Data Structure Linked List

Người trình bày: Phạm Hồng Thi

Table of contents

01

DEFINITION

02

BASIC OPERATION

03

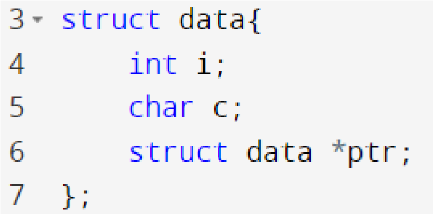
IMPLEMENT DOUBLY LINKED LIST

01

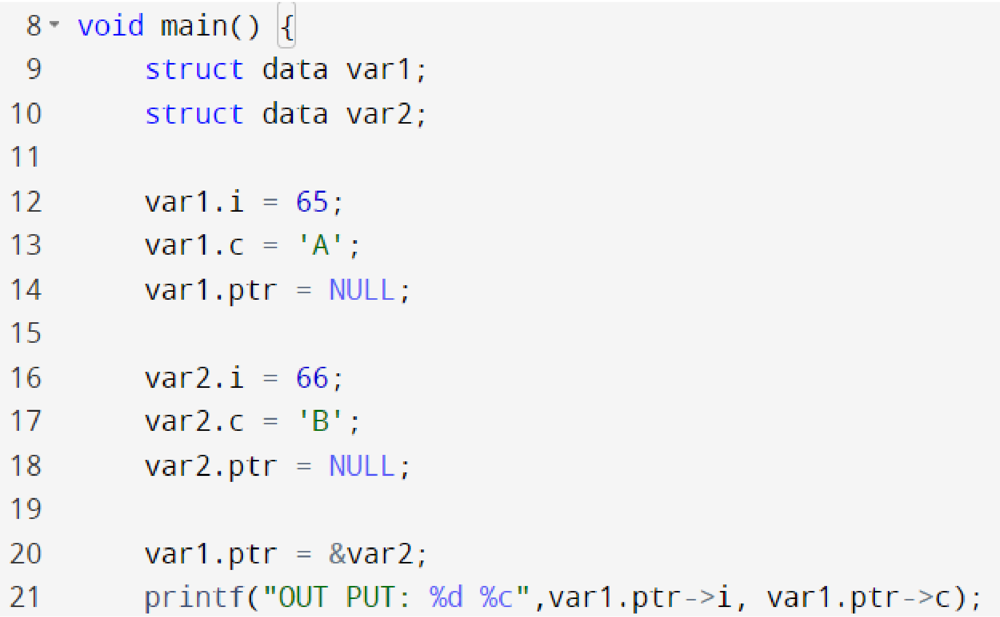
DEFINITION

1. Linked List: Self refrential structures

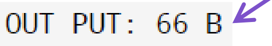
Definition: Self refential structures are those structures in which ***one or more*** pointer point to the structure of the same type



```
struct data{
        int i;
        char c;
        struct data *ptr;
};
```



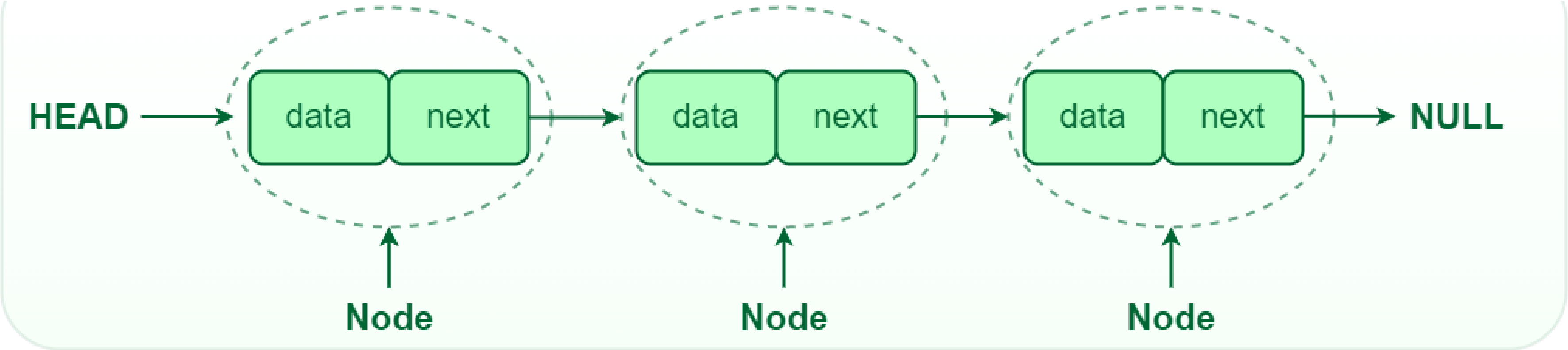
```
void main() {
        struct data var1;
        struct data var2;

        var1.i = 65;
        var1.c = 'A';
        var1.ptr = NULL;

        var2.i = 66;
        var2.c = 'B';
        var2.ptr = NULL;

        var1.ptr = &var2;
        printf("OUT PUT: %d %c",var1.ptr->i, var1.ptr->c);
```



```
OUT PUT: 66 B
```

1. Linked List: Definition

Linked List is a ***linear data structure***, in which elements are not stored at a contiguous location, rather they are linked using pointers. Linked List forms a series of connected nodes, where each node stores the data and the address of the next node.



A node in a linked list typically consists of two components:

- Data
- Next Pointer / Previous Pointer

Head and Tail: The linked list is accessed through the ***head node,*** which points to the first node in the list. The last node in the list points to NULL, indicating the end of the list. This node is known as the ***tail node***.

1. Linked List: Advantages

- ***Dynamic size***: Linked lists do not have a fixed size, so you can add or remove elements as needed, without having to worry about the size of the list.
- ***Efficient Insertion and Deletion***: Inserting or deleting elements in a linked list is fast and efficient, as you only need to modify the reference of the next node, which is an O(n) operation.
- ***Memory Efficiency***: Linked lists use only as much memory as they need, so they are more efficient with memory compared to arrays, which have a fixed size and can waste memory if not all elements are used.

1. Linked List: Disadvantages

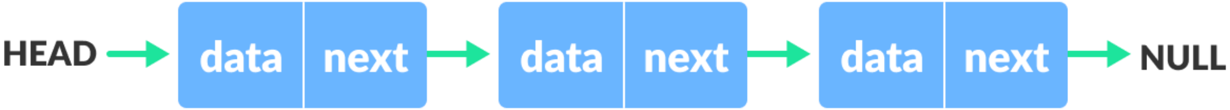
- ***Slow Access Time***: Accessing elements in a linked list can be slow, as you need to traverse the linked list to find the element you are looking for, which is an O(n) operation. This makes linked lists a poor choice for situations where you need to access elements quickly.
- ***Pointers***: Linked lists use pointers to reference the next node, which can make them more complex to understand and use compared to arrays. This complexity can make linked lists more difficult to debug and maintain.
- ***Extra memory required***: Linked lists require an extra pointer for each node, which takes up extra memory. This can be a problem when you are working with large data sets, as the extra memory required for the pointers can quickly add up.

1. Linked List: Array vs LinkedList

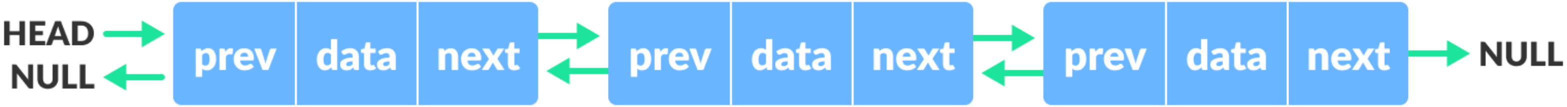
Array	Linked List
1. Mảng lưu trữ dữ liệu ở các ô nhớ liên tiếp	1. Danh sách lưu trữ dữ liệu không tại các ô tùy ý
2. Kích thước đã gán cố định	2. Kích thước động thay đổi dễ dàng
3. Bộ nhớ cấp phát tại compile time	3. Bộ nhớ cấp phát tại run time
4. Sử dụng ít bộ nhớ hơn danh sách	4. Sử dụng nhiều bộ nhớ hơn do cần lưu trữ địa chỉ node liền kề
5. Truy cập phần tử dễ dàng nhờ có chỉ số mảng	5. Truy cập phần tử lâu hơn do phải duyệt toàn bộ danh sách
6. Thao tác chèn và xóa thực hiện lâu	6. Thao tác chèn và xóa thực hiện nhanh hơn mảng

1. Linked List: Type of LinkedList

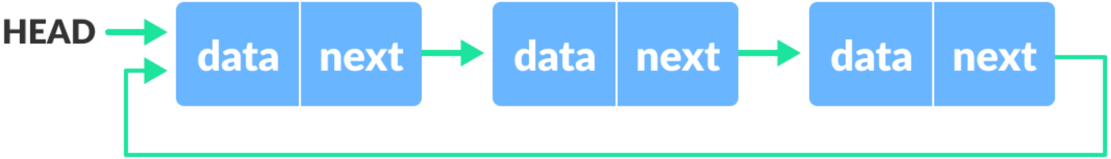
- Singly Linked List



- Doubly linked list



- Circular Linked List



02

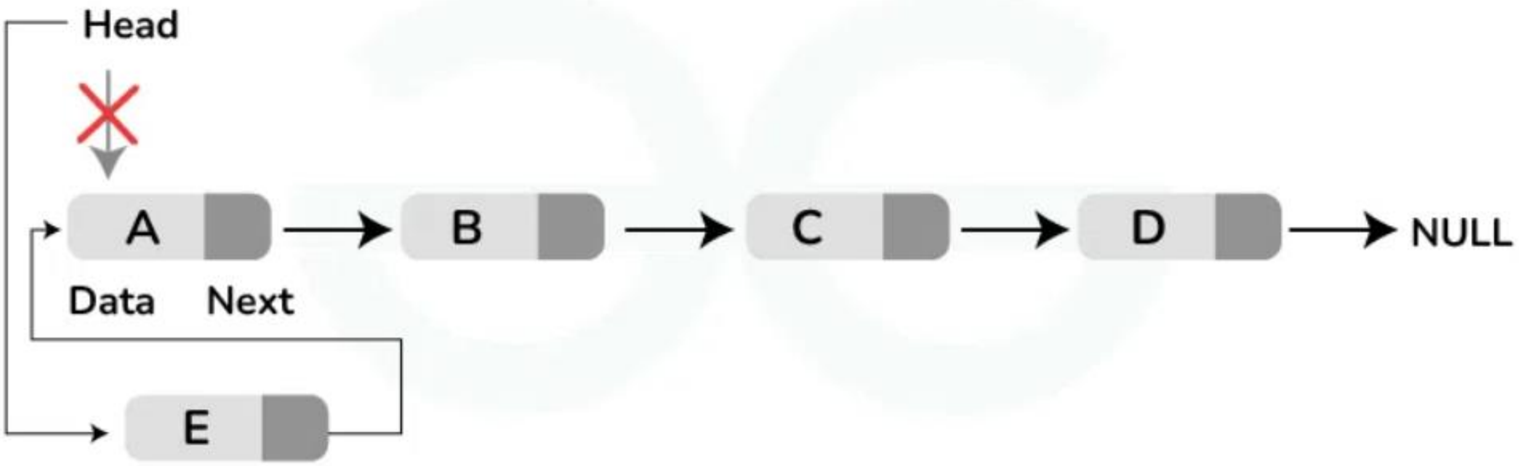
BASIC OPERATION

Single Linked List

2. Basic Operation

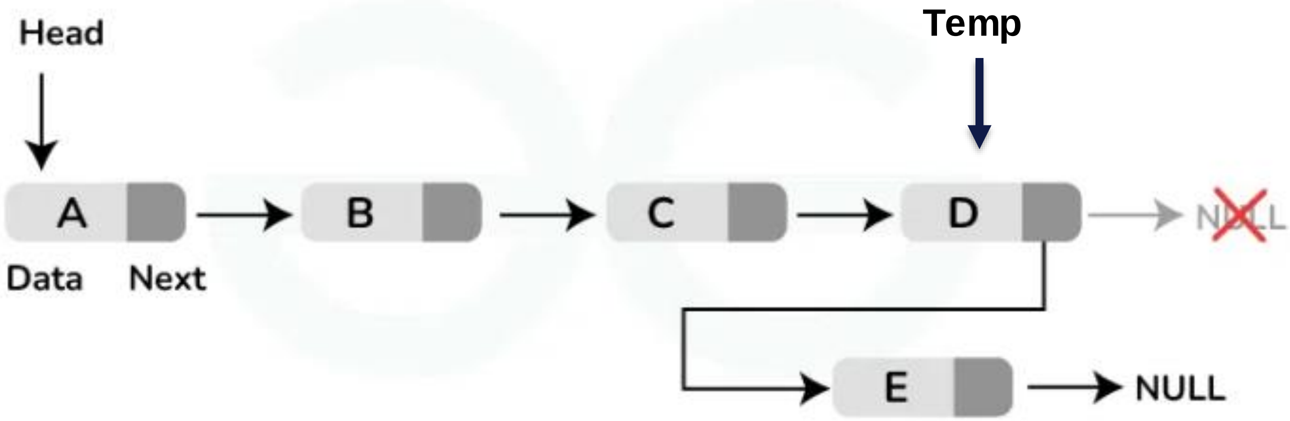
- ***Insertion***
- ***Deletion***
- ***Search an element***
- ***Reverse***

