Circular Linked List

A **circular linked list** is a data structure where the last node connects back to the first, forming a loop. This structure allows for continuous traversal without any interruptions. Circular linked lists are especially helpful for tasks like **scheduling** and **managing playlists,** this allowing for smooth navigation. In this tutorial, we'll cover the basics of circular linked lists, how to work with them, their advantages and disadvantages, and their applications.

What is a Circular Linked List?

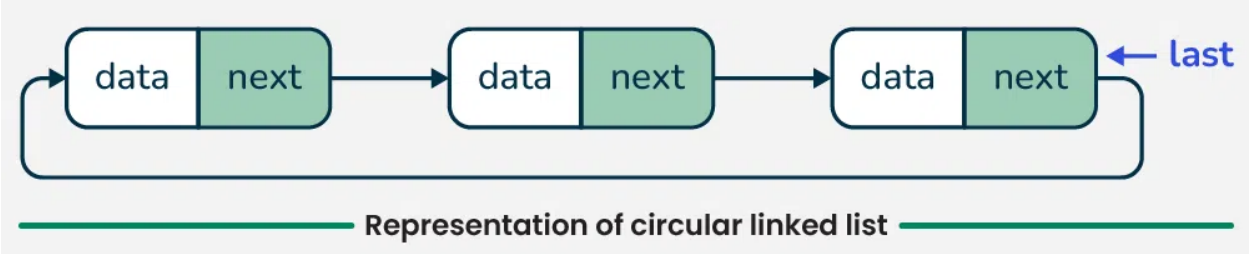
A **circular linked list** is a special type of linked list where all the nodes are connected to form a circle. Unlike a regular linked list, which ends with a node pointing to **NULL**, the last node in a circular linked list points back to the first node. This means that you can keep traversing the list without ever reaching a **NULL** value.

Types of Circular Linked Lists

We can create a circular linked list from both singly linked lists and doubly linked lists. So, circular linked list are basically of two types:

1. Circular Singly Linked List

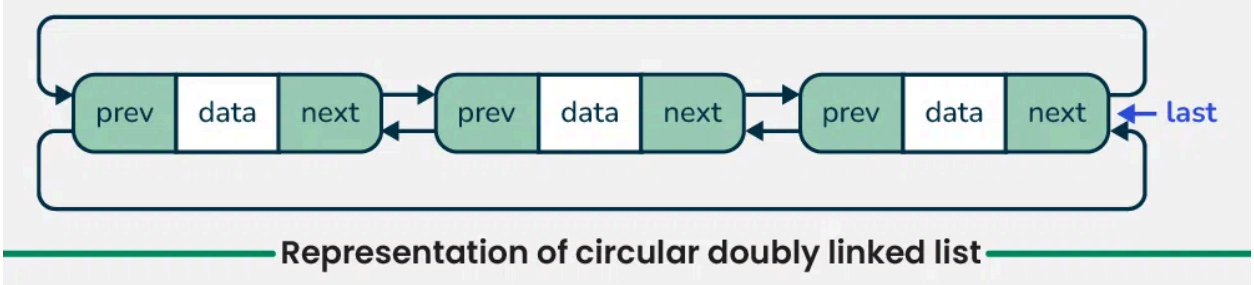
In **Circular Singly Linked List**, each node has just one pointer called the "**next**" pointer. The next pointer of **last node** points back to the **first node** and this results in forming a circle. In this type of Linked list we can only move through the list in one direction.



Representation of Circular Singly Linked List

2. Circular Doubly Linked List:

In **circular doubly linked list,** each node has two pointers **prev** and **next,** similar to doubly linked list. The **prev** pointer points to the previous node and the **next** points to the next node. Here, in addition to the **last** node storing the address of the first node, the **first node** will also store the address of the **last node**.



Representation of Circular Doubly Linked List

Note: In this article, we will use the circular singly linked list to explain the working of circular linked lists.

Representation of a Circular Singly Linked List

Let's take a look on the structure of a circular linked list.

