

Data Structure Linked List

Người trình bày: Phạm Hồng Thi

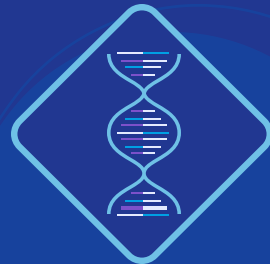




Table of contents

01

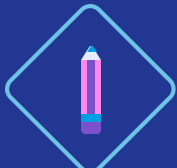
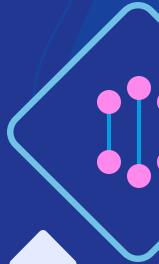
DEFINITION

02

BASIC OPERATION

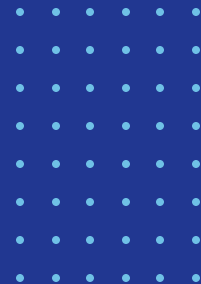
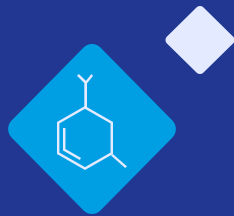
03

**IMPLEMENT DOUBLY
LINKED LIST**



01

DEFINITION



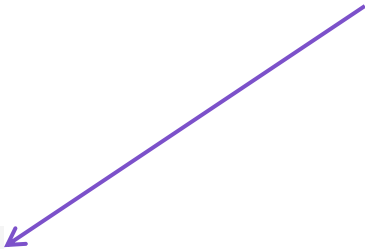
1. Linked List: Self referential structures

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

```
3 struct data{  
4     int i;  
5     char c;  
6     struct data *ptr;  
7 };
```

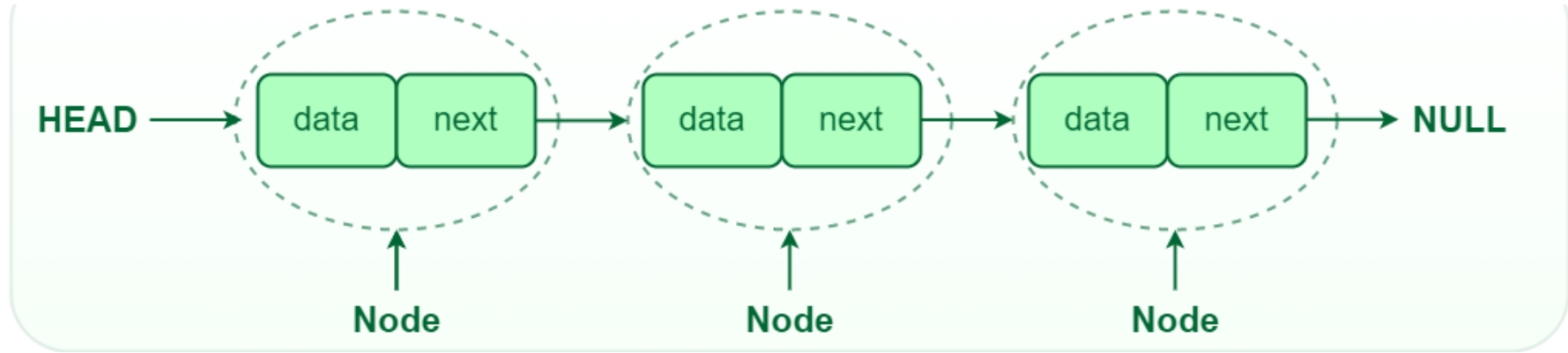
```
8 void main() {  
9     struct data var1;  
10    struct data var2;  
11  
12    var1.i = 65;  
13    var1.c = 'A';  
14    var1.ptr = NULL;  
15  
16    var2.i = 66;  
17    var2.c = 'B';  
18    var2.ptr = NULL;  
19  
20    var1.ptr = &var2;  
21    printf("OUT PUT: %d %c",var1.ptr->i, var1.ptr->c);
```

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

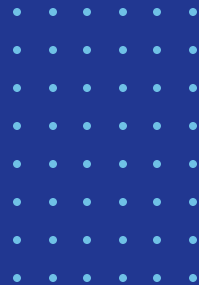
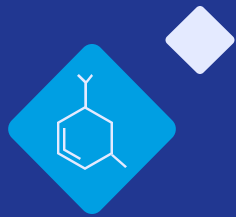


