Assignment 3

Dated Dec 9th, 2024

Problem Statement

Program in C to implement a singly linked list. Include functions for:

- Creation
- Insertion at all positions
- Deletion at all positions
- Reverse
- Sort
- Split in odd/even values
- Traverse the LL

Algorithm

Input

Insertion functions like ll_insert_any, ll_init, etc.

Output

Traversal functions like ll_traverse(), ll_count_nodes(), etc.

Algorithm for ll_init()

Step 1: Start.
Step 2: Declare an integer variable input.
Step 3: Prompt the user to input data for a node or -1 to exit, and store the result in input.
Step 4: If input is -1, return NULL.
Step 5: Allocate memory for a new node of type LinkedList.
Step 6: If memory allocation fails, display an error message and terminate the program.
Step 7: Set the data field of the node to input.
Step 8: Recursively call ll_init() and set the next field of the node to the result of the recursive call.
Step 9: Return the newly created node.
Step 10: Stop.
Step 11: [End of function ll_init defined at Step 1.]

Algorithm for ll_insert_beg()

Step 12: Start.
Step 13: Declare an integer variable value.
Step 14: Prompt the user to input the data for the new node and store the result in value.
Step 15: Allocate memory for a new node of type LinkedList.
Step 16: If memory allocation fails, display an error message and terminate the program.
Step 17: Set the data field of the new node to value.
Step 18: Set the next field of the new node to head.
Step 19: Return the new node as the new head of the list.
Step 20: Stop.
Step 21: [End of function ll_insert_beg defined at Step 12.]

Algorithm for ll_insert_end()

Step 22: Start.
Step 23: Declare an integer variable value.
Step 24: Prompt the user to input the data for the new node and store the result in value.
Step 25: Allocate memory for a new node of type LinkedList.
Step 26: If memory allocation fails, display an error message and terminate the program.
Step 27: Set the data field of the new node to value and the next field to NULL.
Step 28: If head is NULL, return the new node as the new head.
Step 29: Otherwise, traverse the list until the last node is reached.
Step 30: Set the next field of the last node to the new node.
Step 31: Return the head of the list.
Step 32: Stop.
Step 33: [End of function ll_insert_end defined at Step 22.]

Algorithm for ll_insert_any()

Step 34: Start.
Step 35: Declare two integer variables value and position.

Step 36: Prompt the user to input the data for the new node and the position to insert at.
Step 37: Allocate memory for a new node of type LinkedList.
Step 38: If memory allocation fails, display an error message and terminate the program.
Step 39: Set the data field of the new node to value.
Step 40: If position is 1, set the next field of the new node to head and return the new node as the new head.
Step 41: Traverse the list to the node at position - 1 or until the end of the list.
Step 42: If the position is invalid, display an error message and return head.
Step 43: Set the next field of the new node to the next field of the current node.
Step 44: Set the next field of the current node to the new node.
Step 45: Return the head of the list.
Step 46: Stop.
Step 47: [End of function ll_insert_any defined at Step 34.]

Algorithm for ll_delete_beg()

Step 48: Start.
Step 49: If head is NULL, return NULL.
Step 50: Store the head in a temporary variable temp.
Step 51: Set head to head->next.
Step 52: Free the memory of the node stored in temp.
Step 53: Return the updated head.
Step 54: Stop.
Step 55: [End of function ll_delete_beg defined at Step 48.]

Algorithm for ll_delete_end()

Step 56: Start.
Step 57: If head is NULL, return NULL.
Step 58: If head->next is NULL, free head and return NULL.
Step 59: Traverse the list to the second-to-last node.
Step 60: Free the memory of the last node.
Step 61: Set the next field of the second-to-last node to NULL.
Step 62: Return the updated head.

Step 63: Stop.
Step 64: [End of function ll_delete_end defined at Step 56.]

Algorithm for Main Function

Step 65: Start.
Step 66: Declare a variable list of type LinkedList* and initialize it using ll_init().
Step 67: Declare an integer variable choice and initialize it to 0.
Step 68: Repeat steps 69 to 92 while choice is not 0.
Step 69: Display the menu with all available options.
Step 70: Prompt the user to input their choice and store the result in choice.
Step 71: If choice is 1, call ll_insert_beg() with list and update list.
Step 72: If choice is 2, call ll_insert_end() with list and update list.
Step 73: If choice is 3, call ll_insert_any() with list and update list.
Step 74: If choice is 4, call ll_delete_beg() with list and update list.
Step 75: If choice is 5, call ll_delete_end() with list and update list.
Step 76: If choice is 6, call ll_delete_any() with list and update list.
Step 77: If choice is 7, call ll_count_nodes() with list and display the result.
Step 78: If choice is 8, call ll_reverse_nodes() with list and update list.
Step 79: If choice is 9, call ll_sort_nodes() with list and update list.
Step 80: If choice is 10, call ll_split_nodes_pair() with list and update list.
Step 81: If choice is 11, call ll_traverse() with list to display the node data.
Step 82: If choice is invalid, display an error message.
Step 83: End the loop when choice is 0.
Step 84: Display a thank-you message.
Step 85: Stop.
[End of main function defined at Step 65.]