Representation of a Circular Singly Linked List

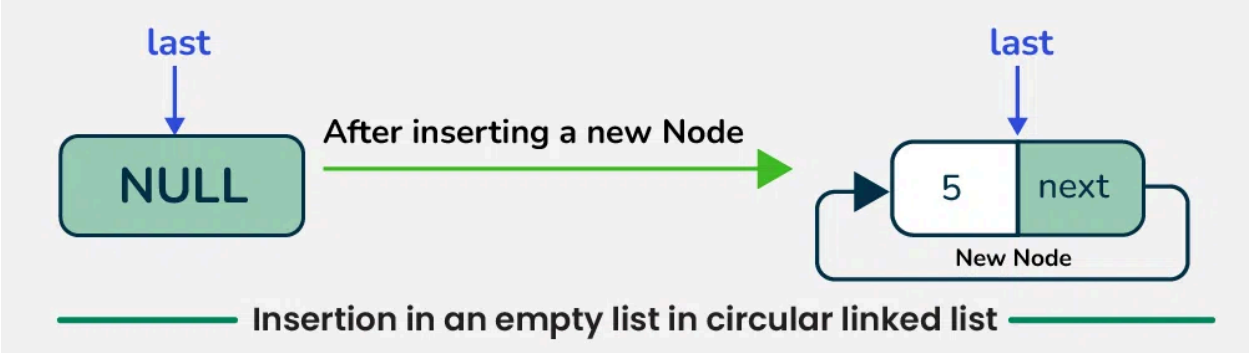
Operations on the Circular Linked list:

We can do some operations on the circular linked list similar to the singly and doubly linked list which are:

- **Insertion**
    - Insertion at the empty list
    - Insertion at the beginning
    - Insertion at the end
    - Insertion at the given position
- **Deletion**
    - Delete the first node
    - Delete the last node
    - Delete the node from any position
- **Searching**

1. Insertion in an empty List in the circular linked list

*To insert a node in empty circular linked list, creates a **new node** with the given data, sets its next pointer to point to itself, and updates the **last** pointer to reference this **new node**.*




```c
#include <stdio.h>

#include <stdlib.h>



// Define the Node structure

struct Node {

    int data;

    struct Node* next;

};



struct Node* createNode(int value);



// Function to insert a node into an empty

// circular singly linked list

struct Node* insertInEmptyList(struct Node* last, int data) {

    if (last != NULL) return last;
```

```c
    // Create a new node
    struct Node* newNode = createNode(data);

    // Update last to point to the new node
    last = newNode;
    return last;
}

void printList(struct Node* last) {
    if (last == NULL) return;

    // Start from the head node
    struct Node* head = last->next;
    while (1) {
        printf("%d ", head->data);
        head = head->next;
        if (head == last->next) break;
    }
    printf("\n");
```

```
}

struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = newNode;
    return newNode;
}

int main() {
    struct Node* last = NULL;

    // Insert a node into the empty list
    last = insertInEmptyList(last, 1);

    // Print the list
    printf("List after insertion: ");
    printList(last);

    return 0;
```

