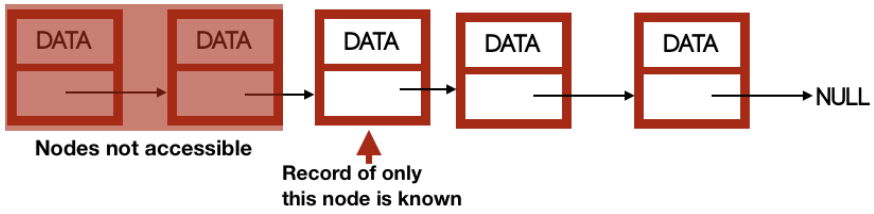
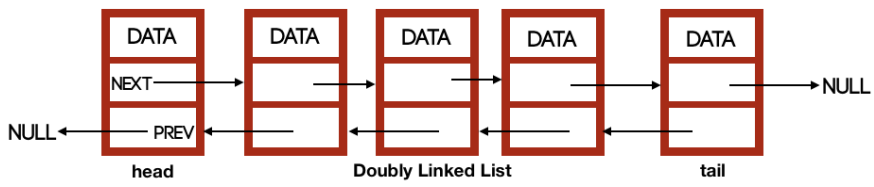
[redacted] (/)

Doubly Linked Lists

In a singly linked list, we can only move forward from any node. For example, if we are at node *b*, as shown in the picture given below, we can't access the node previous to it.



This problem can easily be fixed by using one extra link for each node which will point to the previous node.



This is called **doubly linked list**.

As you can see in the above picture, *prev* of the head and *next* of the tail point to null. So, let's make the node for a doubly linked list.

C **Python** **Java**

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

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

You can see that we have used two pointers *next* and *prev* to make a node instead of just using *next* as we did in the singly linked list.

Let's write the functions of inserting and deleting a node in a doubly linked list.

Inserting New Node

Similar to singly linked lists, we can have three cases:

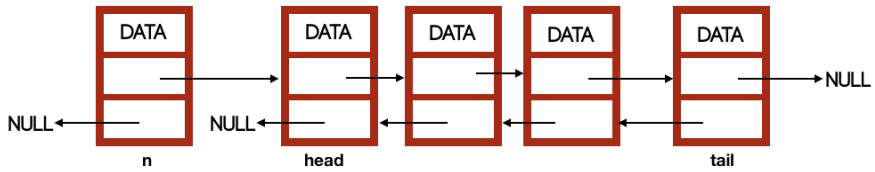
- Inserting a new node at the front of the doubly linked list.
- Inserting a new node at the end of the doubly linked list.
- Inserting a new node after any node in the doubly linked list.

Let's start by writing the code to insert a new node at the front of the linked list.

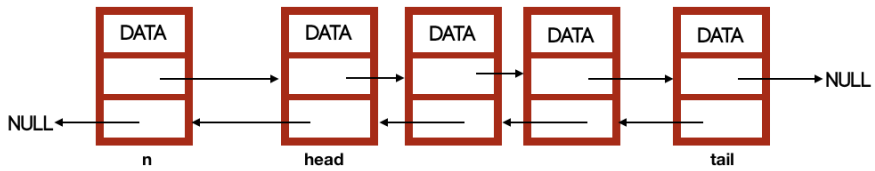
Inserting a New Node at the Front

We will start with a function and pass the linked list and the node to be inserted to it - `INSERT_AT_FRONT(L, n)`.

Our first task is to point *next* of the new node (n) to the head of the linked list i.e., `n.next = L.head`.



Then, we will point prev of the head to the new node - `L.head.prev = n`.



At last, we will just make the new node head of the linked list.

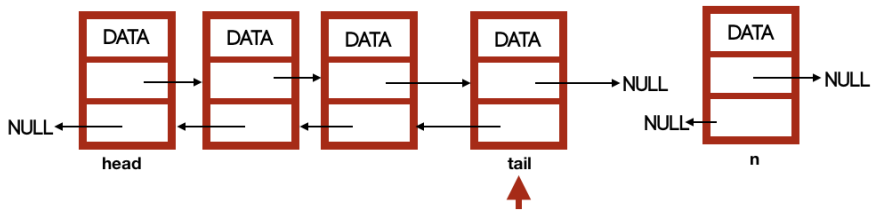
`L.head = n`

```
INSERT_AT_FRONT(L, n)
     n.next = L.head
     L.head.prev = n
     L.head = n
```

