```c
/* Header files */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

/* Symbolic constants */
#define SUCCESS 1
#define TRUE    1
#define FALSE   0
#define LIST_DATA_NOT_FOUND 2
#define LIST_EMPTY  3

/* Node layout definition */
struct node
{
    int data;
    struct node* next;
};

/* interface function declarations */
/* List creation function */
struct node* create_list(void);

/* Data addition feature */
int insert_start(struct node* p_list, int new_data);
int insert_end(struct node* p_list, int new_data);
int insert_after(struct node* p_list, int existing_data, int new_data);
int insert_before(struct node* p_list, int existing_data, int new_data);

/* Get function */
int get_start(struct node* p_list, int* p_start_data);
int get_end(struct node* p_list, int* p_end_data);

/* Pop function */
int pop_start(struct node* p_list, int* p_start_data);
int pop_end(struct node* p_list, int* p_end_data);

/* Remove functions */
int remove_start(struct node* p_list);
int remove_end(struct node* p_list);
int remove_data(struct node* p_list, int r_data);

/* Miscellaneous functions */
int find(struct node* p_list, int f_data);
int get_list_length(struct node* p_list);
int is_list_empty(struct node* p_list);
void show_list(struct node* p_list, const char* msg);

/* List destruction function */
int destroy_list(struct node* p_list);


/* Client of Linked List */
int main(void)
{
    struct node* p_list = NULL;
    int data, start_data, end_data;
    int length;
    int status;

    static const char* line = 
"----------------------------------------------------------------";
    p_list = create_list();
    assert(NULL != p_list);
    printf("list created successfully!\n");
    puts(line);

    printf("Testing assertion on the empty list\n");
    assert(TRUE == is_list_empty(p_list));
    assert(0 == get_list_length(p_list));
    assert(LIST_EMPTY == get_start(p_list, &start_data));
    assert(LIST_EMPTY == get_end(p_list, &end_data));
    assert(LIST_EMPTY == pop_start(p_list, &start_data));
    assert(LIST_EMPTY == pop_end(p_list, &end_data));

    assert(LIST_EMPTY == remove_start(p_list));
    puts(line);
    assert(LIST_EMPTY == remove_end(p_list));
    printf("All assertion on the empty list are successful\n");
    puts(line);

    show_list(p_list, "Showing empty list immediately after creation:");
    puts(line);

    for (data = 0; data < 5; ++data)
    {
        status = insert_start(p_list, data * 10);
        assert(SUCCESS == status);
        printf("%d inserted successfully at the start of the list\n", data *
10);
    }
    show_list(p_list, "Showing list after inserting 5 data elements at the
start:");
    puts(line);

    for (data = 0; data < 5; ++data)
    {
        status = insert_end(p_list, data * 5);
        assert(SUCCESS == status);
        printf("%d inserted successfully at the end of the list\n", data * 5);
    }
    show_list(p_list, "Showing list after inserting 5 data elements at the
end:");
    puts(line);


    status = insert_after(p_list, -5, 100);
    assert(LIST_DATA_NOT_FOUND == status);
    printf("Expected failure to insert data 100 after non-existent data -5\n");
    puts(line);

    status = insert_after(p_list, 0, 100);
    assert(SUCCESS == status);
    show_list(p_list, "Showing list after successfully inserting 100 after 0:");
    puts(line);

    status = insert_before(p_list, 43, 200);
    assert(LIST_DATA_NOT_FOUND == status);
    printf("Expected failure to insert data 200 before non-existent data 43\n");
    puts(line);

    status = insert_before(p_list, 0, 200);
    assert(SUCCESS == status);
    show_list(p_list, "Showing list after successfully inserting data 200 before
0:");
    puts(line);
    status = get_start(p_list, &start_data);
    assert(SUCCESS == status);
    printf("Data at the start:%d\n", start_data);
    show_list(p_list, "Showing list to demonstrate that get_start() returns
start data without removing it:");
    puts(line);

    status = get_end(p_list, &end_data);
    assert(SUCCESS == status);
    printf("Data at the end:%d\n", end_data);
    show_list(p_list, "Showing list to demonstrate that get_end() returns data
without removing it:");
    puts(line);


    status = pop_start(p_list, &start_data);
    assert(SUCCESS == status);
    printf("Data at the start:%d\n", start_data);
    show_list(p_list, "Showing list to demonstrate that pop_start() returns data
and removes it from the list:");
    puts(line);

    status = pop_end(p_list, &end_data);