- Doubly linked list



- Circular Linked List





02

BASIC OPERATION

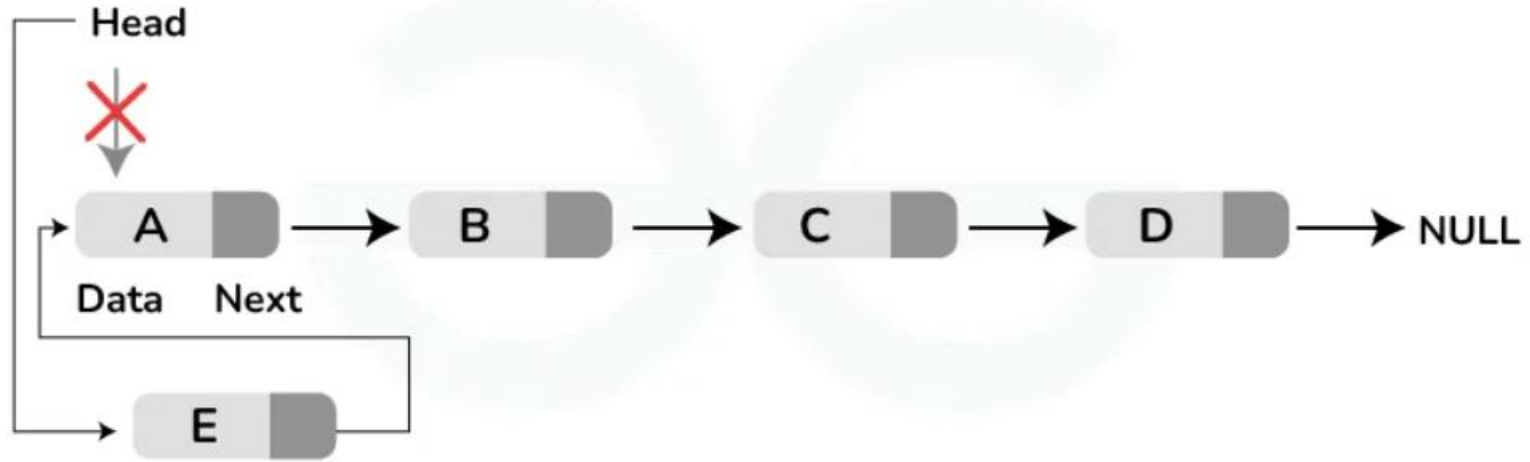
Single Linked List



2. Basic Operation

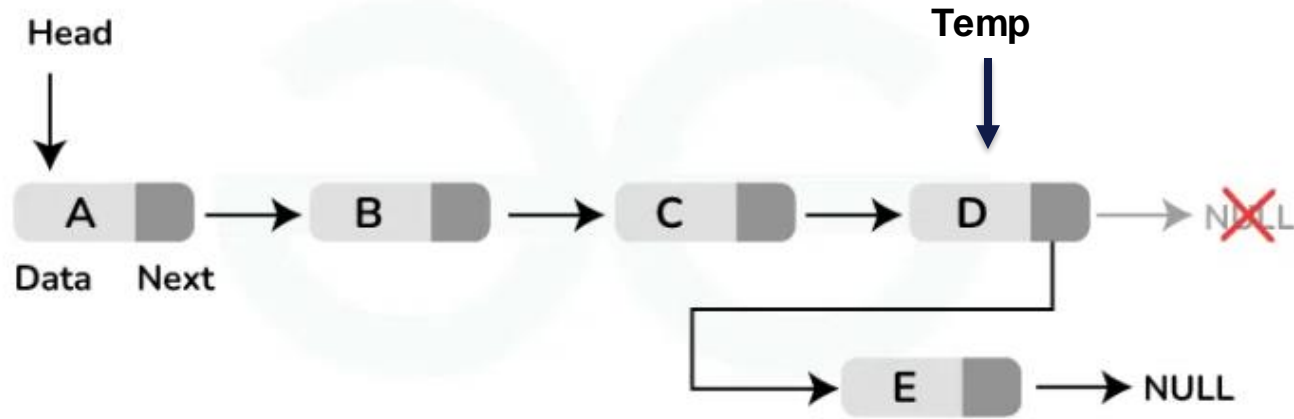
- *Insertion*
- *Deletion*
- *Search an element*
- *Reverse*