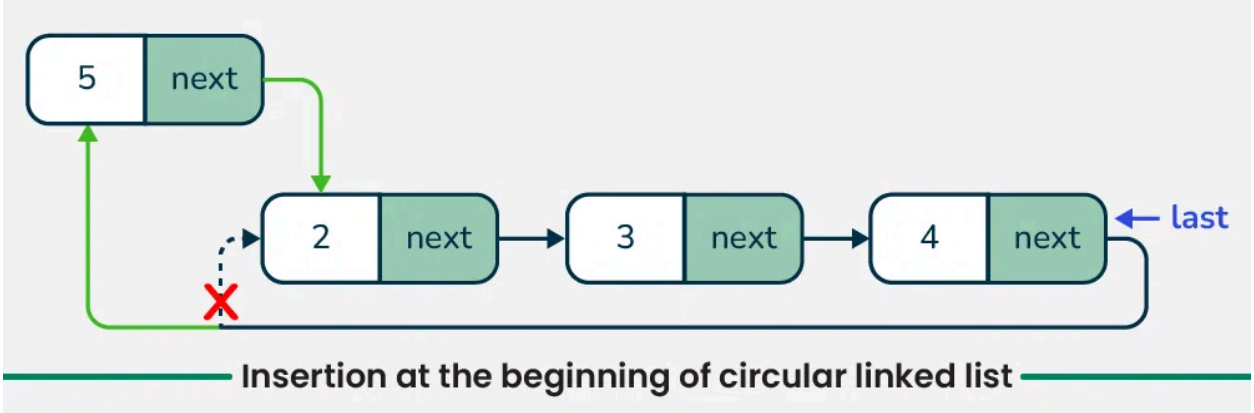
```
}
```

Output

```
List after insertion: 1
```

2. Insertion at the beginning in circular linked list

*To insert a new node at the beginning of a circular linked list, we first create the **new node** and allocate memory for it. If the list is empty (indicated by the last pointer being **NULL**), we make the **new node** point to itself. If the list already contains nodes then we set the **new node's** next pointer to point to the **current head** of the list (which is **last->next**), and then update the last node's next pointer to point to the **new node**. This maintains the circular structure of the list.*




```
#include <stdio.h>

#include <stdlib.h>
```

```c
// Define the Node structure
struct Node
{
    int data;
    struct Node *next;
};

// Function to create a new node
struct Node *createNode(int value)
{
    struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}

// Function to insert a node at the beginning
// of the circular linked list
struct Node *insertAtBeginning(struct Node *last, int value)
```

```
{
    struct Node *newNode = createNode(value);

    // If the list is empty, make the new node point to itself
    // and set it as last
    if (last == NULL)
    {
        newNode->next = newNode;
        return newNode;
    }

    // Insert the new node at the beginning
    newNode->next = last->next;
    last->next = newNode;

    return last;
}

void printList(struct Node *last)
{
```

```
    if (last == NULL) return;

    struct Node *head = last->next;
    while (1){
        printf("%d ", head->data);
        head = head->next;
        if (head == last->next)
            break;
    }
    printf("\n");
}

int main()
{
    // Create circular linked list: 2, 3, 4
    struct Node *first = createNode(2);
    first->next = createNode(3);
    first->next->next = createNode(4);
    struct Node *last = first->next->next;
    last->next = first;
```

```
    printf("Original list: ");
    printList(last);

    // Insert 5 at the beginning
    last = insertAtBeginning(last, 5);

    printf("List after inserting 5 at the beginning: ");
    printList(last);

    return 0;
}
```

Output

```
Original list: 2 3 4
List after inserting 5 at the beginning: 5 2 3 4
```

3. Insertion at the end in circular linked list

To insert a new node at the end of a circular linked list, we first create the new node and allocate memory for it. If the list is empty (mean, ***last*** *or* ***tail***

*pointer being **NULL**), we initialize the list with the **new node** and making it point to itself to form a circular structure. If the list already contains nodes then we set the **new node's** next pointer to point to the **current head** (which is **tail->next**), then update the **current tail's** next pointer to point to the **new node**. Finally, we update the **tail pointer** to the **new node.** This will ensure that the **new node** is now the **last node** in the list while maintaining the circular linkage.*

3. Insertion at the end in circular linked list

*To insert a new node at the end of a circular linked list, we first create the new node and allocate memory for it. If the list is empty (mean, **last** or **tail** pointer being **NULL**), we initialize the list with the **new node** and making it point to itself to form a circular structure. If the list already contains nodes then we set the **new node's** next pointer to point to the **current head** (which is **tail->next**), then update the **current tail's** next pointer to point to the **new node**. Finally, we update the **tail pointer** to the **new node.** This will ensure that the **new node** is now the **last node** in the list while maintaining the circular linkage.*

```
#include <stdio.h>

#include <stdlib.h>



// Define the Node structure

struct Node

{
```

```
    int data;
    struct Node *next;
};

// Function to create a new node
struct Node *createNode(int value);

// Function to insert a node at the end of a circular linked list
struct Node *insertEnd(struct Node *tail, int value)
{
    struct Node *newNode = createNode(value);
    if (tail == NULL)
    {
        // If the list is empty, initialize it with the new node
        tail = newNode;
        newNode->next = newNode;
    }
    else
    {
        // Insert new node after the current tail and update the tail
pointer
```

```
        newNode->next = tail->next;
        tail->next = newNode;
        tail = newNode;
    }
    return tail;
}

// Function to print the circular linked list
void printList(struct Node *last)
{
    if (last == NULL)
        return;

    struct Node *head = last->next;
    while (1)
    {
        printf("%d ", head->data);
        head = head->next;
        if (head == last->next)
            break;
```

```
    }
    printf("\n");
}

struct Node *createNode(int value)
{
    struct Node *newNode = (struct Node *)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}

int main()
{
    // Create circular linked list: 2, 3, 4
    struct Node *first = createNode(2);
    first->next = createNode(3);
    first->next->next = createNode(4);

    struct Node *last = first->next->next;
```

