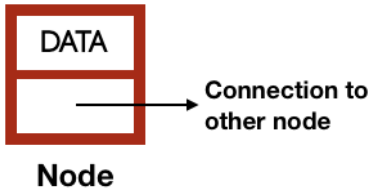
[redacted] (/)

Linked Lists

So far, we know that a data structure is a way to keep our data, and a linked list is the first data structure we are going to study in this course. Breaking the name "**Linked List**", it is a list of elements (containing data) in which the elements are linked together.

As you can see in the above picture, each data is connected (or linked) to a different data and thus, forming a linear list of data, and this is a linked list. Since data in a linked list are stored in a linear fashion, a linked list is a linear data structure. There are also non-linear data structures like trees, graphs, etc. which we are going to study further in this course.

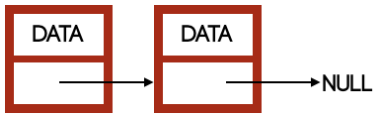
Till now, you know that we have to connect data of a linked list and to do so, we put our data in nodes and these nodes are connected in the desired fashion.



So before going further, let's discuss about nodes.

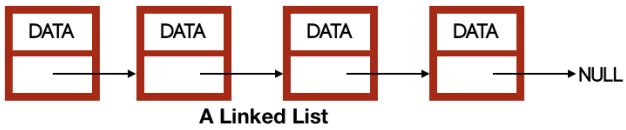
Node

A node can be viewed as a container or a box which contains data and other information in it. Nodes are connected (or organized) in a specific way to make data structures like linked lists, trees, etc.

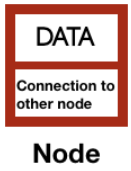


In the above picture, each node has two parts, one stores the data and another is connected to a different node. The last node is not connected to any other node and thus, its connection to the next node is null.

In a linked list, a node is connected to a different node forming a chain of nodes.



Thus to make a linked list, we first need to make a node which should store some data into it and also a link to another node.



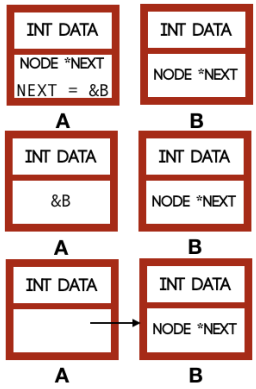
There can be different ways to make this node in different languages, we are going to discuss the making of the node in C, Java and Python.

C

As discussed above, our node should store some data and a connection to a different node. We will use structure to store this information, and to store the connection, we can simply store the address of the node which it will be linked to.


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

So, we have made a structure named node which stores an integer named *data* and a pointer named *next* to another structure (node).



Let's use this structure to make two nodes and connect one with another.

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

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

int main() {
    struct node *a, *b;
    a = malloc(sizeof(struct node));
    b = malloc(sizeof(struct node));

    a->data = 10;
    b->data = 20;

    a->next = b;
    b->next = NULL;

    printf("%d\n%d\n",a->data, a->next->data);
    return 0;
}
```

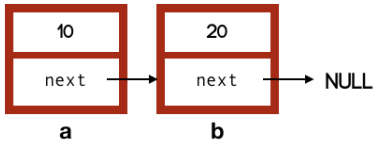
```
struct node *a, *b;
a = malloc(sizeof(struct node));
b = malloc(sizeof(struct node));
```

Here, we are making two nodes (structure named *node*) and allocating space to them using the malloc (https://www.codesdope.com/c-dynamic-memory/) function.

```
a->data = 10;
b->data = 20;
```

And then we are setting the values of *data* of the nodes *a* and *b*.

```
a->next = b;
b->next = NULL;
```



We are storing the address of *b* in the *next* variable of *a*. Thus, *next* of *a* is now storing the address of *b* or we can say the *next* of *a* is pointing to *b*.

Since there is no node for the *next* of *b*, so we are making *next* of *b* null.

Java

```
class Node {
  public int data;
  public Node next;
}
```

You can see that we have created a class named Node to represent a node and the class has two members - the first is the data (an integer) which our node is going to store and the other is another node which is the next node.

Node

As we have created our class which meets our requirement of a node, let's create two objects of this Node class and connect the next of the first node with the second one.

