1. Reverse a linked list

Given 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.

Examples:

Input: Head of following linked list

1->2->3->4->NULL

Output: Linked list should be changed to,

4->3->2->1->NULL

Input: Head of following linked list

1->2->3->4->5->NULL

Output: Linked list should be changed to,

5->4->3->2->1->NULL

Input: NULL

Output: NULL

Input: 1->NULL

Output: 1->NULL

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

Iterative Method

1. *Initialize three pointers prev as NULL, curr as head and next as NULL.*
2. *Iterate trough the linked list. In loop, do following.*

```
// Before changing next of current,
// store next node
next = curr->next
// Now change next of current
// This is where actual reversing happens
curr->next = prev
```

```
// Move prev and curr one step forward
prev = curr
curr = next
```

Below is the implementation of the above approach:

```
// Iterative C++ program to reverse
// a linked list
#include <iostream>
using namespace std;

/* Link list node */
struct Node {
    int data;
    struct Node* next;
    Node(int data)
    {
        this->data = data;
        next = NULL;
    }
};

struct LinkedList {
    Node* head;
    LinkedList()
    {
        head = NULL;
    }

    /* Function to reverse the linked list */
    void reverse()
    {
        // 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;
    }

    /* Function to print linked list */
    void print()
    {
        struct Node* temp = head;
        while (temp != NULL) {
            cout << temp->data << " ";
            temp = temp->next;
        }
    }

    void push(int data)
    {
        Node* temp = new Node(data);
        temp->next = head;
        head = temp;
    }
};

/* Driver program to test above function*/
int main()
{
    /* Start with the empty list */
    LinkedList ll;
    ll.push(20);
    ll.push(4);
    ll.push(15);
    ll.push(85);

    cout << "Given linked list\n";
    ll.print();

    ll.reverse();

    cout << "\nReversed Linked list \n";
    ll.print();
    return 0;
}
```

Output:

```
Given linked list
```

```
85 15 4 20

Reversed Linked list

20 4 15 85
```

Time Complexity: O(n)

Space Complexity: O(1)

Recursive Method:

```
  1) Divide the list in two parts - first node and

     rest of the linked list.

  2) Call reverse for the rest of the linked list.

  3) Link rest to first.

  4) Fix head pointer
```

```
// Recursive C++ program to reverse
// a linked list
#include <iostream>
using namespace std;

/* Link list node */
struct Node {
    int data;
    struct Node* next;
    Node(int data)
    {
        this->data = data;
        next = NULL;
    }
};

struct LinkedList {
    Node* head;
    LinkedList()
    {
        head = NULL;
    }

    Node* reverse(Node* head)
```

```
    {
        if (head == NULL || head->next == NULL)
            return head;

        /* reverse the rest list and put
          the first element at the end */
        Node* rest = reverse(head->next);
        head->next->next = head;

        /* tricky step -- see the diagram */
        head->next = NULL;

        /* fix the head pointer */
        return rest;
    }

    /* Function to print linked list */
    void print()
    {
        struct Node* temp = head;
        while (temp != NULL) {
            cout << temp->data << " ";
            temp = temp->next;
        }
    }

    void push(int data)
    {
        Node* temp = new Node(data);
        temp->next = head;
        head = temp;
    }
};

/* Driver program to test above function*/
int main()
{
    /* Start with the empty list */
    LinkedList ll;
    ll.push(20);
    ll.push(4);
    ll.push(15);
    ll.push(85);

    cout << "Given linked list\n";
    ll.print();

    ll.head = ll.reverse(ll.head);
```

```
    cout << "\nReversed Linked list \n";
    ll.print();
    return 0;
}
```

Time Complexity: O(n)

Space Complexity: O(1)

A Simpler and Tail Recursive Method

Below is the implementation of this method.

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

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

void reverseUtil(Node* curr, Node* prev, Node** head);

// This function mainly calls reverseUtil()
// with prev as NULL
void reverse(Node** head)
{
    if (!head)
        return;
    reverseUtil(*head, NULL, head);
}

// A simple and tail recursive function to reverse
// a linked list.  prev is passed as NULL initially.
void reverseUtil(Node* curr, Node* prev, Node** head)
{
    /* If last node mark it head*/
    if (!curr->next) {
        *head = curr;

        /* Update next to prev node */
        curr->next = prev;
        return;
    }
```

```
    /* Save curr->next node for recursive call */
    Node* next = curr->next;

    /* and update next ..*/
    curr->next = prev;

    reverseUtil(next, curr, head);
}

// A utility function to create a new node
Node* newNode(int key)
{
    Node* temp = new Node;
    temp->data = key;
    temp->next = NULL;
    return temp;
}

// A utility function to print a linked list
void printlist(Node* head)
{
    while (head != NULL) {
        cout << head->data << " ";
        head = head->next;
    }
    cout << endl;
}