2. Basic Operation: *Insertion*



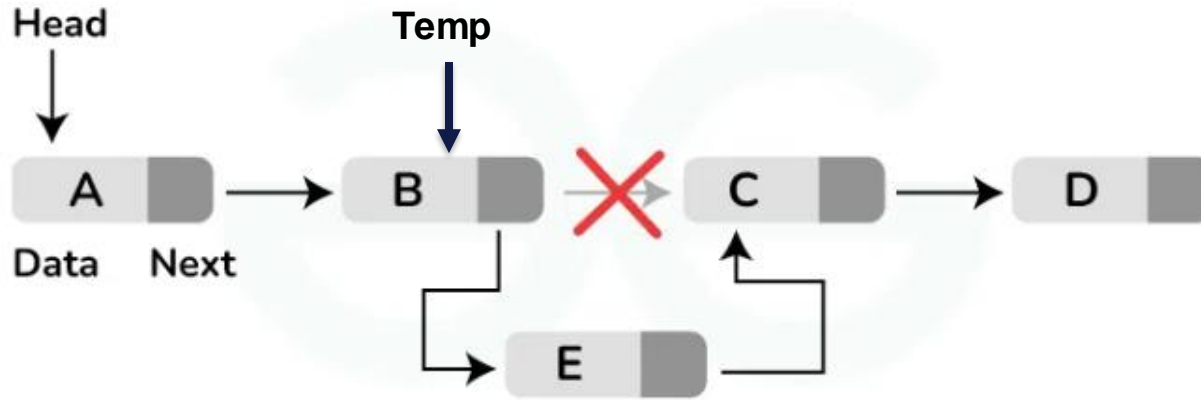
Insert in the front

2. Basic Operation: *Insertion*



Insert in the end

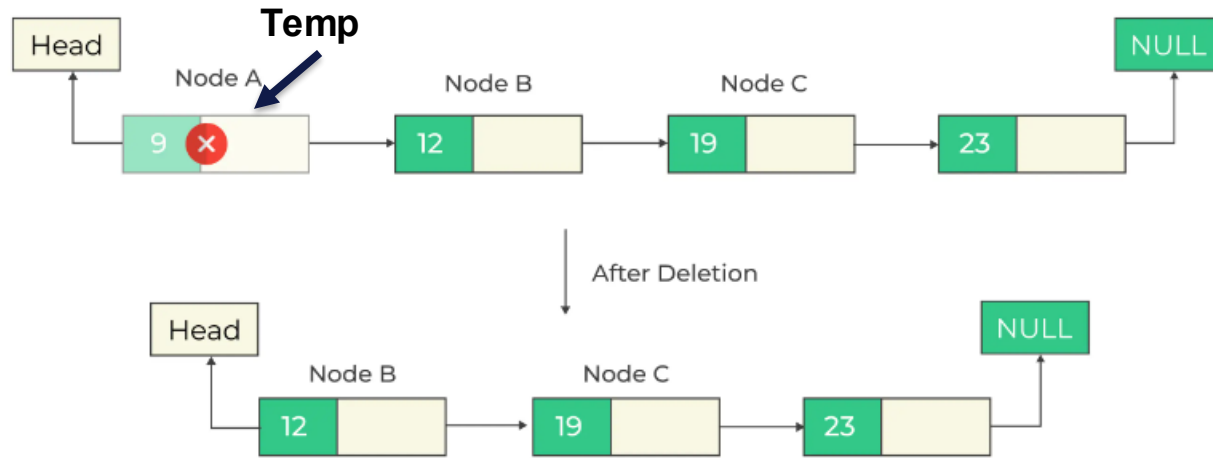
2. Basic Operation: *Insertion*



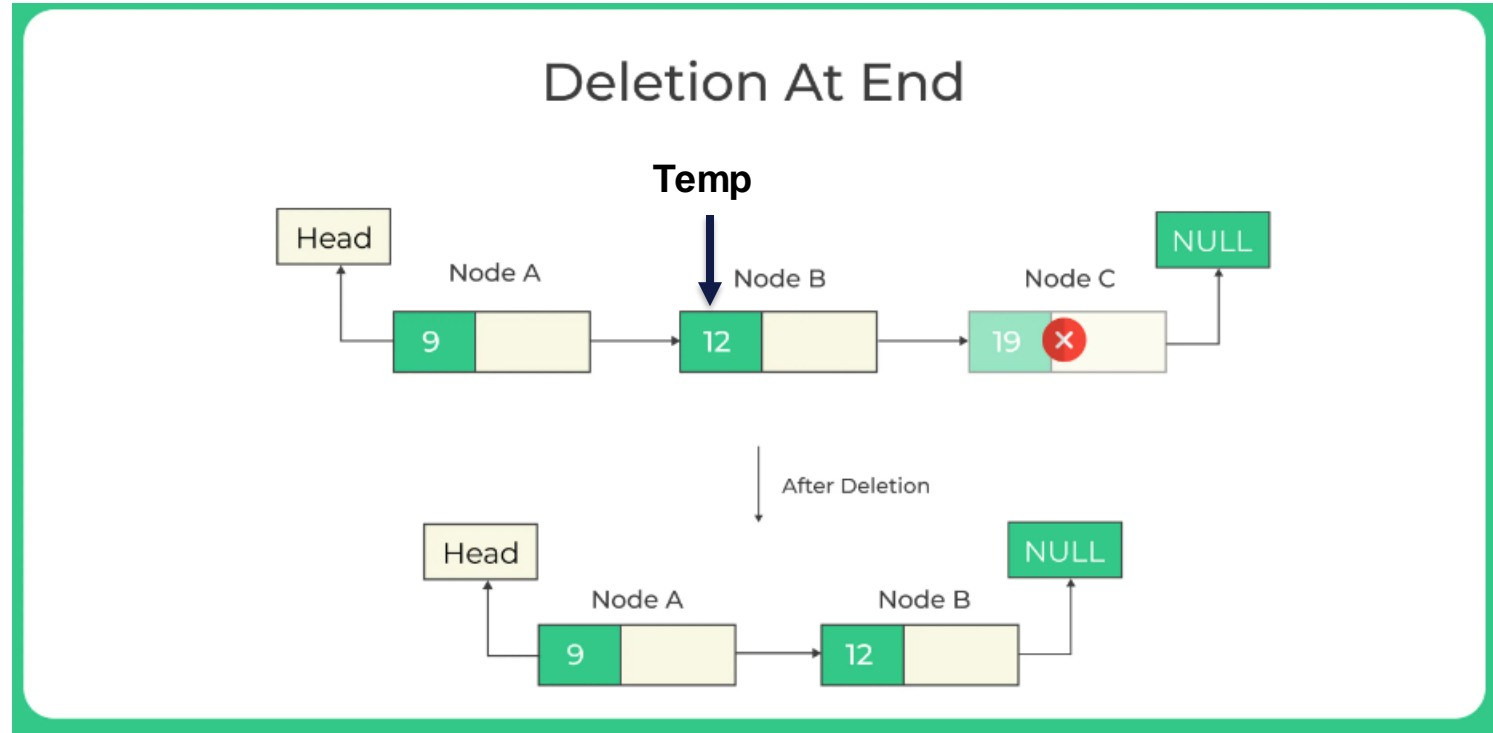
Insert after a given node

2. Basic Operation: *Deletion*

Deletion At Beginning