```
class Node {
  public int data;
  public Node next;

  public Node(int d) {
    data = d;
  }
}

class List {
  public static void main(String[] args) {
    Node a, b;
    a = new Node(10);
    b = new Node(20);

    a.next = b;
    b.next = null;

    System.out.print(a.data+"\n"+a.next.data+"\n");
  }
}
```

`Node a, b;` → Here, we are creating two nodes (or two objects of the Node class).

`a.next = b;` → We are storing *b* in next of *a*. So, *next* of *a* is now linked to the node *b*.

Since *b* is the last node (there is no node to point next of *b* to), we are making *next* of *b* null - `b.next = null;`.

Python

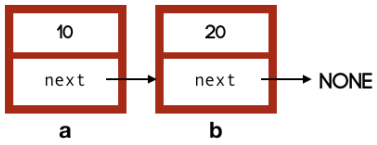
```
class Node():
    def __init__(self, data):
        self.data = data

a = Node(10)
b = Node(20)

a.next = b
b.next = None

print(a.data)
print(a.next.data)
```

Here, we have created a class named Node to represent a node. 'data' is used to store the data of the node.

`a = Node(10)`

`b = Node(20)`

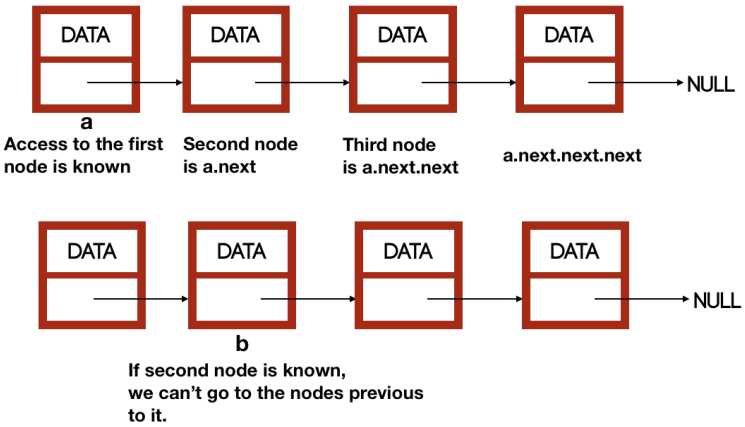
We have created two nodes *a* and *b* with data 10 and 20 respectively and then we are making *b* the next of *a* using `a.next = b`. Since there is no other node to make it *next* of *b*, we are making *next* of *b* `None`.



Now we know how to create nodes and use them. We also know how to connect a node with a different node, so let's move ahead and learn about linked lists.

Making and Traversing a Linked List

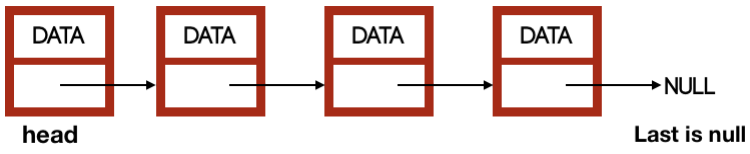
In the above examples, we have made two nodes *a* and *b* and we were able to access the node *b* with `a.next`. Suppose there are few more nodes, we can easily access them if we have access to the first node. This is explained in the picture given below.



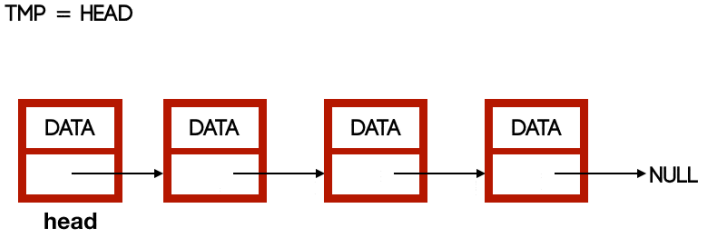
Generally, we call the first node of a linked list the "head" of the linked list, and we always keep the access of the head of the linked list so that we have access to all the nodes of a linked list.

Traversing a Linked List

Traversing is visiting all the nodes of a linked list. As stated above, we always keep a record of the head of a linked list. We also know that the last node of a linked list is null.



So, we start a loop from the head of the linked list and end it when the node is null.



Let's make a function to traverse over all the nodes of a linked list. We will pass the linked list (L) to it - `TRAVERSE(L)`.

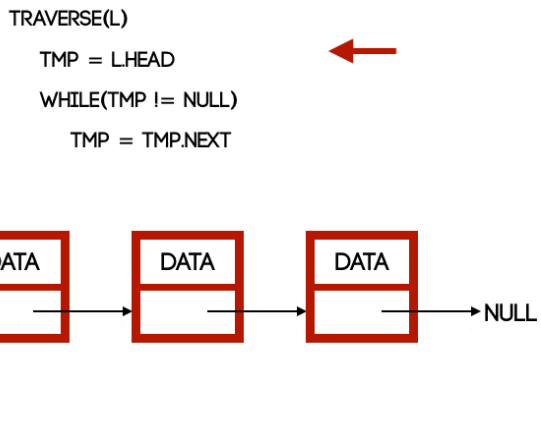
Now, we make a temporary pointer and point it to the head of the linked list (L) - `temp = L.head`.

We have a pointer pointing to the head of the linked list, thus we can start a loop and end it when this pointer will point to null (last element) i.e., `while(temp != null)`.

At the end of each iteration, we have to just point the *temp* pointer to the next of the node i.e., `temp = temp.next`.

