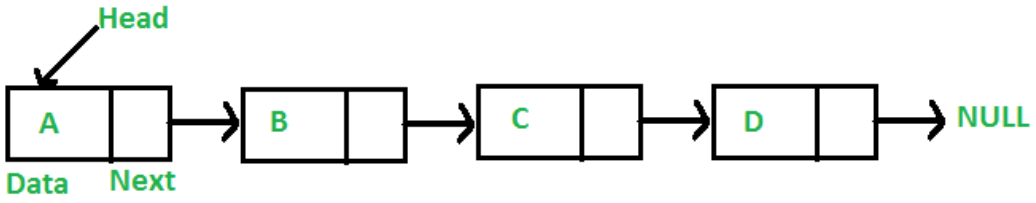
Linked List | Introduction

> *Linked Lists are linear or sequential data structures in which elements are stored at non contiguous memory locations and are linked to each other using pointers.*

Like arrays, linked lists are also linear data structures but in linked lists elements are not stored at contiguous memory locations. They can be stored anywhere in the memory but for sequential access, the nodes are linked to each other using pointers.

Each element in a linked list consists of two parts:

- **Data**: This part stores the data value of the node. That is the information to be stored at the current node.
- **Next Pointer**: This is the pointer variable or any other variable which stores the address of the next node in the memory.



Advantages of Linked Lists over Arrays : Arrays can be used to store linear data of similar types, but arrays have the following limitations:

1. The size of the arrays is fixed, so we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage. On the other hand, linked lists are dynamic and the size of the linked list can be incremented or decremented during runtime.
2. Inserting a new element in an array of elements is expensive, because a room has to be created for the new elements, and to create room, existing elements have to shift. For

example, in a system, if we maintain a sorted list of IDs in an array id[ ].

```
id[] = [1000, 1010, 1050, 2000, 2040].
```

And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000). Deletion is also expensive with arrays unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved. On the other hand, nodes in linked lists can be inserted or deleted without any shift operation and is efficient than that of arrays.

Disadvantages of Linked Lists:

1. Random access is not allowed in Linked Lists. We have to access elements sequentially starting from the first node. So, we cannot do a binary search with linked lists efficiently with its default implementation. Therefore, lookup or search operation is costly in linked lists in comparison to arrays.
2. Extra memory space for a pointer is required with each element of the list.
3. Not cache-friendly. Since array elements are present at contiguous locations, there is a locality of reference which is not there in the case of linked lists.

Representation of Linked Lists

A linked list is represented by a pointer to the first node of the linked list. The first node is called the head node of the list. If the linked list is empty, then the value of the head node is NULL.

Each node in a list consists of at least two parts:

1. data
2. Pointer (Or Reference) to the next node

In C/C++, we can represent a node using structure. Below is an example of a linked list node with integer data.

```
struct Node
{
    int data;
```

```
    struct Node* next;
};
```

In Java, LinkedList can be represented as a class, and the Node as a separate class. The LinkedList class contains a reference of the Node class type.

```
class LinkedList
{
    Node head;  // head of list

    /* Linked list Node*/
    class Node
    {
        int data;
        Node next;

        // Constructor to create a new node
        // Next is by default initialized
        // as null
        Node(int d) {data = d;}
    }
}
```

Below is a sample program in both C/C++ and Java to create a simple linked list with 3 Nodes.

```
// A simple C/C++ program to introduce
// a linked list
#include<bits/stdc++.h>
using namespace std;

// Linked List Node
struct Node
{
    int data; // Data
    struct Node *next;  // Pointer
};
```

```
// Program to create a simple linked
// list with 3 nodes
int main()
{
    struct Node* head = NULL;
    struct Node* second = NULL;
    struct Node* third = NULL;

    // allocate 3 nodes in the heap
    head = new Node;
    second = new Node;
    third = new Node;

    /* Three blocks have been allocated dynamically.
        We have pointers to these three blocks as first,
          second and third
          head         second         third
                  |             |             |
                  |             |             |
          +---+-----+     +----+----+     +----+----+
                 | # | # |     | # | # |     | # | # |
          +---+-----+     +----+----+     +----+----+

          # represents any random value.
          Data is random because we haven't assigned
          anything yet */

    head->data = 1; //assign data in first node

    // Link first node with the second node
    head->next = second;

    /*  data has been assigned to data part of first
            block (block pointed by head). And next
            pointer of first block points to second.
            So they both are linked.
            head         second         third
                    |             |             |
```

```
                |             |             |
        +---+---+     +----+----+     +-----+----+
        | 1 | o----->| # | # |     | # | # |
        +---+---+     +----+----+     +-----+----+
        */
        // assign data to second node
        second->data = 2;

// Link second node with the third node
    second->next = third;

/*  data has been assigned to data part of second
        block (block pointed by second). And next
        pointer of the second block points to third
        block. So all three blocks are linked.
        head         second         third
                |             |             |
                |             |             |
        +---+---+     +---+---+     +----+----+
        | 1 | o----->| 2 | o-----> | # | # |
        +---+---+     +---+---+     +----+----+
        */

third->data = 3; //assign data to third node
third->next = NULL;

/*  data has been assigned to data part of third
block (block pointed by third). And next pointer
of the third block is made NULL to indicate
that the linked list is terminated here.
We have the linked list ready.
head
        |
        |
+---+---+     +---+---+     +----+------+
| 1 | o----->| 2 | o-----> | 3 | NULL |
+---+---+     +---+---+     +----+------+
Note that only head is sufficient to represent
```

```
    the whole list. We can traverse the complete
    list by following next pointers. */

    return 0;
}
```

Linked List Traversal : In the previous program, we have created a simple linked list with three nodes. Let us traverse the created list and print the data of each node. For traversal, let us write a general purpose function printList( ) that prints any given list.

```
// A simple C/C++ program to introduce
// a linked list
#include<bits/stdc++.h>
using namespace std;

// Linked List Node
struct Node
{
    int data; // Data
    struct Node *next;  // Pointer};

// This function prints contents of linked list
// starting from  the given node
void printList(Node *node)
{
    while (node != NULL)
    {
        cout<<node->data<<" ";
        node = node->next;
    }
}

// Program to create a simple linked
// list with 3 nodes int main()
{
    struct Node* head = NULL;
    struct Node* second = NULL;
```

```
    struct Node* third = NULL;

    // allocate 3 nodes in the heap
    head = new Node;
    second = new Node;
    third = new Node;


    head->data = 1; //assign data in first node

    // Link first node with the second node
    head->next = second;

    // assign data to second node
    second->data = 2;

    // Link second node with the third node
    second->next = third;

    third->data = 3; //assign data to third node
    third->next = NULL;

    // Print the linked list
    printList(head);

    return 0;
}
```

Output:

```
1 2 3
```