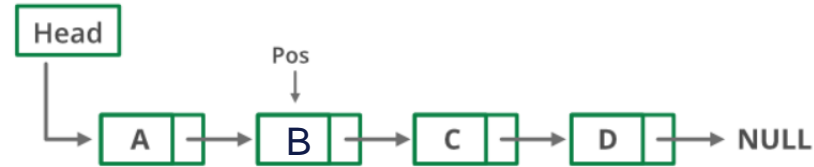


2. Basic Operation: *Deletion*

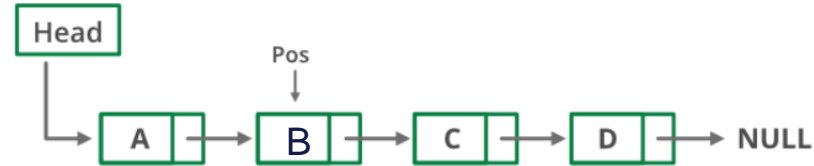


2. Basic Operation: *Deletion*

Initially :

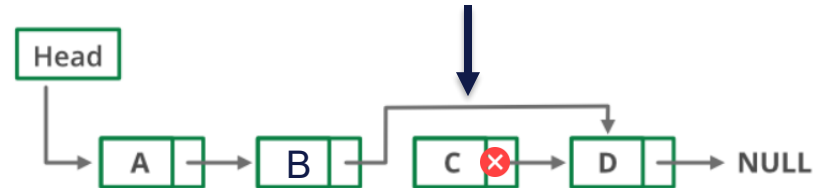


Step 1:



Temp

Step 2:



2. Basic Operation: *Search an element*

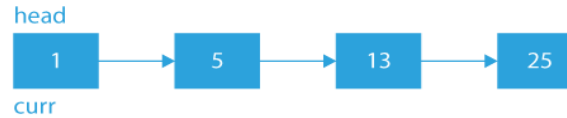


Data Node = 13 ?