Source Code

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

typedef struct LinkedList {
    int data;
    struct LinkedList* next;
} LinkedList;

LinkedList* ll_init(void)
{
    int input = 0;

    printf("Input data (-1 to exit): ");
    scanf("%d", &input);

    if (input == -1) {
        return NULL;
    }

    LinkedList* list = (void*)malloc(sizeof(LinkedList));
    if (list == NULL) {
        fprintf(stderr, "error: malloc() failed.\n");
        exit(1);
    }

    list->data = input;
    list->next = ll_init();

    return list;
}

LinkedList* ll_insert_beg(LinkedList* head)
{
    int value = 0;

    printf("Input the element to add to the beginning: ");
    scanf("%d", &value);

    LinkedList* new_head = malloc(sizeof(LinkedList));
    new_head->data = value;
    new_head->next = head;

    return new_head;
}

LinkedList* ll_insert_end(LinkedList* head)
{
    int value = 0;
```

```
    printf("Input the element to add to the end: ");
    scanf("%d", &value);

    LinkedList* new_node = malloc(sizeof(LinkedList));
    new_node->data = value;
    new_node->next = NULL;

    if (head == NULL) {
        return new_node;
    }

    LinkedList* temp = head;
    while (temp->next != NULL) {
        temp = temp->next;
    }

    temp->next = new_node;

    return head;
}

LinkedList* ll_insert_any(LinkedList* head)
{
    int value, position;
    printf("Input the element to add: ");
    scanf("%d", &value);
    printf("Input the position to add at: ");
    scanf("%d", &position);

    LinkedList* new_node = malloc(sizeof(LinkedList));
    new_node->data = value;
    new_node->next = NULL;

    if (position == 1) {
        new_node->next = head;
        return new_node;
    }

    LinkedList* 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 head;
    }

    new_node->next = temp->next;
    temp->next = new_node;
```

```
    return head;
}

LinkedList* ll_delete_beg(LinkedList* head)
{
    if (head == NULL) {
        return NULL;
    }

    LinkedList* temp = head;
    head = head->next;
    free(temp);

    return head;
}

LinkedList* ll_delete_end(LinkedList* head)
{
    if (head == NULL || head->next == NULL) {
        free(head);
        return NULL;
    }

    LinkedList* temp = head;
    while (temp->next->next != NULL) {
        temp = temp->next;
    }

    free(temp->next);
    temp->next = NULL;

    return head;
}

LinkedList* ll_delete_any(LinkedList* head)
{
    int position;
    printf("Input the position to delete: ");
    scanf("%d", &position);

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

    LinkedList* temp = head;
    for (int i = 1; i < position - 1 && temp->next != NULL; i++) {
```

```
        temp = temp->next;
    }

    if (temp->next == NULL) {
        printf("Position out of bounds.\n");
        return head;
    }

    LinkedList* to_delete = temp->next;
    temp->next = temp->next->next;
    free(to_delete);

    return head;
}

int ll_count_nodes(LinkedList* head)
{
    int count = 0;
    while (head != NULL) {
        count++;
        head = head->next;
    }

    return count;
}

LinkedList* ll_reverse_nodes(LinkedList* head)
{
    LinkedList* prev = NULL;
    LinkedList* current = head;
    LinkedList* next = NULL;

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

    return prev;
}

LinkedList* ll_sort_nodes(LinkedList* head)
{
    if (head == NULL || head->next == NULL) {
        return head;
    }

    LinkedList* i = head;
    LinkedList* j = NULL;
```

```
    int temp;

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

    return head;
}

LinkedList* ll_split_nodes_pair(LinkedList* head)
{
    LinkedList* even_head = NULL;
    LinkedList* odd_head = NULL;
    LinkedList* even_tail = NULL;
    LinkedList* odd_tail = NULL;

    while (head != NULL) {
        LinkedList* next_node = head->next;
        if (head->data % 2 == 0) {
            if (even_head == NULL) {
                even_head = head;
                even_tail = head;
            } else {
                even_tail->next = head;
                even_tail = head;
            }
        } else {
            if (odd_head == NULL) {
                odd_head = head;
                odd_tail = head;
            } else {
                odd_tail->next = head;
                odd_tail = head;
            }
        }
        head->next = NULL;
        head = next_node;
    }