// Driver program to test above functions
int main()
{
    Node* head1 = newNode(1);
    head1->next = newNode(2);
    head1->next->next = newNode(3);
    head1->next->next->next = newNode(4);
    head1->next->next->next->next = newNode(5);
    head1->next->next->next->next->next = newNode(6);
    head1->next->next->next->next->next->next = newNode(7);
    head1->next->next->next->next->next->next->next = newNode(8);
    cout << "Given linked list\n";
    printlist(head1);
    reverse(&head1);
    cout << "\nReversed linked list\n";
    printlist(head1);
    return 0;
}
```

Output:

```
Given linked list
```

```
1 2 3 4 5 6 7 8


Reversed linked list

8 7 6 5 4 3 2 1
```

2. Find the middle of a given linked list in C and Java

Given a singly linked list, find middle of the linked list. For example, if given linked list is 1->2->3->4->5 then output should be 3.

If there are even nodes, then there would be two middle nodes, we need to print second middle element. For example, if given linked list is 1->2->3->4->5->6 then output should be 4.

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

Method 1:

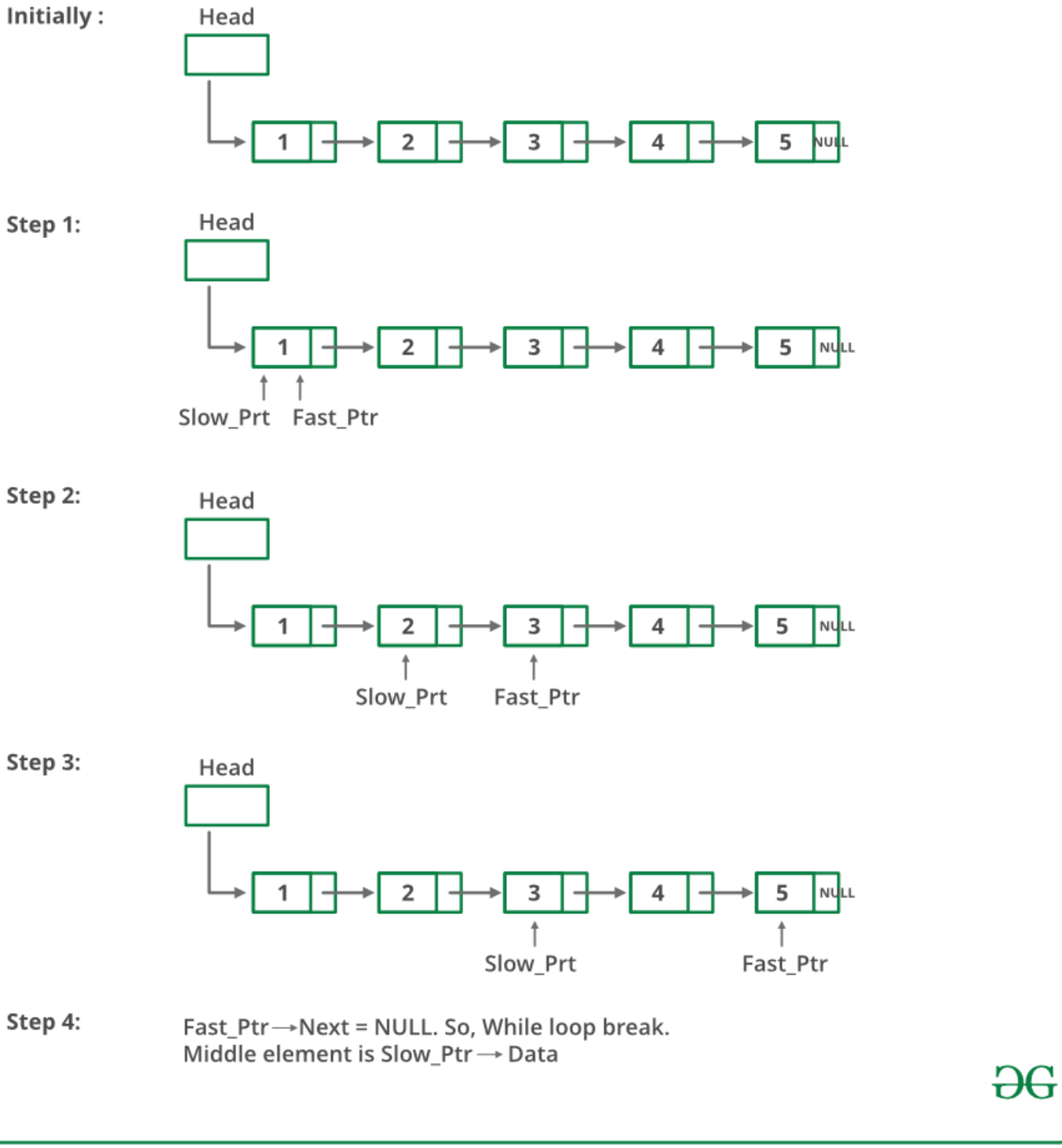
Traverse the whole linked list and count the no. of nodes. Now traverse the list again till count/2 and return the node at count/2.

Method 2:

Traverse linked list using two pointers. Move one pointer by one and other pointer by two. When the fast pointer reaches end slow pointer will reach middle of the linked list.

Below image shows how printMiddle function works in the code :

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

/* Link list node */
struct Node
{
    int data;
    struct Node* next;
};

/* Function to get the middle of the linked list*/
void printMiddle(struct Node *head)
{
    struct Node *slow_ptr = head;
    struct Node *fast_ptr = head;

    if (head!=NULL)
    {
        while (fast_ptr != NULL && fast_ptr->next != NULL)
        {
            fast_ptr = fast_ptr->next->next;
```




```
            slow_ptr = slow_ptr->next;
        }
        printf("The middle element is [%d]\n\n", slow_ptr->data);
    }
}

void push(struct Node** head_ref, int new_data)
{
    /* allocate node */
    struct Node* new_node =
        (struct Node*) malloc(sizeof(struct Node));
```