Value to be searched (x) = 13

Initially, curr will point to head



As curr \rightarrow data \neq x
curr = curr \rightarrow next



As curr \rightarrow data \neq x
curr = curr \rightarrow next

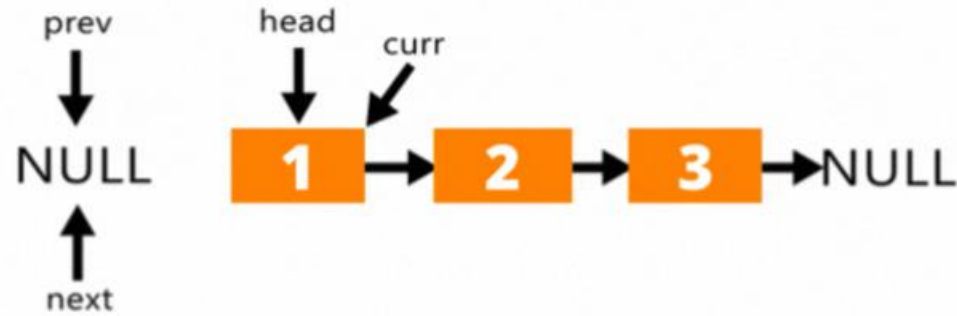


Now, curr \rightarrow data = x
 \therefore return true

Final Output

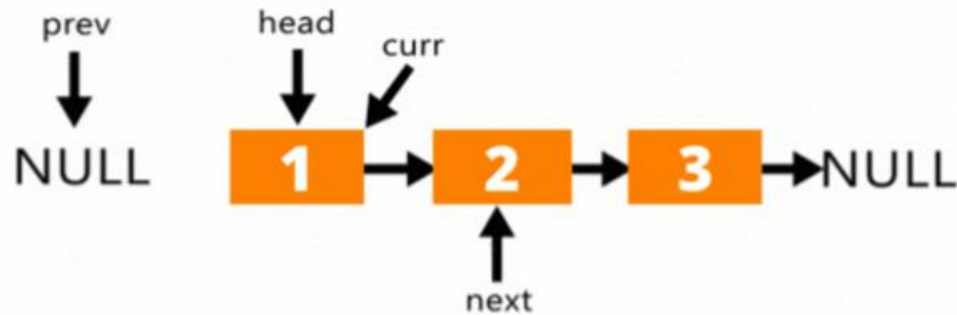
Yes

2. Basic Operation: *Reverse*



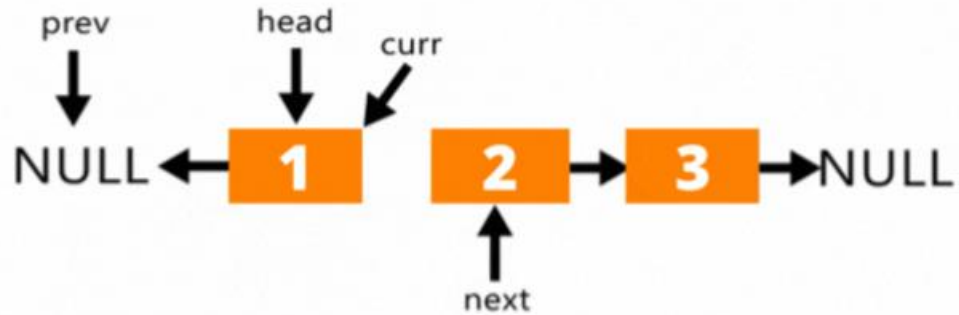
```
while (current != NULL)
{
    next = current->next;
    current->next = prev;
    prev = current;
    current = next;
}
*head_ref = prev;
```

2. Basic Operation: *Reverse*



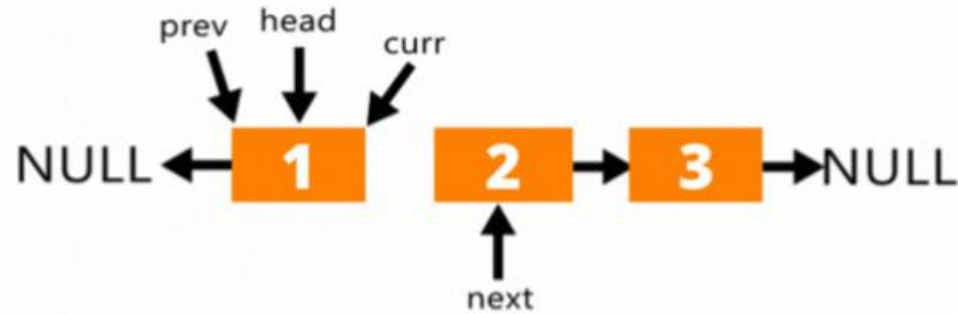
```
while (current != NULL)
{
    next = current->next;
    current->next = prev;
    prev = current;
    current = next;
}
*head_ref = prev;
```

2. Basic Operation: *Reverse*



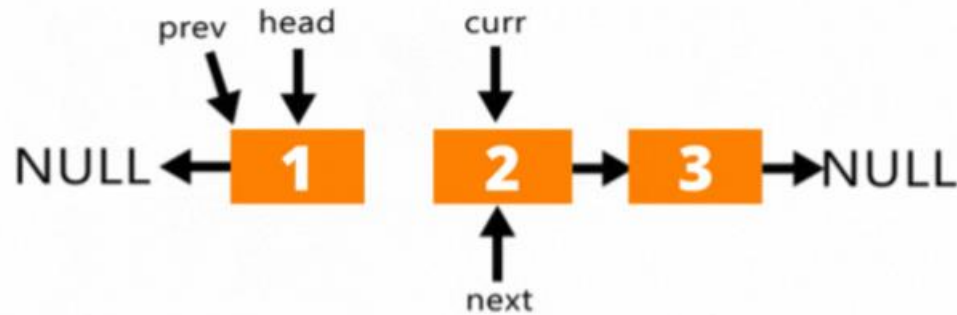
```
while (current != NULL)
{
    next = current->next;
    current->next = prev;
    prev = current;
    current = next;
}
*head_ref = prev;
```

2. Basic Operation: *Reverse*



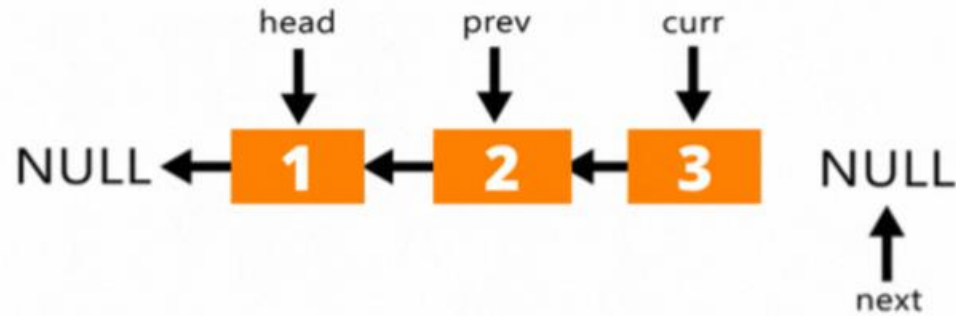
```
while (current != NULL)
{
    next = current->next;
    current->next = prev;
    prev = current;
    current = next;
}
*head_ref = prev;
```

