We have combined Classroom and Theory tab and created a new Learn tab for easy access. You can access Classroom and Theory from the left panel.

Learn

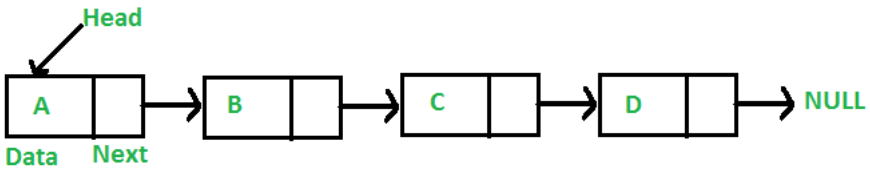
Linked List | Introduction

> *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.

C++

```
// 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;
```

```
26      third = new Node;
27
28      /* Three blocks have been allocated dynamically.
29          We have pointers to these three blocks as first,
30          second and third
```

Run

Java

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.

C++

```
1
2  // A simple C/C++ program to introduce
3  // a linked list
4
5  #include<bits/stdc++.h>
6  using namespace std;
7
8  // Linked List Node
9  struct Node
10 {
11     int data; // Data
12     struct Node *next;  // Pointer
13 };
14
15 // This function prints contents of linked list
16 // starting from  the given node
17 void printList(Node *node)
18 {
19     while (node != NULL)
20     {
21         cout<<node->data<<" ";
22         node = node->next;
23     }
24 }
25
26 // Program to create a simple linked
27 // list with 3 nodes
28 int main()
29 {
30     struct Node* head = NULL;
```

Run

Java

Output:

```
1 2 3
```

- Insertion in Singly Linked Lists

Given the head node of a linked list, the task is to insert a new node in this already created linked list.

There can be many different situations that may arise while inserting a node in a linked list. Three most frequent situations are:

1. Inserting a node at the start of the list.
2. Inserting a node after any given node in the list.
3. Inserting a node at the end of the list.

We have seen that a linked list node can be represented using structures and classes as:

C++

```
1
2  // A linked list node
3  struct Node
```

```
4 {
5   int data;
6   struct Node *next;
7 };
8
```

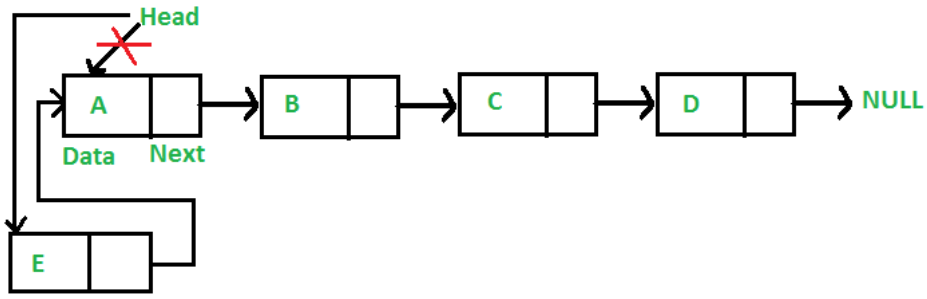
Run

Java

Let us now take a look at each of the above three listed methods of inserting a node in the linked list:

- **Inserting a Node at Beginning**: Inserting a node at the start of the list is a *four-step* process. In this process, the new node is always added before the head of the given Linked List and the newly added node becomes the new head of the Linked List.

  For example, if the given Linked List is *10->15->20->25* and we add an item **5** at the front, then the Linked List becomes *5->10->15->20->25*.



  Let us call the function that adds a new node at the front of the list as *push()*. The push() function must receive a pointer to the head node because the function must change the head pointer to point to the new node.

