Name: Deshmukh Mehmood Rehan

MIS No. : 612303050

SY Computer Science – Div 1

Question: Define ADT SLL (Singly Linked List of integers). Write the following functions with suitable prototypes for ADT SLL:

init_SLL() // to initialize the list

append() // to add an element at the end of the list.

traverse() // to display all the list elements

insert_at_beg() // to add an element at the beginning of the list

remove_at_pos() // to remove an element from the list from the given position

len() //returns the length of the list.

/*

You are free to include more functions.

The skeleton of function main() is given below. Use the same by replacing commented statements with actual function calls:

```
int main() {
  SLL L1;
  //call init()
  // call append() multiple times to insert elements in the respective list as in the
  // call traverse()
  // call
  // insert_at_beg()
  // remove_at_pos()
  // len()
  return 0;
```

}

Write logic of your program in comments

header.h: This File includes the declarations of structures and function prototypes

```c
/*

 This will be the structure of the node of our Singly Linked List
it will contain an integer i.e data and a pointer to the next Node
in case of last Node the next pointer will point to NULL
*/
typedef struct Node {
    int data; /* the data (in our case it is an Integer)*/
    struct Node *next; /*the next pointer*/
} Node;

/*
To make an ADT all we need is a pointer to the first Node.
So we will typedef it as 'List'
*/
typedef Node * List;

/*These are the functions mentioned in the assignment to implement : */
void init(List *l);  /* this function initializes the list */
void display(List l);  /* this function displays the entire list. */
void append(List *l, int data); /* this function adds an element at the end of
the list. */
void insertAtBeginning(List *l, int data); /* this function adds an element at
the beginning of the list. */
int removeEnd(List *l); /* this function removes an element at the end from
the list. */
int removeBeginning(List *l); /* this function removes an element at the
beginning from the list. */
int removeNode(List *l, Node *n); /* this function removes an element from the
list. */
void addNodeAtPosition(List *l, int data, int position); /* this function adds
an element at the given position of the list. */
int length(List l); /* this function returns the length of the list. */


/*This are some of the extra functions that i have implemented*/
void swapNodes(List * l, Node * n1, Node * n2);
void fill(List *l, int number);
void reverseEven(List *l);
int isPalindrome(List l);
```

```
void removeDuplicates(List *l);
void destroy(List *l);
void removeAndInsert(List *l, Node *n, int index);
```