    assert(SUCCESS == status);
    printf("Dta at the end:%d\n", end_data);
    show_list(p_list, "Showing list to demonstrate that pop_end() returns data
and removes it from the list:");
    puts(line);

    status = remove_start(p_list);
    assert(SUCCESS == status);
    show_list(p_list, "Showing list after remove_start():");
    puts(line);

    status = remove_end(p_list);
    assert(SUCCESS == status);
    show_list(p_list, "Showing list after remove_end():");
    puts(line);

    status = remove_data(p_list, 78);
    assert(LIST_DATA_NOT_FOUND == status);
    printf("Expected error in removing non-existing data 78\n");
    puts(line);

    status = remove_data(p_list, 0);
    assert(SUCCESS == status);
    show_list(p_list, "Showing list after removing existing data 0:");
    puts(line);

    status = find(p_list, 91);
    assert(LIST_DATA_NOT_FOUND == status);
    printf("Expected return value FALSE from find() for non-existent data
91\n");
    puts(line);

    status = find(p_list, 100);
    assert(SUCCESS == status);
    printf("Expected return value TRUE from find() for existing data 100\n");
    puts(line);

    status = is_list_empty(p_list);
    assert(FALSE == status);
    printf("Expected return value FALSE from is_list_empty()\n");
    puts(line);
```

```c
    length = get_list_length(p_list);
    assert(length);
    printf("Length of the list:%d\n", length);
    puts(line);

    status = destroy_list(p_list);
    assert(SUCCESS == status);
    p_list = NULL;
    printf("List is destroyed successfully\n");
    puts(line);

    return (0);
}

/* Server of Linked List */
/* List creation function */
struct node* create_list(void)
{
    struct node* head_node = NULL;

    head_node = (struct node*)malloc(sizeof(struct node));
    if (NULL == head_node)
    {
        puts("out of memory");
        exit(EXIT_FAILURE);
    }

    head_node->next = NULL;
    head_node->data = 0;

    return (head_node);
}


/* Data addition feature */
int insert_start(struct node* p_list, int new_data)
{
    struct node* new_node = NULL;
    new_node = (struct node*)malloc(sizeof(struct node));
    if (NULL == new_node)
    {
        puts("out of memory");
        exit(EXIT_FAILURE);
    }

    new_node->data = new_data;
    new_node->next = p_list->next;
    p_list->next = new_node;

    return (SUCCESS);
}

int insert_end(struct node* p_list, int new_data)
{
    struct node* new_node = NULL;
    struct node* run = NULL;

    new_node = (struct node*)malloc(sizeof(struct node));
    if (NULL == new_node)
    {
        puts("out of memory");
        exit(EXIT_FAILURE);
    }
    new_node->data = new_data;
    new_node->next = NULL;

    // locate the last node
    run = p_list;
    while (run->next != NULL)
    {
        run = run->next;
    }

    // append the new node at the last position
    run->next = new_node;

    return (SUCCESS);
}

int insert_after(struct node* p_list, const int existing_data, int new_data)
{
    struct node* existing_node = NULL;
    struct node* run = NULL;
    struct node* new_node = NULL;

    // locate the node containing the first occurrence of the existing_data
    run = p_list->next;
    while (NULL != run)
    {
        if (existing_data == run->data)
        {
            break;
        }
        run = run->next;
    }

    // If data is not found return error
    if (NULL == run)
    {
        return (LIST_DATA_NOT_FOUND);
    }

    //  allocate and initialize new node
    new_node = (struct node*)malloc(sizeof(struct node));
    if (NULL == new_node)
    {
        puts("out of memory");
        exit(EXIT_FAILURE);
    }

    new_node->data = new_data;
    new_node->next = NULL;

    // insert the new node at its appropriate location
    existing_node = run;
    new_node->next = existing_node->next;
    existing_node->next = new_node;

    return (SUCCESS);
}

int insert_before(struct node* p_list, const int existing_data, int new_data)
{
    struct node* new_node = NULL;
    struct node* run = NULL;
    struct node* run_prev = NULL;
    // locate node existing_data
    run_prev = p_list;
    run = p_list->next;
    while (NULL != run)
    {
        if (existing_data == run->data)
        {
            break;
        }
        run_prev = run;
        run = run->next;
    }

    // Return error if data not found
    if (NULL == run)
    {
        return (LIST_DATA_NOT_FOUND);
    }

    // allocate and initialize new node
    new_node = (struct node*)malloc(sizeof(struct node));
