:
    CDList();           // Constructor: creates an empty list
    ~CDList();          // Destructor: deallocates all nodes

    bool empty() const;    // Checks if the list is empty
    const Elem& front() const; // Returns the element after the cursor
    const Elem& back() const; // Returns the element at the cursor

    void forward();        // Moves the cursor forward by one node
    void backward();       // Moves the cursor backward by one node

    void add(const Elem& e); // Inserts a new node after the cursor
    void remove();          // Deletes the node after the cursor

    // Overloaded output operator for printing the entire list
    // Declared as a friend to allow access to private members
    friend ostream& operator<<(ostream& out, const CDList& c);
};

#endif // CDLIST_H

```

Appendix C. CDList.cpp / CDList.h / main.cpp

-> I will attach the cpp and h files with new features separately to the zip file.