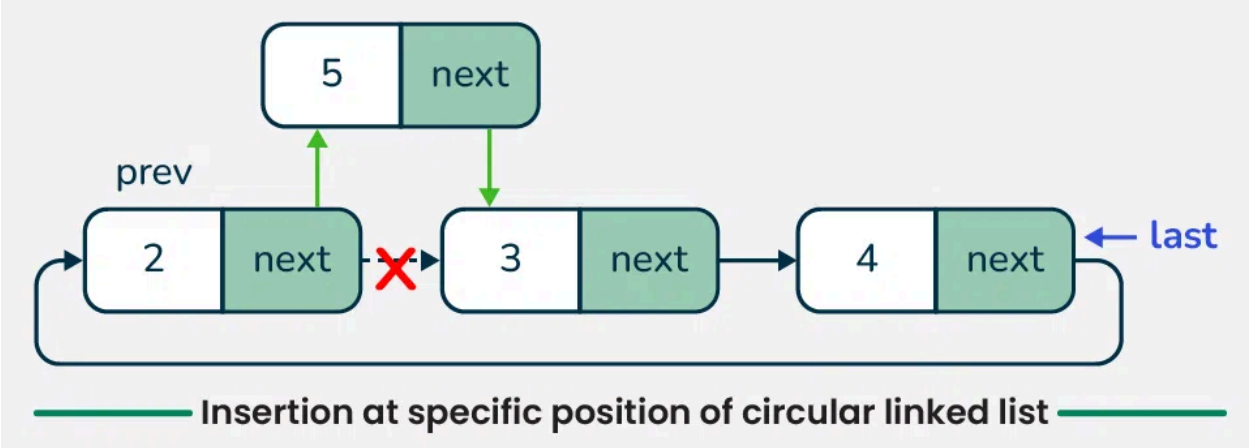
```
    last->next = first;


    printf("Original list: ");
    printList(last);


    // Insert elements at the end of the circular linked list
    last = insertEnd(last, 5);
    last = insertEnd(last, 6);


    printf("List after inserting 5 and 6: ");
    printList(last);


    return 0;
}
```