```
TRAVERSE(L)
     temp = L.head
     while(temp != null)
          temp = temp.next
```

So, `temp = temp.next` will make the *temp* point the next node in each iteration and the loop will run until the last node (until *temp* is not null).



Till now, we know what a linked list is, what a node is and how to traverse over each node of a linked list. Let's learn about adding nodes to a linked list so that we can make a linked list.

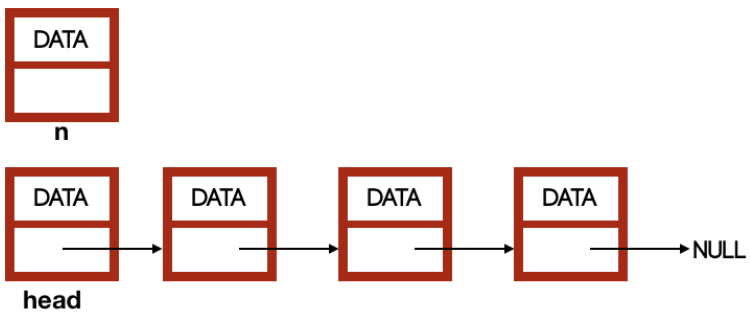
Inserting Nodes in a Linked List

There can be three different positions where we can insert a new node in a linked list:

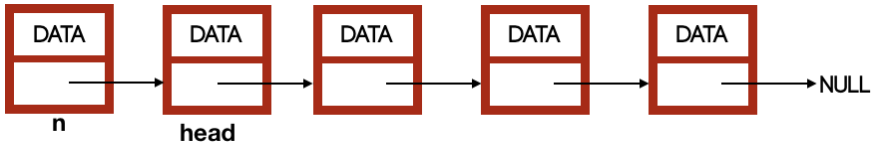
- At the beginning of the list.
- At the end of the list.
- Anywhere except the above-mentioned positions.

Inserting a New Node at the Beginning of a Linked List

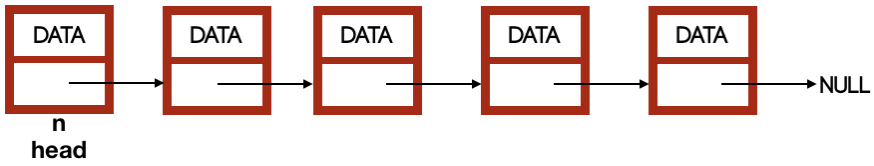
We start by passing the node (n) and the linked list (L) to a function i.e., `INSERT_AT_BEGINNING(L, n)`.



After this, we point the next of the node to the head of the linked list - `n.next = L.head`.



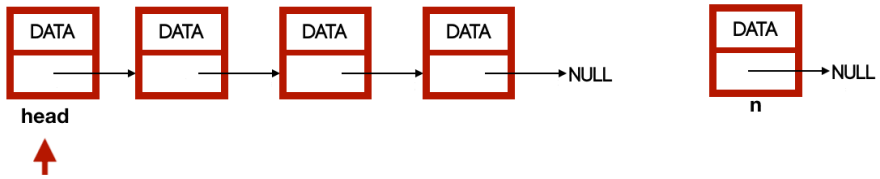
Since the head should always point to the first element of the linked list, so we change the head to point to the new node n i.e, `L.head = n`.




