Doubly Linked List:

```
#include<stdio.h>
#include<stdlib.h>
struct Node
{
	int data;
	struct Node*pre;
	struct Node*next;
};

void doubly_Traverse(struct Node*ptr)
{
	while(ptr!=NULL){
	printf("Element:%d\n",ptr->data);
	ptr=ptr->next;
	}
}

struct Node* insert_At_Begain(struct Node*head,int data)
{
	struct Node*p=(struct Node*)malloc(sizeof(struct Node));
	p->data=data;
	p->pre=NULL;
	p->next=head;
	return p;
};

struct Node* insert_At_Index(struct Node*head,int data,int index)
{
	struct Node*r=head->next;
	struct Node*ptr=(struct Node*)malloc(sizeof(struct Node));
	struct Node*q=head;
	ptr->data=data;
	int i=1;
	while(i!=index-1)
	{
	q=q->next;
	r=r->next;
	i++;
	}
```

```
        ptr->next=r;
        r->pre=ptr;
        q->next=ptr;
        ptr->pre=q;
        return head;
};

struct Node* insert_At_End(struct Node*head,int data)
{
        struct Node*q=head;
        struct Node*ptr=(struct Node*)malloc(sizeof(struct Node));
        ptr->data=data;
        while(q->next!=NULL)
        {
        q=q->next;
        }
        q->next=ptr;
        ptr->pre=q;
        ptr->next=NULL;
        return head;
};

struct Node* delete_At_Begain(struct Node*head)
{
        struct Node*q=head;
        head=head->next;
        free(q);
        return head;
};

struct node* deletionofvalue(struct Node* head,int value){
        struct Node*p= head;
        struct Node* q= head->next;
        while(q->data!= value){
        p=p->next;
        q=q->next;
        }
        if(q->data==value){
        p->next=q->next;
        free(q);
        }
        return head;
}
```

```
struct Node*delete_At_Index(struct Node*head,int index)
{
	struct Node*ptr=head;
	struct Node*q=head->next;
	int i=1;
	while(i!=index-1)
	{
	ptr=ptr->next;
	q=q->next;
	i++;
	}
	ptr->next=q->next;
	free(q);
	return head;
};

struct Node*delete_At_End(struct Node *head)
{
	struct Node*ptr=head;
	struct Node*q=head->next;
	while(q->next!=NULL)
	{
	q=q->next;
	ptr=ptr->next;
	}
	ptr->next=NULL;
	free(q);
	return head;
};

int main()
{
int data1,data2,data3,n;

	struct Node*head=(struct Node*)malloc(sizeof(struct Node));
	struct Node*one=(struct Node*)malloc(sizeof(struct Node));
	struct Node*two=(struct Node*)malloc(sizeof(struct Node));
	struct Node*three=(struct Node*)malloc(sizeof(struct Node));
	struct Node*four=(struct Node*)malloc(sizeof(struct Node));

	head->pre=NULL;
	head->data=10;
	head->next=one;
```

```
one->pre=head;
one->data=20;
one->next=two;

two->pre=one;
two->data=30;
two->next=three;

three->pre=two;
three->data=40;
three->next=four;

four->pre=three;
four->data=50;
four->next=NULL;

doubly_Traverse(head);

printf("\n\nEnter The data for the first Node:");
scanf("%d",&data1);
head=insert_At_Begain(head,data1);

doubly_Traverse(head);

printf("\n\nEnter the index :");
scanf("%d",&n);
printf("Enter the data for new Node:");
scanf("%d",&data2);
head=insert_At_Index(head,data2,n);

doubly_Traverse(head);

printf("\n\nEnter The data for the last Node:");
scanf("%d",&data3);
head=insert_At_End(head,data3);
doubly_Traverse(head);
printf("\n\nAfter Deleting First Node\n");
head=delete_At_Begain(head);

doubly_Traverse(head);

printf("\n\nAfter Deleting by value\n");
deletionofvalue(head,50);
doubly_Traverse(head);
```

```
        int i;
        printf("\n\nEnter the Index:");
        scanf("%d",&i);
        printf("After Deleting Index Node:\n");
        head=delete_At_Index(head,i);
        doubly_Traverse(head);

        printf("\n\nAfter Deleting Last Node\n");
        head=delete_At_End(head);
        doubly_Traverse(head);
}
```

