```c
/* Header files */
#include<stdio.h>
#include<stdlib.h>
#include<assert.h>
#include<string.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 functions declarations */
/* List creation function */
struct node* create_list(void);

/* Data addition functions */
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 functions */
int get_start(struct node* p_list, int* p_start_data);
int get_end(struct node* p_list, int* p_end_data);

/* Pop Functions */
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);

/* Miscallaneous 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);

```

```c
/* Client of linked list */

int main(void){
    struct node* p_list = NULL;

    int status;
    int data, start_data, end_data;
    int length;

    static const char* line = "----------------------------------------------------------------

    p_list = create_list();
    assert(p_list != NULL);
    printf("List created successfully\n");
    puts(line);

    printf("Testing assertions on the empty list\n");
    assert(is_list_empty(p_list) == TRUE);
    assert(get_list_length(p_list) == 0);
    assert(get_start(p_list, &start_data) == LIST_EMPTY);
    assert(get_end(p_list, &end_data) == LIST_EMPTY);
    assert(pop_start(p_list, &start_data) == LIST_EMPTY);
    assert(pop_end(p_list, &end_data) == LIST_EMPTY);
    assert(remove_start(p_list) == LIST_EMPTY);
    assert(remove_end(p_list) == LIST_EMPTY);
    printf("All assertions 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(status == SUCCESS);
        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 = 1; data <= 5; ++data)
    {
        status = insert_end(p_list, data * 5);
        assert(status == SUCCESS);
        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:\n");
    puts(line);

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

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

    status = insert_before(p_list, 43, 200);
    assert(status == LIST_DATA_NOT_FOUND);
```

```
112         printf("Expected failure to insert data 200 after non-existent data 43\n");
113         puts(line);
114
115         status = insert_before(p_list, 0, 200);
116         assert(status == SUCCESS);
117         show_list(p_list, "Showing list after successfully inserting data 200 before 0:");
118         puts(line);
119
120         status = get_start(p_list, &start_data);
121         assert(status == SUCCESS);
122         printf("Data at the start:%d\n", start_data);
123         show_list(p_list, "Showing list to demonstrate that get_start() returns start data without removing it:");
124         puts(line);
125
126         status = get_end(p_list, &end_data);
127         assert(status == SUCCESS);
128         printf("Data at the end:%d\n", end_data);
129         show_list(p_list, "Showing list to demonstrate that get_end() returns the end data without removing it:");
130         puts(line);
131
132         status = pop_start(p_list, &start_data);
133         assert(status == SUCCESS);
134         printf("Data at the start = %d\n", start_data);
135         show_list(p_list, "Showing list to demonstrate that pop_start() removes and returns the start data:");
136         puts(line);
137
138         status = pop_end(p_list, &end_data);
139         assert(status == SUCCESS);
140         printf("Data at the end = %d\n", end_data);
141         show_list(p_list, "Showing list to demonstrate that pop_end() removes and returns the end data:");
142         puts(line);
143
144         status = remove_start(p_list);
145         assert(status == SUCCESS);
146         show_list(p_list, "Showing list after remove_start():");
147         puts(line);
148
149         status = remove_end(p_list);
150         assert(status == SUCCESS);
151         show_list(p_list, "Showing list after remove_end():");
152         puts(line);
153
154         status = remove_data(p_list, 78);
155         assert(status == LIST_DATA_NOT_FOUND);
156         printf("Expected error in removing non-existent data 78\n");
157         puts(line);
158
159         status = remove_data(p_list, 0);
160         assert(status == SUCCESS);
161         show_list(p_list, "Showing list after removing existing data 0\n");
162         puts(line);
163
164         status = find(p_list, 91);
165         assert(status == FALSE);
166         printf("Expected return value FALSE from find() for non-existent data 91\n");
167         puts(line);
168
169         status = find(p_list, 100);
170         assert(status == TRUE);
171         printf("Expected return value TRUE from find() for existing data 30\n");
```

```c
    puts(line);

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

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

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

    return(0);
}

/* Server of linked list */
/* Interface functions declarations */
/* 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->data = 0;
    head_node->next = NULL;

    return (head_node);
}

/* Data addition functions */
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);
}
```

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

    // Step 1: 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;

    // Step 2: Locate the last node
    run = p_list;
    while(run->next != NULL)
    {
        run = run->next;
    }