```
INSERT_AT_BEGINNING(L, n)
     n.next = L.head
     L.head = n
```

Inserting a New Node at the End of a Linked List.

To insert a node at the end of a linked list, we just iterate to the last of the linked list and add a new node there. This is described in the picture given below.



We start by passing the linked list and the node to the function - `INSERT_AT_LAST(L, n)`.

Our next task is to iterate to the last of the linked list.

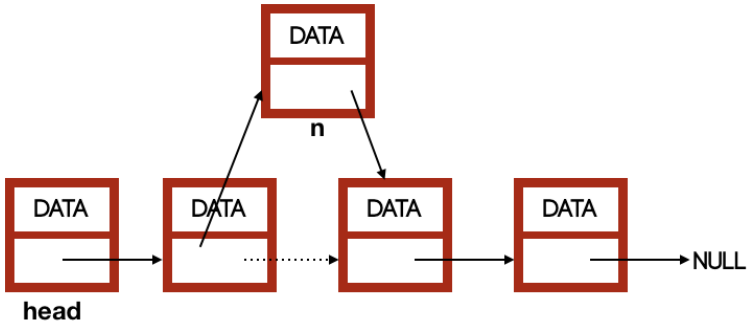
```
temp = L.head
while(temp.next != null)
     temp = temp.next
```

After this, we just need to add the node there.

```
INSERT_AT_LAST(L, n)
     temp = L.head
     while(temp.next != null)
          temp = temp.next

     temp.next = n
```

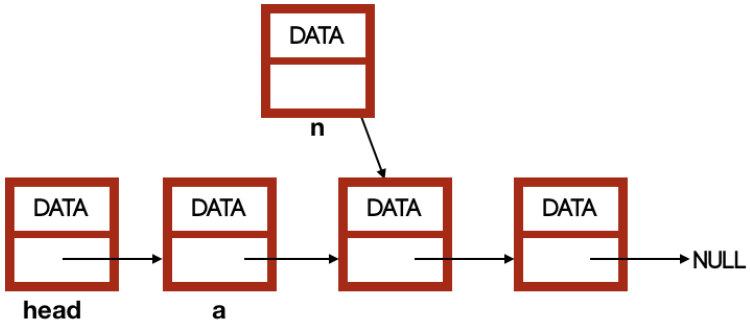
Inserting a New Node in the Middle of a Linked List

To insert a new node in the middle of a linked list, we need to break the existing links and create new links. This will be clear from the picture given below.



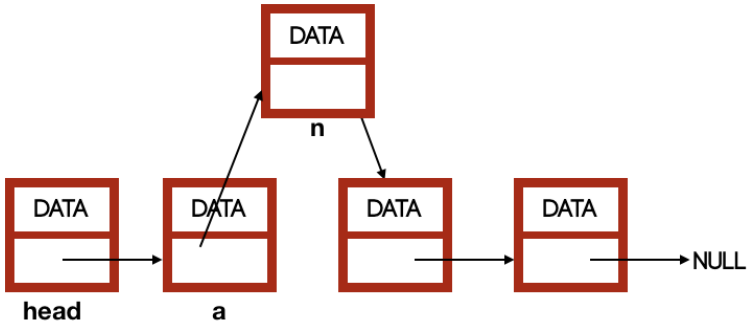
So, we start by passing the node to be inserted (n) and the node after which we are going to insert this node (a) i.e., `INSERT_NODE_AFTER(n, a)`.

We point *next* of the new node (n) to *next* of the node *a*.

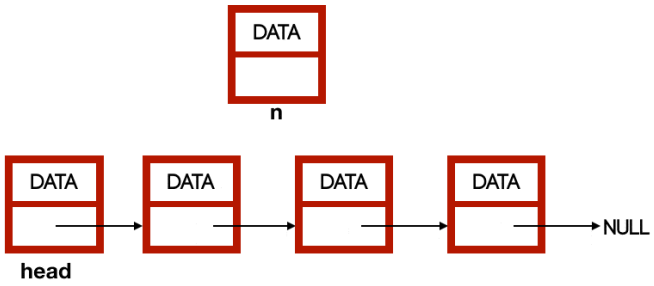