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

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

// A utility function to print a given linked list
void printList(struct Node *ptr)
{
    while (ptr != NULL)
    {
        printf("%d->", ptr->data);
        ptr = ptr->next;
    }
    printf("NULL\n");
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;
    int i;

    for (i=5; i>0; i--)
    {
        push(&head, i);
        printList(head);
        printMiddle(head);
    }

    return 0;
}
```

Output:

```
5->NULL

The middle element is [5]



4->5->NULL

The middle element is [5]
```

```
3->4->5->NULL

The middle element is [4]



2->3->4->5->NULL

The middle element is [4]



1->2->3->4->5->NULL

The middle element is [3]
```

Method 3:

Initialize mid element as head and initialize a counter as 0. Traverse the list from head, while traversing increment the counter and change mid to mid->next whenever the counter is odd. So the mid will move only half of the total length of the list.

Thanks to Narendra Kangralkar for suggesting this method.

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

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Function to get the middle of the linked list*/
void printMiddle(struct node *head)
{
    int count = 0;
    struct node *mid = head;

    while (head != NULL)
    {
        /* update mid, when 'count' is odd number */
        if (count & 1)
            mid = mid->next;

        ++count;
        head = head->next;
    }
```

```
    /* if empty list is provided */
    if (mid != NULL)
        printf("The middle element is [%d]\n\n", mid->data);
}


void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct node));

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

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref)    = new_node;
}

// A utility function to print a given linked list
void printList(struct node *ptr)
{
    while (ptr != NULL)
    {
        printf("%d->", ptr->data);
        ptr = ptr->next;
    }
    printf("NULL\n");
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;
    int i;

    for (i=5; i>0; i--)
    {
        push(&head, i);
        printList(head);
        printMiddle(head);
    }

    return 0;
```

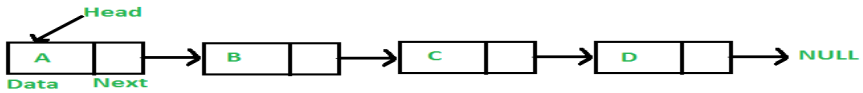
}
Output:

```
5->NULL

The middle element is [5]



4->5->NULL

The middle element is [5]



3->4->5->NULL

The middle element is [4]



2->3->4->5->NULL

The middle element is [4]



1->2->3->4->5->NULL

The middle element is [3]
```

3. Program for n’th node from the end of a Linked List

Given a Linked List and a number n, write a function that returns the value at the n’th node from the end of the Linked List.

For example, if the input is below list and n = 3, then output is “B”



Recommended: Please solve it on “*PRACTICE*” first, before moving on to the solution.

Method 1 (Use length of linked list)

1) Calculate the length of Linked List. Let the length be len.

2) Print the (len – n + 1)th node from the beginning of the Linked List.

```
// Simple C++ program to find n'th node from end
#include <bits/stdc++.h>
using namespace std;

/* Link list node */
struct Node {
    int data;
    struct Node* next;
};

/* Function to get the nth node from the last of a linked list*/
void printNthFromLast(struct Node* head, int n)
{
    int len = 0, i;
    struct Node* temp = head;

    // count the number of nodes in Linked List
    while (temp != NULL) {
        temp = temp->next;
        len++;
    }

    // check if value of n is not
    // more than length of the linked list
    if (len < n)
```

```
        return;

    temp = head;

    // get the (len-n+1)th node from the beginning
    for (i = 1; i < len - n + 1; i++)
        temp = temp->next;

    cout << temp->data;

    return;
}

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

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

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

// Driver Code
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    // create linked 35->15->4->20
    push(&head, 20);
    push(&head, 4);
    push(&head, 15);
    push(&head, 35);

    printNthFromLast(head, 4);
    return 0;
}
```

Output:

```
35
```

Following is a recursive C code for the same method. Thanks to Anuj Bansal for providing following code.

```
void printNthFromLast(struct Node* head, int n)
{
    static int i = 0;
    if (head == NULL)
        return;
    printNthFromLast(head->next, n);
    if (++i == n)
        printf("%d", head->data);
}
```

Time Complexity: O(n) where n is the length of linked list.

