Doubly linked list

Code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

typedef struct node{
    int value;
    struct node *prev;
    struct node *next;
} node;

typedef struct{
    node *head;
    node *tail;
}doubullyLinkedList;

node* createnode(int element){
    node * n = (node*)malloc(sizeof(node));
    if(n){
        n->value = element;
        n->prev = NULL;
        n->next = NULL;
    }
    return n;
}

void prepend(doubullyLinkedList *list, int element){
    node* n = createnode(element);
    if(n){
        n->next= list->head;
        if(list->head)list->head->prev = n;
        else list->tail = n;
        list->head = n;
    }
}

void append(doubullyLinkedList *list, int element){
    node* n = createnode(element);
    if(n){
        n->prev = list->tail;
        if(list->tail) list->tail->next = n;
```

```
        else list->head = n;
        list->tail = n;
    }
}

void insertSorted(doubullyLinkedList *list, int element){
    node* n = createnode(element);
    if(n){
        node *iterator = list->head;
        while(iterator && iterator->value < element) iterator = iterator-
>next;
        if(iterator == NULL)append(list, element);
        else if (iterator == list->head) prepend(list, element);
        else{
            n->next = iterator;
            n->prev = iterator->prev;
            iterator->prev->next = n;
            iterator->prev = n;
        }
    }
}

void insertAfter(doubullyLinkedList *list, int element, int after){
    node* n = createnode(element);
    node *iterator = list->head;
    while(iterator && iterator->value != after) iterator = iterator->next;
    if(iterator){
        n->next = iterator->next;
        if(iterator->next)iterator->next->prev = n;
        else list->tail = n;
        iterator->next = n;
        n->prev = iterator;
    }
}

void destroy(doubullyLinkedList *list, node *n){
    if(n->prev) n->prev->next = n->next;
    else list->head = n->next;
    if(n->next) n->next->prev = n->prev;
    else list->tail= n->prev;
    free(n);
}

void delNode(doubullyLinkedList *list, int element){
    node *iterator = list->head;
    while(iterator && iterator->value != element) iterator = iterator->next;
    if(iterator) destroy(list, iterator);
}
```

```
void delFront(doubullyLinkedList *list){
    if(list->head)destroy(list, list->head);
}

void delBack(doubullyLinkedList *list){
    if(list->tail)destroy(list, list->tail);
}

void printList(doubullyLinkedList *list){
    if(list == NULL){
        printf("List is empty!\n");
        return;
    }
    for(node* ptr = list->head; ptr != NULL; ptr = ptr->next) printf("%d ",
ptr->value);
    printf("\n");
}

bool find(doubullyLinkedList *list, int element){
    for(node* iterator = list->head; iterator; iterator = iterator->next)
if(iterator->value == element) return true;
    return false;
}

int main(){
    doubullyLinkedList list;
    list.head=list.tail=NULL;

    prepend(&list, 3);
    prepend(&list, 37);
    append(&list, 5);
    insertSorted(&list, 4);
    insertAfter(&list, 2, 3);
    insertAfter(&list, 6, 7);
    insertSorted(&list, 56);
    printList(&list);
    printf("Is 4 in the list? %s\n", find(&list, 4) ? "FOUND" : "NOT FOUND");
    printf("Is 6 in the list? %s\n", find(&list, 6) ? "FOUND" : "NOT FOUND");
    delNode(&list, 3);
    printList(&list);
    delFront(&list);
    printList(&list);
    delBack(&list);
    printList(&list);