logic.c : contains the definition of all the functions declared in the header file along with some helper functions

```
#include "./header.h" /*Including the header file*/
#include <stdio.h>
#include <stdlib.h>  /* the C standard Library for functions like malloc  */
#include <limits.h>  /* includes constants like INT_MIN / INT_MAX */


/*
to initialize our ADT we need a notion of an empty list which is possible
when our List will point to NULL
so we will initialize our ADT by making the List point to
NULL
*/
void init(List *l){
    *l = NULL;
    return; /* make l point to NULL*/
}

/*
to traverse and display the list we need a temporary pointer which will
point to the first node . while the temp is not pointing to null we will print
the
node's data and increment temp to point to the next Node

if the list is empty we simply return
*/

void display(List l){
    if(!l){  // if empty.. return
        printf("LinkedList is Empty\n");
        return;
    }

    printf("Linked List: "); // traverse and print each element
    while (l) {
        printf(" %d ->", l->data);
        l = l->next;
    }
    printf("\b\b   \n"); // get rid of the trailing '<->'
    return;
}
```

```c
/*
to append a new element we simply traverse to the end of the list and then
just add the newNode as the next of the current last Node

we also handle the case when the list is empty
*/

void append(List *l, int data){
    Node *newNode = (Node *) malloc(sizeof(Node));
    if (!newNode) { // return if memory allocation fails
        return;
    }

    newNode->data = data;
    newNode->next = NULL;

    if(!*l){ // if list is empty this becomes the first Node
        *l = newNode;
        return;
    }

    Node *temp = *l;

    while(temp->next != NULL){ // traverse to the last Node
        temp = temp->next;
    }
    temp->next = newNode; //add the newNode to the end
    return;
}

/*
This function adds an element at the begginning of the list
the newNode's next will point to the first Node and the newNode will become
the first Node
*/
void insertAtBeginning(List *l, int data){
    Node *newNode = (Node *) malloc(sizeof(Node));
    if (!newNode) return;  // return if memory allocation fails

    newNode->data = data;
    newNode->next = *l; // newNode will point the first Node

    *l = newNode; // newNode becomes the first Node
    return;
}

/*
```

```c
/*
This function removes element at a given index by first traversing to the
previous index and then making it point to the next node of the Node to be
removed
*/
int removeAtIndex(List *l, int index){
    // check for invalid conditions
    if(isEmpty(*l) || index < 0 || index > length(*l)) return INT_MIN;
    if(index == 0){
        return removeBeginning(l);
    }

    Node *removedNode, *temp = *l;
    int removedElement;

    for(int i = 0; i < index - 1; i++){ //traverse to the previous index
        temp = temp->next;
    }

    removedNode = temp->next;
    temp->next = removedNode->next; // change the links
    removedElement = removedNode->data;
    free(removedNode); //free the removed Node

    return removedElement; // return removed Element
}


/*
This function returns the length of the List By traversing the entire list and
incrementing the count each time
*/
int length(List l){
    Node *temp = l;
    int length = 0;
    while (temp){ //traverse the entire list
        length++; //increment the length
        temp = temp->next;
    }
    return length;
}

/* Extra function */

/*
to remove the last element we simply traverse to second last Node. keep a
pointer to last Node.
make the second last Node point to NULL. free the last Node and return the
last element
```

```c
we also handle the case when we have a single element in the list
*/
int removeEnd(List *l){
    Node *temp, *removedNode;
    int removedElement;

    if(!*l) return INT_MIN;


    temp = *l;
    if (!temp->next){
        removedElement = temp->data;
        *l = NULL;
        free(temp);
        return removedElement;
    }


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

    removedNode = temp->next;
    temp->next = NULL;
    removedElement = removedNode->data;
    free(removedNode);

    return removedElement;

}
int removeBeginning(List *l){
    if(!*l) return INT_MIN;

    Node *removedNode = *l;
    int removedElement = removedNode->data;
    *l = (*l)->next;

    free(removedNode);

    return removedElement;
}

int removeNode(List *l, Node *n){
    if(!*l || !n) return INT_MIN;

    Node *removedNode;
    int removedElement;

    Node *temp = *l;
```

```
    if (temp == n){
        removedNode = temp;
        removedElement = removedNode->data;
        *l = temp->next;
        free(removedNode);
        return removedElement;
    }


    while(temp && temp->next != n){
        temp = temp->next;
    }

    if(!temp) return INT_MIN;

    removedNode = temp->next;
    removedElement = removedNode->data;
    temp->next = temp->next->next;
    free(removedNode);

    return removedElement;
}

void addNodeAtPosition(List *l, int data, int position){
    Node *newNode, *temp;
    int i;

    if (position < 0 || position > length(*l)){
        return;
    }

    newNode = (Node *) malloc(sizeof(Node));
    newNode->data = data;
    newNode->next = NULL;

    if (position == 0) {
        newNode->next = *l;
        *l = newNode;
    } else {
        temp = *l;
        for (i = 0; i < position - 1; i++){
            temp = temp->next;
        }
        newNode->next = temp->next;
        temp->next = newNode;
    }
```

```
    return;
}


void swapNodes(List * l, Node * n1, Node * n2){
    if (n1 == n2 || !n1 || !n2 || !(*l)) return;

    List * master = l;

    while ((*master) != n1)
    {
        master = &(*master)->next;
    }

    (*master) = n2;

    master = &n1->next;

    while ((*master) != n2)
    {
        master = &(*master)->next;
    }

    (*master) = n1;

    Node * temp = n1->next;
    n1->next = n2->next;
    n2->next = temp;

    return;
}

void fill(List *l, int number){
    if(number < 1) return;

    for(int i = 0; i < number; i++){
        append(l, rand() % 100 + 1);
    }

    return;
}

void reverseEven(List *l){
    if (length(*l) - 2 < 0) return;

    Node * p = *l, *q, *r, *s;

    while (p->next)
    {
```

```
        q = r = s = NULL;
        if (p->data % 2 == 0){
            q = p;
            r = p;
        }
        else if (p->next->data % 2 == 1){
            p = p->next;
            continue;
        }
        else {
            q = p->next;
            r = q;
        }

        while (r->next->data % 2 == 0)
        {
            r = r->next;
        }

        if (q == r){
            p = r->next;
            continue;
        }
        else s = q->next;

        if (q == p){
            *l = r;
        }
        else {
            p->next = r;
        }

        q->next = r->next;
        p = q;
        q = s;
        s = s->next;

        while (p != r)
        {
            q->next = p;
            p = q;
            q = s;
            s = s->next;
        }

        p = q;
    }
```

```
    return;

}

int isPalindrome(List l){
    int len = length(l);
    if (len - 2 < 0) return 1;

    int halfLength = len / 2;
    int * arr = (int *) malloc(halfLength * sizeof(int));

    Node * p = l;

    for (int i = 0; i < halfLength; i++)
    {
        arr[i] = p->data;
        p = p->next;
    }

    if (len % 2 == 1) p = p->next;

    for (int i = halfLength - 1; i >= 0; i--)
    {
        if (arr[i] != p->data) return 0;
        p = p->next;
    }

    return 1;
}

void removeDuplicates(List *l){
    int len = length(*l);

    int * arr = (int *) calloc(len, sizeof(int));
    if (!arr) return;

    int newlen = 0;
    Node * p = *l;

    arr[newlen++] = p->data;

    while (p->next)
    {
        int data = p->next->data;
        int duplicate = 0;

        for (int i = 0; i < newlen; i++)
        {
```

```
            if (arr[i] == data){
                duplicate = 1;
                break;
            }
        }

        if (duplicate)
        {
            Node * q = p->next;
            p->next = p->next->next;
            free(q);
        }
        else{
            arr[newlen++] = data;
            p = p->next;
        }
    }

}

void destroy(List *l){
    if (!*l) return;

    while (*l){
        removeBeginning(l);
    }

    return;
}

void removeAndInsert(List *l, Node *n, int index){
    if (!*l) return;

    if (*l == n) {
        *l = n->next;
    } else {
        Node *p = *l;
        while (p->next != n && p->next != NULL) {
            p = p->next;
        }
        if (p->next == NULL) return;
        p->next = n->next;
    }

    if (!index)
    {
        n->next = *l;
        *l = n;
```

```
        return;
    }

    Node *q = *l;

    for (int i = 0; i < index - 1; i++)
    {
        q = q->next;
    }

    n->next = q->next;
    q->next = n;

    return;

}
```