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;
```

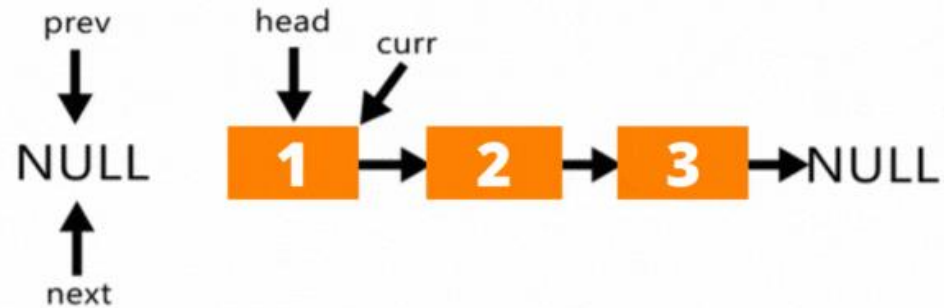

2. Basic Operation: *Reverse*



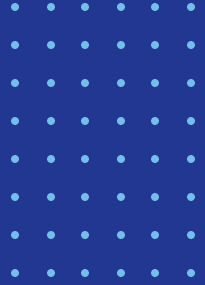
```
while (current != NULL)
{
    next = current->next;
    current->next = prev;
    prev = current;
    current = next;
}
```

`*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

```
6  typedef int (*list_del_cb_t) (void *val);
7  typedef int (*list_cmp_cb_t) (void *val1, void *val2, int type); // type = 0: val2 is data structure, \
8  type = 1: val is key of data structure
9  typedef void (*display_cb_t) (void *val);
10
11  typedef struct listnode
12  {
13      struct listnode *next;
14      struct listnode *prev;
15      void *data;
16  } listnode_t;
17
18  typedef struct list {
19      listnode_t *head;
20      listnode_t *tail;
21      int count;
22      list_cmp_cb_t cmp;
23      list_del_cb_t del;
24      display_cb_t display;
25  } list_t;
26
27  // linkedlist function
28  list_t *list_new();
29  list_t *list_create(list_cmp_cb_t cmp_cb, list_del_cb_t del_cb, display_cb_t display_cb);
30  list_t *link_list_init(list_t *lst, list_cmp_cb_t cmp_cb, list_del_cb_t del_cb);
31  void list_display(list_t *list);
32  void list_free(list_t *list);
33
34  listnode_t *listnode_new (void *data);
35  listnode_t *listnode_add_sort (list_t *, void *data);
36
37  /* add node and sort by value of void *data in struct node and return position of the node in the list */
38  int listnode_add_sort_index (list_t *list, void *data);
39  /* add node to list with cmp_cb, return 0 if node has been add, and return 1 if the node is duplicate */
40  int listnode_add_sort_nodup (list_t *list, void *data);
41
42  listnode_t *listnode_find_node (list_t *list, void *data); // return 0, 1
43  listnode_t *listnode_find_node_key (list_t *list, void *key);
44
45  // creat find node by key, return listnode
46
47  listnode_t *listnode_add_before (list_t *list, listnode_t *node, void *data);
48  listnode_t *listnode_add_after (list_t *list, listnode_t *node, void *data);
49
50  /* Delete specific data pointer from list */
51  listnode_t *listnode_delete_data (list_t *list, void *key);
52  /* delete specific node from the list contain data, 0 - success, 1 - fail */
53  int list_delete_data (struct list *list, void *val);
54  #endif
```

3. Implement Doubly LinkedList: Source file

```
3  list_t *list_new(){
4      list_t *new_list = malloc(sizeof(list_t));
5      if (new_list == NULL) {
6          fprintf(stderr, "Failed to allocate memory for the new list.\n");
7          return NULL;
8      }
9
10     new_list->head = NULL;
11     new_list->tail = NULL;
12     new_list->count = 0;
13     new_list->cmp = NULL;
14     new_list->del = NULL;
15     return new_list;
16 }
```

```
18 ▼ list_t *list_create(list_cmp_cb_t cmp_cb, list_del_cb_t del_cb, display_cb_t display_cb){
19     list_t *new_list = list_new();
20     if (!new_list) return NULL;
21
22     new_list->cmp = cmp_cb;
23     new_list->del = del_cb;
24     new_list->display = display_cb;
25     return new_list;
26 }
```

3. Implement Doubly LinkedList: Source file

```
36 void list_free(list_t *list) {
37     if (list == NULL) return;
38     listnode_t *current = list->head;
39
40     while (current != NULL) {
41         listnode_t *next = current->next;
42         if (list->del) {
43             list->del(current->data);
44         } else {
45             free(current->data);
46         }
47
48         free(current);
49         current = next;
50     }
51     free(list);
52 }
```

3. Implement Doubly LinkedList: Source file

```
54 ▼ void list_display(list_t *list) {
55 ▼     if (list == NULL || list->display == NULL) {
56         fprintf(stderr, "List is NULL or display function not set.\n");
57         return;
58     }
59
60     listnode_t *current = list->head;
61 ▼     while (current != NULL) {
62         list->display(current->data);
63         current = current->next;
64     }
65 }
```