2. Basic Operation: *Insertion*



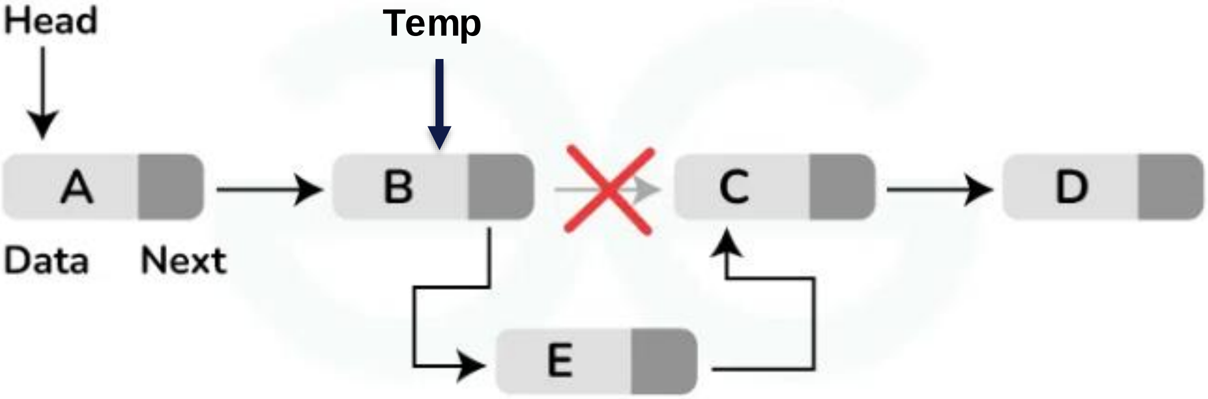
Insert in the front

2. Basic Operation: *Insertion*



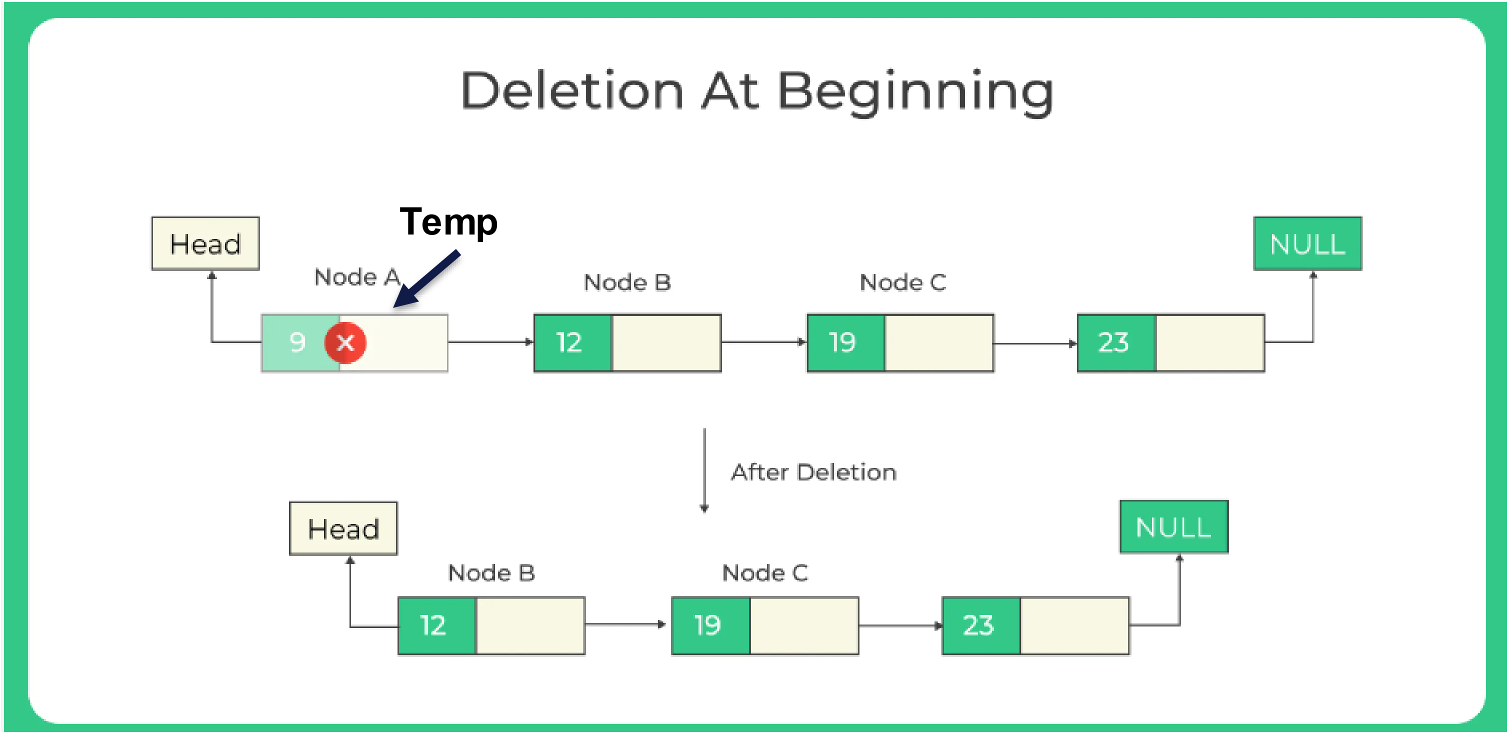
Insert in the end

2. Basic Operation: *Insertion*

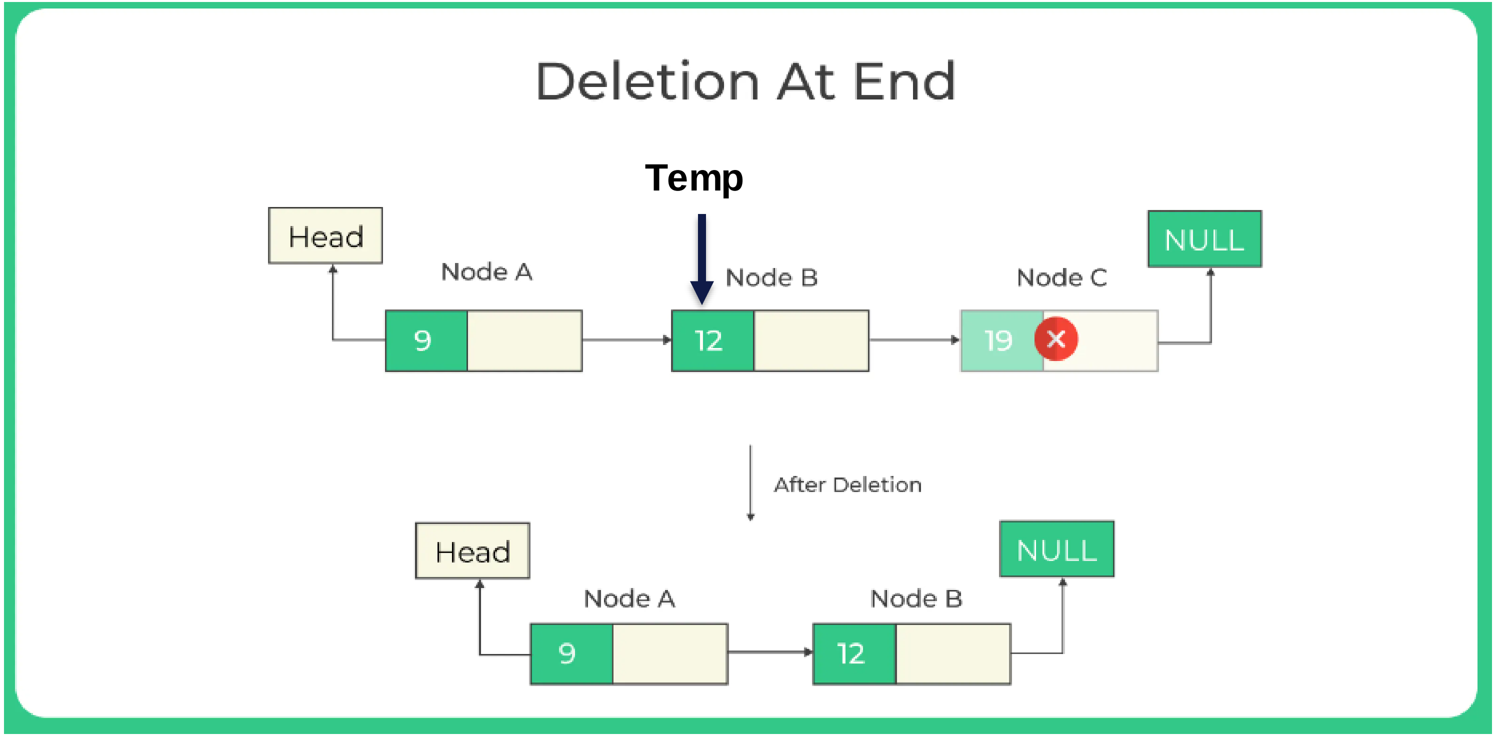


Insert after a given node

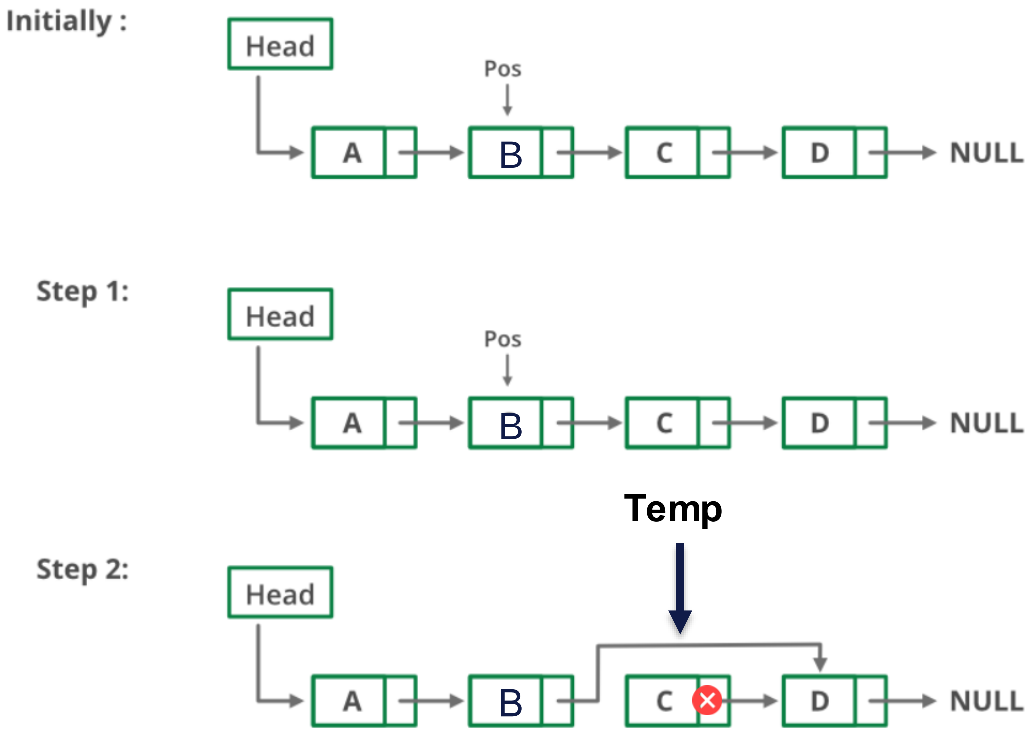
2. Basic Operation: ***Deletion***



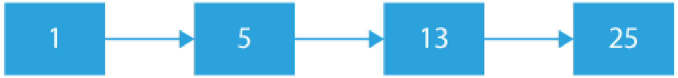
2. Basic Operation: ***Deletion***



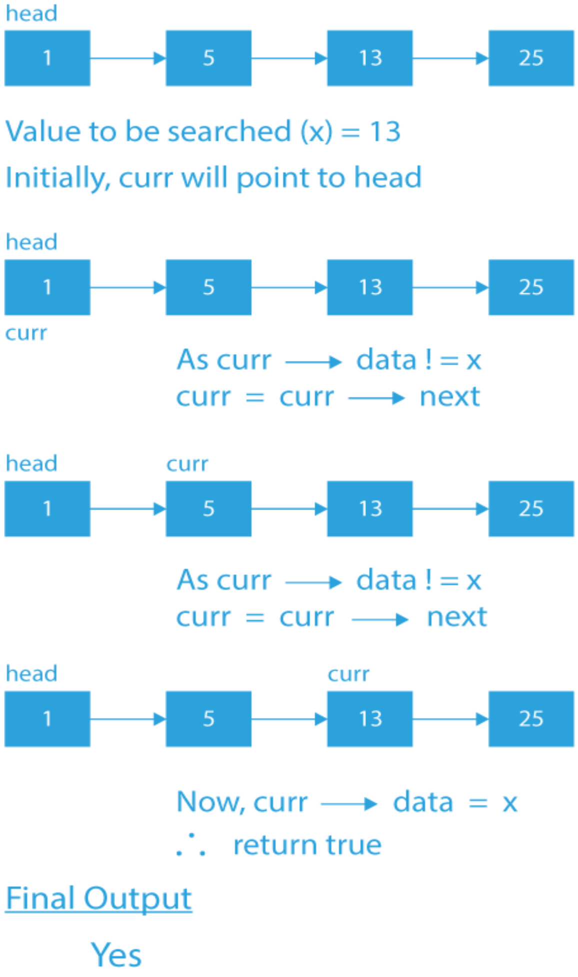
2. Basic Operation: ***Deletion***



2. Basic Operation: ***Search an element***



Data Node = 13 ?