Output:
Element:10
Element:20
Element:30
Element:40
Element:50

Enter The data for the first Node:1
Element:1
Element:10
Element:20
Element:30
Element:40
Element:50

Enter the index :2
Enter the data for new Node:2
Element:1
Element:2
Element:10
Element:20
Element:30
Element:40
Element:50

Enter The data for the last Node:6
Element:1
Element:2

Element:10
Element:20
Element:30
Element:40
Element:50
Element:6

After Deleting First Node
Element:2
Element:10
Element:20
Element:30
Element:40
Element:50
Element:6

After Deleting by value
Element:2
Element:10
Element:20
Element:30
Element:40
Element:6

Enter the Index:4
After Deleting Index Node:
Element:2
Element:10
Element:20
Element:40
Element:6

After Deleting Last Node
Element:2
Element:10
Element:20
Element:40

Doubly Linked List

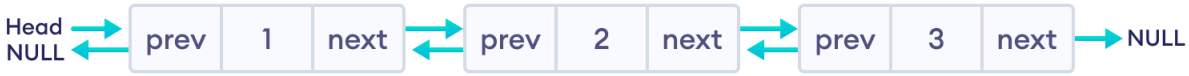
A doubly linked list is a type of linked list in which each node consists of 3 components:

- `*prev` - address of the previous node
- `data` - data item
- `*next` - address of next node

A doubly linked list node

Representation of Doubly Linked List

Let's see how we can represent a doubly linked list on an algorithm/code. Suppose we have a doubly linked list:



Newly created doubly linked list

Here, the single node is represented as

```
struct node {
   int data;
   struct node *next;
   struct node *prev;
}
```

Each struct node has a data item, a pointer to the previous struct node, and a pointer to the next struct node.

Now we will create a simple doubly linked list with three items to understand how this works.

```
/* Initialize nodes */
struct node *head;
struct node *one = NULL;
```

```
struct node *two = NULL;
struct node *three = NULL;

/* Allocate memory */
one = malloc(sizeof(struct node));
two = malloc(sizeof(struct node));
three = malloc(sizeof(struct node));

/* Assign data values */
one->data = 1;
two->data = 2;
three->data = 3;

/* Connect nodes */
one->next = two;
one->prev = NULL;

two->next = three;
two->prev = one;

three->next = NULL;
three->prev = two;

/* Save address of first node in head */
head = one;
```

In the above code, `one`, `two`, and `three` are the nodes with data items **1**, **2**, and **3** respectively.

- **For node one**: `next` stores the address of `two` and `prev` stores `null` (there is no node before it)
- **For node two**: `next` stores the address of `three` and `prev` stores the address of `one`
- **For node three**: `next` stores `null` (there is no node after it) and `prev` stores the address of `two`.

Note: In the case of the head node, `prev` points to `null`, and in the case of the tail pointer, `next` points to null. Here, `one` is a head node and `three` is a tail node.

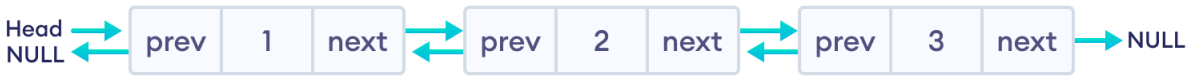
Insertion on a Doubly Linked List

Pushing a node to a doubly-linked list is similar to pushing a node to a linked list, but extra work is required to handle the pointer to the previous node.

We can insert elements at 3 different positions of a doubly-linked list:

1. Insertion at the beginning
2. Insertion in-between nodes
3. Insertion at the End

Suppose we have a double-linked list with elements **1**, **2**, and **3**.



Original doubly linked list

1. Insertion at the Beginning

Let's add a node with value **6** at the beginning of the doubly linked list we made above.