    // Step 3: Append the new node at the last position
    run->next = new_node;

    return (SUCCESS);
}

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

    // Step 1: Search for the node containing the first occurrence of the existing data
    run = p_list->next;
    while(run != NULL)
    {
        if(run->data == existing_data)
        {
            break;
        }
        run = run->next;
    }

    // If existing data is not found then return error as such
    if(NULL == run)
        return (LIST_DATA_NOT_FOUND);


    // Step 2: Allocate and initialize new node
    existing_node = run;
    new_node = (struct node*)malloc(sizeof(struct node));
    if(NULL == new_node)
    {
        puts("out of memory");
        exit(EXIT_FAILURE);
    }

    new_node->data = new_data;

    // Step 3: Insert the new node at its appropriate position
    new_node->next = existing_node->next;
    existing_node->next = new_node;

    return (SUCCESS);
}
```

```c
int insert_before(struct node* p_list, int existing_data, int new_data)
{
    struct node* run = NULL;
    struct node* run_previous = NULL;
    struct node* new_node = NULL;

    // Step 1:  Search for the first occurrence of the existing data
    //          maintaining the back pointer
    run_previous = p_list;
    run = p_list->next;
    while(run != NULL)
    {
        if(run->data == existing_data)
        {
            break;
        }

        run_previous = run;
        run = run->next;
    }

    //  if the existing data is not found throughout the list then return an error
    //  saying as such
    if(run == NULL)
        return (LIST_DATA_NOT_FOUND);

    //  Step 2: Allocate and initialize the 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;
    //  Step 3: Insert the new node at the appropriate position
    new_node->next = run;
    run_previous->next = new_node;

    return (SUCCESS);
}

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

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

    *p_end_data = run->data;

    return (SUCCESS);
}
```

```c
/* Pop Functions */
int pop_start(struct node* p_list, int* p_start_data)
{
    struct node* delete_previous = NULL;
    struct node* delete_node = NULL;
    struct node* delete_next = NULL;

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

    *p_start_data = p_list->next->data;

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

    delete_previous->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 = NULL;
    struct node* run_previous = NULL;

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

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

    *p_end_data = run->data;

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

    return (SUCCESS);
}

/* Remove functions */
int remove_start(struct node* p_list)
{
    struct node* delete_previous = NULL;
    struct node* delete_node = NULL;
    struct node* delete_next = NULL;

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

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

    delete_previous->next = delete_next;

    free(delete_node);
    delete_node = NULL;

    return (SUCCESS);
}
```

```
440     }
441
442     int remove_end(struct node* p_list)
443     {
444         struct node* run = NULL;
445         struct node* run_previous = NULL;
446
447         if(p_list->next == NULL)
448             return (LIST_EMPTY);
449
450         run_previous = p_list;
451         run = p_list->next;
452         while(run->next != NULL)
453         {
454             run_previous = run;
455             run = run->next;
456         }
457
458         free(run);
459         run = NULL;
460         run_previous->next = NULL;
461
462         return (SUCCESS);
463     }
464
465     int remove_data(struct node* p_list, int r_data)
466     {
467         struct node* run = NULL;
468         struct node* run_previous = NULL;
469
470         run_previous = p_list;
471         run = p_list->next;
472         while(run != NULL)
473         {
474             if(run->data == r_data)
475                 break;
476
477             run_previous = run;
478             run = run->next;
479         }
480
481         if(run == NULL)
482             return (LIST_DATA_NOT_FOUND);
483
484         run_previous->next = run->next;
485         free(run);
486         run = NULL;
487
488         return (SUCCESS);
489     }
490
491     /* Miscallaneous functions */
492     int find(struct node* p_list, int f_data)
493     {
494         struct node* run = NULL;
495
496         run = p_list->next;
497         while(run != NULL)
498         {
499             if(run->data == f_data)
500                 return (TRUE);
501             run = run->next;
502         }
503
504         return (FALSE);
505     }
506
```

```c
}

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

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

    return (len);
}

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

void show_list(struct node* p_list, const char* msg)
{
    struct node* run = NULL;

    if(msg != NULL)
        puts(msg);

    printf("[START]->");
    run = p_list->next;
    while(run != NULL)
    {
        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;
    while(run != NULL)
    {
        run_next = run->next;
        free(run);
        run = run_next;
    }

    return (SUCCESS);
}
```