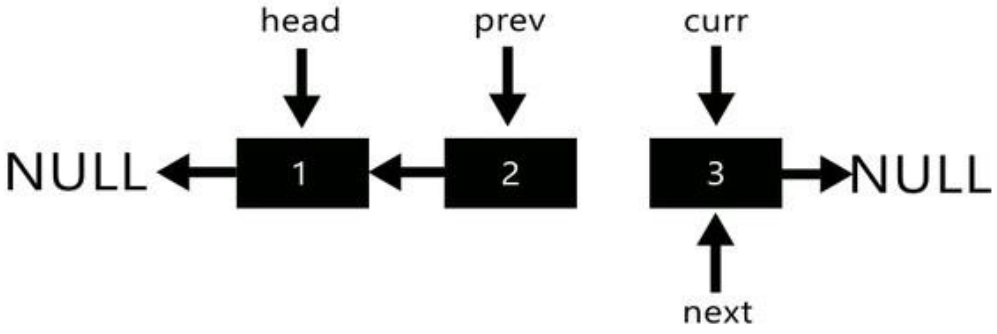
Sample Problems on Linked List

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.

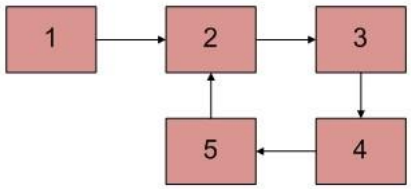
1. We return the pointer of next node to his previous(current) node and then make the previous node as the next node of returned node and then returning the current node.
2. We first traverse till the last node and making the last node as the head node of reversed linked list and then applying the above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



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.

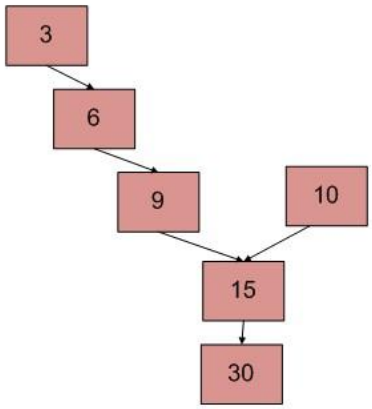
Pseudo Code

```
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
}
```