Value to be searched (x) = 13

Initially, curr will point to head

As curr → data != x

curr = curr → next

As curr → data != x

curr = curr → next

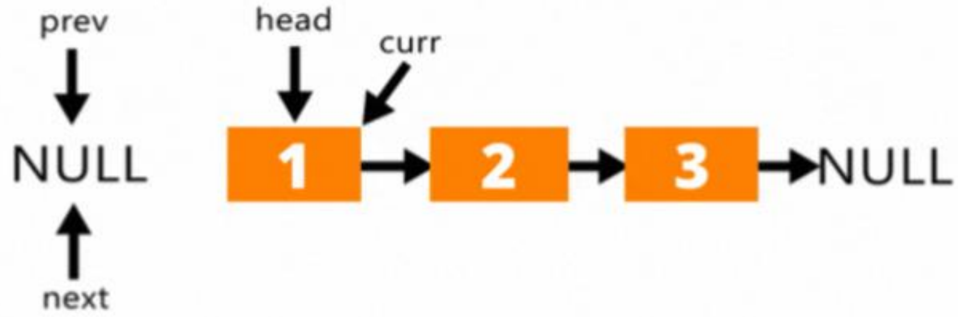
Now, curr → data = x

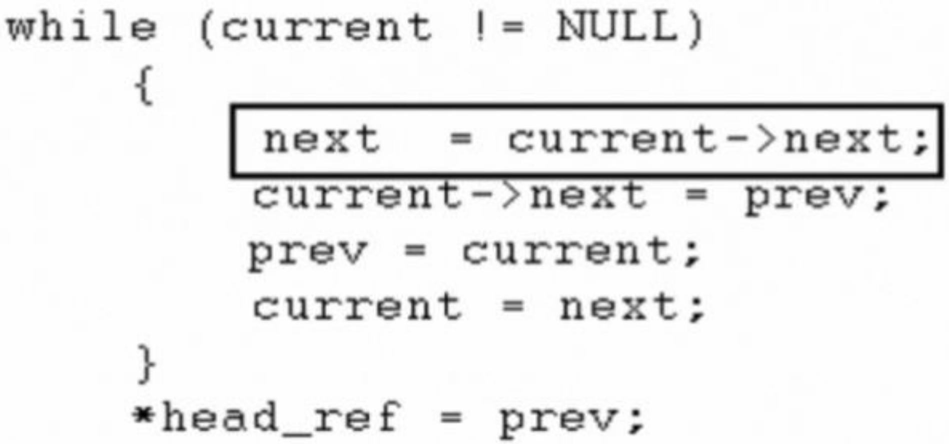
∴ return true

Final Output

Yes

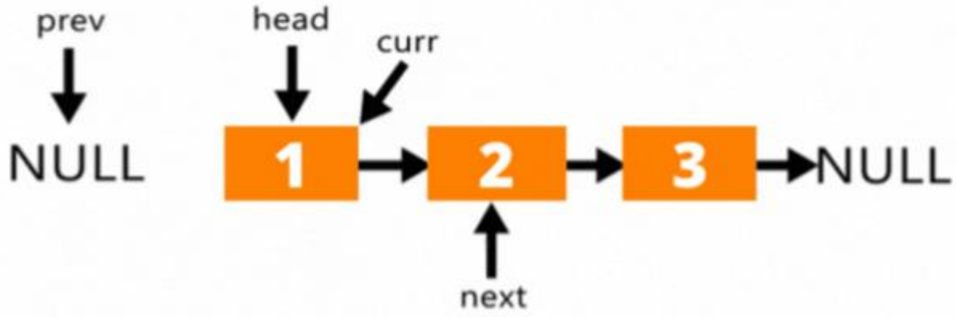
2. Basic Operation: ***Reverse***

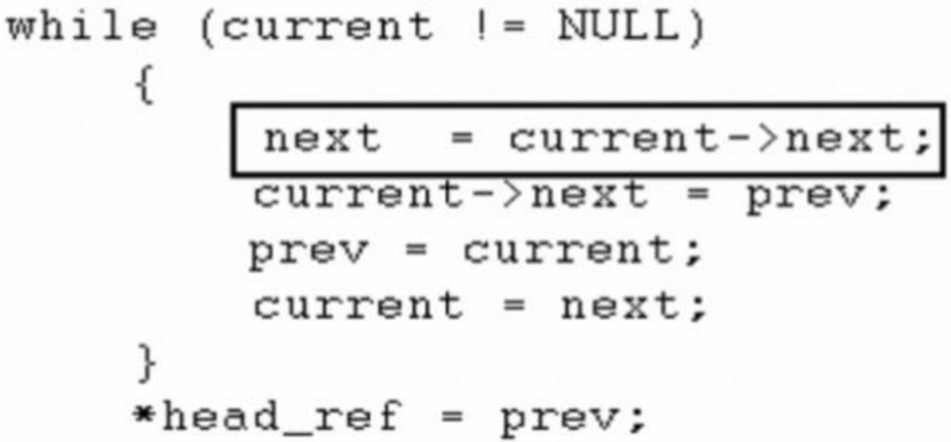




```
while (current != NULL)
    {
        next  = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    *head_ref = prev;
```

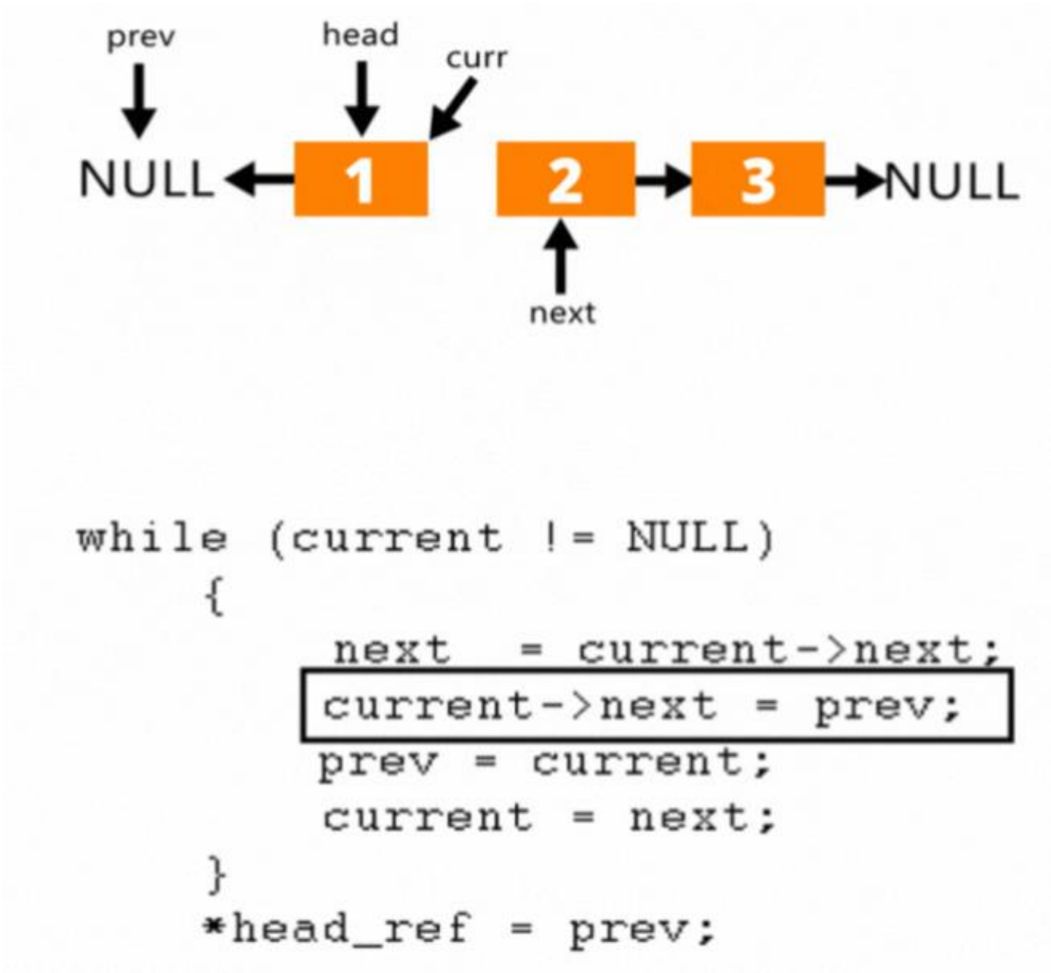
2. Basic Operation: ***Reverse***





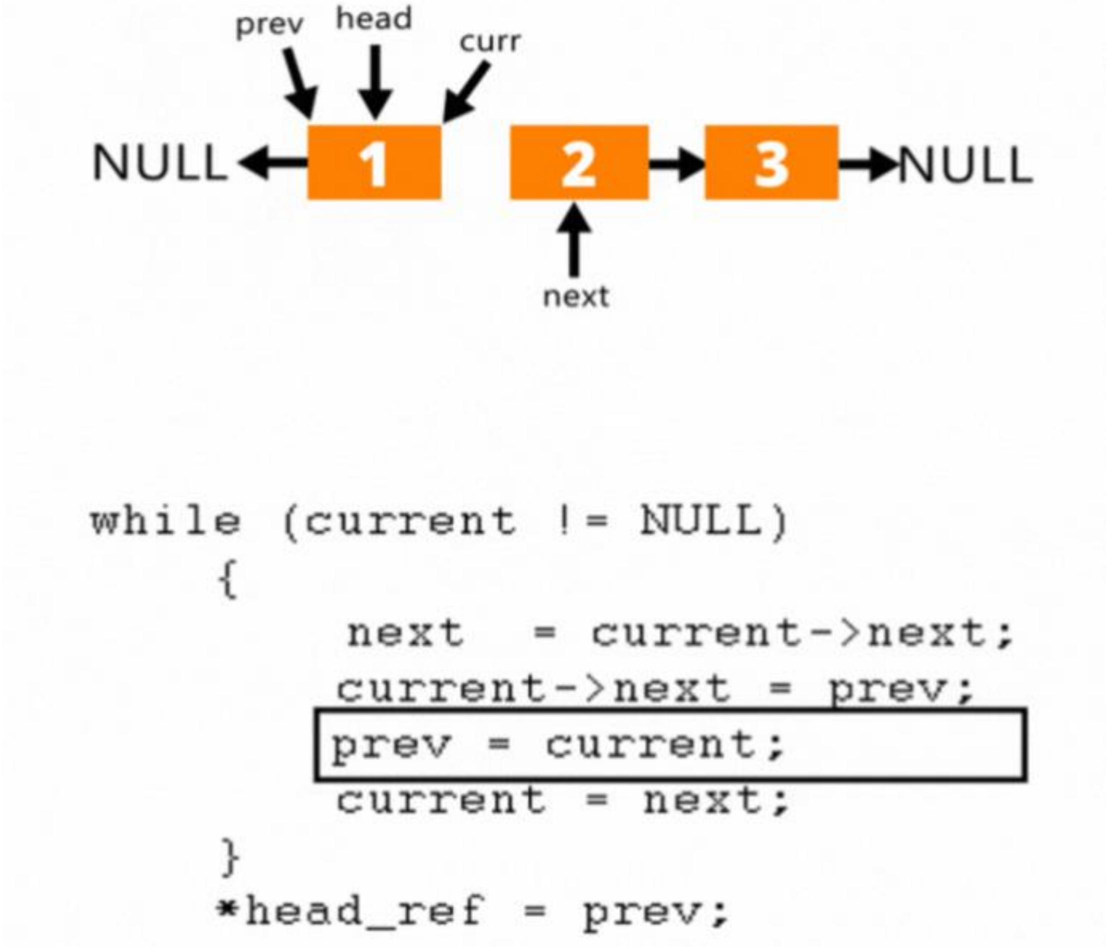
```
while (current != NULL)
     {
          next  = current->next;
          current->next = prev;
          prev = current;
          current = next;
     }
     *head_ref = prev;
```

2. Basic Operation: ***Reverse***



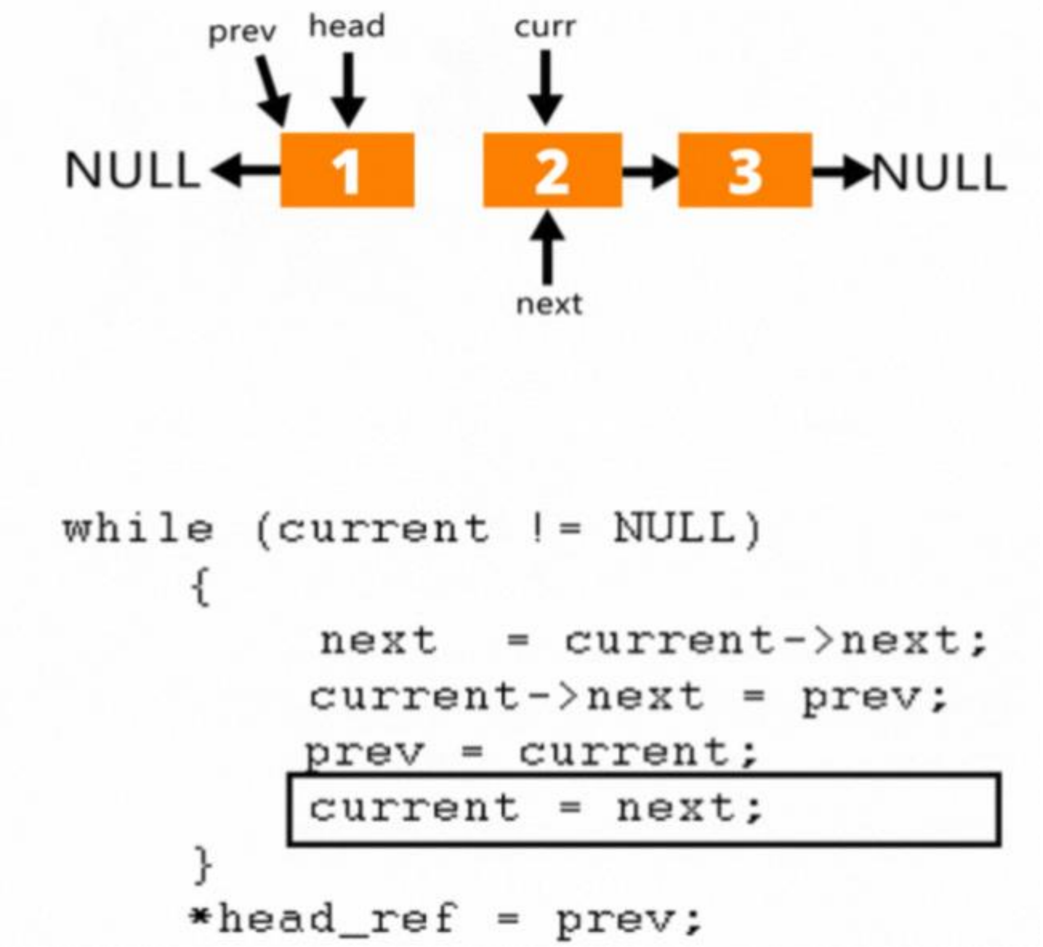
```
while (current != NULL)
    {
        next   = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    *head_ref = prev;
```

2. Basic Operation: *Reverse*



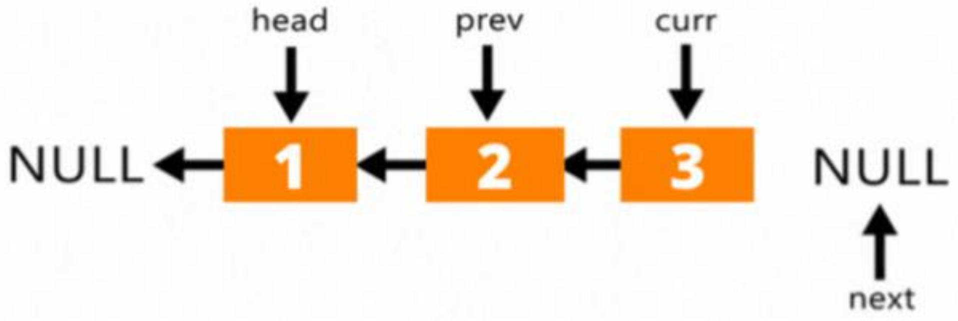
```
while (current != NULL)
    {
        next  = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    *head_ref = prev;
```