```
67 listnode_t *listnode_new(void *data) {
68     listnode_t *new_node = malloc(sizeof(listnode_t));
69     if (new_node == NULL) {
70         fprintf(stderr, "Failed to allocate memory for new node.\n");
71         return NULL;
72     }
73
74     new_node->next = NULL;
75     new_node->prev = NULL;
76     new_node->data = data;
77     return new_node;
78 }
```

3. Implement Doubly LinkedList: Source file

```
80 listnode_t *listnode_add_sort(list_t *list, void *data) {
81     if (list == NULL || data == NULL) return NULL;
82     listnode_t *new_node = listnode_new(data);
83     if (new_node == NULL) return NULL;
84
85     if (list->head == NULL) {
86         list->head = new_node;
87         list->tail = new_node;
88         list->count++;
89         return new_node;
90     }
91
92     listnode_t *current = list->head;
93     //find position for new node
94     while (current != NULL && list->cmp(current->data, data) < 0) { //compare current->data ? data new
95         current = current->next;
96     }
97
98     //insert in LinkedList
99     if (current == NULL) {
100         // insert to tail
101         list->tail->next = new_node;
102         new_node->prev = list->tail;
103         list->tail = new_node;
104     } else {
105         // insert before current
106         new_node->next = current;
107         new_node->prev = current->prev;
108         if (current->prev) {
109             current->prev->next = new_node;
110         } else {
111             list->head = new_node; //insert head
112         }
113         current->prev = new_node;
114     }
115     list->count++;
116     return new_node;
117 }
```


3. Implement Doubly LinkedList: Source file

```
119 int listnode_add_sort_index(list_t *list, void *data) {
120     if (list == NULL || data == NULL) return -1;
121     listnode_t *new_node = listnode_new(data);
122     if (new_node == NULL) return -1;
123
124     int index = 0;
125     if (list->head == NULL) {
126         list->head = new_node;
127         list->tail = new_node;
128         list->count++;
129         return (index+1);
130     }
131
132     listnode_t *current = list->head;
133     //find position for new node
134     while (current != NULL && list->cmp(current->data, data) < 0) { //compare current->data ? data new
135         current = current->next;
136         index++;
137     }
138
139     //insert in LinkedList
140     if (current == NULL) {
141         // insert to tail
142         list->tail->next = new_node;
143         new_node->prev = list->tail;
144         list->tail = new_node;
145     } else {
146         // insert before current
147         new_node->next = current;
148         new_node->prev = current->prev;
149         if (current->prev) {
150             current->prev->next = new_node;
151         } else {
152             list->head = new_node; //insert head
153         }
154         current->prev = new_node;
155     }
156     list->count++;
157     return index+1;
158 }
```

3. Implement Doubly LinkedList: Source file

```
160 int listnode_add_sort_nodup(list_t *list, void *data) {
161     if (list == NULL || data == NULL) return -1;
162     listnode_t *new_node = listnode_new(data);
163     if (new_node == NULL) return -1;
164
165     if (list->head == NULL) {
166         list->head = new_node;
167         list->tail = new_node;
168         list->count++;
169         return 0;
170     }
171
172     listnode_t *current = list->head;
173     //find position for new node
174     while (current != NULL && list->cmp(current->data, data) < 0) { //compare current->data ? data new
175         current = current->next;
176     }
177
178     if (current == NULL) {
179         // insert to tail
180         list->tail->next = new_node;
181         new_node->prev = list->tail;
182         list->tail = new_node;
183     } else {
184         // insert before current
185         new_node->next = current;
186         new_node->prev = current->prev;
187         if (current->prev) {
188             current->prev->next = new_node;
189         } else {
190             list->head = new_node; //insert head
191         }
192         current->prev = new_node;
193     }
194     list->count++;
195     return ((new_node->data == new_node->prev->data) ? 1 : 0);
196 }
```

3. Implement Doubly LinkedList: Source file

```
201 listnode_t *listnode_find_node (list_t *list, void* data){
202     if(list == NULL){
203         return NULL;
204     }
205     listnode_t *current = list->head;
206     while(current != NULL && list->cmp(current->data, data, 0) != 0){
207         current = current->next;
208     }
209     return ((current != NULL) ? current : NULL);
210 }
211
212 listnode_t *listnode_find_node_key (list_t *list, void* key);{
213     if(list == NULL){
214         return NULL;
215     }
216     listnode_t *current = list->head;
217     while(current != NULL && list->cmp(current->data, key, 1) != 0){
218         current = current->next;
219     }
220     return ((current != NULL) ? current : NULL);
221 }
```

3. Implement Doubly LinkedList: Source file

```
225 listnode_t *listnode_add_before (list_t *list, listnode_t *node, void *data){
226     listnode_t *current = node;
227     if(current == NULL) {
228         return NULL;
229     }else{
230         listnode_t *new_node = listnode_new(data);
231         if (new_node == NULL) return NULL;
232         new_node->next = current;
233         new_node->prev = current->prev;
234         if (current->prev) {
235             current->prev->next = new_node;
236         } else {
237             list->head = new_node; //insert head
238         }
239         current->prev = new_node;
240         list->count++;
241         return new_node;
242     }
243 }
244
245 listnode_t *listnode_add_after (list_t *list, listnode_t *node, void *data){
246     listnode_t *current = node;
247     if(current == NULL) {
248         return NULL;
249     }else{
250         listnode_t *new_node = listnode_new(data);
251         if(new_node == NULL) return NULL;
252         new_node->next = current->next;
253         new_node->prev = current;
254         if(current->next){
255             current->next->prev = new_node;
256         } else {
257             list->tail = new_node; //insert tail
258         }
259         current->next = new_node;
260         list->count++;
261         return new_node;
262     }
263 }
```

3. Implement Doubly LinkedList: Source file