Method 2 (Use two pointers)

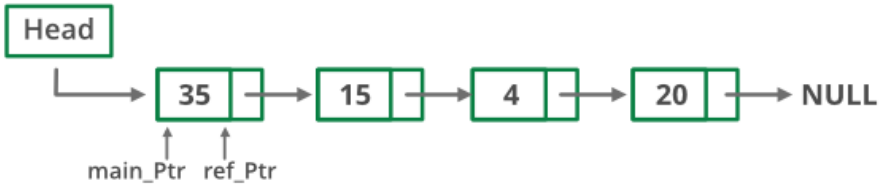
Maintain two pointers – reference pointer and main pointer. Initialize both reference and main pointers to head. First, move reference pointer to n nodes from head. Now move both pointers one by one until the reference pointer reaches the end. Now the main pointer will point to nth node from the end. Return the main pointer.

Below image is a dry run of the above approach:

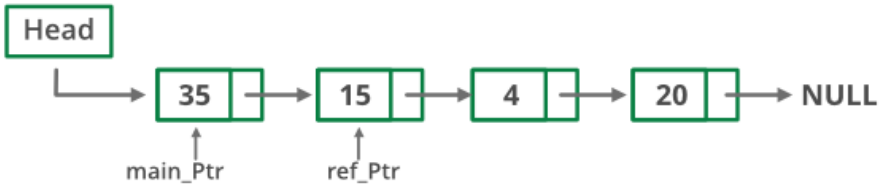
Below is the implementation of the above approach:

```
// Simple C++ program to find n'th node from end
#include<bits/stdc++.h>
using namespace std;

/* Link list node */
struct Node
{
  int data;
  struct Node* next;
};

/* Function to get the nth node from the last of a linked list*/
void printNthFromLast(struct Node *head, int n)
{
  struct Node *main_ptr = head;
  struct Node *ref_ptr = head;

  int count = 0;
  if(head != NULL)
  {
     while( count < n )
     {
        if(ref_ptr == NULL)
```

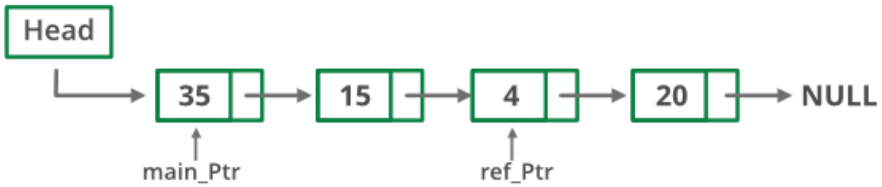
Initially : n= 3 , Count = 0



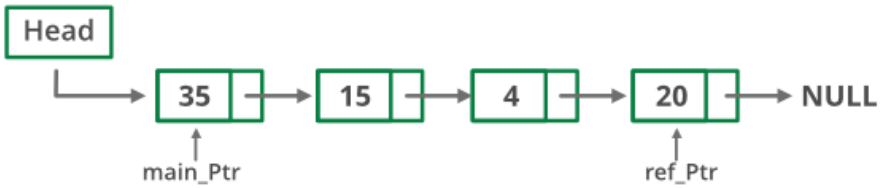
Step 1: n= 3 , Count = 1



Step 2: n= 3 , Count = 2

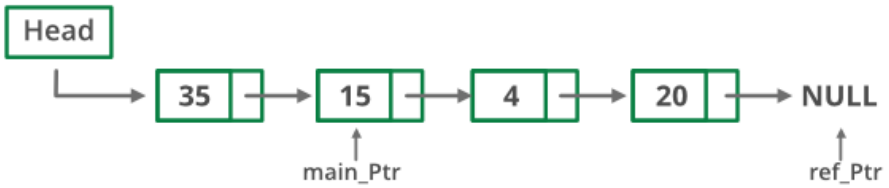


Step 3: n= 3 , Count = 3



Here, n is equals to count. First while loop breaks

Step 4:



Here, ref_Ptr points to NULL. main_Ptr is the n^{th} Node from end.

```
    {
        printf("%d is greater than the no. of "
                "nodes in list", n);
        return;
    }
    ref_ptr = ref_ptr->next;
    count++;
} /* End of while*/
```

```
      while(ref_ptr != NULL)
      {
         main_ptr = main_ptr->next;
         ref_ptr  = ref_ptr->next;
      }
      printf("Node no. %d from last is %d ",
               n, main_ptr->data);
  }
}

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

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

  /* link the old list off the new node */
  new_node->next = (*head_ref);

  /* move the head to point to the new node */
  (*head_ref)    = new_node;
}

/* Drier program to test above function*/
int main()
{
  /* Start with the empty list */
  struct Node* head = NULL;
  push(&head, 20);
  push(&head, 4);
  push(&head, 15);
  push(&head, 35);

  printNthFromLast(head, 4);
}
```

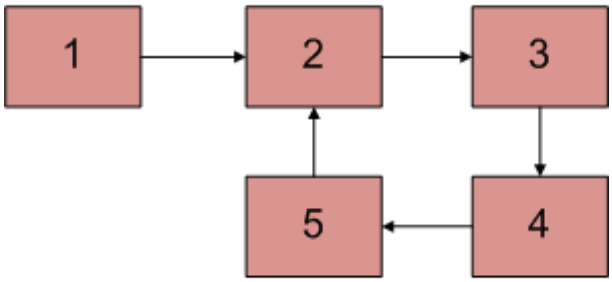
Output:

```
Node no. 4 from last is 35
```

Time Complexity: O(n) where n is the length of linked list.