main.c : This contains the code to test the implementation

```
#include <stdio.h>
#include <stdlib.h>
#include "logic.c"

int main() {
    List list;
    init(&list);

    append(&list, 1);
    append(&list, 2);
    append(&list, 3);
    display(list);

    printf("\nInserting 0 at the beginning of the linked list:\n");
    insertAtBeginning(&list, 0);
    display(list);

    printf("\nRemoving element from the 2nd Index of the linked list:\n");
    int removedElement = removeAtIndex(&list, 2);
    printf("Removed Element: %d\n", removedElement);
    display(list);

    printf("The length of the list is %d\n", length(list));


    // extra functions test

    printf("\nRemoving element from the end of the linked list:\n");
    int removedEnd = removeEnd(&list);
```

```
    printf("Removed Element: %d\n", removedEnd);
    display(list);


    printf("\nRemoving element from the beginning of the linked list:\n");
    int removedBeginning = removeBeginning(&list);
    printf("Removed Element: %d\n", removedBeginning);
    display(list);

    append(&list, 2);
    append(&list, 3);
    append(&list, 4);
    append(&list, 5);

    display(list);

    printf("\nInserting at index 3:\n");
    addNodeAtPosition(&list, 3, 3);
    display(list);
    append(&list, 4);
    append(&list, 5);
    append(&list, 6);
    display(list);

    printf("Testing swap: \n");
    swapNodes(&list, list->next, list->next->next->next->next);
    display(list);

    List testEven;
    init(&testEven);

    append(&testEven, 1);
    append(&testEven, 2);
    append(&testEven, 6);
    append(&testEven, 9);
    append(&testEven, 10);
    append(&testEven, 24);
    append(&testEven, 22);
    append(&testEven, 11);
    append(&testEven, 65);
    append(&testEven, 13);

    printf("Testing ReverseEven (Before): \n");
    display(testEven);

    reverseEven(&testEven);

    printf("Testing ReverseEven (After):\n");
    display(testEven);
```

```
    List palindrome1, palindrome2;
    init(&palindrome1);
    init(&palindrome2);

    append(&palindrome1, 1);
    append(&palindrome1, 2);
    append(&palindrome1, 3);
    append(&palindrome1, 2);
    append(&palindrome1, 1);

    display(palindrome1);
    printf("The List is Palindrome : %s\n", isPalindrome(palindrome1) ? "True"
: "False");

    append(&palindrome2, 1);
    append(&palindrome2, 2);
    append(&palindrome2, 3);
    append(&palindrome2, 1);
    append(&palindrome2, 2);
    append(&palindrome2, 1);

    display(palindrome2);
    printf("The List is Palindrome : %s\n", isPalindrome(palindrome2) ? "True"
: "False");

    append(&list, 5);
    printf("\nTesting remove Duplicates (Before): \n");
    display(list);

    removeDuplicates(&list);
    printf("Testing remove Duplicates (After):  \n");
    display(list);

    printf("Testing remove and insert (before):  \n");
    display(list);
    removeAndInsert(&list, list->next->next->next, 1);
    printf("Testing remove and insert (after):  \n");
    display(list);

    destroy(&list);
    destroy(&testEven);
    destroy(&palindrome1);
    destroy(&palindrome2);

    display(list);

    fill(&list, 10);
```

```
    printf("\nTesting Fill ");
    display(list);

    printf("Testing Remove Node:\n");
    printf("Removed Element: %d\n", removeNode(&list, list->next->next));
    display(list);

    return 0;
}
```