```
265 listnode_t *listnode_delete_data (list_t *list, void *key){
266     if(list == NULL) return NULL;
267     listnode_t *tmp = listnode_find_node_key(list, key);
268     if(tmp != NULL){
269         if(list->del){
270             list->del(tmp->data);
271         }else{
272             free(tmp->data);
273         }
274     }
275     return tmp;
276 }
277
278 int listnode_delete (list_t *list, void *val){
279     listnode_t *current = listnode_delete_data(list, val);
280     if(current == NULL) return -1;
281     if(current == list->head){
282         list->head = current->next;
283         list->head->prev = NULL;
284         free(current);
285         return 0;
286     } else if(current == list->tail){
287         list->tail = current->prev;
288         list->tail->next = NULL;
289         free(current);
290         return 0;
291     } else {
292         current->next->prev = current->prev;
293         current->next->prev->next = current->next;
294         free(current);
295         return 0;
296     }
297 }
```

3. Implement Doubly LinkedList: main.c

```
6  typedef struct SV{
7      int mssv;
8      char name[30];
9  } sv_t;
10
11 int sv_compare(void *val1, void *val2, int type) {
12     if(type == 0){
13         sv_t *sv1 = (sv_t *)val1;
14         sv_t *sv2 = (sv_t *)val2;
15         return (sv1->mssv > sv2->mssv) - (sv1->mssv < sv2->mssv);
16     } else {
17         sv_t *sv = (sv_t *)val1;
18         int *key = (int *)val2;
19         return (sv->mssv == *key) ? 0 : -1;
20     }
21 }
22
23
24 int sv_del(void *val){
25     if(val == NULL){
26         return -1;
27     }
28     sv_t *sv = (sv_t *)val;
29     free(sv);
30     return 0;
31 }
32
33 void display_sv(void *val){
34     sv_t *sv = (sv_t *)val;
35     printf("MSSV: %d, Name: %s\n", sv->mssv, sv->name);
36 }
37
38
39 int main(){
40     list_t *my_list = list_create(sv_compare, sv_del, display_sv);
41     sv_t *sv1 = malloc(sizeof(sv_t));
42     sv_t *sv2 = malloc(sizeof(sv_t));
43     sv_t *sv3 = malloc(sizeof(sv_t));
44     sv_t *sv4 = malloc(sizeof(sv_t));
45     sv_t *sv5 = malloc(sizeof(sv_t));
46     sv_t *sv6 = malloc(sizeof(sv_t));
47     int key_tmp = 666;
48
49     if(sv1){
50         sv1->mssv = 100;
51         strcpy(sv1->name, "AAA");
52     }
53     if(sv2){
54         sv2->mssv = 103;
55         strcpy(sv2->name, "BBB");
56     }
57     if(sv3){
58         sv3->mssv = 99;
59         strcpy(sv3->name, "CCC");
60     }
61     if(sv4){
62         sv4->mssv = 103;
63         strcpy(sv4->name, "BBB");
64     }
65     if(sv5){
66         sv5->mssv = 666;
67         strcpy(sv5->name, "EEE");
68     }
69     if(sv6){
70         sv6->mssv = 888;
71         strcpy(sv6->name, "FFF");
72     }
```

3. Implement Doubly LinkedList: main.c

```
75 listnode_add_sort(my_list, sv1);
76 printf("Add node sv1\n");
77 list_display(my_list);
78 listnode_add_sort(my_list, sv2);
79 printf("Add node sv2\n");
80 list_display(my_list);
81 printf("Index of new node[mssv %d]: %d\n",sv3->mssv,listnode_add_sort_index(my_list, sv3));
82 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

```
81  printf("Index of new node[mssv %d]: %d\n",sv3->mssv,listnode_add_sort_index(my_list, sv3));
82  list_display(my_list);
83  printf("%s",listnode_add_sort_nodup(my_list, sv4) ? "New node not duplicate\n" : "New node
84  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

```
85 listnode_t *temp = listnode_find_node(my_list, sv1);
86 listnode_add_before(my_list, temp, sv5);
87 printf("Add node sv5 before sv1\n");
88 list_display(my_list);
89 temp = listnode_find_node_key(my_list, &key_tmp);
90 listnode_add_after(my_list, temp, sv6);
91 printf("Add node sv6 after node have mssv = 666\n");
92 list_display(my_list);
93 key_tmp = 100;
94 listnode_delete(my_list, &key_tmp);
95 printf("Delete node have mssv = 100\n");
96 list_display(my_list);
97 key_tmp = 103;
98 listnode_delete(my_list, &key_tmp);
99 printf("Delete node have mssv = 103\n");
100 list_display(my_list);
101
102 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