```
n.next = a.next
```

After this, we point *next* of the *a* to the new node.




```
a.next = n
```




```
INSERT_NODE_AFTER(n, a)
    n.next = a.next
    a.next = n
```

Now, we are able to create a linked list and also add nodes to it. Let's complete this chapter by learning to delete a node from a linked list.

Deleting a Node from a Linked List

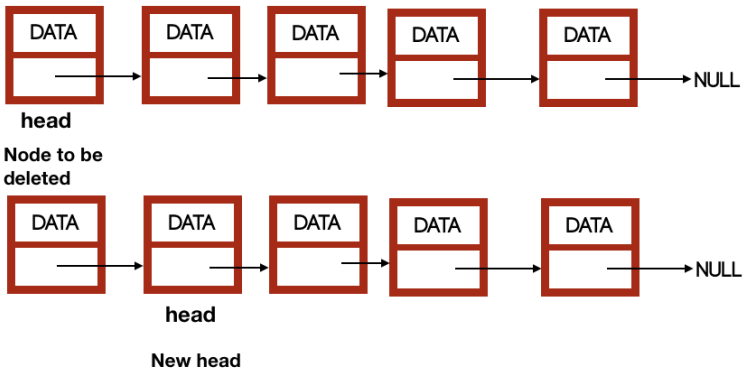
We delete any node of a linked list by connecting the predecessor node of the node to be deleted by the successor node of the node. For example, if we have a linked list a → b → c, then to delete the node 'b', we will connect 'a' to 'c' i.e., a → c. But this will make the node 'b' inaccessible and this type of inaccessible nodes are called **garbage**.

We might need to clean this garbage ourself in some languages like C by using the free function (https://www.codesdope.com/c-dynamic-memory/) while some languages like Java does it automatically. In the pseudocode, we will assume that a language does it automatically. However, you can see the full code in C, Java and Python at the end of this chapter.

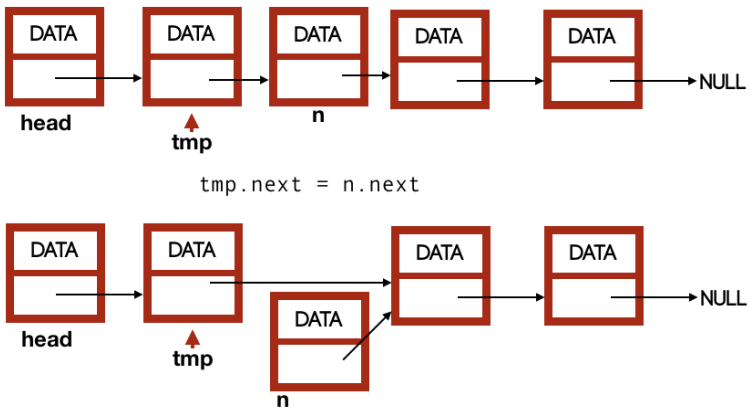
```
DEL(L, n)
     tmp = L.head
     if tmp == n // node to be deleted is head
          L.head = n.next