Output:

```
Labwork 8 Singly LinkedList> gcc .\main.c -Wall -o main
Labwork 8 Singly LinkedList> .\main.exe
Linked List:  1 -> 2 -> 3

Inserting 0 at the beginning of the linked list:
Linked List:  0 -> 1 -> 2 -> 3

Removing element from the 2nd Index of the linked list:
Removed Element: 2
Linked List:  0 -> 1 -> 3
The length of the list is 3

Removing element from the end of the linked list:
Removed Element: 3
Linked List:  0 -> 1

Removing element from the beginning of the linked list:
Removed Element: 0
Linked List:  1
Linked List:  1 -> 2 -> 3 -> 4 -> 5

Inserting at index 3:
Linked List:  1 -> 2 -> 3 -> 3 -> 4 -> 5
Linked List:  1 -> 2 -> 3 -> 3 -> 4 -> 5 -> 4 -> 5 -> 6
Testing swap:
Linked List:  1 -> 4 -> 3 -> 3 -> 2 -> 5 -> 4 -> 5 -> 6
```

```
Testing ReverseEven (Before):
Linked List:  1 -> 2 -> 6 -> 9 -> 10 -> 24 -> 22 -> 11 -> 65 -> 13
Testing ReverseEven (After):
Linked List:  1 -> 6 -> 2 -> 9 -> 22 -> 24 -> 10 -> 11 -> 65 -> 13
Linked List:  1 -> 2 -> 3 -> 2 -> 1
The List is Palindrome : True
Linked List:  1 -> 2 -> 3 -> 1 -> 2 -> 1
The List is Palindrome : False

Testing remove Duplicates (Before):
Linked List:  1 -> 4 -> 3 -> 3 -> 2 -> 5 -> 4 -> 5 -> 6 -> 5
Testing remove Duplicates (After):
Linked List:  1 -> 4 -> 3 -> 2 -> 5 -> 6
Testing remove and insert (before):
Linked List:  1 -> 4 -> 3 -> 2 -> 5 -> 6
Testing remove and insert (after):
Linked List:  1 -> 2 -> 4 -> 3 -> 5 -> 6
LinkedList is Empty

Testing Fill Linked List:  42 -> 68 -> 35 -> 1 -> 70 -> 25 -> 79 -> 59 -> 63 -> 65
Testing Remove Node:
Removed Element: 35
Linked List:  42 -> 68 -> 1 -> 70 -> 25 -> 79 -> 59 -> 63 -> 65
Labwork 8 Singly LinkedList>
```