    if (NULL == new_node)
    {
        puts("out of memory");
        exit(EXIT_FAILURE);
    }

    new_node->data = new_data;
    new_node->next = NULL;

    // insert new node before found node
    new_node->next = run_prev->next;
    run_prev->next = new_node;

    return (SUCCESS);
}

/* Get function */
int get_start(struct node* p_list, int* p_start_data)
{
    if (NULL == p_list->next)
    {
        return (LIST_EMPTY);
    }

    *p_start_data = p_list->next->data;

    return (SUCCESS);
}

int get_end(struct node* p_list, int* p_end_data)
{
    struct node* run = NULL;

    if (NULL == p_list->next)
    {
        return (LIST_EMPTY);
    }

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

    *p_end_data = run->data;

    return (SUCCESS);
}

/* Pop function */
int pop_start(struct node* p_list, int* p_start_data)
{
    struct node* delete_prev = NULL;
    struct node* delete_node = NULL;
    struct node* delete_next = NULL;

    if (NULL == p_list->next)
    {
        return (LIST_EMPTY);
    }

    *p_start_data = p_list->next->data;

    delete_prev = p_list;
    delete_node = p_list->next;
    delete_next = p_list->next->next;

    delete_prev->next = delete_next;

    free(delete_node);
    delete_node = NULL;

    return (SUCCESS);
}

int pop_end(struct node* p_list, int* p_end_data)
{
    struct node* run_prev = NULL;
    struct node* run = NULL;

    if (NULL == p_list->next)
    {
        return (LIST_EMPTY);
    }

    run_prev = p_list;
    run = p_list->next;
    while (NULL != run->next)
    {
        run_prev = run;
        run = run->next;
    }

    *p_end_data = run->data;

    run_prev->next = NULL;
    free(run);
    run = NULL;

    return (SUCCESS);
}

/* Remove functions */
int remove_start(struct node* p_list)
{
    struct node* delete_prev = NULL;
```

```c
    struct node* delete_node = NULL;
    struct node* delete_next = NULL;

    if (NULL == p_list->next)
    {
        return (LIST_EMPTY);
    }

    delete_prev = p_list;
    delete_node = p_list->next;
    delete_next = p_list->next->next;

    delete_prev->next = delete_next;

    free(delete_node);
    delete_node = NULL;

    return (SUCCESS);
}

int remove_end(struct node* p_list)
{
    struct node* run_prev = NULL;
    struct node* run = NULL;

    if (NULL == p_list->next)
    {
        return (LIST_EMPTY);
    }

    run_prev = p_list;
    run = p_list->next;

    // find last node
    while (NULL != run->next)
    {
        run_prev = run;
        run = run->next;
    }

    // second last node becomes last node
    run_prev->next = NULL;

    // destroy last node
    free(run);
    run = NULL;

    return (SUCCESS);
}

int remove_data(struct node* p_list, const int r_data)
{
    struct node* run = NULL;
    struct node* run_prev = NULL;

    run_prev = p_list;
    run = p_list->next;

    while (NULL != run)
    {
        if (r_data == run->data)
        {
            break;
        }
        run_prev = run;
        run = run->next;
    }

    // node not found in list with given data
    if (NULL == run)
    {
        return (LIST_DATA_NOT_FOUND);
    }

    // skip node with found data
    run_prev->next = run->next;

    // destroy node with given data
    free(run);
    run = NULL;

    return (SUCCESS);
}

/* Miscellaneous functions */
int find(struct node* p_list, const int f_data)
{
    struct node* run = NULL;

    run = p_list->next;
    while (NULL != run)
    {
        if (f_data == run->data)
        {
            break;
        }
        run = run->next;
    }

    if (NULL == run)
    {
        return (LIST_DATA_NOT_FOUND);
    }

    return (SUCCESS);
}

int get_list_length(struct node* p_list)
{
    int len = 0;
    struct node* run = NULL;

    run = p_list->next;
    while (NULL != run)
    {
        len = len + 1;
        run = run->next;
    }

     return (len);
}

int is_list_empty(struct node* p_list)
{
    return (NULL == p_list->next);
}

void show_list(struct node* p_list, const char* msg)
```

```
{
    struct node* run = NULL;

    if (msg)
    {
        puts(msg);
    }

    printf("[START]->");
    run = p_list->next;
    while (NULL != run)
    {
        printf("[%d]->", run->data);
        run = run->next;
    }
    printf("[END]\n");

}

/* List destruction function */
int destroy_list(struct node* p_list)
{
    struct node* run = NULL;
    struct node* run_next = NULL;

    run = p_list;
    run_next = p_list->next;

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

    return (SUCCESS);
}
```