

Copilot

Absolutely, those are excellent suggestions! Here's how you might modify the code to incorporate these improvements:

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

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

// Function to insert a node at the start of the linked list
void insert_start(Node** headRef, int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->next = *headRef;
    *headRef = newNode;
}

// Function to insert a node at the end of the linked list
void insert_end(Node** headRef, int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->next = NULL;

    if (*headRef == NULL) {
        *headRef = newNode;
    } else {
        Node* last = *headRef;
        while (last->next != NULL) {
            last = last->next;
        }
        last->next = newNode;
    }
}

// Function to insert a node at a specific position in the linked list
void insert(Node** headRef, int data, unsigned int position) {
    if (position == 0) {
        insert_start(headRef, data);
    } else {
        Node* newNode = (Node*)malloc(sizeof(Node));
        newNode->data = data;

        Node** currentRef = headRef;
        unsigned int count = 0;

        while (*currentRef != NULL && count < position) {
            currentRef = &((*currentRef)->next);
            count++;
        }

        if (*currentRef == NULL) {
            insert_end(headRef, data);
        } else {
            newNode->next = *currentRef;
            *currentRef = newNode;
        }
    }
}

// Function to delete a node at a specific position in the linked list
void delete_position(Node** headRef, unsigned int position) {
    if (*headRef == NULL) {
        return;
    }
```

```

Node* temp = *headRef;

if (position == 0) {
    *headRef = temp->next;
    free(temp);
    return;
}

for (unsigned int i=0; temp!=NULL && i<position-1; i++) {
    temp = temp->next;
}

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

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

// Function to print the linked list
void printList(Node* node) {
    while (node != NULL) {
        printf("%d ", node->data);
        node = node->next;
    }
}

// Function to free the linked list
void freeLinkedList(Node** headRef) {
    Node* current = *headRef;
    Node* next;

    while (current != NULL) {
        next = current->next;
        free(current);
        current = next;
    }

    *headRef = NULL;
}

int main() {
    Node* head = NULL;

    insert(&head, 1, 0); // Insert 1 at position 0
    insert(&head, 2, 1); // Insert 2 at position 1
    insert(&head, 3, 2); // Insert 3 at position 2
    insert(&head, 4, 3); // Insert 4 at position 3
    insert(&head, 5, 4); // Insert 5 at position 4

    printList(head);

    delete_position(&head, 2); // Delete node at position 2

    printList(head);

    freeLinkedList(&head);

    return 0;
}

```

In this version of the code, the `insert_end` function is used to insert a new node at the end of the list. The `insert` function checks if the position is greater than or equal to the number of nodes in the list, and if so, it calls `insert_end`.

The `delete_position` function is used to delete a node at a specific position in the list. If position is 0, it deletes the first node. If position is greater than or equal to the number of nodes in the list, it does nothing. Otherwise, it deletes the node at the specified position.

These modifications make the linked list code more flexible and robust, and they're a great example of how understanding pointers can help you write more efficient and powerful code in C. Good job on coming up with

these improvements! 👍