  Below is the 4 step process of adding a new node at the front of Linked List declared at the beginning of this post:

C++

```
1
2  /* Given a reference (pointer to pointer) to the
3     head of a list and an int, insert a new node
4     on the front of the list. */
5
6  void push(struct Node** head_ref, int new_data)
7  {
8      /* 1. allocate node */
9      Node* new_node = new Node;
10
11     /* 2. put in the data  */
12     new_node->data  = new_data;
13
14     /* 3. Make next of new node as head */
15     new_node->next = (*head_ref);
16
17     /* 4. move the head to point to the new node */
18     (*head_ref)    = new_node;
19 }
20
```

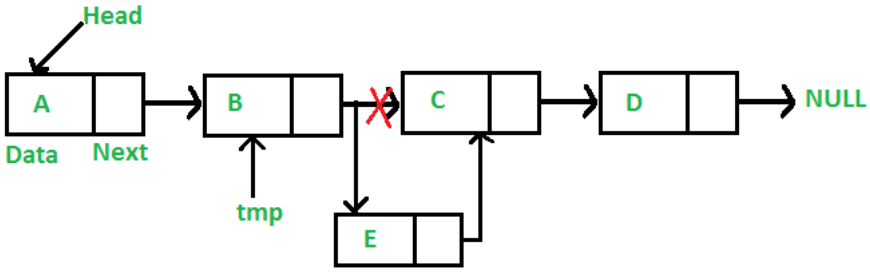
Run

Java

  *The **time complexity** of inserting a node at start of the List is O(1).*

- **Inserting a Node after given Node**: Inserting a Node after a given Node is also similar to the above process. One has to first allocate the new Node and change the next pointer of the newly created node to the next of the previous node and the next pointer of the previous node to point to the newly created node.

  Below is the pictorial representation of the complete process:



Let us call the function that adds a new node after a given node in the list as insertAfter(). The *insertAfter()* function must receive a pointer to the previous node after which the new node is to be inserted.

Below is the complete process of adding a new node after a given node in the Linked List declared at the beginning of this post:

C++

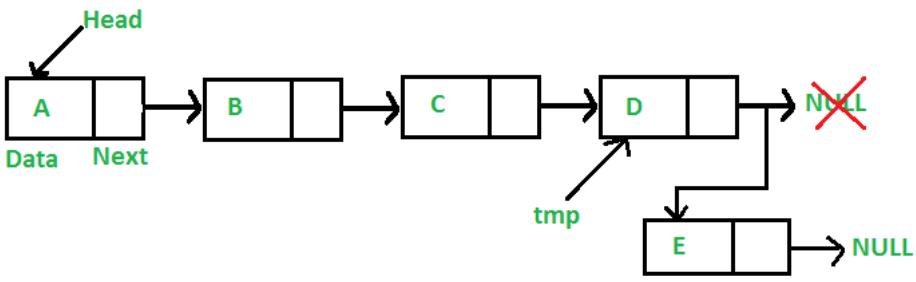
```
/* Given a node prev_node, insert a new node
   after the given prev_node */

void insertAfter(struct Node* prev_node, int new_data)
{
    /* 1. check if the given prev_node is NULL */
    if (prev_node == NULL)
    {
       printf("the given previous node cannot be NULL");
       return;
    }

    /* 2. allocate new node */
    Node* new_node = new Node;

    /* 3. put in the data  */
    new_node->data  = new_data;

    /* 4. Make next of new node as next of prev_node */
    new_node->next = prev_node->next;

    /* 5. move the next of prev_node as new_node */
    prev_node->next = new_node;
}

```

Run

Java

*The **time complexity** of inserting a node at start of the List is O(1).*

- **Inserting a Node at the End**: Inserting a new Node at the end of a Linked List is generally a six step process in total. The new node is always added after the last node of the given Linked List. For example if the given Linked List is *5->10->15->20->25* and we add an item 30 at the end, then the Linked List becomes *5->10->15->20->25->30*.



Since a Linked List is typically represented by its head, we have to first traverse the list till the end in order to get the pointer pointing

to the last node and then change the next of last node to the new node.