Output

```
Original list: 2 3 4
List after inserting 5 and 6: 2 3 4 5 6
```

4. Insertion at specific position in circular linked list

*To insert a new node at a specific position in a circular linked list, we first check if the list is empty. If it is and the **position** is not **1** then we print an error message because the position doesn't exist in the list. If the **position** is **1** then we create the **new node** and make it point to itself. If the list is not empty, we create the **new node** and traverse the list to find the correct insertion point. If the **position** is **1**, we insert the **new node** at the beginning by adjusting the pointers accordingly. For other positions, we traverse through the list until we reach the desired position and inserting the **new node** by updating the pointers. If the new node is inserted at the end, we also update the **last** pointer to reference the new node, maintaining the circular structure of the list.*




```c
#include <stdio.h>

#include <stdlib.h>



// Define the Node structure

struct Node {

    int data;
```

```
    struct Node *next;
};

struct Node* createNode(int value);

// Function to insert a node at a specific position in a circular linked
list
struct Node* insertAtPosition(struct Node *last, int data, int pos) {
    if (last == NULL) {
        // If the list is empty
        if (pos != 1) {
            printf("Invalid position!\n");
            return last;
        }
        // Create a new node and make it point to itself
        struct Node *newNode = createNode(data);
        last = newNode;
        last->next = last;
        return last;
    }
```

```
    // Create a new node with the given data
    struct Node *newNode = createNode(data);

    // curr will point to head initially
    struct Node *curr = last->next;

    if (pos == 1) {
        // Insert at the beginning
        newNode->next = curr;
        last->next = newNode;
        return last;
    }

    // Traverse the list to find the insertion point
    for (int i = 1; i < pos - 1; ++i) {
        curr = curr->next;

        // If position is out of bounds
        if (curr == last->next) {
```

```
            printf("Invalid position!\n");
            return last;
        }
    }

    // Insert the new node at the desired position
    newNode->next = curr->next;
    curr->next = newNode;

    // Update last if the new node is inserted at the end
    if (curr == last) last = newNode;

    return last;
}

// Function to print the circular linked list
void printList(struct Node *last) {
    if (last == NULL) return;

    struct Node *head = last->next;
```

```
    while (1) {
        printf("%d ", head->data);
        head = head->next;
        if (head == last->next) break;
    }
    printf("\n");
}

// Function to create a new node
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}

int main() {
    // Create circular linked list: 2, 3, 4
    struct Node *first = createNode(2);
    first->next = createNode(3);
```

```
    first->next->next = createNode(4);

    struct Node *last = first->next->next;
    last->next = first;

    printf("Original list: ");
    printList(last);

    // Insert elements at specific positions
    int data = 5, pos = 2;
    last = insertAtPosition(last, data, pos);
    printf("List after insertions: ");
    printList(last);

    return 0;
}
```

Output

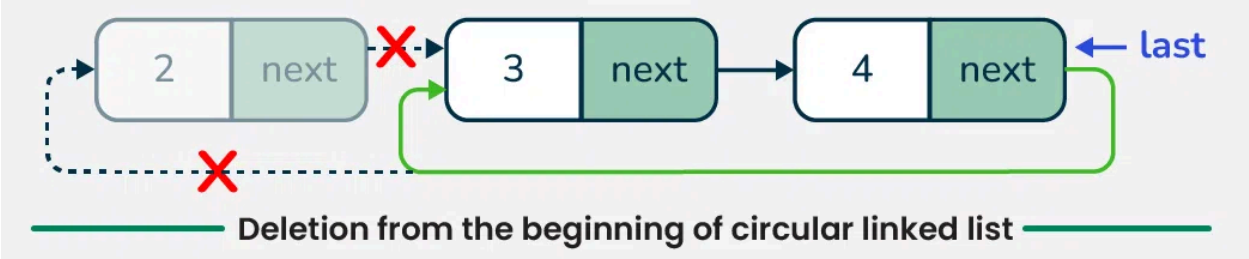
```
Original list: 2 3 4
List after insertions: 2 5 3 4
```

Deletion from a Circular Linked List

Deletion involves removing a node from the linked list. The main difference is that we need to ensure the list remains circular after the deletion. We can delete a node in a circular linked list in three ways:

1. Delete the first node in circular linked list

*To delete the first node of a circular linked list, we first check if the list is empty. If it is then we print a message and return **NULL**. If the list contains only one node (the **head** is the same as the **last**) then we delete that node and set the **last** pointer to **NULL**. If there are multiple nodes then we update the **last->next** pointer to skip the **head** node and effectively removing it from the list. We then delete the **head** node to free the allocated memory. Finally, we return the updated **last** pointer, which still points to the **last** node in the list.*




```
#include <stdio.h>

#include <stdlib.h>
```

```c
struct Node {
    int data;
    struct Node* next;
};

struct Node* deleteFirstNode(struct Node* last) {
    if (last == NULL) {
        // If the list is empty
        printf("List is empty\n");
        return NULL;
    }

    struct Node* head = last->next;

    if (head == last) {
        // If there is only one node in the list
        free(head);
        last = NULL;
    } else {
```

```
        // More than one node in the list
        last->next = head->next;
        free(head);
    }

    return last;
}

void printList(struct Node* last) {
    if (last == NULL) return;

    struct Node* head = last->next;
    while (1) {
        printf("%d ", head->data);
        head = head->next;
        if (head == last->next) break;
    }
    printf("\n");
}
```

```c
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}

int main() {
    struct Node* first = createNode(2);
    first->next = createNode(3);
    first->next->next = createNode(4);

    struct Node* last = first->next->next;
    last->next = first;

    printf("Original list: ");
    printList(last);

    last = deleteFirstNode(last);
```