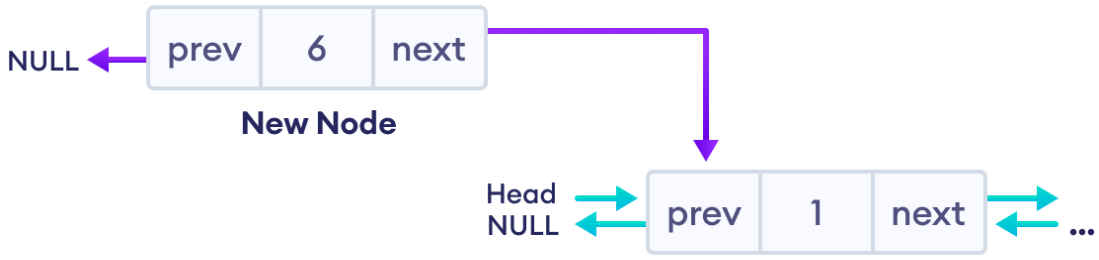
1. Create a new node

- allocate memory for `newNode`
- assign the data to `newNode`.

New node

2. Set prev and next pointers of new node

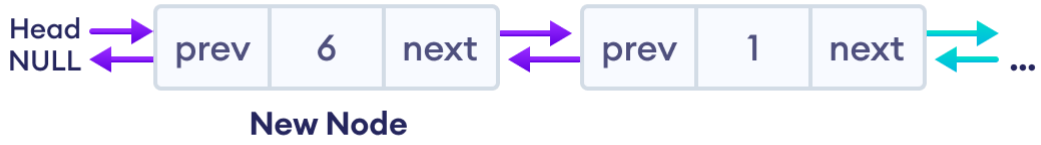
- point `next` of `newNode` to the first node of the doubly linked list
- point `prev` to `null`



Reorganize the pointers (changes are denoted by purple arrows)

3. Make new node as head node

- Point `prev` of the first node to `newNode` (now the previous `head` is the second node)
- Point `head` to `newNode`



Reorganize the pointers

Code for Insertion at the Beginning

```
// insert node at the front
void insertFront(struct Node** head, int data) {

    // allocate memory for newNode
    struct Node* newNode = new Node;

    // assign data to newNode
    newNode->data = data;

    // point next of newNode to the first node of the doubly linked list
    newNode->next = (*head);

    // point prev to NULL
    newNode->prev = NULL;
```

```
    // point previous of the first node (now first node is the second node) to newNode
    if ((*head) != NULL)
        (*head)->prev = newNode;

    // head points to newNode
    (*head) = newNode;
}
```

2. Insertion in between two nodes

Let's add a node with value 6 after node with value 1 in the doubly linked list.

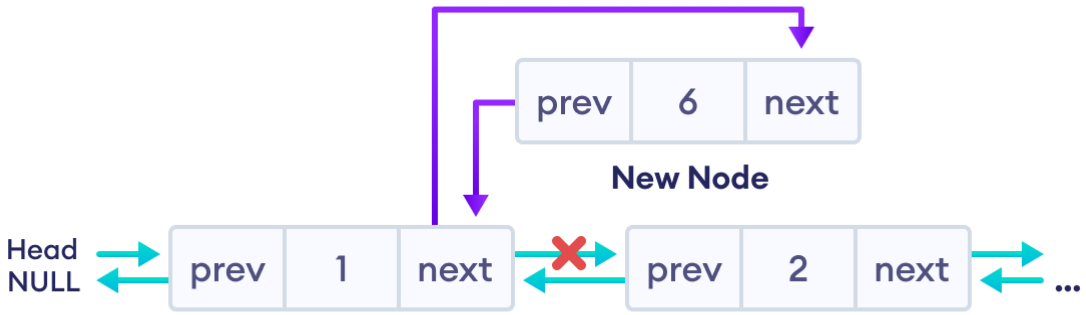
1. Create a new node

- allocate memory for `newNode`
- assign the data to `newNode`.