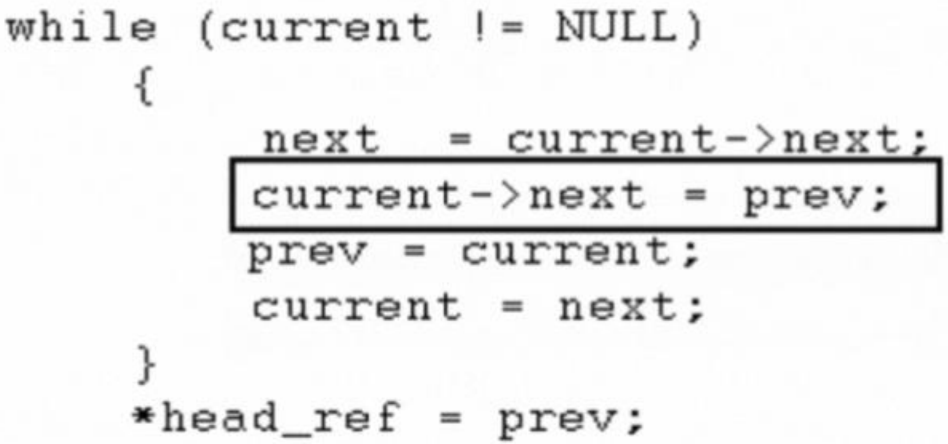
2. Basic Operation: *Reverse*



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

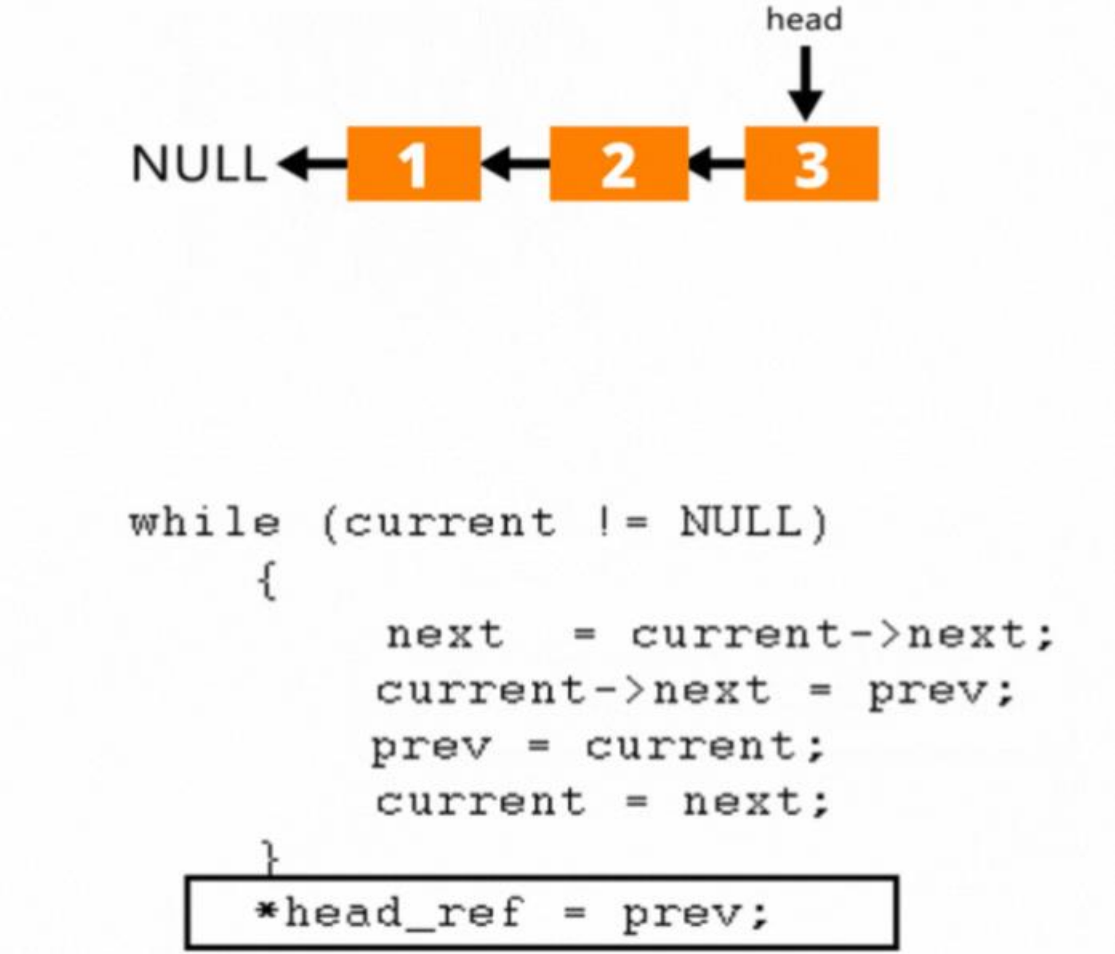
2. Basic Operation: ***Reverse***





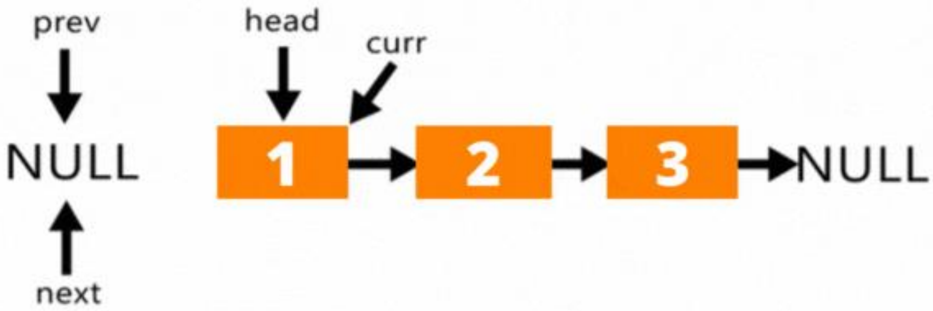
```
while (current != NULL)
    {
        next  = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    *head_ref = prev;
```

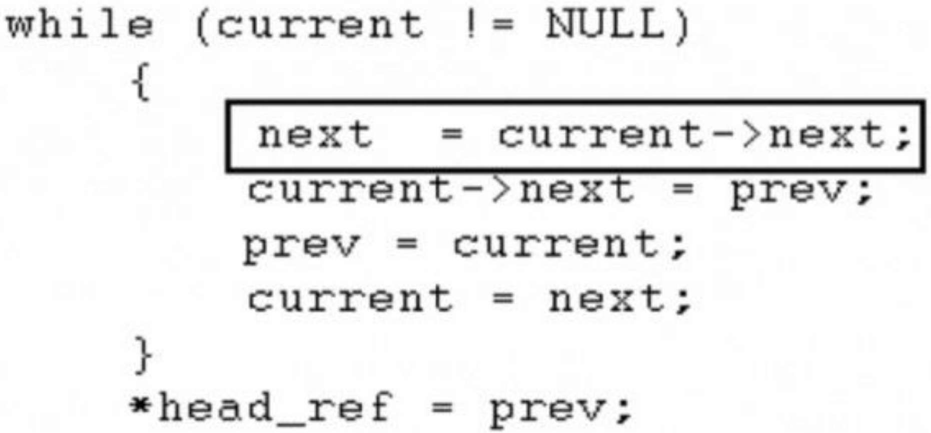
2. Basic Operation: ***Reverse***



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

2. Basic Operation: ***Reverse***

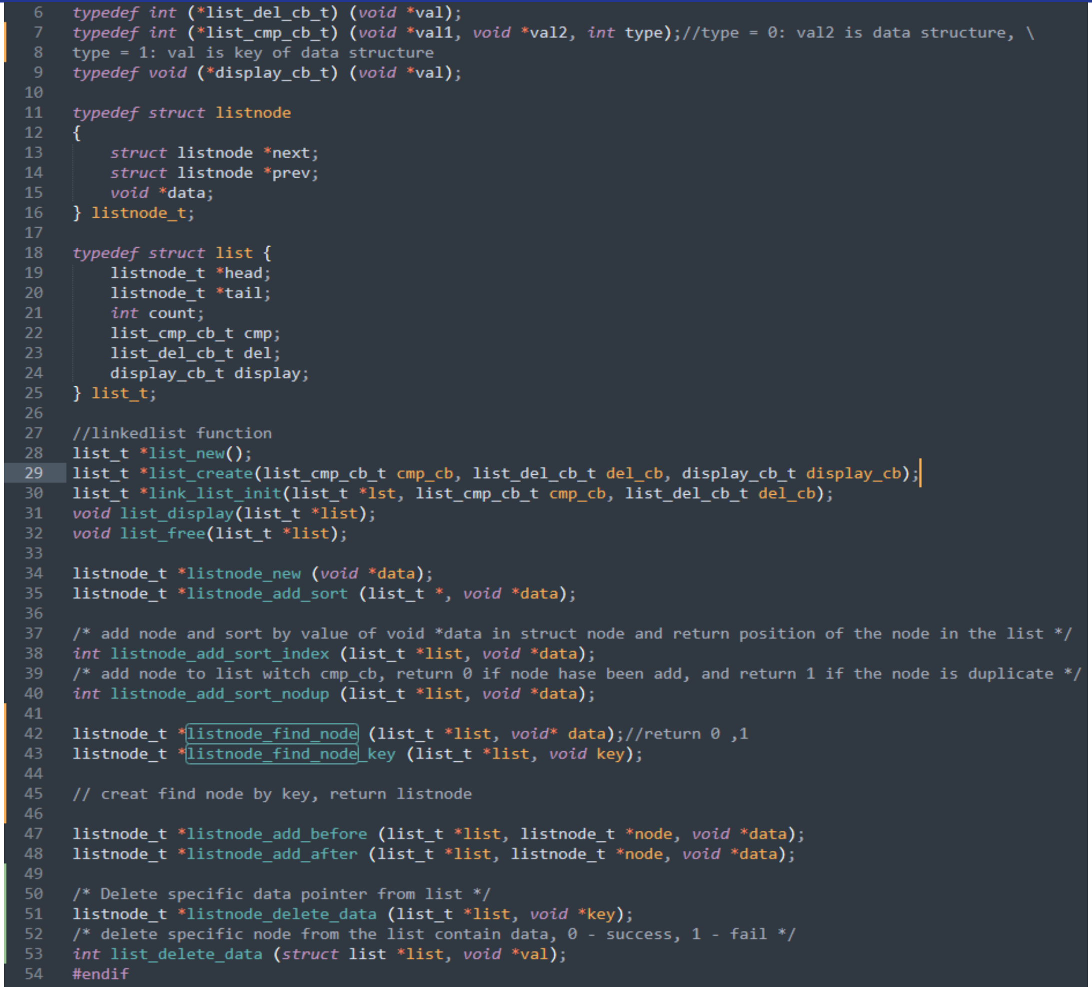




```
while (current != NULL)
    {
        next  = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    *head_ref = prev;
```