Below is the complete 6 step process of adding a new Node at the end of the list:

C++

```
/* Given a reference (pointer to pointer) to
   the head of a list and an int, appends a new
   node at the end  */

void append(struct Node** head_ref, int new_data)
{
    /* 1. allocate node */
    Node* new_node = new Node;

    struct Node *last = *head_ref;  /* used in step 5*/

    /* 2. put in the data  */
    new_node->data  = new_data;

    /* 3. This new node is going to be the last node,
          so make next of it as NULL*/
    new_node->next = NULL;

    /* 4. If the Linked List is empty, then make
          the new node as head */
    if (*head_ref == NULL)
    {
       *head_ref = new_node;
       return;
    }

    /* 5. Else traverse till the last node */
    while (last->next != NULL)
        last = last->next;
```

Run

Java

*The **time complexity** of this operation is **O(N)** where N is the number of nodes in the Linked List as one has to traverse the complete list in order to find the last node.*

− Deletion in Singly Linked Lists

Given the head pointer pointing to the Head of a singly linked list and a node to be deleted from the list. Delete the given node from the Linked List.

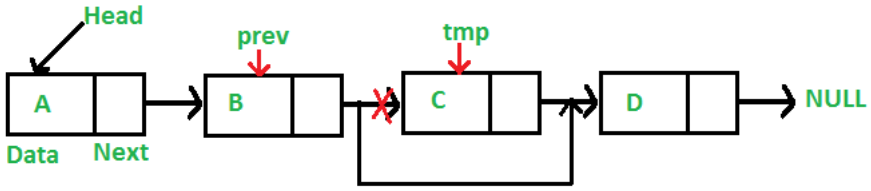
Like inserting a node in a linked list, there can be many situations when it comes to deleting a node from a Linked List. Some of the most frequent situations are:

- Given the data value of a node, delete the first occurrence of that data in the list.
- Given the position of a node, delete the node present at the given position in the list.
- Given a pointer to the node to be deleted, delete the node.

Let us look at each one of these situations and their solutions with complete explanations:

- **Deleting the first occurrence of a given data value:** Deleting a node by its value can be done by traversing the list till the node just before the node with the value as the given key. Once the node just before the node to be deleted is found. Update its next pointer to point to the next of its next node.

  That is:
  1. Find previous node of the node to be deleted.
  2. Change the next of previous node.
  3. Free memory for the node to be deleted.



Below is the implementation of this approach:

C++

```
/* Given a reference (pointer to pointer) to the head of a list
   and a key, deletes the first occurrence of key in linked list */

void deleteNode(struct Node **head_ref, int key)
{
    // Store head node
    struct Node* temp = *head_ref, *prev;

    // If head node itself holds the key to be deleted
    if (temp != NULL && temp->data == key)
    {
        *head_ref = temp->next;   // Changed head
        free(temp);               // free old head
        return;
    }

    // Search for the key to be deleted, keep track of the
    // previous node as we need to change 'prev->next'
    while (temp != NULL && temp->data != key)
    {
        prev = temp;
        temp = temp->next;
    }

    // If key was not present in linked list
    if (temp == NULL) return;

    // Unlink the node from linked list
    prev->next = temp->next;
```

Run

Java

*The **time complexity** of this operation in worst case is O(N) where N is the number of nodes in the Linked List.*

- **Deleting a node at a given position**: If the node to be deleted is the root node, we can simply delete it by updating the head pointer to point to the next of the root node. To delete a node present somewhere in between, we must have the pointer to the node previous to the node to be deleted. So if the position is not zero, run a loop position-1 times and get the pointer to the previous node and follow the method discussed in the first situation above to delete the node.

Below is the implementation of this approach:

C++

```
/* Given a reference (pointer to pointer) to the head of a list
   and a position, delete the node at the given position */
void deleteNode(struct Node **head_ref, int position)
{
   // If linked list is empty
   if (*head_ref == NULL)
      return;

   // Store head node
   struct Node* temp = *head_ref;

    // If head needs to be removed
```

```
    // If head needs to be removed
    if (position == 0)
    {
        *head_ref = temp->next;   // Change head
        free(temp);               // free old head
        return;
    }

    // Find previous node of the node to be deleted
    for (int i=0; temp!=NULL && i<position-1; i++)
         temp = temp->next;

    // If position is more than number of ndoes
    if (temp == NULL || temp->next == NULL)
         return;

    // Node temp->next is the node to be deleted
    // Store pointer to the next of node to be deleted
```

Run

Java

*The **time complexity** of this operation in worst case is O(N) where N is the number of nodes in the Linked List.*

- **Deleting a node whose pointer is given**: In this case, a pointer is given which is pointing to a particular node in the linked list and the task is to delete that particular node.