New node

2. Set the next pointer of new node and previous node

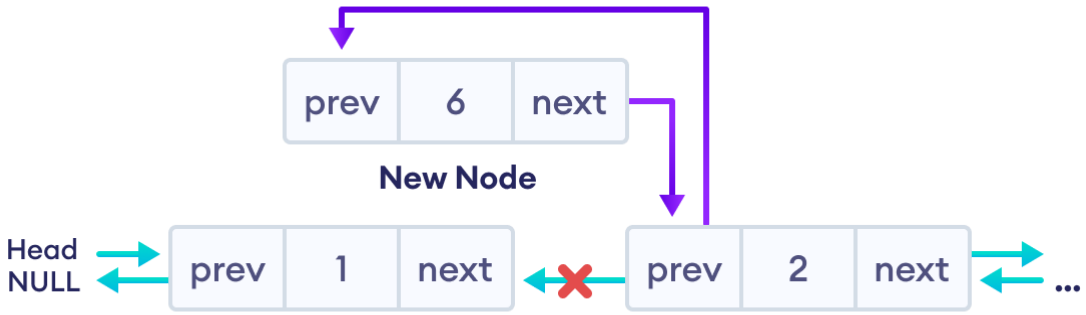
- assign the value of `next` from previous node to the `next` of `newNode`
- assign the address of `newNode` to the `next` of previous node



Reorganize the pointers

3. Set the prev pointer of new node and the next node

- assign the value of `prev` of next node to the `prev` of `newNode`
- assign the address of `newNode` to the `prev` of next node



Reorganize the pointers

The final doubly linked list is after this insertion is:

Final list

Code for Insertion in between two Nodes

```
// insert a node after a specific node
void insertAfter(struct Node* prev_node, int data) {

    // check if previous node is NULL
    if (prev_node == NULL) {
        cout << "previous node cannot be NULL";
        return;
    }

    // allocate memory for newNode
    struct Node* newNode = new Node;

    // assign data to newNode
    newNode->data = data;

    // set next of newNode to next of prev node
    newNode->next = prev_node->next;

    // set next of prev node to newNode
    prev_node->next = newNode;

    // set prev of newNode to the previous node
    newNode->prev = prev_node;

    // set prev of newNode's next to newNode
    if (newNode->next != NULL)
        newNode->next->prev = newNode;
}
```

3. Insertion at the End

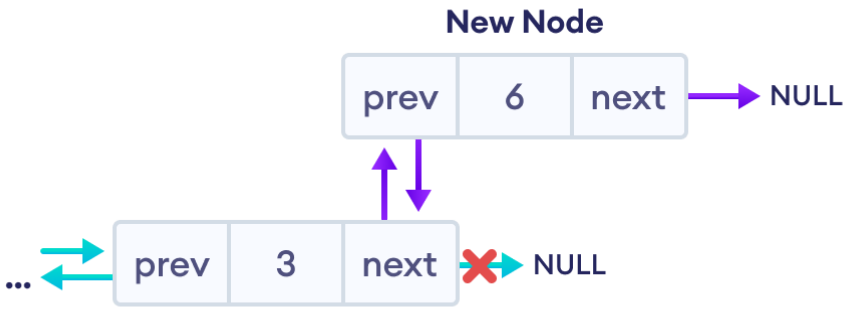
Let's add a node with value 6 at the end of the doubly linked list.

1. Create a new node

New node

2. Set prev and next pointers of new node and the previous node

If the linked list is empty, make the newNode as the head node. Otherwise, traverse to the end of the doubly linked list and



Reorganize the pointers

The final doubly linked list looks like this.

The final list

Code for Insertion at the End

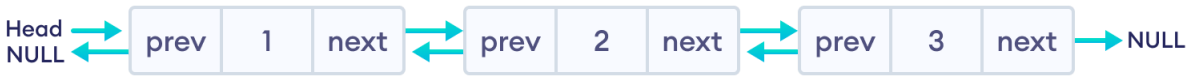
```
// insert a newNode at the end of the list
void insertEnd(struct Node** head, int data) {
    // allocate memory for node
    struct Node* newNode = new Node;

    // assign data to newNode
    newNode->data = data;

    // assign NULL to next of newNode
    newNode->next = NULL;

    // store the head node temporarily (for later use)
    struct Node* temp = *head;
```

```
        // if the linked list is empty, make the newNode as head node
        if (*head == NULL) {
        newNode->prev = NULL;
        *head = newNode;
        return;
        }

        // if the linked list is not empty, traverse to the end of the linked list
        while (temp->next != NULL)
        temp = temp->next;

        // now, the last node of the linked list is temp

        // point the next of the last node (temp) to newNode.
        temp->next = newNode;

        // assign prev of newNode to temp
        newNode->prev = temp;
}
```

Deletion from a Doubly Linked List

Similar to insertion, we can also delete a node from **3** different positions of a doubly linked list.