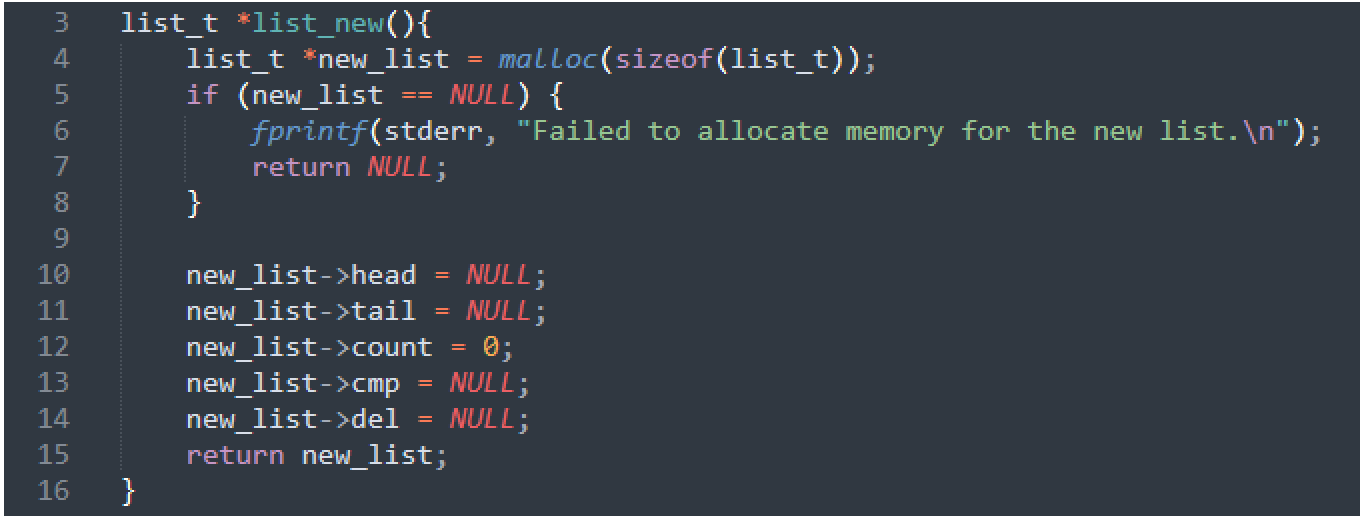
03
IMPLEMENT DOUBLY LINKED LIST

3. Implement Doubly LinkedList: Header file

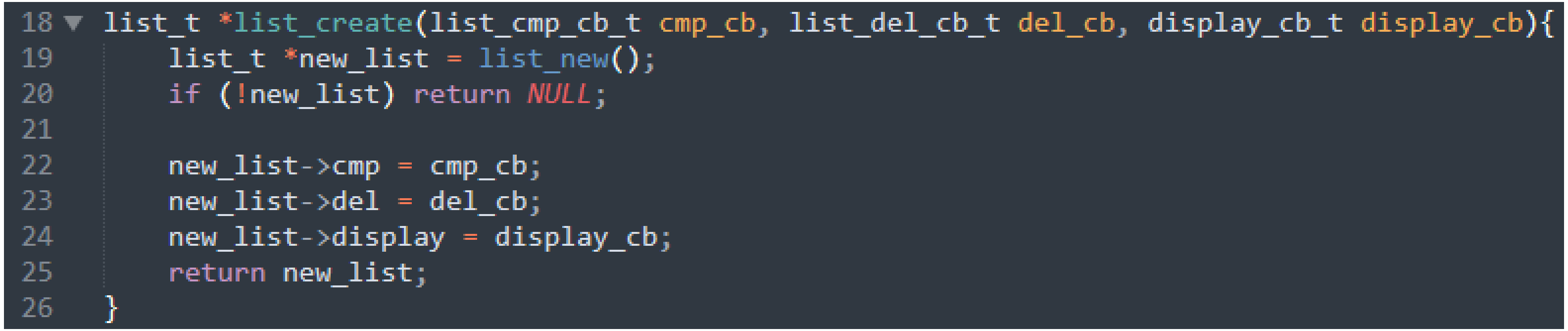


```c
typedef int (*list_del_cb_t) (void *val);
typedef int (*list_cmp_cb_t) (void *val1, void *val2, int type);//type = 0: val2 is data structure, \
type = 1: val is key of data structure
typedef void (*display_cb_t) (void *val);

typedef struct listnode
{
    struct listnode *next;
    struct listnode *prev;
    void *data;
} listnode_t;

typedef struct list {
    listnode_t *head;
    listnode_t *tail;
    int count;
    list_cmp_cb_t cmp;
    list_del_cb_t del;
    display_cb_t display;
} list_t;

//linkedlist function
list_t *list_new();
list_t *list_create(list_cmp_cb_t cmp_cb, list_del_cb_t del_cb, display_cb_t display_cb);
list_t *link_list_init(list_t *lst, list_cmp_cb_t cmp_cb, list_del_cb_t del_cb);
void list_display(list_t *list);
void list_free(list_t *list);

listnode_t *listnode_new (void *data);
listnode_t *listnode_add_sort (list_t *, void *data);

/* add node and sort by value of void *data in struct node and return position of the node in the list */
int listnode_add_sort_index (list_t *list, void *data);
/* add node to list witch cmp_cb, return 0 if node hase been add, and return 1 if the node is duplicate */
int listnode_add_sort_nodup (list_t *list, void *data);

listnode_t *listnode_find_node (list_t *list, void* data);//return 0 ,1
listnode_t *listnode_find_node_key (list_t *list, void key);

// creat find node by key, return listnode

listnode_t *listnode_add_before (list_t *list, listnode_t *node, void *data);
listnode_t *listnode_add_after (list_t *list, listnode_t *node, void *data);

/* Delete specific data pointer from list */
listnode_t *listnode_delete_data (list_t *list, void *key);
/* delete specific node from the list contain data, 0 - success, 1 - fail */
int list_delete_data (struct list *list, void *val);
#endif
```

3. Implement Doubly LinkedList: Source file

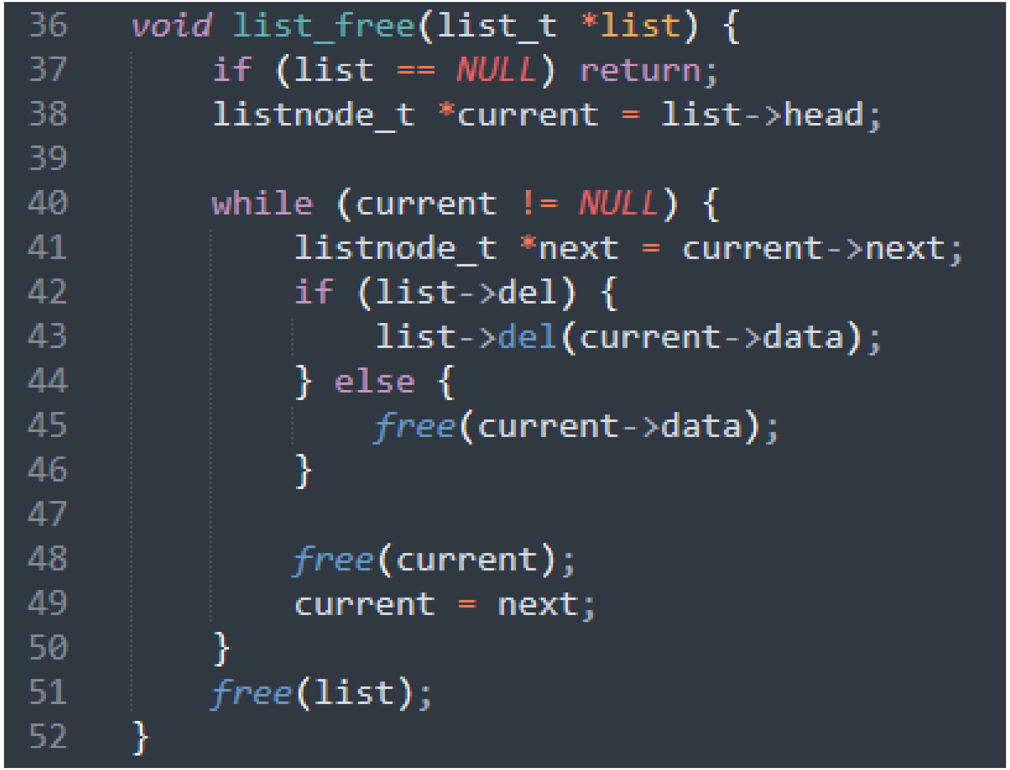


```c
list_t *list_new(){
    list_t *new_list = malloc(sizeof(list_t));
    if (new_list == NULL) {
        fprintf(stderr, "Failed to allocate memory for the new list.\n");
        return NULL;
    }

    new_list->head = NULL;
    new_list->tail = NULL;
    new_list->count = 0;
    new_list->cmp = NULL;
    new_list->del = NULL;
    return new_list;
}
```



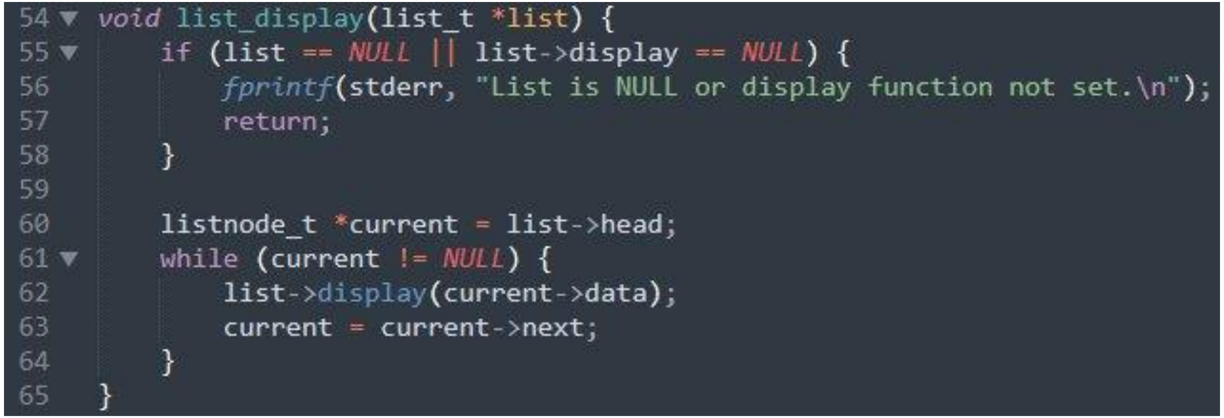
```c
list_t *list_create(list_cmp_cb_t cmp_cb, list_del_cb_t del_cb, display_cb_t display_cb){
    list_t *new_list = list_new();
    if (!new_list) return NULL;

    new_list->cmp = cmp_cb;
    new_list->del = del_cb;
    new_list->display = display_cb;
    return new_list;
}
```

3. Implement Doubly LinkedList: Source file

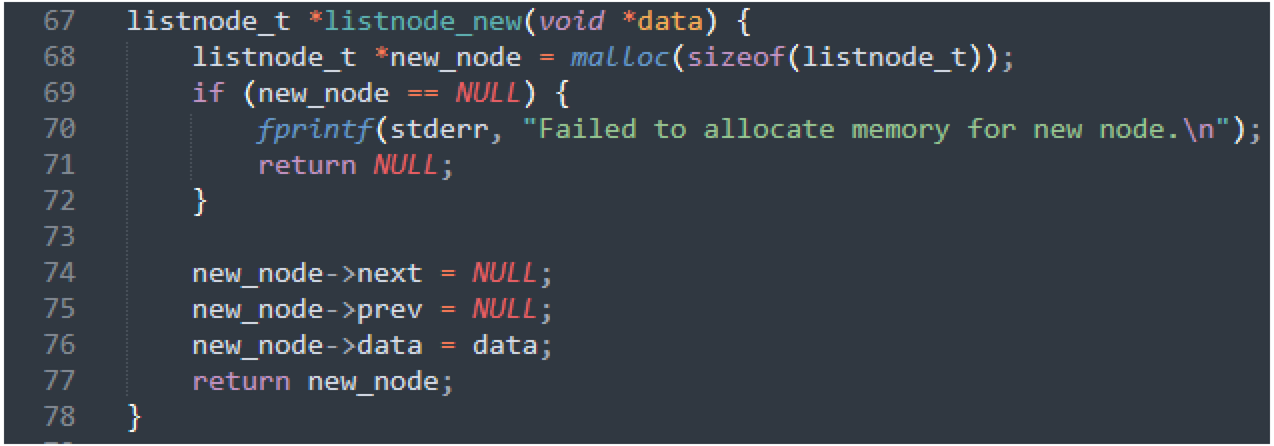


```c
void list_free(list_t *list) {
    if (list == NULL) return;
    listnode_t *current = list->head;

    while (current != NULL) {
        listnode_t *next = current->next;
        if (list->del) {
            list->del(current->data);
        } else {
            free(current->data);
        }

        free(current);
        current = next;
    }
    free(list);
}
```

3. Implement Doubly LinkedList: Source file

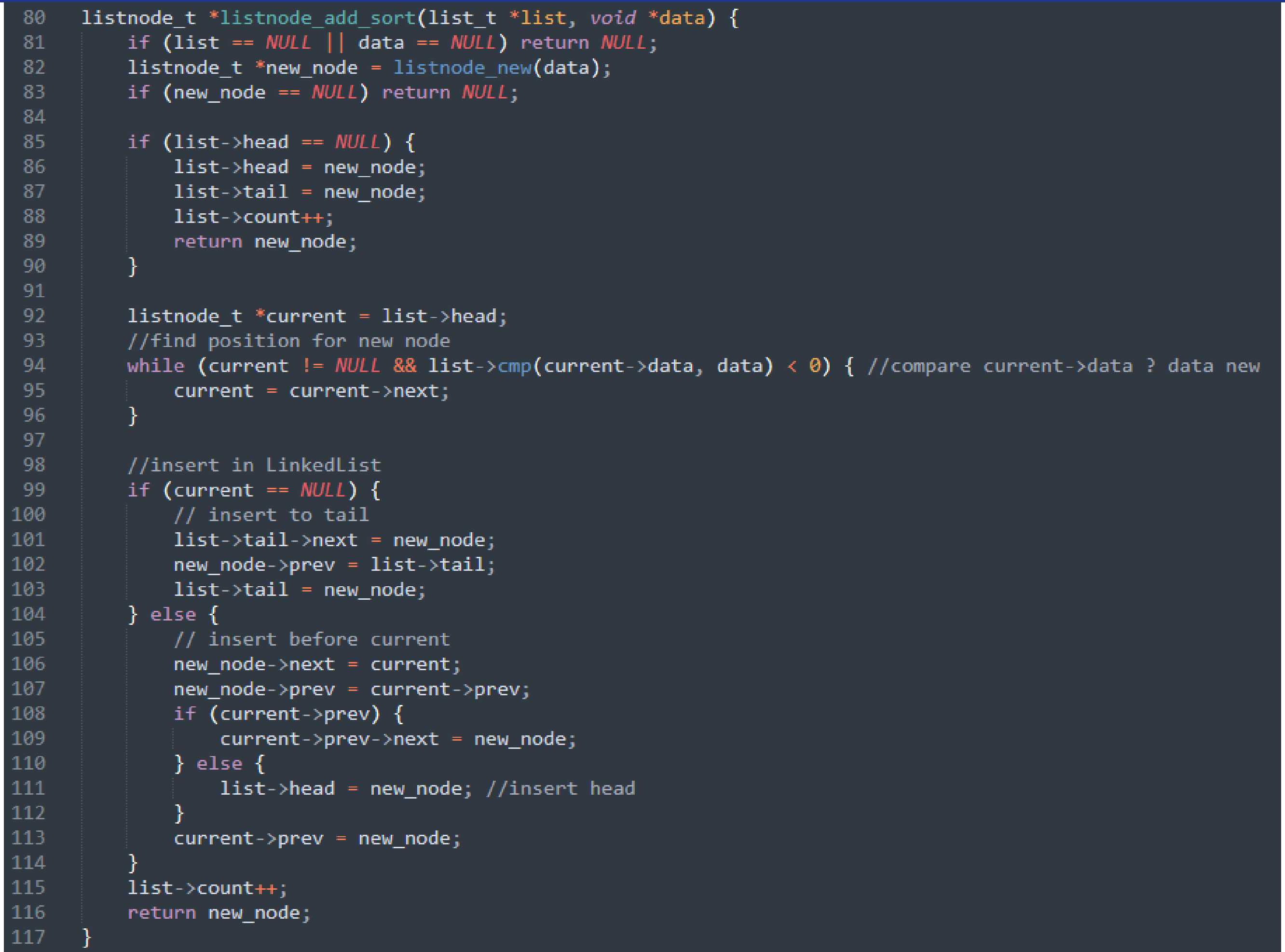


```c
void list_display(list_t *list) {
    if (list == NULL || list->display == NULL) {
        fprintf(stderr, "List is NULL or display function not set.\n");
        return;
    }

    listnode_t *current = list->head;
    while (current != NULL) {
        list->display(current->data);
        current = current->next;
    }
}
```



