Circular Linked List in C with proper explanation

A **Circular Linked List** is a variation of a linked list where the last node points back to the head of the list, forming a circle. This allows for circular traversal, where we can traverse the list starting from any node and eventually return to the starting point.

There are two types of circular linked lists:

1. **Singly Circular Linked List**: Each node points to the next node, and the last node points back to the first node.
2. **Doubly Circular Linked List**: Each node points to both the next and the previous node, and the last node's `next` points to the first node, while the first node's `prev` points to the last node.

Singly Circular Linked List

In a **Singly Circular Linked List**, each node contains two fields:

1. `data`: the value stored in the node.
2. `next`: a pointer to the next node in the list.

The key difference from a singly linked list is that the `next` pointer of the last node points back to the head node, creating a circular structure.

Structure of a Circular Linked List Node in C

We define the structure for a node as follows:

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

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

Basic Operations

1. Insertion at the End

To insert a new node at the end of a circular linked list, we traverse the list to find the last node and adjust its `next` pointer to point to the new node. The `next` pointer of the new node will point back to the head.

```
void insertAtEnd(struct Node** head, int new_data) {
   // Allocate memory for new node
   struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
   struct Node* temp = *head;

   // Set data for the new node
   new_node->data = new_data;
   new_node->next = NULL;

   // If the list is empty, make the new node point to itself
   if (*head == NULL) {
```

```
        new_node->next = new_node;
        *head = new_node;
        return;
    }

    // Traverse to the last node
    while (temp->next != *head)
        temp = temp->next;

    // Insert the new node at the end
    temp->next = new_node;
    new_node->next = *head;
}
```

2. Insertion at the Beginning

To insert a new node at the beginning of the list, we first find the last node (to update its next pointer) and then adjust the new node's next pointer to point to the head. Finally, we update the head to the new node.

```
void insertAtBeginning(struct Node** head, int new_data) {
    // Allocate memory for the new node
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    struct Node* temp = *head;

    // Set data for the new node
    new_node->data = new_data;

    // If the list is empty, make the new node point to itself
    if (*head == NULL) {
        new_node->next = new_node;
        *head = new_node;
        return;
    }

    // Traverse to the last node
    while (temp->next != *head)
        temp = temp->next;

    // Insert the new node at the beginning
    temp->next = new_node;
    new_node->next = *head;

    // Update the head to the new node
    *head = new_node;
}
```

3. Traversal

To traverse a circular linked list, we start from the head node and keep printing the data field until we encounter the head node again, indicating that we have completed one full circle.

```
void traverse(struct Node* head) {
    struct Node* temp = head;

    // If the list is empty, there's nothing to traverse
    if (head == NULL) {
```

```
        printf("List is empty.\n");
        return;
    }

    // Traverse the list starting from head
    do {
        printf("%d -> ", temp->data);
        temp = temp->next;
    } while (temp != head); // Stop when we reach the head again

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

Full Example Program

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

// Node structure
struct Node {
    int data;
    struct Node* next;
};

// Function to insert at the end of the list
void insertAtEnd(struct Node** head, int new_data) {
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    struct Node* temp = *head;
    new_node->data = new_data;
    new_node->next = NULL;

    if (*head == NULL) {
        new_node->next = new_node;
        *head = new_node;
        return;
    }

    while (temp->next != *head)
        temp = temp->next;

    temp->next = new_node;
    new_node->next = *head;
}

// Function to insert at the beginning of the list
void insertAtBeginning(struct Node** head, int new_data) {
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    struct Node* temp = *head;
    new_node->data = new_data;

    if (*head == NULL) {
        new_node->next = new_node;
        *head = new_node;
        return;
    }

    while (temp->next != *head)
        temp = temp->next;
```

```
    temp->next = new_node;
    new_node->next = *head;
    *head = new_node;
}

// Function to traverse the circular linked list
void traverse(struct Node* head) {
    struct Node* temp = head;
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    do {
        printf("%d -> ", temp->data);
        temp = temp->next;
    } while (temp != head);
    printf("HEAD\n");
}

// Driver program to test the circular linked list
int main() {
    struct Node* head = NULL;

    insertAtEnd(&head, 10);
    insertAtEnd(&head, 20);
    insertAtEnd(&head, 30);

    printf("List after inserting at the end: ");
    traverse(head);

    insertAtBeginning(&head, 5);

    printf("List after inserting at the beginning: ");
    traverse(head);

    return 0;
}
```

Explanation of the Program:

1. insertAtEnd: This function inserts a new node at the end of the circular linked list by traversing the list to the last node, adjusting the pointers to maintain the circular structure.
2. insertAtBeginning: This function inserts a new node at the beginning of the list. It first finds the last node and updates its next pointer to point to the new node, then updates the head to the new node.
3. traverse: This function prints all the nodes in the list starting from the head, going through each node until it returns to the head, indicating the completion of one full cycle.

Circular Nature

- **Circularity**: Unlike a normal linked list, in a circular linked list, the last node points back to the head. This means that traversing the list from any node will eventually bring you back to that node.

- **Advantages**: Circular linked lists are useful in applications like circular buffers, where the data structure needs to be cycled through continuously.