  This can be done by following a similar approach as in the above two cases, by first finding the node just previous to it and updating its next pointer. The time complexity of this would be again O(N).

  However, this particular case can be solved with ***O(1)** time complexity* if the pointer to the node to be deleted is given.

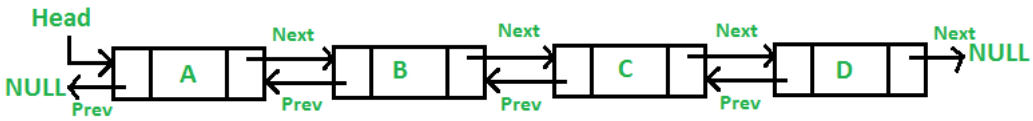
  The ***efficient solution*** is to copy the data from the next node to the node to be deleted and delete the next node. Suppose the node to be deleted is *node_ptr*, it can be deleted as:

```
// Find next node using next pointer
struct Node *temp  = node_ptr->next;

// Copy data of next node to this node
node_ptr->data  = temp->data;

// Unlink next node
node_ptr->next  = temp->next;

// Delete next node
free(temp);
```

  Note: *This solution fails if the node to be deleted is the last node of the List.*

− Doubly Linked Lists

Similar to *Singly Linked Lists*, ***Doubly Linked Lists*** are also a sequential data structure with the only difference that the doubly linked lists contain two pointers instead of one to store the address of both next node and previous node respectively.



As you can see in the above image:

- Each node contains two pointers, one pointing to the next node and the other pointing to the previous node.
- The prev of Head node is NULL, as there is no previous node of Head.

- The next of last node is NULL, as there is no node after the last node.

Below is the sample Doubly Linked List node in C++ and Java:

C++

```
/* Node of a doubly linked list */
struct Node {
    int data;
    struct Node* next; // Pointer to next node in DLL
    struct Node* prev; // Pointer to previous node in DLL
};
```

Run

Java

Advantages of doubly linked lists over singly linked list:

1. A DLL can be traversed in both forward and backward directions.
2. The delete operation in DLL is more efficient if the pointer to the node to be deleted is given.
3. We can quickly insert a new node before a given node.
4. In a singly linked list, to delete a node, a pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In DLL, we can get the previous node using the previous pointer.

Disadvantages of doubly linked lists over singly linked list:

1. Every node of DLL requires extra space for a previous pointer.
2. All operations require an extra pointer previous to be maintained. For example, an insertion, we need to modify previous pointers together with next pointers.

Creating and Traversing a Doubly Linked List

Creating and Traversing a doubly linked list is almost similar to that of the singly linked lists. The only difference is that in doubly linked lists we need to maintain an extra previous pointer for every node while creating the list.

Below is the complete program to create and traverse a Doubly Linked List in both C++ and Java:

C++

```
// A complete working C++ program to demonstrate
// Doubly Linked Lists

#include <bits/stdc++.h>
using namespace std;

// A linked list node
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};

// This function prints contents of Doubly linked
// list starting from the given node
void printList(struct Node* node)
{
    struct Node* last;

    // Traverse the linked list in forward direction
    // using the next node's pointer present
    // at each node
    cout<<"\nTraversal in forward direction \n";
    while (node != NULL) {
        cout<<node->data<<" ";
        last = node;
        node = node->next;
    }

```

Run

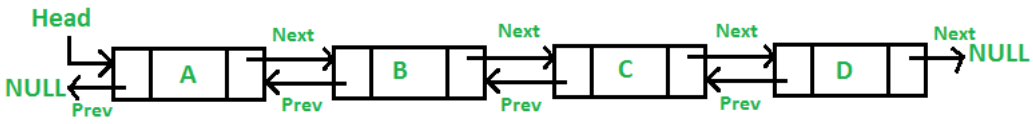
Java

Output:

```
Created DLL is:
Traversal in forward direction
6 7 1 4
Traversal in reverse direction
4 1 7 6
```


− XOR Linked Lists

XOR Linked Lists are Memory Efficient implementation of *Doubly Linked Lists*. An ordinary Doubly Linked List requires space for two address fields to store the addresses of previous and next nodes. A memory-efficient version of Doubly Linked List can be created using only one space for the address field with every node. This memory efficient Doubly Linked List is called XOR Linked List or Memory Efficient Linked List as the list uses bitwise XOR operation to save space for one address. In the XOR linked list instead of storing actual memory addresses every node stores the XOR of addresses of previous and next nodes.