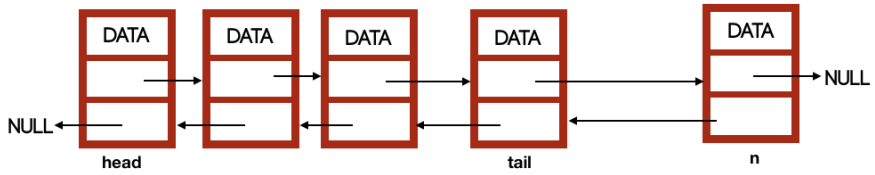
Inserting a New Node at the End

To insert a new node at the tail, we will first iterate to the last node.

```
INSERT_AT_TAIL(L, n)
     tmp = L.head
     while(tmp.next != null)
          tmp = tmp.next
     ...
```



Now, we will point *next* of *tmp* to *n* and *prev* of *n* to *tmp*.



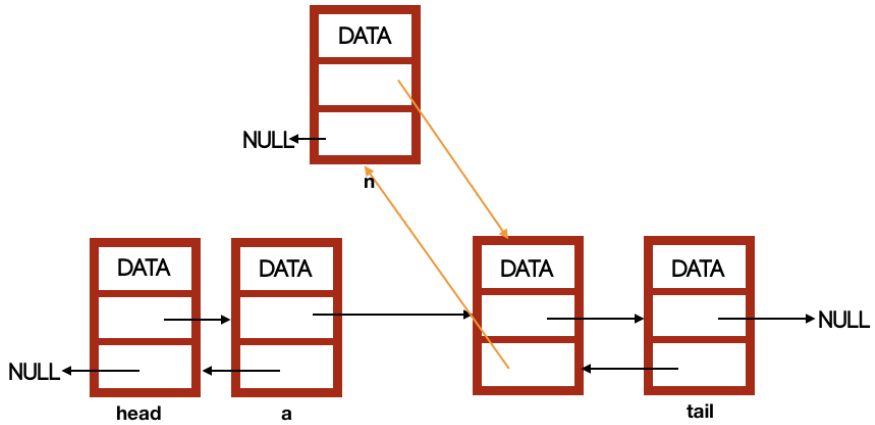

```
INSERT_AT_TAIL(L, n)
     tmp = L.head
     while(tmp.next != null)
          tmp = tmp.next
tmp.next = n
n.prev = tmp
```

Thus, we can insert a new node at the start and the end of a doubly linked list. Let's learn to add a node after any given node.

Inserting a New Node After Any Node

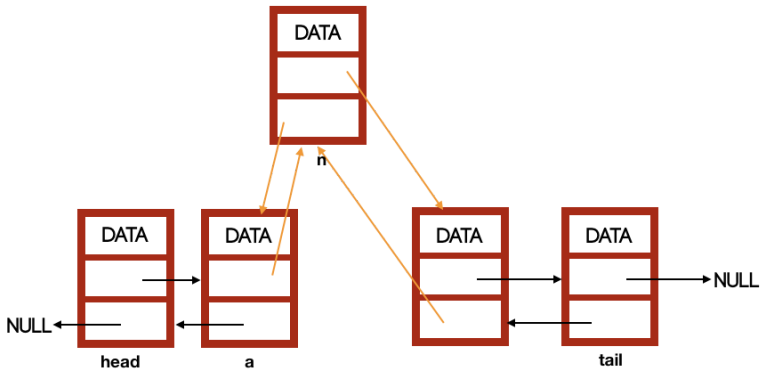
Our function will take the node which we are going to insert (n) and the node after which we are going to insert it (a) - `INSERT_AFTER(n, a)`.

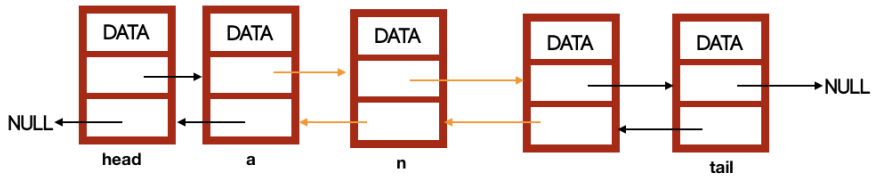
Firstly, we will link the new node *n* and *a.next*.




```
n.next = a.next
a.next.prev = n
```

After this, we will link *a* to *n*.





