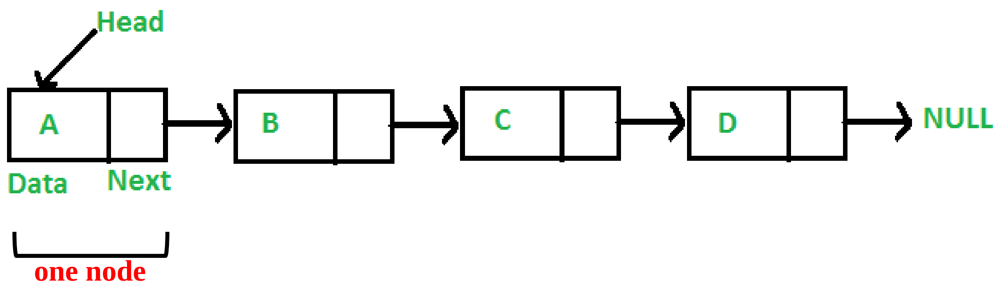
Linked List | Set 1 (Introduction)

Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at a contiguous location; the elements are connected/linked using pointers.



Why Linked List?

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. In addition, generally, the allocated memory is equal to the upper limit irrespective of the usage.

2) Inserting a new element in an array of elements is expensive because the room has to be created for the new elements and to create room existing elements have to be shifted.

Advantages over arrays
1) Dynamic size
2) Ease of insertion/deletion

Drawbacks:
1) Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists efficiently with its default implementation.

2) Extra memory space for a pointer is required with each element of the list.

Representation:
A linked list is represented by a pointer to the first node of the linked list. The first node is called the head. If the linked list is empty, then the value of the head 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, we can represent a node using structures.

First Example:

Let us create a simple linked list with 3 nodes.

```
// A simple C program to introduce
// a linked list
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct 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 = (struct Node*)malloc(sizeof(struct Node));
    second = (struct Node*)malloc(sizeof(struct Node));
    third = (struct Node*)malloc(sizeof(struct Node));

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

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

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

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

    // the second node
    /* data has been assigned to the data part of the first
```

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

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

// third node must set to NULL because it is last 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 move over 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 program for traversal of a linked list
#include <stdio.h>
#include <stdlib.h>

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

// This function prints contents of linked list starting from
// the given node
void printList(struct Node* n)
{
    while (n != NULL) {
        printf(" %d ", n->data);
        n = n->next;
    }
}

int main()
{
    struct Node* head = NULL;
    struct Node* second = NULL;
    struct Node* third = NULL;

    // allocate 3 nodes in the heap
    head = (struct Node*)malloc(sizeof(struct Node));
    second = (struct Node*)malloc(sizeof(struct Node));
    third = (struct Node*)malloc(sizeof(struct Node));

    head->data = 1; // assign data in first node
    head->next = second; // Link first node with second

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

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

    printList(head);

    return 0;}
```

Output:

```
 1   2   3
```