Suppose we have a double-linked list with elements **1**, **2**, and **3**.

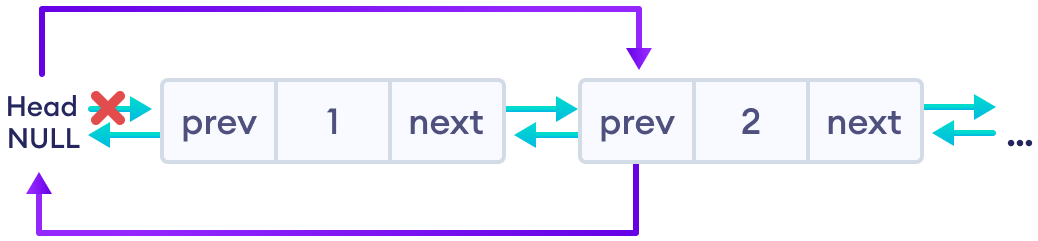


Original doubly linked list

1. Delete the First Node of Doubly Linked List

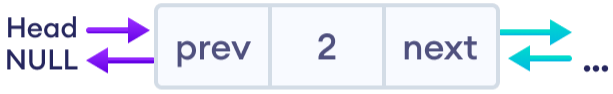
If the node to be deleted (i.e. `del_node`) is at the beginning

Reset value node after the del_node (i.e. node two)



Reorganize the pointers

Finally, free the memory of `del_node`. And, the linked will look like this



Free the space of the first node

Final list

Code for Deletion of the First Node

```
if (*head == del_node)
    *head = del_node->next;

if (del_node->prev != NULL)
    del_node->prev->next = del_node->next;

free(del);
```

2. Deletion of the Inner Node

If `del_node` is an inner node (second node), we must have to reset the value of `next` and `prev` of the nodes before and after the `del_node`.

For the node before the del_node (i.e. first node)

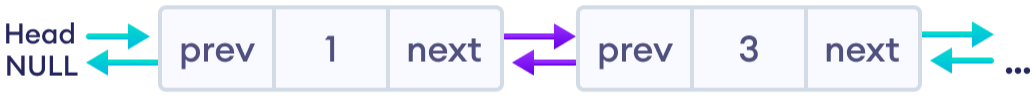
Assign the value of `next` of `del_node` to the `next` of the `first` node.

For the node after the del_node (i.e. third node)

Assign the value of `prev` of `del_node` to the `prev` of the `third` node.

Reorganize the pointers

Finally, we will free the memory of `del_node`. And, the final doubly linked list looks like this.



Final list

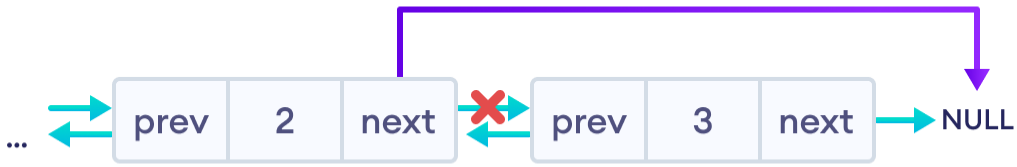
Code for Deletion of the Inner Node

```
if (del_node->next != NULL)
    del_node->next->prev = del_node->prev;

if (del_node->prev != NULL)
    del_node->prev->next = del_node->next;
```

3. Delete the Last Node of Doubly Linked List

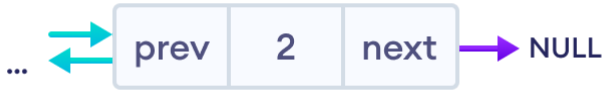
In this case, we are deleting the last node with value **3** of the doubly linked list.