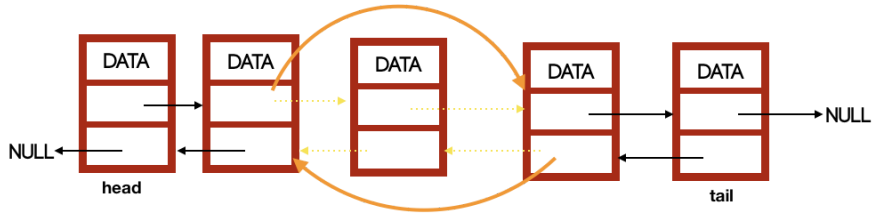

```
a.next = n
n.prev = a
```

```
INSERT_AFTER(n, a)
    n.next = a.next
    a.next.prev = n
    a.next = n
    n.prev = a
```

Our next task is to delete a node from a given linked list.

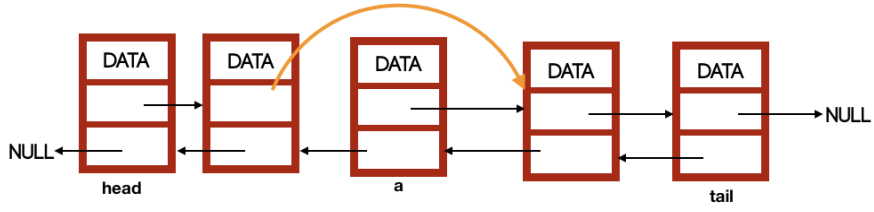
Deleting a Node

Till now, you must have got an idea that deleting a node from a doubly linked list is also similar to a singly linked list. We link the node previous to the node to be deleted to the next of it.



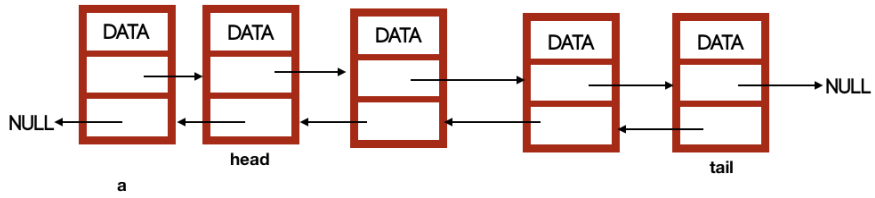
So, let's write the code to do it. We will start making a function and then pass the node to be deleted and the linked list- `DELETE_NODE(L, a)`.

We will first check if the node *a* is head or not. We can do it easily by checking if the *prev* pointer of the node is null or not because *prev* of the head is always null. If the node is not head, then we will point *next* pointer of the node previous to *a* to the next of *a*.



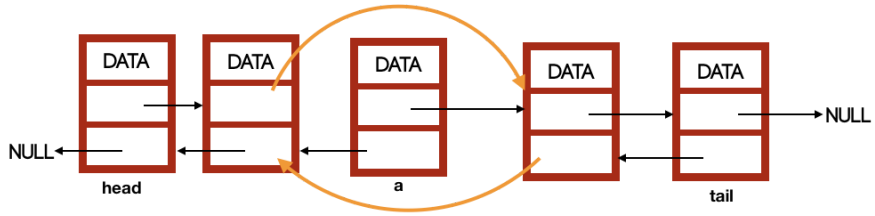

```
if a.prev != null // node is not head
    a.prev.next = a.next
```

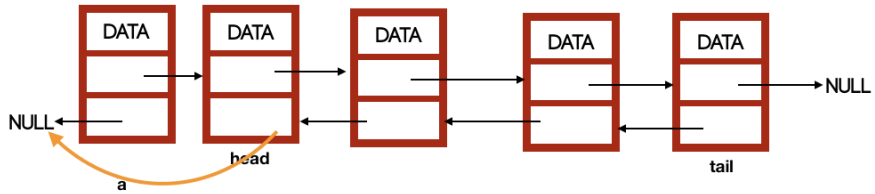
If the node *a* is head, then we will mark the node next to *a* as the head of the linked list.




```
else // node a is head
     L.head = a.next
```

At last, we will point the prev pointer of the node next to *a* to the node previous of *a*.