Consider the above Doubly Linked List. Following are the Ordinary and XOR (or Memory Efficient) representations of the Doubly Linked List.

Ordinary Representation:

- *Node A:* prev = NULL, next = add(B) // previous is NULL and next is address of B
- *Node B:* prev = add(A), next = add(C) // previous is address of A and next is address of C
- *Node C:* prev = add(B), next = add(D) // previous is address of B and next is address of D
- *Node D:* prev = add(C), next = NULL // previous is address of C and next is NULL

XOR List Representation: Let us call the address variable in XOR representation as ***npx*** (XOR of next and previous)

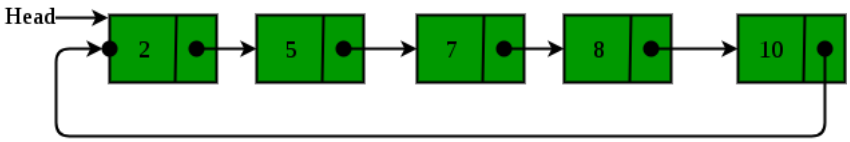
- *Node A*:
  npx = 0 XOR add(B) // bitwise XOR of zero and address of B
- *Node B*:
  npx = add(A) XOR add(C) // bitwise XOR of address of A and address of C
- *Node C*:
  npx = add(B) XOR add(D) // bitwise XOR of address of B and address of D
- *Node D*:
  npx = add(C) XOR 0 // bitwise XOR of address of C and 0

Traversal of XOR Linked List: We can traverse the XOR list in both forward and reverse directions. While traversing the list we need to remember the address of the previously accessed node in order to calculate the next node's address. For example, when we are at node C, we must have the address of B. XOR of add(B) and npx of C gives us the add(D). The reason is simple: npx(C) is "add(B) XOR add(D)". If we do xor of npx(C) with add(B), we get the result as "add(B) XOR add(D) XOR add(B)" which is "add(D) XOR 0" which is "add(D)". So we have the address of the next node. Similarly, we can traverse the list in a backward direction.

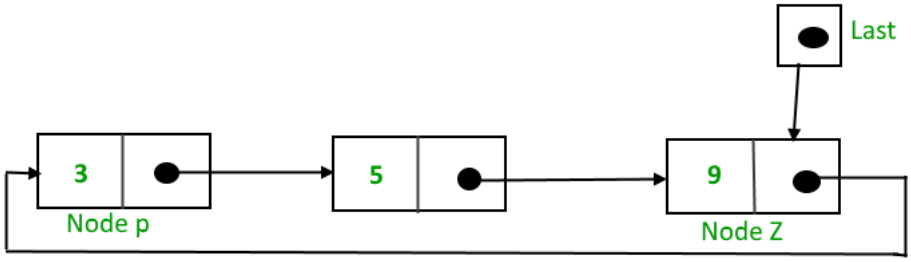
− Circular Linked Lists

> *A **circular linked** list is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.*

Below is a pictorial representation of Circular Linked List:



Implementation:
To implement a circular singly linked list, we take an external pointer that points to the last node of the list. If we have a pointer last pointing to the last node, then *last -> next* will point to the first node.



The pointer ***last*** points to node **Z** and ***last -> next*** points to the node **P**.

Why have we taken a pointer that points to the last node instead of first node?

For insertion of node in the beginning we need to traverse the whole list. Also, for insertion at the end, the whole list has to be traversed. If instead of start pointer we take a pointer to the last node then in both the cases there won't be any need to traverse the whole list. So, insertion in the begging or at the end takes constant time irrespective of the length of the list.