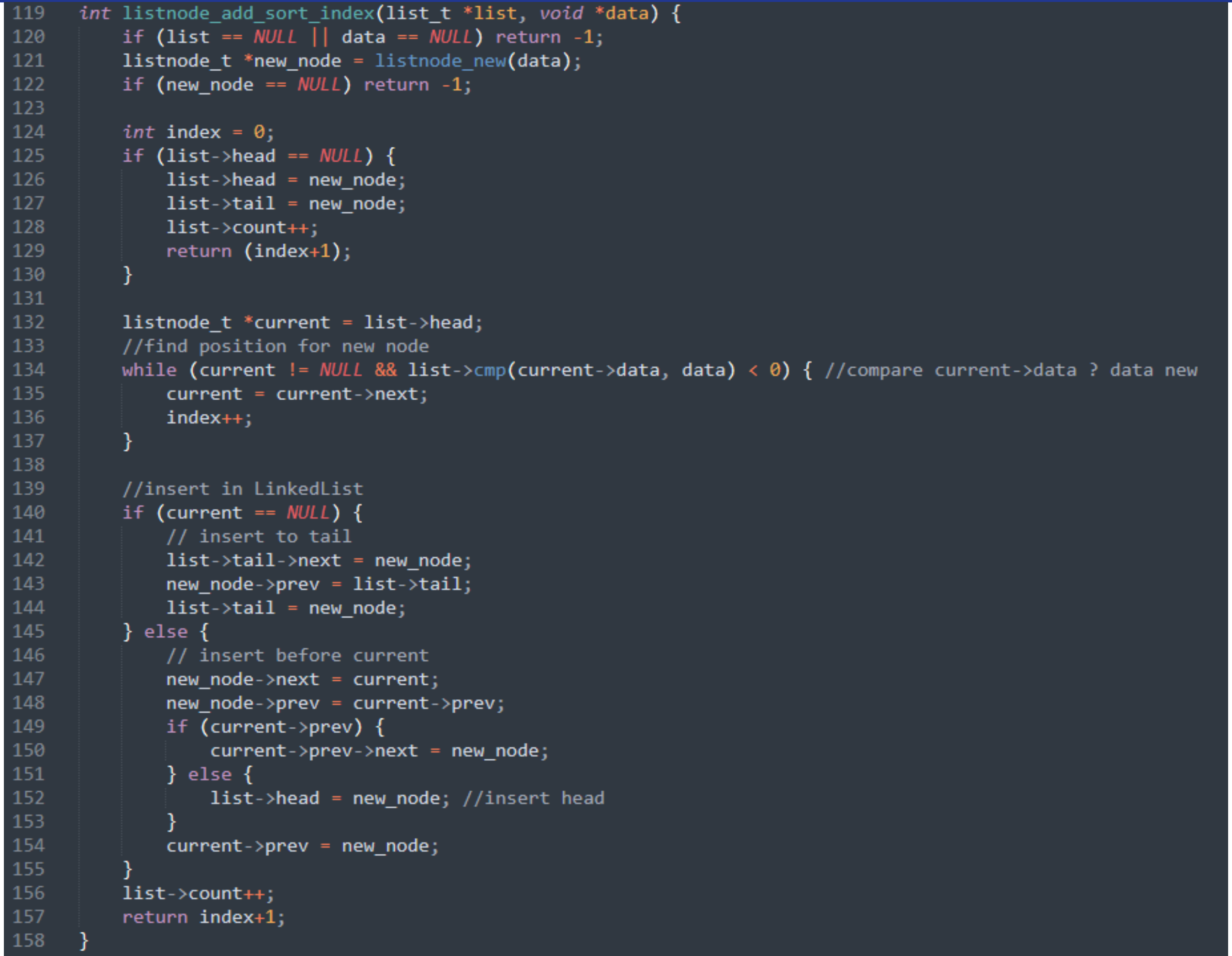
```c
listnode_t *listnode_new(void *data) {
    listnode_t *new_node = malloc(sizeof(listnode_t));
    if (new_node == NULL) {
        fprintf(stderr, "Failed to allocate memory for new node.\n");
        return NULL;
    }

    new_node->next = NULL;
    new_node->prev = NULL;
    new_node->data = data;
    return new_node;
}
```

3. Implement Doubly LinkedList: Source file



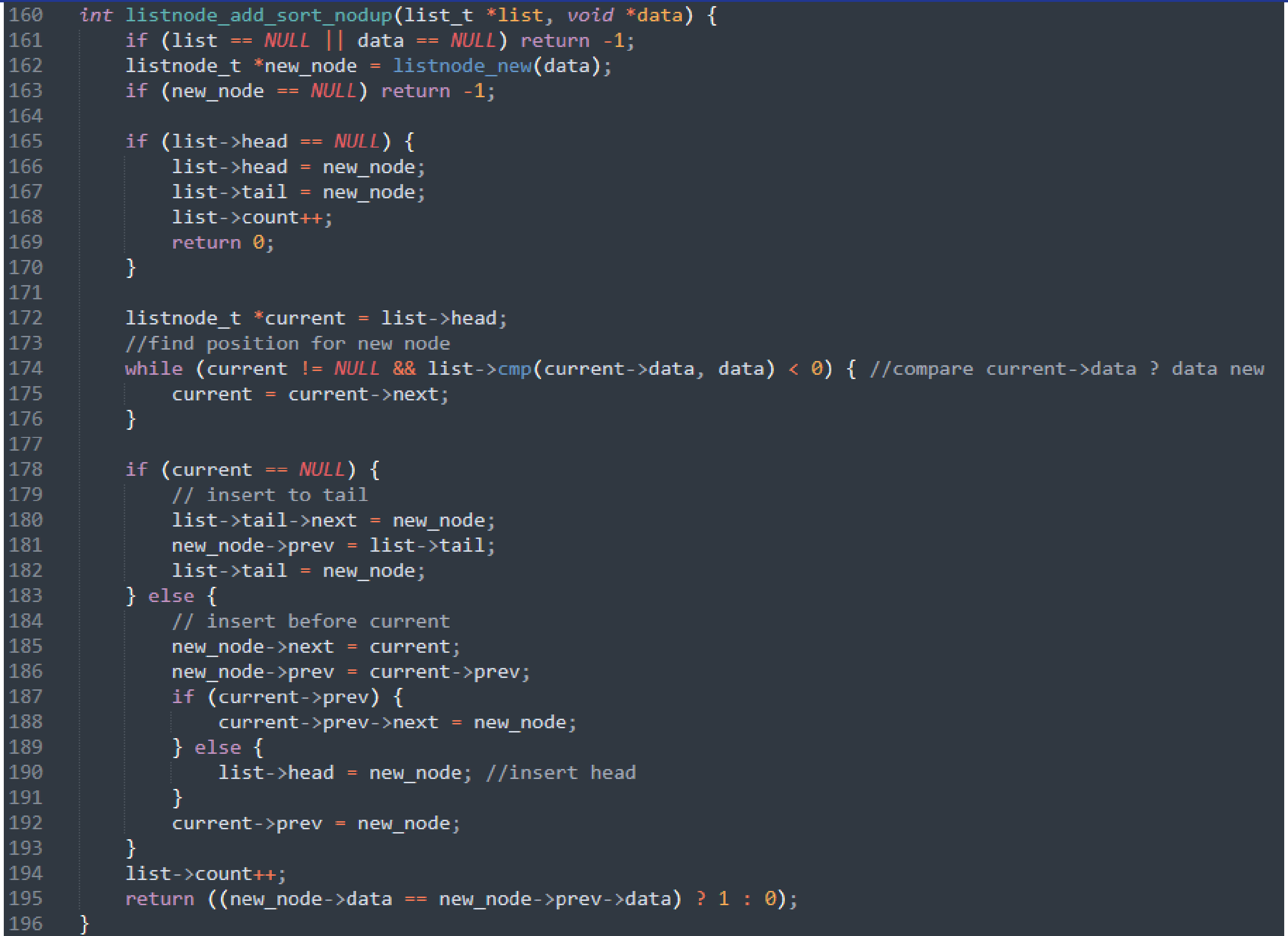
```c
listnode_t *listnode_add_sort(list_t *list, void *data) {
    if (list == NULL || data == NULL) return NULL;
    listnode_t *new_node = listnode_new(data);
    if (new_node == NULL) return NULL;

    if (list->head == NULL) {
        list->head = new_node;
        list->tail = new_node;
        list->count++;
        return new_node;
    }

    listnode_t *current = list->head;
    //find position for new node
    while (current != NULL && list->cmp(current->data, data) < 0) { //compare current->data ? data new
        current = current->next;
    }

    //insert in LinkedList
    if (current == NULL) {
        // insert to tail
        list->tail->next = new_node;
        new_node->prev = list->tail;
        list->tail = new_node;
    } else {
        // insert before current
        new_node->next = current;
        new_node->prev = current->prev;
        if (current->prev) {
            current->prev->next = new_node;
        } else {
            list->head = new_node; //insert head
        }
        current->prev = new_node;
    }
    list->count++;
    return new_node;
}
```

3. Implement Doubly LinkedList: Source file



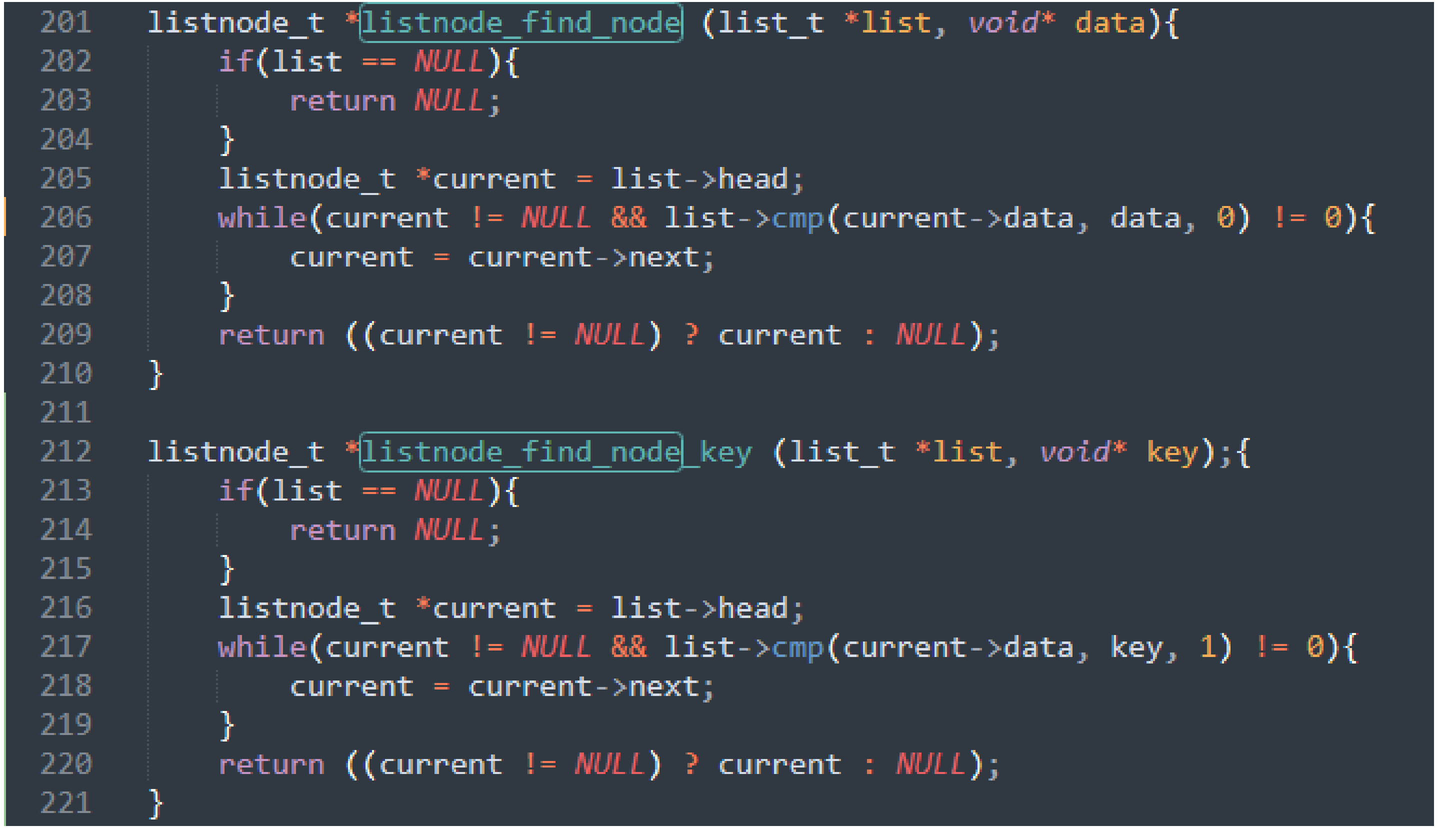
```c
int listnode_add_sort_index(list_t *list, void *data) {
    if (list == NULL || data == NULL) return -1;
    listnode_t *new_node = listnode_new(data);
    if (new_node == NULL) return -1;

    int index = 0;
    if (list->head == NULL) {
        list->head = new_node;
        list->tail = new_node;
        list->count++;
        return (index+1);
    }

    listnode_t *current = list->head;
    //find position for new node
    while (current != NULL && list->cmp(current->data, data) < 0) { //compare current->data ? data new
        current = current->next;
        index++;
    }

    //insert in LinkedList
    if (current == NULL) {
        // insert to tail
        list->tail->next = new_node;
        new_node->prev = list->tail;
        list->tail = new_node;
    } else {
        // insert before current
        new_node->next = current;
        new_node->prev = current->prev;
        if (current->prev) {
            current->prev->next = new_node;
        } else {
            list->head = new_node; //insert head
        }
        current->prev = new_node;
    }
    list->count++;
    return index+1;
}
```