Here, we can simply delete the `del_node` and make the `next` of node before `del_node` point to `NULL`.



Reorganize the pointers

The final doubly linked list looks like this.



Final list

Code for Deletion of the Last Node

```
if (del_node->prev != NULL)
    del_node->prev->next = del_node->next;
```

Here, `del_node ->next` is `NULL` so `del_node->prev->next = NULL`.

Note: We can also solve this using the first condition (for the node before `del_node`) of the second case (Delete the inner node).

Doubly Linked list Program:

```
#include <stdio.h>
#include <stdlib.h>

// node creation
struct Node {
  int data;
  struct Node* next;
  struct Node* prev;
};
```

```
// insert node at the front
void insertFront(struct Node** head, int data) {
  // allocate memory for newNode
  struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

  // assign data to newNode
  newNode->data = data;

  // make newNode as a head
  newNode->next = (*head);

  // assign null to prev
  newNode->prev = NULL;

  // previous of head (now head is the second node) is newNode
  if ((*head) != NULL)
          (*head)->prev = newNode;

  // head points to newNode
  (*head) = newNode;
}

// insert a node after a specific node
void insertAfter(struct Node* prev_node, int data) {
  // check if previous node is null
  if (prev_node == NULL) {
          printf("previous node cannot be null");
          return;
  }

  // allocate memory for newNode
  struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

  // assign data to newNode
  newNode->data = data;

  // set next of newNode to next of prev node
  newNode->next = prev_node->next;

  // set next of prev node to newNode
  prev_node->next = newNode;

  // set prev of newNode to the previous node
  newNode->prev = prev_node;
```

```
  // set prev of newNode's next to newNode
  if (newNode->next != NULL)
        newNode->next->prev = newNode;
}

// insert a newNode at the end of the list
void insertEnd(struct Node** head, int data) {
  // allocate memory for node
  struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

  // assign data to newNode
  newNode->data = data;

  // assign null to next of newNode
  newNode->next = NULL;

  // store the head node temporarily (for later use)
  struct Node* temp = *head;

  // if the linked list is empty, make the newNode as head node
  if (*head == NULL) {
        newNode->prev = NULL;
        *head = newNode;
        return;
  }

  // if the linked list is not empty, traverse to the end of the linked list
  while (temp->next != NULL)
        temp = temp->next;

  // now, the last node of the linked list is temp

  // assign next of the last node (temp) to newNode
  temp->next = newNode;

  // assign prev of newNode to temp
  newNode->prev = temp;
}

// delete a node from the doubly linked list
void deleteNode(struct Node** head, struct Node* del_node) {
  // if head or del is null, deletion is not possible
  if (*head == NULL || del_node == NULL)
```

```
        return;

  // if del_node is the head node, point the head pointer to the next of del_node
  if (*head == del_node)
        *head = del_node->next;

  // if del_node is not at the last node, point the prev of node next to del_node to the previous of
del_node
  if (del_node->next != NULL)
        del_node->next->prev = del_node->prev;

  // if del_node is not the first node, point the next of the previous node to the next node of
del_node
  if (del_node->prev != NULL)
        del_node->prev->next = del_node->next;

  // free the memory of del_node
  free(del_node);
}

// print the doubly linked list
void displayList(struct Node* node) {
  struct Node* last;

  while (node != NULL) {
        printf("%d->", node->data);
        last = node;
        node = node->next;
  }
  if (node == NULL)
        printf("NULL\n");
}

int main() {
  // initialize an empty node
  struct Node* head = NULL;

  insertEnd(&head, 5);
  insertFront(&head, 1);
  insertFront(&head, 6);
  insertEnd(&head, 9);

  // insert 11 after head
  insertAfter(head, 11);
```

```
  // insert 15 after the seond node
  insertAfter(head->next, 15);

  displayList(head);

  // delete the last node
  deleteNode(&head, head->next->next->next->next->next);

  displayList(head);
}
```