Below is a sample program to create and traverse in a Circular Linked List in both Java and C++:

C++

```
// A complete C++ program to demonstrate the
// working of Circular Linked Lists

#include<bits/stdc++.h>
using namespace std;

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

// Function to add a node at the end of a
// Circular Linked List
struct Node *addEnd(struct Node *last, int data)
{
    if (last == NULL)
    {
        // Creating a node dynamically.
        struct Node *temp = new Node;

        // Assigning the data.
        temp -> data = data;
        last = temp;

        // Creating the link.
        last -> next = last;

```

Run

Java

Output:

```
6 4 2 8 12 10
```

Advantages of Circular Linked Lists:

1. Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.
2. Useful for implementation of a queue. Unlike this implementation, we don't need to maintain two pointers for front and rear if we use a circular linked list. We can maintain a pointer to the last inserted node and the front can always be obtained as the next of last.
3. Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.
4. Circular Doubly Linked Lists are used for implementation of advanced data structures like Fibonacci Heap.

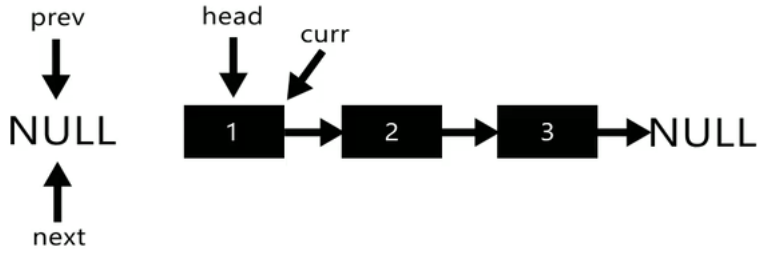
- Sample Problems on Linked List

Problem #1 : Reverse a Linked List

Description - Given a pointer to the head node of a linked list, the task is to reverse the linked list. We need to reverse the list by changing links between nodes.

```
Input: Head of following linked list
       1->2->3->4->NULL
Output: Linked list should be changed to,
       4->3->2->1->NULL
```

Three Pointers Solution : We will be using three auxiliary three pointers **prev, current and next** to reverse the links of the linked list. The solution can be understood by the below animation, how links are reversed.




```
while (current != NULL)
     {
          next   = current->next;
          current->next = prev;
          prev = current;
          current = next;
     }
     *head_ref = prev;
```

Pseudo Code

```
void reverse(Node* head)
{
    // Initialize current, previous and
    // next pointers
    Node *current = head;
    Node *prev = NULL, *next = NULL
    while (current != NULL)
    {
        // Store next
        next = current->next

        // Reverse current node's pointer
        current->next = prev

        // Move pointers one position ahead.
        prev = current
        current = next
    }
    head = prev
}
```


Two Pointers Solution : We will be using auxiliary two pointers **current and next** to reverse the links of the linked list. This is little bit tricky solution. Try out with examples-

Pseudo Code

```
void reverse(Node* head)
{
    // Initialize current, previous and
    // next pointers
    Node *current = head;
    Node *prev = NULL, *next = NULL
    while (current != NULL)
    {
        // Store next
        next = current->next

        // Reverse current node's pointer
        current->next = prev

        // Move pointers one position ahead.
        prev = current
        current = next
    }

    head = prev
}
```

Recursive Solution : In this approach of reversing a linked list by passing a single pointer what we are trying to do is that we are making the previous node of the current node as his next node to reverse the linked list.

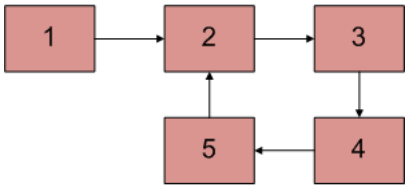
1. We return the pointer of next node to his previous(current) node and then make the previous node as the next node of returned node and then returning the current node.
2. We first traverse till the last node and making the last node as the head node of reversed linked list and then applying the above procedure in the recursive manner.

Pseudo Code

```
Node* reverse(Node* node)
{
    if (node == NULL) :
        return NULL
    if (node->next == NULL) :
        head = node
        return node
    Node* temp = reverse(node->next)
    temp->next = node
    node->next = NULL
    return node
}
```

Problem #2 : Detect Loop in a Linked List

Description : Given a linked list, check if the the linked list has loop or not. Below diagram shows a linked list with a loop.