    return 0;
}
```

Explaination:

This code implements a Doubly Circular Linked List in C. Let's go through each function and explain its approach:

1. Structure Definitions: The code defines a structure `node` to represent a node in the linked list and a structure `doubullyLinkedList` to represent the doubly linked list itself. Each node contains an integer value, and pointers to the previous and next nodes.

2. Creating a New Node (`createnode`): This function allocates memory for a new node, initializes its value to the given element, and sets its `prev` and `next` pointers to `NULL`.

3. Prepending (`prepend`): This function adds a new node with the specified element to the beginning of the linked list. It creates a new node, updates its `next` pointer to point to the current head of the list, updates the previous pointer of the current head (if it exists) to point to the new node, and updates the `head` pointer to point to the new node.

4. Appending (`append`): Similar to prepend, this function adds a new node with the specified element to the end of the linked list. It creates a new node, updates its `prev` pointer to point to the current tail of the list (if it exists), updates the next pointer of the current tail (if it exists) to point to the new node, and updates the `tail` pointer to point to the new node.

5. Inserting in Sorted Order (`insertSorted`): This function inserts a new node with the specified element into the list while maintaining sorted order. It traverses the list until it finds the correct position to insert the new node. Then, it updates the pointers of the adjacent nodes to include the new node.

6. Inserting After a Specific Node (`insertAfter`): This function inserts a new node with the specified element after a node with a given value. It traverses the list until it finds the node with the specified value, then inserts the new node after it by updating the appropriate pointers.

7. Destroying a Node (`destroy`): This function removes a node from the list and frees the memory allocated for it. It updates the pointers of the adjacent nodes to bypass the node to be destroyed and then frees its memory.

8. Deleting a Node (`delNode`): This function deletes a node with a specific value from the list. It traverses the list to find the node with the specified value and then calls the `destroy` function to remove it.

9. Deleting the Front Node (`delFront`): This function deletes the front node of the list by calling `destroy` with the head of the list.

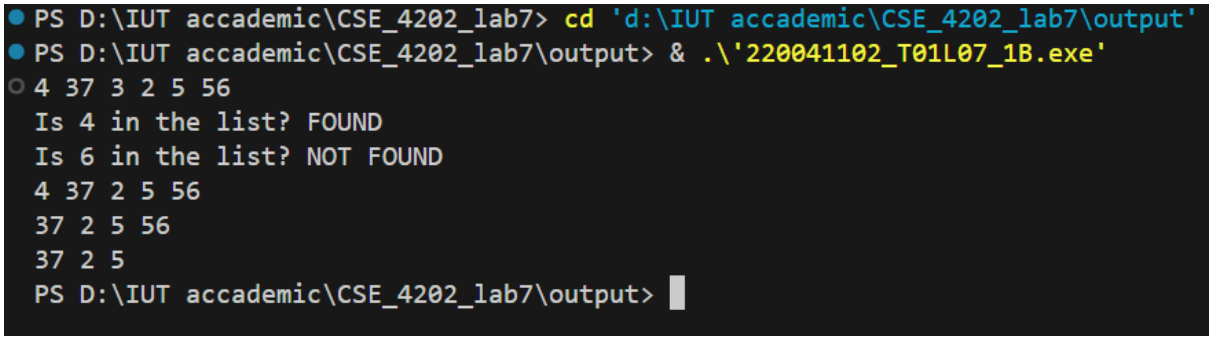
10. Deleting the Back Node (`delBack`): This function deletes the back node of the list by calling `destroy` with the tail of the list.

11. Printing the List (`printList`): This function prints the elements of the list by traversing it from head to tail and printing each element.

12. Finding an Element (`find`): This function searches for a specific element in the list. It traverses the list and returns `true` if the element is found, otherwise `false`.

The code works correctly because it appropriately handles the creation, insertion, deletion, and traversal of nodes in the doubly circular linked list. It ensures that the pointers are updated correctly to maintain the structure of the list, and it handles edge cases such as an empty list or deleting the only node in the list. Overall, this implementation provides a functional and efficient way to manage a doubly circular linked list in C.

Output:


PS D:\IUT accademic\CSE_4202_lab7> cd 'd:\IUT accademic\CSE_4202_lab7\output'
PS D:\IUT accademic\CSE_4202_lab7\output> & .\'220041102_T01L07_1B.exe'
4 37 3 2 5 56
Is 4 in the list? FOUND
Is 6 in the list? NOT FOUND
4 37 2 5 56
37 2 5 56
37 2 5
PS D:\IUT accademic\CSE_4202_lab7\output>