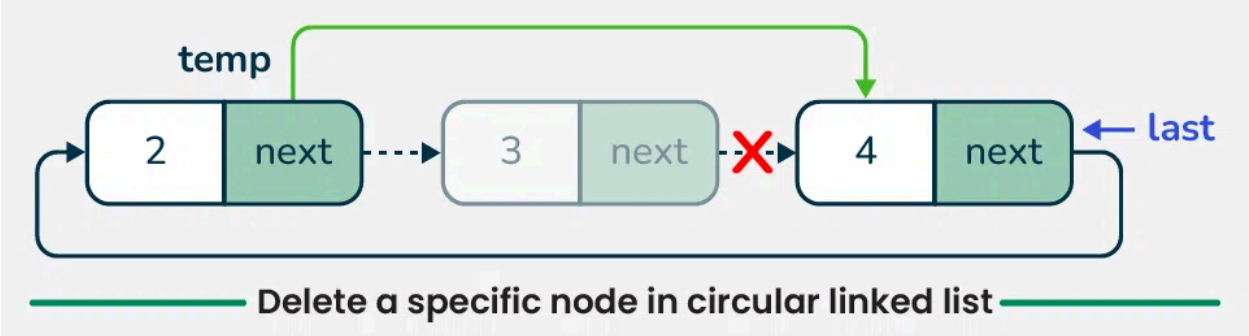
```
    printf("List after deleting first node: ");
    printList(last);

    return 0;
}
```

Output

```
Original list: 2 3 4
List after deleting first node: 3 4
```

2. Delete a specific node in circular linked list

*To delete a specific node from a circular linked list, we first check if the list is empty. If it is then we print a message and return **nullptr**. If the list contains only one node and it matches the **key** then we delete that node and set **last** to **nullptr**. If the node to be deleted is the first node then we update the **next** pointer of the **last** node to skip the **head** node and delete the **head**. For other nodes, we traverse the list using two pointers: **curr** (to find the node) and **prev** (to keep track of the previous node). If we find the node with the matching key then we update the next pointer of **prev** to skip the **curr** node and delete it. If the node is found and it is the last node, we update the **last** pointer accordingly. If the node is not found then do nothing and **tail** or **last** as it is. Finally, we return the updated **last** pointer.*




```c
#include <stdio.h>

#include <stdlib.h>


// Define the structure for a node in the circular linked list

struct Node {

    int data;

    struct Node* next;

};


// Function to delete a specific node in the circular linked list

struct Node* deleteSpecificNode(struct Node* last, int key) {

    if (last == NULL) {

        // If the list is empty

        printf("List is empty, nothing to delete.\n");

        return NULL;

    }
```

```
    struct Node* curr = last->next;
    struct Node* prev = last;

    // If the node to be deleted is the only node in the list
    if (curr == last && curr->data == key) {
        free(curr);
        last = NULL;
        return last;
    }

    // If the node to be deleted is the first node
    if (curr->data == key) {
        last->next = curr->next;
        free(curr);
        return last;
    }

    // Traverse the list to find the node to be deleted
    while (curr != last && curr->data != key) {
```

```
        prev = curr;
        curr = curr->next;
    }

    // If the node to be deleted is found
    if (curr->data == key) {
        prev->next = curr->next;
        if (curr == last) {
            last = prev;
        }
        free(curr);
    } else {
        // If the node to be deleted is not found
        printf("Node with data %d not found.\n", key);
    }

    return last;
}

void printList(struct Node* last) {
```

```
    if (last == NULL) {
        printf("List is Empty");
        return;
    }

    struct Node* head = last->next;
    while (1) {
        printf("%d ", head->data);
        head = head->next;
        if (head == last->next) break;
    }
    printf("\n");
}

struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}
```

```c
int main() {
    // Create circular linked list: 2, 3, 4
    struct Node* first = createNode(2);
    first->next = createNode(3);
    first->next->next = createNode(4);

    struct Node* last = first->next->next;
    last->next = first;

    printf("Original list: ");
    printList(last);

    // Delete a specific node
    int key = 3;
    last = deleteSpecificNode(last, key);

    printf("List after deleting node %d: ", key);
    printList(last);
```