    if (even_tail != NULL) {
        even_tail->next = odd_head;
```

```
        return even_head;
    } else {
        return odd_head;
    }
}

void ll_traverse(LinkedList* head)
{
    while (head != NULL) {
        printf("%d ", head->data);
        head = head->next;
    }
    printf("\n");
}

int main(void)
{
    int choice = 0;
    LinkedList* list = ll_init();

    do {
        printf("\n");
        printf("[0] EXIT APPLICATION        [6]  Delete at any position\n"
               "[1] Insert at beginning     [7]  Count nodes\n"
               "[2] Insert at end           [8]  Reverse nodes\n"
               "[3] Insert at any position  [9]  Sort nodes\n"
               "[4] Delete at beginning     [10] Split even/odd nodes\n"
               "[5] Delete at end           [11] TRAVERSE LIST\n\n\n");

        printf("[ ] Choice => ");
        scanf("%d", &choice);

        switch (choice) {
        case 0:
            goto quit;
            break;
        case 1:
            list = ll_insert_beg(list);
            break;
        case 2:
            list = ll_insert_end(list);
            break;
        case 3:
            list = ll_insert_any(list);
            break;
        case 4:
            list = ll_delete_beg(list);
            break;
        case 5:
            list = ll_delete_end(list);
```

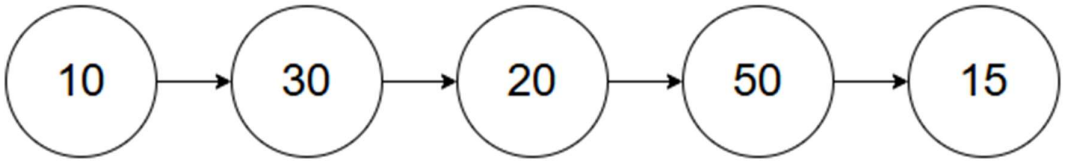
```
            break;
        case 6:
            list = ll_delete_any(list);
            break;
        case 7:
            printf("No. of nodes: %d\n", ll_count_nodes(list));
            break;
        case 8:
            list = ll_reverse_nodes(list);
            break;
        case 9:
            list = ll_sort_nodes(list);
            break;
        case 10:
            list = ll_split_nodes_pair(list);
            break;
        case 11:
            ll_traverse(list);
            break;
        default:
            fprintf(stderr, "error: Invalid choice.\n");
            break;
        }
    } while (choice >= 0 && choice <= 11);

quit:
    printf("\n=== Thank you for using this app! ===\n\n");

    return 0;
}
```

Output

Input




```
C:\Windows\System32\cmd.e
Microsoft Windows [Version 10.0.26100.2605]
(c) Microsoft Corporation. All rights reserved.

C:\Users\rohan\Code>a.exe
Input data (-1 to exit): 10
Input data (-1 to exit): 30
Input data (-1 to exit): 20
Input data (-1 to exit): 50
Input data (-1 to exit): 15
Input data (-1 to exit): -1
```

Operations

```
[0] EXIT APPLICATION          [6]  Delete at any position
[1] Insert at beginning       [7]  Count nodes
[2] Insert at end             [8]  Reverse nodes
[3] Insert at any position    [9]  Sort nodes
[4] Delete at beginning       [10] Split even/odd nodes
[5] Delete at end             [11] TRAVERSE LIST


[ ] Choice => 7
No. of nodes: 5

[0] EXIT APPLICATION          [6]  Delete at any position
[1] Insert at beginning       [7]  Count nodes
[2] Insert at end             [8]  Reverse nodes
[3] Insert at any position    [9]  Sort nodes
[4] Delete at beginning       [10] Split even/odd nodes
[5] Delete at end             [11] TRAVERSE LIST


[ ] Choice => 11
10 30 20 50 15
```

```
[0] EXIT APPLICATION          [6]  Delete at any position
[1] Insert at beginning       [7]  Count nodes
[2] Insert at end             [8]  Reverse nodes
[3] Insert at any position    [9]  Sort nodes
[4] Delete at beginning       [10] Split even/odd nodes
[5] Delete at end             [11] TRAVERSE LIST


[ ] Choice => 11
10 15 20 30 50
```

After performing several sort, insert, and delete operations:

```
[0] EXIT APPLICATION          [6]  Delete at any position
[1] Insert at beginning       [7]  Count nodes
[2] Insert at end             [8]  Reverse nodes
[3] Insert at any position    [9]  Sort nodes
[4] Delete at beginning       [10] Split even/odd nodes
[5] Delete at end             [11] TRAVERSE LIST


[ ] Choice => 11
35 10 20 30 50
```

Discussion

Care should be taken while using a single pointer, as the pointer will itself pass by value, thus an easy trap. Use functions that returns pointer to make changes. Ensure user input validation and memory allocation.

Teacher's signature