4. Detect loop in a linked list

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



Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

Following are different ways of doing this

Use Hashing:

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.

```
// C++ program to detect loop in a linked list
#include <bits/stdc++.h>
using namespace std;

/* Link list node */
struct Node {
    int data;
    struct Node* next;
};

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

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

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
```

```
    (*head_ref) = new_node;
}

// Returns true if there is a loop in linked list
// else returns false.
bool detectLoop(struct Node* h)
{
    unordered_set<Node*> s;
    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 (s.find(h) != s.end())
            return true;

        // If we are seeing the node for
        // the first time, insert it in hash
        s.insert(h);

        h = h->next;
    }

    return false;
}

/* Driver program to test above function*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;

    push(&head, 20);
    push(&head, 4);
    push(&head, 15);
    push(&head, 10);

    /* Create a loop for testing */
    head->next->next->next->next = head;

    if (detectLoop(head))
        cout << "Loop found";
    else
        cout << "No Loop";

    return 0;
}
// This code is contributed by Geetanjali
```

Output:

```
Loop found
```

Approach using Mark Visited Nodes: This solution requires modifications to the basic linked list data structure.

- Have a visited flag with each node.
- Traverse the linked list and keep marking visited nodes.
- If you see a visited node again then there is a loop. This solution works in O(n) but requires additional information with each node.
- A variation of this solution that doesn't require modification to basic data structure can be implemented using a hash, just store the addresses of visited nodes in a hash and if you see an address that already exists in hash then there is a loop

.

Floyd's Cycle-Finding Algorithm: This is the fastest method and has been described below:

- Traverse linked list using two pointers.
- Move one pointer(slow_p) by one and another pointer(fast_p) by two.
- 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

.

Below image shows how the detectloop function works in the code :

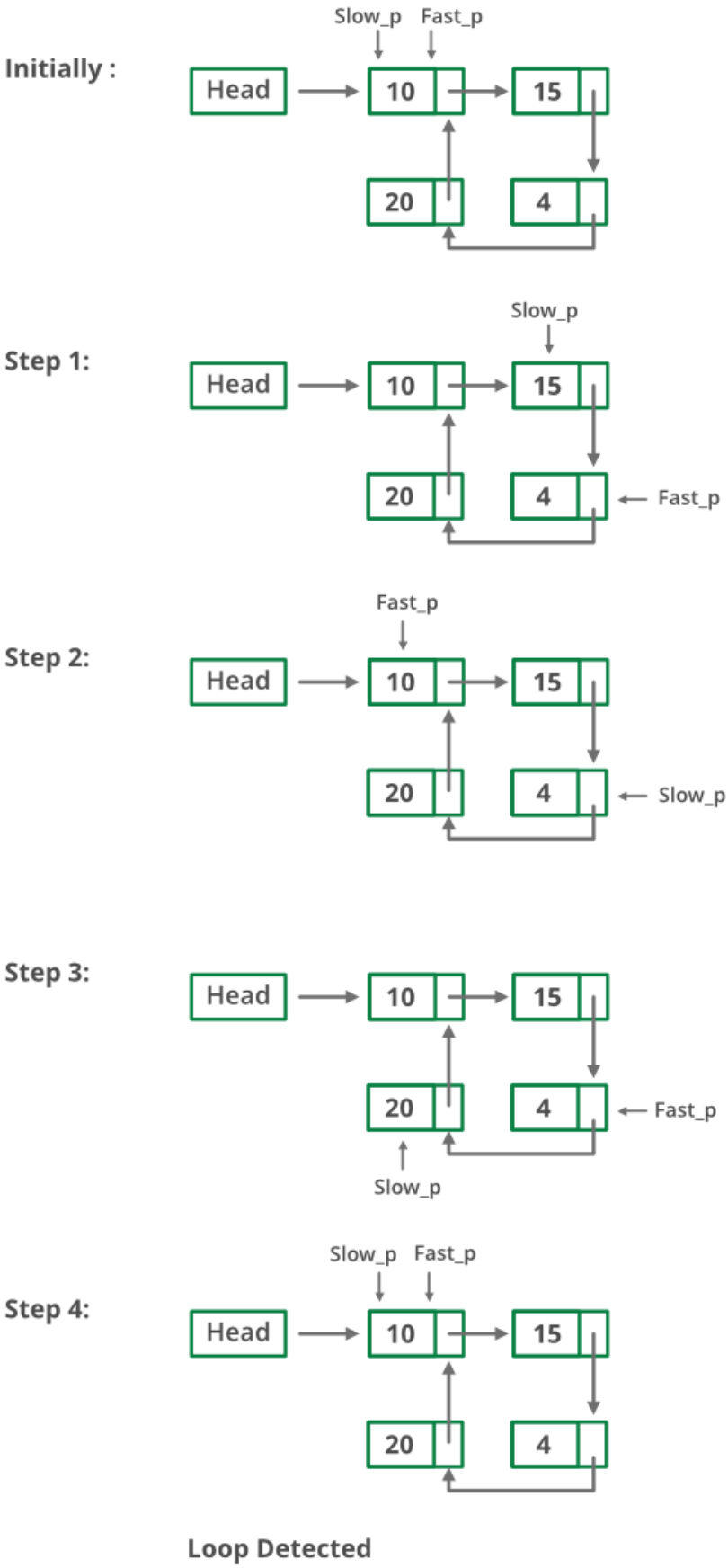
Implementation of Floyd's Cycle-Finding Algorithm:

```
// C++ program to detect loop in a linked list
#include <bits/stdc++.h>
using namespace std;

/* Link list node */
class Node {
public:
    int data;
    Node* next;
};

void push(Node** head_ref, int new_data)
{
    /* allocate node */
    Node* new_node = new Node();
```