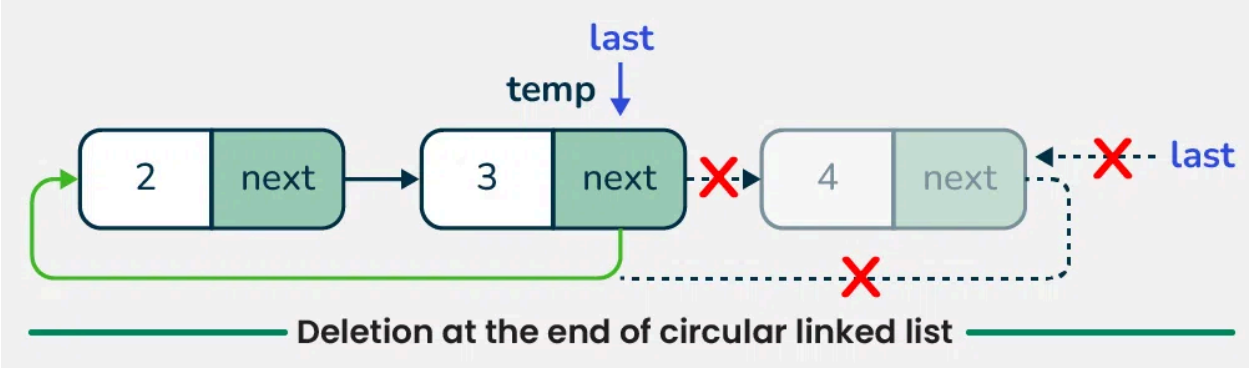
```
    return 0;

}
```

Output

```
Original list: 2 3 4

List after deleting node 3: 2 4
```

3. Deletion at the end of Circular linked list

To delete the last node in a circular linked list, we first check if the list is empty. If it is, we print a message and return ***nullptr****. If the list contains only one node (where the* ***head*** *is the same as the* ***last****), we delete that node and set* ***last*** *to* ***nullptr****. For lists with multiple nodes, we need to traverse the list to find the* ***second last node****. We do this by starting from the* ***head*** *and moving through the list until we reach the node whose next pointer points to* ***last****. Once we find the* ***second last*** *node then we update its next pointer to point back to the* ***head,*** *this effectively removing the last node from the list. We then delete the last node to free up memory and return the updated* ***last*** *pointer, which now points to the last node.*




```c
#include <stdio.h>

#include <stdlib.h>


// Define the structure for a node in the circular linked list

struct Node {

    int data;

    struct Node* next;

};


// Function to delete the last node in the circular linked list

struct Node* deleteLastNode(struct Node* last) {

    if (last == NULL) {

        // If the list is empty

        printf("List is empty, nothing to delete.\n");

        return NULL;
```

```
    }
    struct Node* head = last->next;

    // If there is only one node in the list
    if (head == last) {
        free(last);
        last = NULL;
        return last;
    }
    // Traverse the list to find the second last node
    struct Node* curr = head;
    while (curr->next != last) {
        curr = curr->next;
    }
    // Update the second last node's next pointer to point to head
    curr->next = head;
    free(last);
    last = curr;

    return last;
```

```
}

void printList(struct Node* last) {
    if (last == NULL) return;

    struct Node* head = last->next;
    while (1) {
        printf("%d ", head->data);
        head = head->next;
        if (head == last->next) break;
    }
    printf("\n");
}

struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}
```

```c
int main() {
    // Create circular linked list: 2, 3, 4
    struct Node* first = createNode(2);
    first->next = createNode(3);
    first->next->next = createNode(4);

    struct Node* last = first->next->next;
    last->next = first;

    printf("Original list: ");
    printList(last);

    // Delete the last node
    last = deleteLastNode(last);

    printf("List after deleting last node: ");
    printList(last);

    return 0;
```

```
}
```

Output

```
Original list: 2 3 4

List after deleting last node: 2 3
```

Searching in Circular Linked list

Searching in a circular linked list is similar to searching in a regular linked list. We start at a given node and traverse the list until you either find the target value or return to the starting node. Since the list is circular, make sure to keep track of where you started to avoid an infinite loop.