3. Implement Doubly LinkedList: Source file



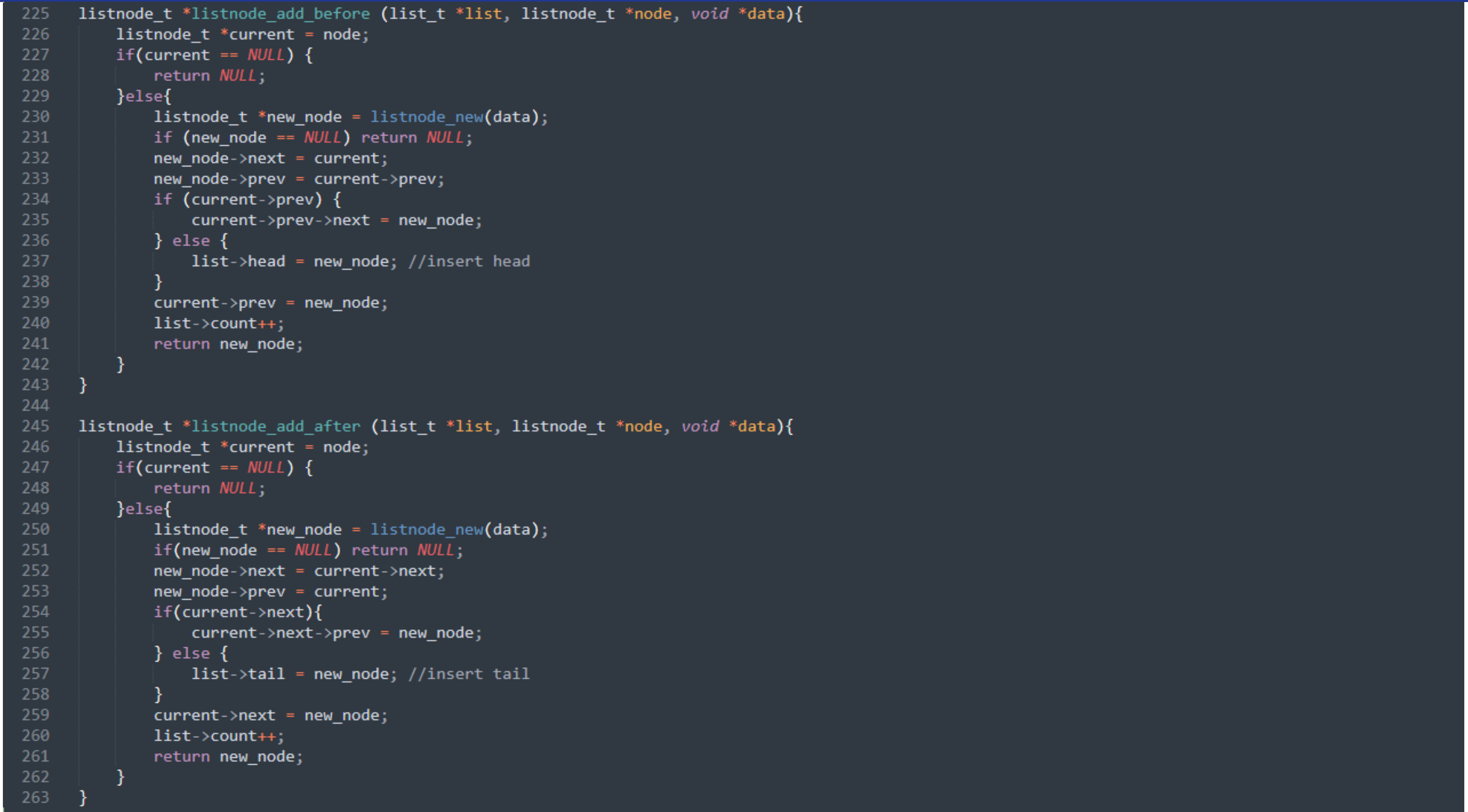
```c
int listnode_add_sort_nodup(list_t *list, void *data) {
    if (list == NULL || data == NULL) return -1;
    listnode_t *new_node = listnode_new(data);
    if (new_node == NULL) return -1;

    if (list->head == NULL) {
        list->head = new_node;
        list->tail = new_node;
        list->count++;
        return 0;
    }

    listnode_t *current = list->head;
    //find position for new node
    while (current != NULL && list->cmp(current->data, data) < 0) { //compare current->data ? data new
        current = current->next;
    }

    if (current == NULL) {
        // insert to tail
        list->tail->next = new_node;
        new_node->prev = list->tail;
        list->tail = new_node;
    } else {
        // insert before current
        new_node->next = current;
        new_node->prev = current->prev;
        if (current->prev) {
            current->prev->next = new_node;
        } else {
            list->head = new_node; //insert head
        }
        current->prev = new_node;
    }
    list->count++;
    return ((new_node->data == new_node->prev->data) ? 1 : 0);
}
```

3. Implement Doubly LinkedList: Source file



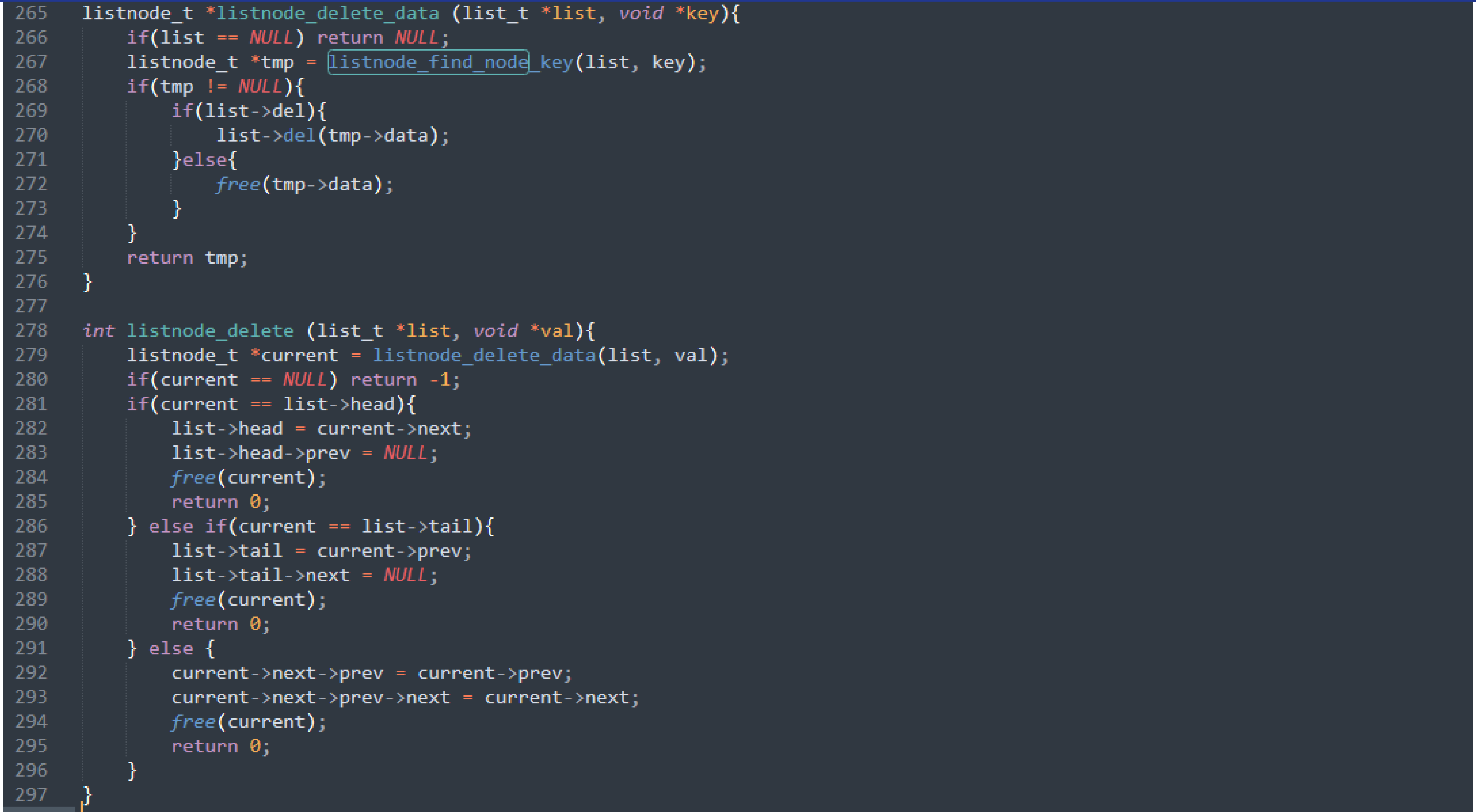
```c
listnode_t *listnode_find_node (list_t *list, void* data){
    if(list == NULL){
        return NULL;
    }
    listnode_t *current = list->head;
    while(current != NULL && list->cmp(current->data, data, 0) != 0){
        current = current->next;
    }
    return ((current != NULL) ? current : NULL);
}

listnode_t *listnode_find_node_key (list_t *list, void* key);{
    if(list == NULL){
        return NULL;
    }
    listnode_t *current = list->head;
    while(current != NULL && list->cmp(current->data, key, 1) != 0){
        current = current->next;
    }
    return ((current != NULL) ? current : NULL);
}
```

3. Implement Doubly LinkedList: Source file



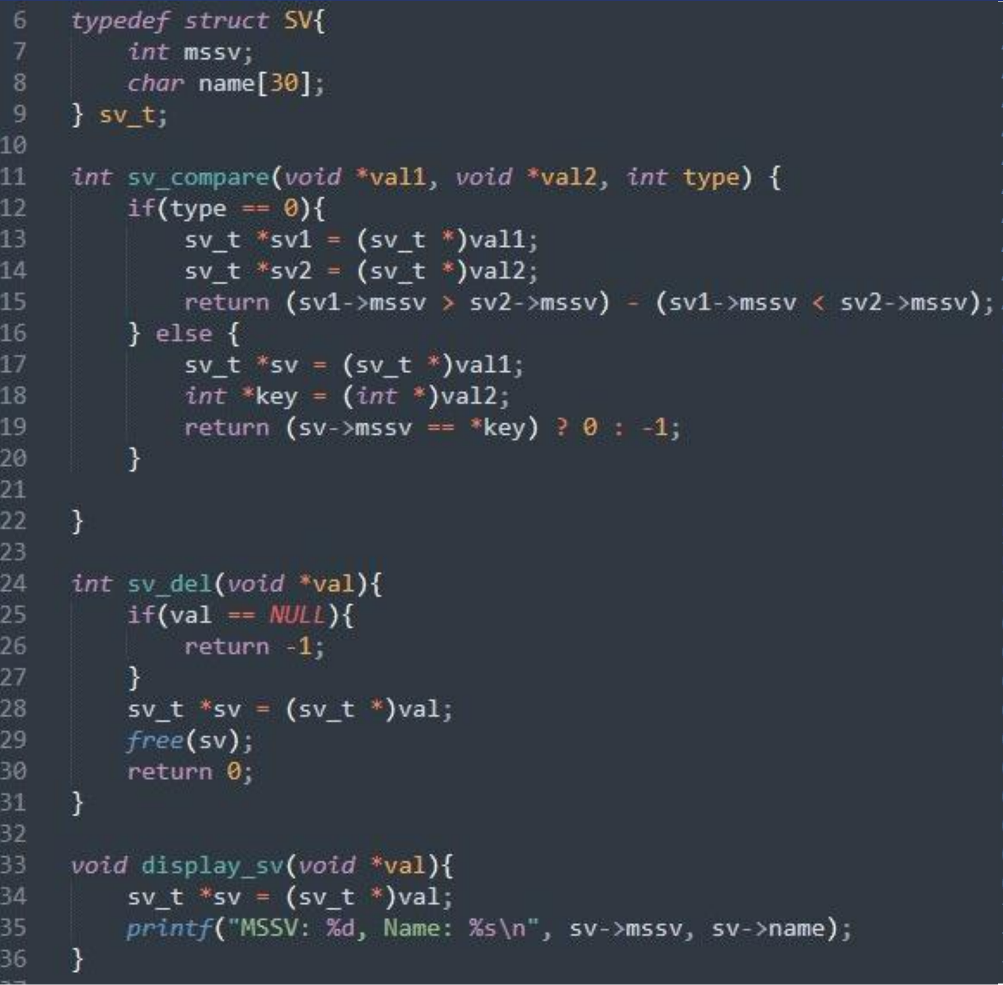
```c
listnode_t *listnode_add_before (list_t *list, listnode_t *node, void *data){
    listnode_t *current = node;
    if(current == NULL) {
        return NULL;
    }else{
        listnode_t *new_node = listnode_new(data);
        if (new_node == NULL) return NULL;
        new_node->next = current;
        new_node->prev = current->prev;
        if (current->prev) {
            current->prev->next = new_node;
        } else {
            list->head = new_node; //insert head
        }
        current->prev = new_node;
        list->count++;
        return new_node;
    }
}

listnode_t *listnode_add_after (list_t *list, listnode_t *node, void *data){
    listnode_t *current = node;
    if(current == NULL) {
        return NULL;
    }else{
        listnode_t *new_node = listnode_new(data);
        if(new_node == NULL) return NULL;
        new_node->next = current->next;
        new_node->prev = current;
        if(current->next){
            current->next->prev = new_node;
        } else {
            list->tail = new_node; //insert tail
        }
        current->next = new_node;
        list->count++;
        return new_node;
    }
}
```

3. Implement Doubly LinkedList: Source file

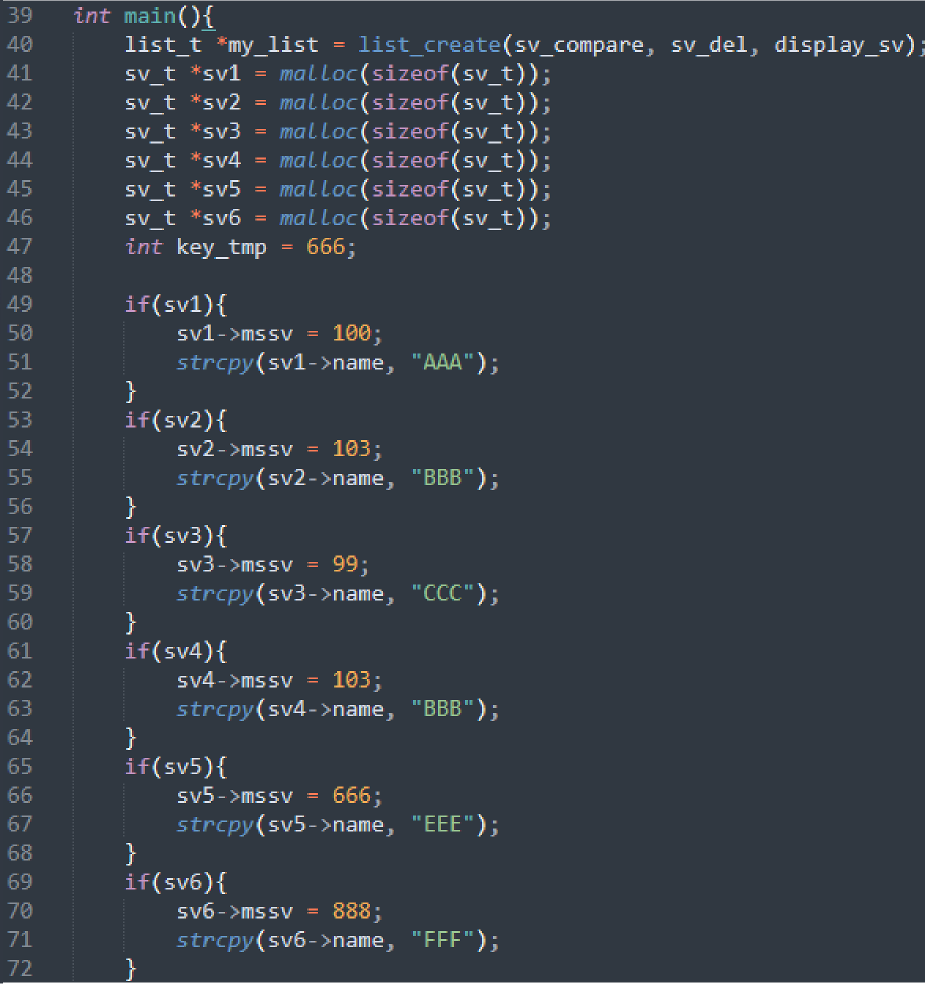


```c
listnode_t *listnode_delete_data (list_t *list, void *key){
    if(list == NULL) return NULL;
    listnode_t *tmp = listnode_find_node_key(list, key);
    if(tmp != NULL){
        if(list->del){
            list->del(tmp->data);
        }else{
            free(tmp->data);
        }
    }
    return tmp;
}

int listnode_delete (list_t *list, void *val){
    listnode_t *current = listnode_delete_data(list, val);
    if(current == NULL) return -1;
    if(current == list->head){
        list->head = current->next;
        list->head->prev = NULL;
        free(current);
        return 0;
    } else if(current == list->tail){
        list->tail = current->prev;
        list->tail->next = NULL;
        free(current);
        return 0;
    } else {
        current->next->prev = current->prev;
        current->next->prev->next = current->next;
        free(current);
        return 0;
    }
}
```