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

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

int detectloop(Node* list)
{
    Node *slow_p = list, *fast_p = list;

    while (slow_p && fast_p && fast_p->next) {
        slow_p = slow_p->next;
        fast_p = fast_p->next->next;
        if (slow_p == fast_p) {
            cout << "Found Loop";
            return 1;
        }
    }
    return 0;
}

/* Driver code*/
int main()
{
    /* Start with the empty list */
    Node* head = NULL;

    push(&head, 20);
    push(&head, 4);
    push(&head, 15);
    push(&head, 10);

    /* Create a loop for testing */
    head->next->next->next->next = head;
    detectloop(head);

    return 0;
}

// This is code is contributed by rathbhupendra
```

Output:

```
Found Loop
```



Time Complexity: O(n)

Auxiliary Space: O(1)

How does above algorithm work?

Please See : How does Floyd's slow and fast pointers approach work?

References:

http://en.wikipedia.org/wiki/Cycle_detection

http://ostermiller.org/find_loop_singly_linked_list.html

Marking visited nodes without modifying the linked list data structure

In this method, a temporary node is created. The next pointer of each node that is traversed is made to point to this temporary node. This way we are using the next pointer of a node as a flag to indicate whether the node has been traversed or not. Every node is checked to see if the next is pointing to a temporary node or not. In the case of the first node of the loop, the second time we traverse it this condition will be true, hence we find that loop exists. If we come across a node that points to null then loop doesn't exist.

The code runs in O(n) time complexity and uses constant memory space.

Below is the implementation of the above approach:

```
// C++ program to return first node of loop
#include <bits/stdc++.h>
using namespace std;

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

Node* newNode(int key)
{
    Node* temp = new Node;
    temp->key = key;
    temp->next = NULL;
    return temp;
}

// A utility function to print a linked list
void printList(Node* head)
{
    while (head != NULL) {
        cout << head->key << " ";
```

```
        head = head->next;
    }
    cout << endl;
}

// Function to detect first node of loop
// in a linked list that may contain loop
bool detectLoop(Node* head)
{

    // Create a temporary node
    Node* temp = new Node;
    while (head != NULL) {

        // This condition is for the case
        // when there is no loop
        if (head->next == NULL) {
            return false;
        }

        // Check if next is already
        // pointing to temp
        if (head->next == temp) {
            return true;
        }

        // Store the pointer to the next node
        // in order to get to it in the next step
        Node* nex = head->next;

        // Make next point to temp
        head->next = temp;

        // Get to the next node in the list
        head = nex;
    }

    return false;
}

/* Driver program to test above function*/
int main()
{
    Node* head = newNode(1);
    head->next = newNode(2);
    head->next->next = newNode(3);
    head->next->next->next = newNode(4);
    head->next->next->next->next = newNode(5);
```

```
    /* Create a loop for testing(5 is pointing to 3) */
    head->next->next->next->next->next = head->next->next;

    bool found = detectLoop(head);
    if (found)
        cout << "Loop Found";
    else
        cout << "No Loop";

    return 0;
}
```

Output:

```
Loop Found
```

Time Complexity: O(n)

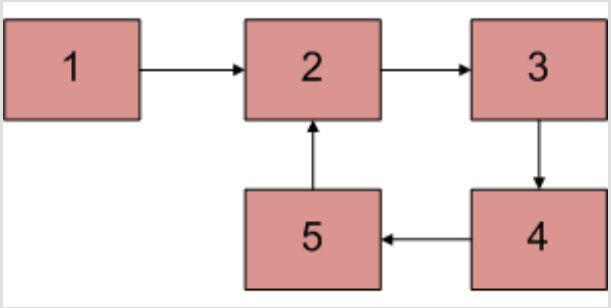
Auxiliary Space: O(1

5. Find first node of loop in a linked list

Write a function findFirstLoopNode() that checks whether a given Linked List contains loop. If loop is present then it returns point to first node of loop. Else it returns NULL.

Example :

```
Input : Head of bellow linked list
```




```
Output : Pointer to node 2
```

Recommended: Please try your approach on *{IDE}* first, before moving on to the solution.

We have discussed Floyd's loop detection algorithm. Below are steps to find first node of loop.

1. If a loop is found, initialize slow pointer to head, let fast pointer be at its position.
2. Move both slow and fast pointers one node at a time.
3. The point at which they meet is the start of the loop.

