# **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:

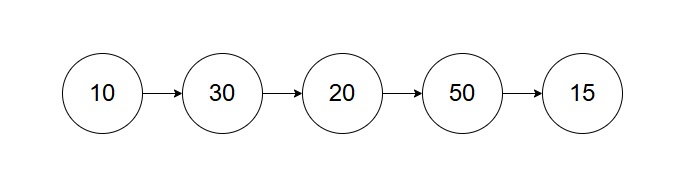
    printf("\n=== Thank you for using this app! ===\n\n");

    return 0;

}

## **Output**

### Input



A screenshot of a computer

Description automatically generated

### Operations

A white screen with black text

Description automatically generated  
A white text on a white background

Description automatically generated

A white screen with black text

Description automatically generated

After performing several sort, insert, and delete operations:

A white screen with black text

Description automatically generated

### 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**