3. Implement Doubly LinkedList: main.c

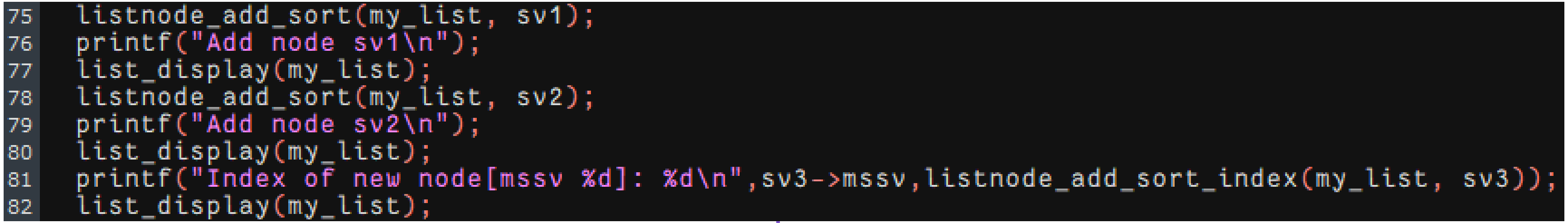


```c
typedef struct SV{
    int mssv;
    char name[30];
} sv_t;

int sv_compare(void *val1, void *val2, int type) {
    if(type == 0){
        sv_t *sv1 = (sv_t *)val1;
        sv_t *sv2 = (sv_t *)val2;
        return (sv1->mssv > sv2->mssv) - (sv1->mssv < sv2->mssv);
    } else {
        sv_t *sv = (sv_t *)val1;
        int *key = (int *)val2;
        return (sv->mssv == *key) ? 0 : -1;
    }

}

int sv_del(void *val){
    if(val == NULL){
        return -1;
    }
    sv_t *sv = (sv_t *)val;
    free(sv);
    return 0;
}

void display_sv(void *val){
    sv_t *sv = (sv_t *)val;
    printf("MSSV: %d, Name: %s\n", sv->mssv, sv->name);
}
```

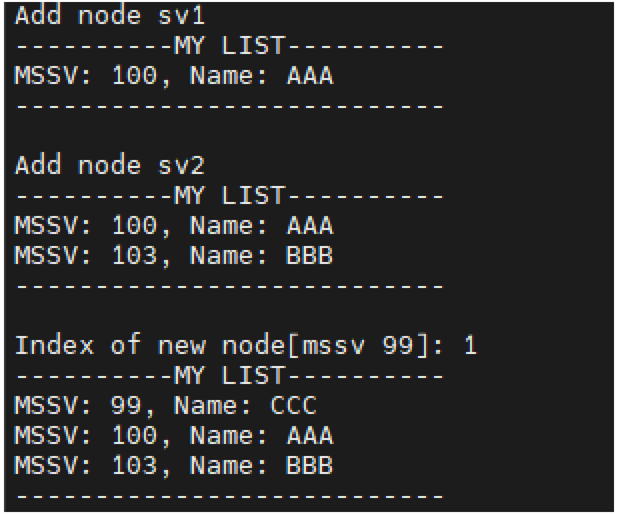


```c
int main(){
    list_t *my_list = list_create(sv_compare, sv_del, display_sv);
    sv_t *sv1 = malloc(sizeof(sv_t));
    sv_t *sv2 = malloc(sizeof(sv_t));
    sv_t *sv3 = malloc(sizeof(sv_t));
    sv_t *sv4 = malloc(sizeof(sv_t));
    sv_t *sv5 = malloc(sizeof(sv_t));
    sv_t *sv6 = malloc(sizeof(sv_t));
    int key_tmp = 666;

    if(sv1){
        sv1->mssv = 100;
        strcpy(sv1->name, "AAA");
    }
    if(sv2){
        sv2->mssv = 103;
        strcpy(sv2->name, "BBB");
    }
    if(sv3){
        sv3->mssv = 99;
        strcpy(sv3->name, "CCC");
    }
    if(sv4){
        sv4->mssv = 103;
        strcpy(sv4->name, "BBB");
    }
    if(sv5){
        sv5->mssv = 666;
        strcpy(sv5->name, "EEE");
    }
    if(sv6){
        sv6->mssv = 888;
        strcpy(sv6->name, "FFF");
    }
```

3. Implement Doubly LinkedList: main.c

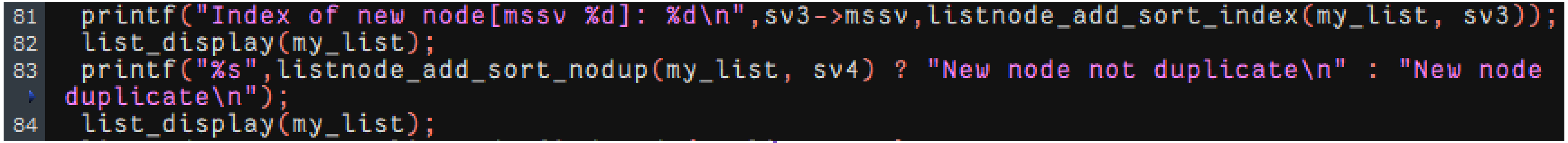


```
    listnode_add_sort(my_list, sv1);
    printf("Add node sv1\n");
    list_display(my_list);
    listnode_add_sort(my_list, sv2);
    printf("Add node sv2\n");
    list_display(my_list);
    printf("Index of new node[mssv %d]: %d\n",sv3->mssv,listnode_add_sort_index(my_list, sv3));
    list_display(my_list);
```

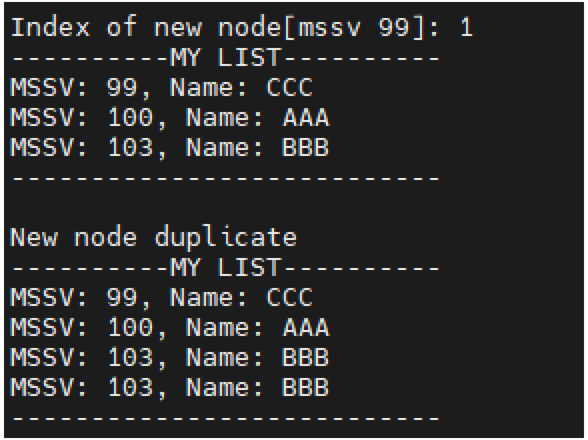


```
Add node sv1
----------MY LIST----------
MSSV: 100, Name: AAA
---------------------------

Add node sv2
----------MY LIST----------
MSSV: 100, Name: AAA
MSSV: 103, Name: BBB
---------------------------

Index of new node[mssv 99]: 1
----------MY LIST----------
MSSV: 99, Name: CCC
MSSV: 100, Name: AAA
MSSV: 103, Name: BBB
---------------------------
```

3. Implement Doubly LinkedList: main.c

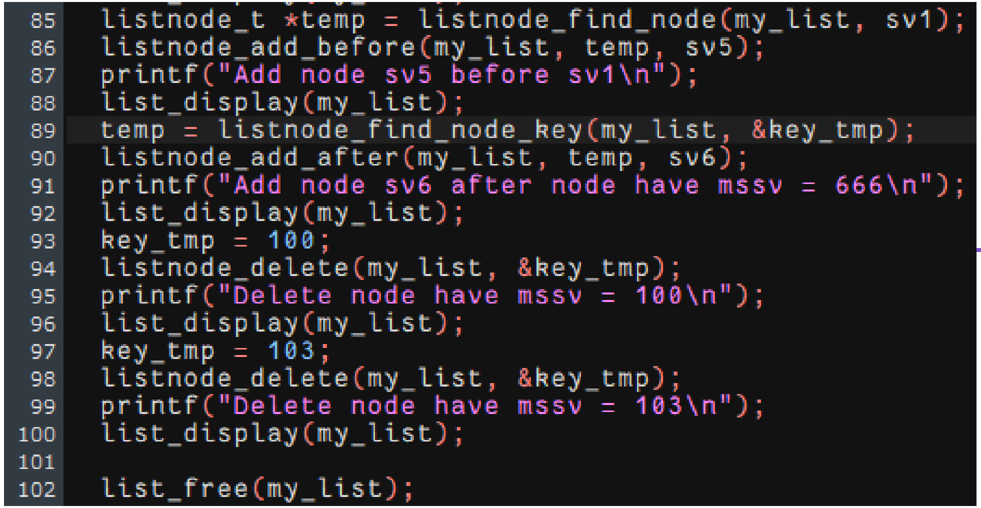


```c
    printf("Index of new node[mssv %d]: %d\n",sv3->mssv,listnode_add_sort_index(my_list, sv3));
    list_display(my_list);
    printf("%s",listnode_add_sort_nodup(my_list, sv4) ? "New node not duplicate\n" : "New node duplicate\n");
    list_display(my_list);
```

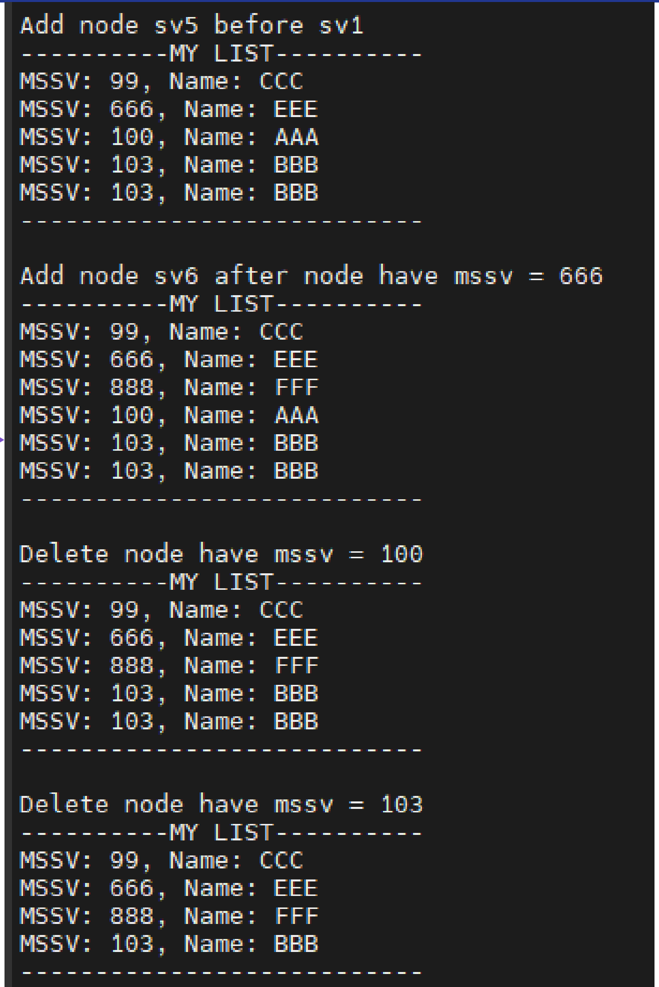


```
Index of new node[mssv 99]: 1
----------MY LIST----------
MSSV: 99, Name: CCC
MSSV: 100, Name: AAA
MSSV: 103, Name: BBB
---------------------------

New node duplicate
----------MY LIST----------
MSSV: 99, Name: CCC
MSSV: 100, Name: AAA
MSSV: 103, Name: BBB
MSSV: 103, Name: BBB
---------------------------
```

3. Implement Doubly LinkedList: main.c



```c
listnode_t *temp = listnode_find_node(my_list, sv1);
listnode_add_before(my_list, temp, sv5);
printf("Add node sv5 before sv1\n");
list_display(my_list);
temp = listnode_find_node_key(my_list, &key_tmp);
listnode_add_after(my_list, temp, sv6);
printf("Add node sv6 after node have mssv = 666\n");
list_display(my_list);
key_tmp = 100;
listnode_delete(my_list, &key_tmp);
printf("Delete node have mssv = 100\n");
list_display(my_list);
key_tmp = 103;
listnode_delete(my_list, &key_tmp);
printf("Delete node have mssv = 103\n");
list_display(my_list);

list_free(my_list);
```



```
Add node sv5 before sv1
----------MY LIST----------
MSSV: 99, Name: CCC
MSSV: 666, Name: EEE
MSSV: 100, Name: AAA
MSSV: 103, Name: BBB
MSSV: 103, Name: BBB
---------------------------

Add node sv6 after node have mssv = 666
----------MY LIST----------
MSSV: 99, Name: CCC
MSSV: 666, Name: EEE
MSSV: 888, Name: FFF
MSSV: 100, Name: AAA
MSSV: 103, Name: BBB
MSSV: 103, Name: BBB
---------------------------

Delete node have mssv = 100
----------MY LIST----------
MSSV: 99, Name: CCC
MSSV: 666, Name: EEE
MSSV: 888, Name: FFF
MSSV: 103, Name: BBB
MSSV: 103, Name: BBB
---------------------------

Delete node have mssv = 103
----------MY LIST----------
MSSV: 99, Name: CCC
MSSV: 666, Name: EEE
MSSV: 888, Name: FFF
MSSV: 103, Name: BBB
---------------------------
```

THANK FOR LISTENING