     else // node to be deleted is not head
          while(tmp != null)
               if tmp.next == n
                    tmp.next = n.next //linking
                    break
               tmp = tmp.next
```

We will first iterate to the node previous to the node *n*. So, we will first start from the head of the linked list - `tmp = L.head`.

If the node we are going to delete is the head of the linked list (`if tmp == n`), then we will just update the head pointer - `L.head = n.next`.



Otherwise, we will iterate to the node previous to the node *n* - `if tmp.next == n` and then link the node previous of the node to be deleted to the next of it (`tmp.next = n.next`).



C **Python** **Java**

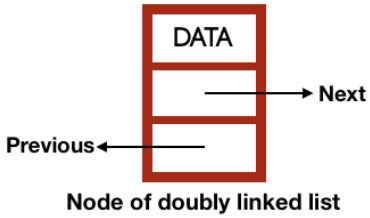
```c
#include <stdio.h>
#include <stdlib.h>

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

  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_beginning(linked_list *l, node *n) {
  n->next = l->head;
  l->head = n;
}

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

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

  temp->next = n;
}

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

//function to delete
void del(linked_list *l, node *n) {
  node *temp = l->head;
  if(temp == n) { //node to be deleted is head
    l->head = n->next;
    free(n);
  }
  else { //node to be deleted is not head
    while(temp != NULL) {
      if(temp->next == n) { //node previous to node to be deleted
```

```
        temp->next = n->next;
        free(n);
        break; //breaking the loop after deleting the node
      }
      temp = temp->next;
    }
  }
}

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_beginning(l, z);
  z = new_node(-10);
  insert_at_beginning(l, z);

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

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

  traversal(l);

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

  return 0;
}
```

Analysis of Linked List

A linked list is a linear data structure like an array but the size of the linked can be changed, unlike an array. However, to access an element from a linked list, we might traverse to the entire list, so accessing an element takes $O(n)$ time. Inserting is also $O(n)$ operation because we need to traverse the entire list when the node to be operated is at the last of the linked list.

We learned about a singly linked list i.e., we had a single link between two nodes. We can also have one more link between nodes to point to the previous node of any node which is called a doubly linked list. We are going to study about it in the next chapter (/course/data-structures-doubly-linked-lists/).



Node of doubly linked list

You can always check the articles in further reading to learn more about any topic. Also, make sure to download the BlogsDope app (https://play.google.com/store/apps/details?id=com.blogsdope) to stay tuned with us.

> “ Somewhere, something incredible is waiting to be known ”
>
> - Carl Sagan

PREV (/course/data-structures-introduction/) (/course/data-structures-doubly-linked-lists/) NEXT

Further Readings

- ➔ Linked lists in C (Singly linked list) (/blog/article/linked-lists-in-c-singly-linked-list/)
- ➔ Linked list traversal using while loop and recursion. (/blog/article/linked-list-traversal-using-while-loop-and-recursi/)
- ➔ Concatenating two linked lists in C. (/blog/article/concatenating-two-linked-lists-in-c/)
- ➔ Inserting a new node in a linked list in C. (/blog/article/inserting-a-new-node-in-a-linked-list-in-c/)
- ➔ Array vs Linked list in C (/blog/article/array-vs-linked-list-in-c/)
- ➔ C++ : Linked lists in C++ (Singly linked list) (/blog/article/c-linked-lists-in-c-singly-linked-list/)
- ➔ Linked list traversal using loop and recursion in c++ (/blog/article/linked-list-traversal-using-loop-and-recursion-in-/)
- ➔ C++ : Concatenating two linked lists in C++ (/blog/article/c-concatenating-two-linked-lists-in-c/)
- ➔ Inserting a new node to a linked list in C++ (/blog/article/inserting-a-new-node-to-a-linked-list-in-c/)
- ➔ C++: Deletion of a given node from a linked list in C++ (/blog/article/c-deletion-of-a-given-node-from-a-linked-list-in-c/)