You are required to implement a singly linked list in C with the following functionalities: 1. Insert at the Beginning: Write a function to insert a node at the beginning of the linked list. 2. Insert at the End: Write a function to insert a node at the end of the linked list. 3. Insert After a Certain Node: Write a function to insert a node after a specified node in the linked list. You may assume that the specified node exists in the list. 4. Insert Before a Certain Node: Write a function to insert a node before a specified node in the linked list. You may assume that the specified node exists in the list. 5. Insert at the nth Position: Write a function to insert a node at the nth position in the linked list. If the position is out of bounds, handle it appropriately. 6. Delete the First Node: Write a function to delete the first node of the linked list. 7. Delete the Last Node: Write a function to delete the last node of the linked list. 8. Delete the nth Node: Write a function to delete the nth node of the linked list. If the position is out of bounds, handle it appropriately. 9. Delete a Node After a Node: Write a function to delete a node after a specified node in the linked list. You may assume that the specified node exists in the list and has a successor. 10. Delete a Node Before a Node: Write a function to delete a node before a specified node in the linked list. You may assume that the specified node exists in the list and has a predecessor. 11. Search for a Node: Write a function to search for a node in the linked list by its value. The function should return the position of the node in the list if found, or indicate that the node is not present. 12. Traverse the Linked List: Write a function to traverse the linked list and print the value of each node. 13. Reverse the Linked List: Write a function to reverse the linked list. 14. Sort the Linked List: Write a function to sort the linked list.

ChatGPT said:

ChatGPT

Here's a complete implementation of the singly linked list in C with the requested functionalities. Each function performs a specific operation on the linked list.

c

Copy code

```c
#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 data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
```

```
    return newNode;
}

// 1. Insert at the Beginning
void insertAtBeginning(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    newNode->next = *head;
    *head = newNode;
}

// 2. Insert at the End
void insertAtEnd(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
        return;
    }
    struct Node* temp = *head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
}

// 3. Insert After a Certain Node
void insertAfterNode(struct Node* prevNode, int data) {
    if (prevNode == NULL) {
        printf("The given previous node cannot be NULL\n");
        return;
    }
    struct Node* newNode = createNode(data);
    newNode->next = prevNode->next;
    prevNode->next = newNode;
}

// 4. Insert Before a Certain Node
void insertBeforeNode(struct Node** head, int target, int data) {
    struct Node* newNode = createNode(data);
```

```
    if (*head == NULL) return;

    if ((*head)->data == target) {
        newNode->next = *head;
        *head = newNode;
        return;
    }

    struct Node* temp = *head;
    while (temp->next != NULL && temp->next->data != target) {
        temp = temp->next;
    }
    if (temp->next == NULL) return;
    newNode->next = temp->next;
    temp->next = newNode;
}

// 5. Insert at the nth Position
void insertAtNthPosition(struct Node** head, int position, int data) {
    if (position < 1) {
        printf("Position out of bounds\n");
        return;
    }

    struct Node* newNode = createNode(data);
    if (position == 1) {
        newNode->next = *head;
        *head = newNode;
        return;
    }

    struct Node* temp = *head;
    for (int i = 1; i < position - 1 && temp != NULL; i++) {
        temp = temp->next;
    }
    if (temp == NULL) {
        printf("Position out of bounds\n");
        return;
```

```
    }
    newNode->next = temp->next;
    temp->next = newNode;
}

// 6. Delete the First Node
void deleteFirstNode(struct Node** head) {
    if (*head == NULL) return;
    struct Node* temp = *head;
    *head = (*head)->next;
    free(temp);
}

// 7. Delete the Last Node
void deleteLastNode(struct Node** head) {
    if (*head == NULL) return;
    if ((*head)->next == NULL) {
        free(*head);
        *head = NULL;
        return;
    }
    struct Node* temp = *head;
    while (temp->next->next != NULL) {
        temp = temp->next;
    }
    free(temp->next);
    temp->next = NULL;
}

// 8. Delete the nth Node
void deleteNthNode(struct Node** head, int position) {
    if (*head == NULL || position < 1) return;
    struct Node* temp = *head;

    if (position == 1) {
        *head = temp->next;
        free(temp);
        return;
```

```
    }

    for (int i = 1; i < position - 1 && temp != NULL; i++) {
        temp = temp->next;
    }

    if (temp == NULL || temp->next == NULL) return;

    struct Node* next = temp->next->next;
    free(temp->next);
    temp->next = next;
}

// 9. Delete a Node After a Certain Node
void deleteAfterNode(struct Node* prevNode) {
    if (prevNode == NULL || prevNode->next == NULL) return;
    struct Node* temp = prevNode->next;
    prevNode->next = temp->next;
    free(temp);
}

// 10. Delete a Node Before a Certain Node
void deleteBeforeNode(struct Node** head, int target) {
    if (*head == NULL || (*head)->data == target) return;

    struct Node* temp = *head;
    if (temp->next != NULL && temp->next->data == target) {
        *head = temp->next;
        free(temp);
        return;
    }

    while (temp->next->next != NULL && temp->next->next->data !=
target) {
        temp = temp->next;
    }

    if (temp->next->next == NULL) return;
```

```
    struct Node* toDelete = temp->next;
    temp->next = toDelete->next;
    free(toDelete);
}

// 11. Search for a Node
int searchNode(struct Node* head, int value) {
    int position = 1;
    struct Node* temp = head;
    while (temp != NULL) {
        if (temp->data == value) return position;
        temp = temp->next;
        position++;
    }
    return -1;
}

// 12. Traverse the Linked List
void traverseList(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

// 13. Reverse the Linked List
void reverseList(struct Node** head) {
    struct Node* prev = NULL;
    struct Node* current = *head;
    struct Node* next = NULL;
    while (current != NULL) {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
```

```
    }
    *head = prev;
}

// 14. Sort the Linked List
void sortList(struct Node** head) {
    struct Node* i = *head;
    struct Node* j = NULL;
    int temp;

    if (*head == NULL) return;

    for (i = *head; i->next != NULL; i = i->next) {
        for (j = i->next; j != NULL; j = j->next) {
            if (i->data > j->data) {
                temp = i->data;
                i->data = j->data;
                j->data = temp;
            }
        }
    }
}

// Main function to test the linked list functionalities
int main() {
    struct Node* head = NULL;

    // Insert elements
    insertAtBeginning(&head, 3);
    insertAtEnd(&head, 5);
    insertAtEnd(&head, 8);
    insertAtNthPosition(&head, 2, 4);
    traverseList(head);

    // Search for a node
    int pos = searchNode(head, 5);
    if (pos != -1) printf("Element found at position %d\n", pos);
    else printf("Element not found\n");
```

```
    // Delete nodes
    deleteFirstNode(&head);
    deleteLastNode(&head);
    traverseList(head);

    // Reverse and sort the list
    reverseList(&head);
    traverseList(head);
    sortList(&head);
    traverseList(head);

    return 0;
}
```