*To search for a specific value in a circular linked list, we first check if the list is empty. If it is then we return **false**. If the list contains nodes then we start from the **head** node (which is the **last->next**) and traverse the list. We use a pointer **curr** to iterate through the nodes until we reach back to the **head**. During traversal, if we find a node whose **data** matches the given **key** then we return **true** to indicating that the value was found. After the loop, we also check the last node to ensure we don't miss it. If the **key** is not found after traversing the entire list then we return **false**.*

```
#include <stdio.h>

#include <stdlib.h>
```

```c
// Definition of the Node structure
struct Node{
    int data;
    struct Node *next;
};

// Function to search for a specific value in the circular linked list
int search(struct Node *last, int key){
    if (last == NULL){
        // If the list is empty
        return 0;
    }

    struct Node *head = last->next;
    struct Node *curr = head;

    // Traverse the list to find the key
    while (curr != last){
        if (curr->data == key){
```

```
            // Key found
            return 1;
        }
        curr = curr->next;
    }

    // Check the last node
    if (last->data == key){
        // Key found
        return 1;
    }
    // Key not found
    return 0;
}

// Function to print the circular linked list
void printList(struct Node *last){
    if (last == NULL) return;

    struct Node *head = last->next;
```

```c
    while (1){
        printf("%d ", head->data);
        head = head->next;
        if (head == last->next)
            break;
    }
    printf("\n");
}
// Function to create a new node
struct Node *createNode(int value){
    struct Node *temp = (struct Node *)malloc(sizeof(struct Node));
    temp->data = value;
    temp->next = NULL;
    return temp;
}

int main(){
    // Create circular linked list: 2, 3, 4
    struct Node *first = createNode(2);
    first->next = createNode(3);
```

```
    first->next->next = createNode(4);

    struct Node *last = first->next->next;
    last->next = first;

    printf("Original list: ");
    printList(last);

    // Search for a specific value
    int key = 3;
    int found = search(last, key);
    if (found){
        printf("Value %d found in the list.\n", key);
    }
    else{
        printf("Value %d not found in the list.\n", key);
    }

    return 0;
}
```

Output

```
Original list: 2 3 4

Value 3 found in the list.
```

Advantages of Circular Linked Lists

- In circular linked list, the last node points to the first node. There are no null references, making traversal easier and reducing the chances of encountering null pointer exceptions.
- We can traverse the list from any node and return to it without needing to restart from the head, which is useful in applications requiring a circular iteration.
- Circular linked lists can easily implement circular queues, where the last element connects back to the first, allowing for efficient resource management.
- In a circular linked list, each node has a reference to the next node in the sequence. Although it doesn't have a direct reference to the previous node like a doubly linked list, we can still find the previous node by traversing the list.

Disadvantages of Circular Linked Lists

- Circular linked lists are more complex to implement than singly linked lists.
- Traversing a circular linked list without a clear stopping condition can lead to infinite loops if not handled carefully.
- Debugging can be more challenging due to the circular nature, as traditional methods of traversing linked lists may not apply.

Applications of Circular Linked Lists

- It is used for time-sharing among different users, typically through a **Round-Robin scheduling mechanism.**
- In multiplayer games, a circular linked list can be used to switch between players. After the last player's turn, the list cycles back to the first player.
- Circular linked lists are often used in buffering applications, such as streaming data, where data is continuously produced and consumed.
- In media players, circular linked lists can manage playlists, this allowing users to loop through songs continuously.
- Browsers use circular linked lists to manage the cache. This allows you to navigate back through your browsing history efficiently by pressing the BACK button.