(https://www.geeksforgeeks.org/wp-content/uploads/2009/04/Linked-List-Loop.gif) **Hashing Solution :**Traverse the list one by one and keep putting the node addresses in a Hash Table. At any point, if NULL is reached then return false and if next of current node points to any of the previously stored nodes in Hash then return true.

Pseudo Code

```
bool detectLoop(Node* h)
{
    seen //HashMap
    while (h != NULL)
    {
        // If this node is already present
        // in hashmap it means there is a cycle
        // (Because you we encountering the
        // node for the second time).
        if (seen.find(h) == True)
            return true
        // If we are seeing the node for
        // the first time, insert it in hash
        seen.insert(h)
        h = h->next
    }
    return false
}
```

Floyd's Cycle-Finding Algorithm: This is the fastest method. Traverse linked list using two pointers. Move one pointer by one step and another pointer by two-step. If these pointers meet at the same node then there is a loop. If pointers do not meet then linked list doesn't have a loop.

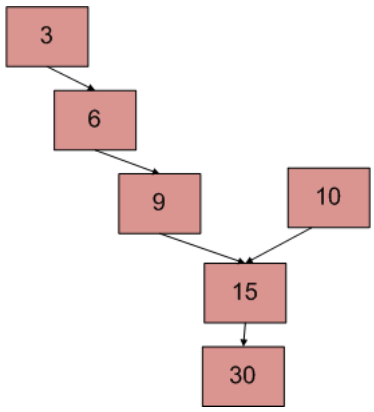
You may visualize the solution as two runners are running on a circular track, If they are having different speeds they will definitely meet up on circular track itself.

Pseudo Code

```
bool detectloop(Node* head)
{
    Node *slow_p = head, *fast_p = head

    while (slow_p && fast_p && fast_p->next)
    {
        slow_p = slow_p->next
        fast_p = fast_p->next->next
        if (slow_p == fast_p)
            return true
    }
    return false
}
```

Problem #3 : Find Intersection Point of Two Linked List

Description : There are two singly linked lists in a system. By some programming error, the end node of one of the linked list got linked to the second list, forming an inverted Y shaped list. Write a program to get the point where two linked list merge.



Above diagram shows an example with two linked list having 15 as intersection point.
Naive Solutions : This solution requires modifications to basic linked list data structure. Have a visited flag with each node. Traverse the first linked list and keep marking visited nodes. Now traverse the second linked list, If you see a visited node again then there is an intersection point, return the intersecting node. This solution works in O(m+n) but requires additional information with each node. A variation of this solution that doesn't require modification to the basic data structure can be implemented using a hash. Traverse the first linked list and store the addresses of visited nodes in a hash. Now traverse the second linked list and if you see an address that already exists in the hash then return the intersecting node.

Node Count Difference Solution : Problem can be solved following these steps -

1. Get count of the nodes in the first list, let count be c1.
2. Get a count of the nodes in the second list, let count be c2.
3. Get the difference of counts d = abs(c1 – c2).
4. Now traverse the bigger list from the first node till d nodes so that from here onwards both the lists have equal no of nodes.
5. Then we can traverse both the lists in parallel till we come across a common node. (Note that getting a common node is done by comparing the address of the nodes)

Pseudo Code

```
/* function to get the intersection point of two linked
   lists head1 and head2 */
int getIntesectionNode( Node* head1, Node* head2)
{
    c1 = getCount(head1)
    c2 = getCount(head2)
    d // difference

    if(c1 > c2)
        d = c1 - c2
        return utility(d, head1, head2)
    else :
        d = c2 - c1
        return utility(d, head2, head1)
}
/* function to get the intersection point of two linked
   lists head1 and head2 where head1 has d more nodes than
   head2 */
int utility(d, Node* head1, Node* head2)
{
    Node* current1 = head1
    Node* current2 = head2

    for ( i = 0 to d-1 )
    {
        if(current1 == NULL)
            return -1
        current1 = current1->next
    }

    while(current1 !=  NULL && current2 != NULL)
    {
        if(current1 == current2)
            return current1->data
        current1= current1->next
        current2= current2->next
    }
    return -1
}
```