```
// C++ program to return first node of loop.
#include <bits/stdc++.h>
using namespace std;

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

Node* newNode(int key)
{
    Node* temp = new Node;
```

```
    temp->key = key;
    temp->next = NULL;
    return temp;
}

// A utility function to print a linked list
void printList(Node* head)
{
    while (head != NULL) {
        cout << head->key << " ";
        head = head->next;
    }
    cout << endl;
}

// Function to detect and remove loop
// in a linked list that may contain loop
Node* detectAndRemoveLoop(Node* head)
{
    // If list is empty or has only one node
    // without loop
    if (head == NULL || head->next == NULL)
        return NULL;

    Node *slow = head, *fast = head;

    // Move slow and fast 1 and 2 steps
    // ahead respectively.
    slow = slow->next;
    fast = fast->next->next;

    // Search for loop using slow and
    // fast pointers
    while (fast && fast->next) {
        if (slow == fast)
            break;
        slow = slow->next;
        fast = fast->next->next;
    }

    // If loop does not exist
    if (slow != fast)
        return NULL;

    // If loop exists. Start slow from
    // head and fast from meeting point.
    slow = head;
    while (slow != fast) {
        slow = slow->next;
```

```
            fast = fast->next;
        }

        return slow;
}

/* Driver program to test above function*/
int main()
{
        Node* head = newNode(50);
        head->next = newNode(20);
        head->next->next = newNode(15);
        head->next->next->next = newNode(4);
        head->next->next->next->next = newNode(10);

        /* Create a loop for testing */
        head->next->next->next->next->next = head->next->next;

        Node* res = detectAndRemoveLoop(head);
        if (res == NULL)
            cout << "Loop does not exist";
        else
            cout << "Loop starting node is " << res->key;

        return 0;
}
```

Output:

```
Loop starting node is 15
```

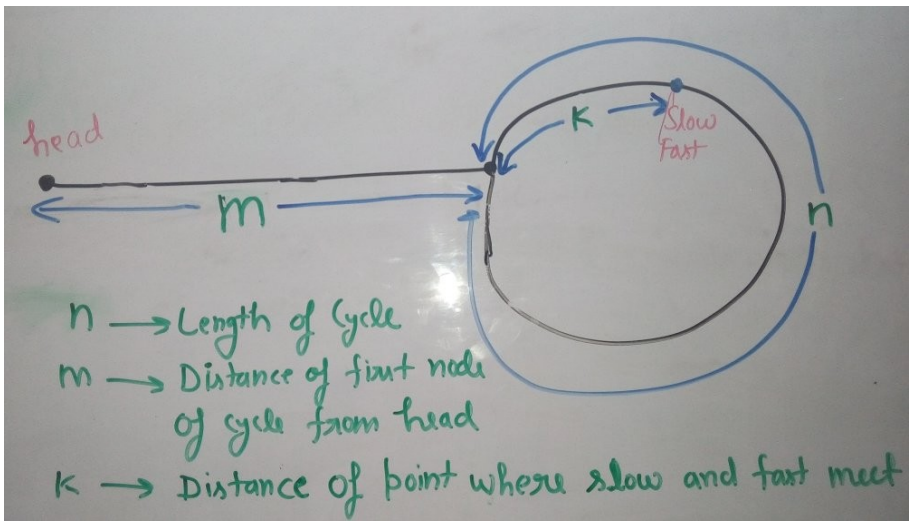
Output:

```
Loop starting node is 15
```

How does this approach work?

Let slow and fast meet at some point after Floyd's Cycle finding algorithm.

Below diagram shows the situation when cycle is found.



We can conclude below from above diagram

```
Distance traveled by fast pointer = 2 * (Distance traveled
                                         by slow pointer)


(m + n*x + k) = 2*(m + n*y + k)


Note that before meeting the point shown above, fast
was moving at twice speed.


x -->  Number of complete cyclic rounds made by
       fast pointer before they meet first time


y -->  Number of complete cyclic rounds made by
       slow pointer before they meet first time
```

From above equation, we can conclude below