We will do this if the node *a* is not the last node.

```
if a.next != null
     a.next.prev = a.prev
```

```
DELETE_NODE(L, a)
     if a.prev != null // node is not head
          a.prev.next = a.next
     else // node a is head
          L.head = a.next

     if a.next != null
          a.next.prev = a.prev
```

C **Python** **Java**

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

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

typedef struct linked_list {
  struct node *head;
}linked_list;

//to make new node
node* new_node(int data) {
  node *z;
  z = malloc(sizeof(struct node));
  z->data = data;
  z->next = NULL;
  z->prev = NULL;

  return z;
}

//to make a new linked list
linked_list* new_linked_list(int data) {
  node *a; //new node for head of linked list
  a = new_node(data);

  linked_list *l = malloc(sizeof(linked_list)); //linked list
  l->head = a;

  return l;
}

void traversal(linked_list *l) {
  node *temp = l->head; //temporary pointer to point to head

  while(temp != NULL) { //iterating over linked list
    printf("%d\t", temp->data);
    temp = temp->next;
  }

  printf("\n");
}

//new node before head
void insert_at_front(linked_list *l, node *n) {
  n->next = l->head;
  l->head->prev = n;
  l->head = n;
}

//insert new node at last
void insert_at_tail(linked_list *l, node *n) {
  node *temp = l->head;

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

  temp->next = n;
  n->prev = temp;
}

//function to insert a node after a node
void insert_after(node *n, node *a) {
  n->next = a->next;
  a->next->prev = n;
  a->next = n;
  n->prev = a;
}

//function to delete
void del(linked_list *l, node *a) {
  if(a->prev != NULL) { //node is not head
    a->prev->next = a->next;
```

```
  }
  else { //node a is head
    l->head = a->next;
  }

  if(a->next != NULL) {
    a->next->prev = a->prev;
  }
  free(a);
}

int main() {
  linked_list *l = new_linked_list(10);

  node *a, *b, *c; //new nodes to insert in linekd list
  a = new_node(20);
  b = new_node(50);
  c = new_node(60);

  //connecting to linked list
  /*
     ----      ----      ----      ----
    |head|-->| a  |-->|  b |-->|  c |-->NULL
    |____|   |____|   |____|   |____|
  */
  l->head->next = a;
  a->next = b;
  b->next = c;

  traversal(l);

  node *z;

  z = new_node(0);
  insert_at_front(l, z);
  z = new_node(-10);
  insert_at_front(l, z);

  z = new_node(100);
  insert_at_tail(l, z);

  z = new_node(30);
  insert_after(z, a);
  z = new_node(40);
  insert_after(z, a->next);
  z = new_node(500);
  insert_after(z, a->next->next);

  traversal(l);

  del(l, l->head);
  del(l, z);
  traversal(l);

  return 0;
}
```

So, we have discussed both singly and doubly linked lists. There must be one question in your mind i.e., what are the pros and cons of both linked lists? So, let's compare both and find out.

Singly Linked List v/s Doubly Linked List

- Doubly Linked List requires extra space as compared to Singly Linked List because it stores one extra pointer i.e., the previous node of each node.
- We don't need to store the head pointer in a doubly linked list because we can access each node if we have access to any node of the doubly linked list.
- Doubly Linked List requires more memory but searching for an element in a doubly linked list is comparatively efficient than a singly linked list because we can iterate in both directions.
- Deletion and Insertion tasks are relatively complex in a doubly linked list and more efficient in a singly linked list.

In conclusion, we should use a singly linked list when we have limited memory and searching for elements is not our priority. However, when the limitation of memory is not a problem and insertion and deletion task doesn't happen frequently, we should move with the doubly linked list.

In the next chapter, we are going to discuss another linked list which is circular linked list. So, let's move ahead and learn about it.

> Science is a beautiful gift to humanity; we should not distort it.
>
> - A. P. J. Abdul Kalam

PREV (/course/data-structures-linked-lists/) (/course/data-structures-circular-linked-lists/) NEXT

New Questions

Difference between = and == method In java **- Java**

(/discussion/difference-between-and-method-in-java)

This is a program for displaying multiplication table of any number but when I write program as given it doesn't give proper result but when I declare **- C**

(/discussion/this-is-a-program-for-displaying-multiplication-ta)

What do you mean by Constructor? **- Java**

(/discussion/what-do-you-mean-by-constructor)

setting up an ide for mac.. **- Cpp**

(/discussion/setting-up-an-ide-for-mac)

Please fill the blanks and help me am stuck **- Java**

(/discussion/fill-the-blanks-and-help-me-am-stuck)