```
    m + k = (x-2y)*n


Which means m+k is a multiple of n.
```

So if we start moving both pointers again at same speed such that one pointer (say slow) begins from head node of linked list and other pointer (say fast) begins from meeting point. When slow pointer reaches beginning of loop (has made m steps), fast pointer would have made also moved m steps as they are now moving same pace. Since m+k is a multiple of n and fast starts from k, they would meet at the beginning. Can they meet before also? No because slow pointer enters the cycle first time after m steps.

Method 2:

In this method, a temporary node is created. The next pointer of each node that is traversed is made to point to this temporary node. This way we are using the next pointer of a node as a flag to indicate whether the node has been traversed or not. Every node is checked to see if the next is pointing to temporary node or not. In the case of the first node of the loop, the second time we traverse it this condition will be true, hence we return that node. The code runs in O(n) time complexity and uses constant memory space.

Below is the implementation of the above approach:

```
// C++ program to return first node of loop
#include <bits/stdc++.h>
using namespace std;

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

Node* newNode(int key)
{
    Node* temp = new Node;
    temp->key = key;
    temp->next = NULL;
    return temp;
}

// A utility function to print a linked list
void printList(Node* head)
{
    while (head != NULL) {
        cout << head->key << " ";
        head = head->next;
```

```
    }
    cout << endl;
}

// Function to detect first node of loop
// in a linked list that may contain loop
Node* detectLoop(Node* head)
{

    // Create a temporary node
    Node* temp = new Node;
    while (head != NULL) {

        // This condition is for the case
        // when there is no loop
        if (head->next == NULL) {
            return NULL;
        }

        // Check if next is already
        // pointing to temp
        if (head->next == temp) {
            break;
        }

        // Store the pointer to the next node
        // in order to get to it in the next step
        Node* nex = head->next;

        // Make next point to temp
        head->next = temp;

        // Get to the next node in the list
        head = nex;
    }

    return head;
}

/* Driver program to test above function*/
int main()
{
    Node* head = newNode(50);
    head->next = newNode(20);
    head->next->next = newNode(15);
    head->next->next->next = newNode(4);
    head->next->next->next->next = newNode(10);

    /* Create a loop for testing */
```

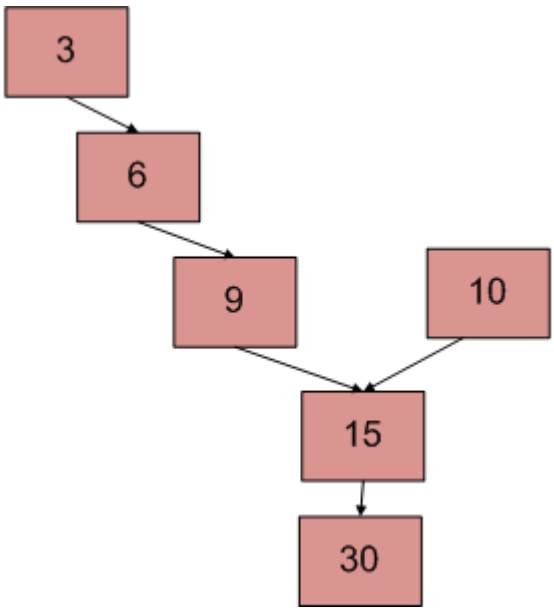
```
    head->next->next->next->next->next = head->next->next;

    Node* res = detectLoop(head);
    if (res == NULL)
        cout << "Loop does not exist";
    else
        cout << "Loop starting node is " << res->key;

    return 0;
}
```

Output:

```
Loop starting node is 15
```

6. Write a function to get the intersection point of two Linked Lists

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.

Method 1(Simply use two loops)

Use 2 nested for loops. The outer loop will be for each node of the 1st list and inner loop will be for 2nd list. In the inner loop, check if any of nodes of the 2nd list is same as the current node of the first linked list. The time complexity of this method will be O(M * N) where m and n are the numbers of nodes in two lists.

Method 2 (Mark Visited Nodes)

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.

Method 3(Using difference of node counts)

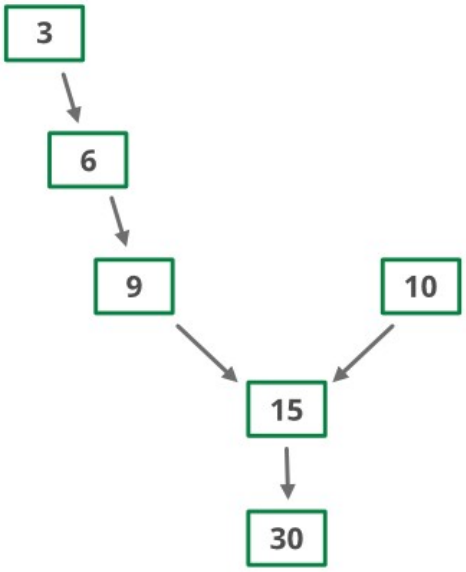
- Get count of the nodes in the first list, let count be c1.
- Get count of the nodes in the second list, let count be c2.
- Get the difference of counts d = abs(c1 – c2)
- Now traverse the bigger list from the first node till d nodes so that from here onwards both the lists have equal no of nodes.
- 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)

Below image is a dry run of the above approach:

Below is the implementation of the above approach :

```
// C++ program to get intersection point of two linked list
#include <bits/stdc++.h>
using namespace std;

/* Link list node */
class Node {
public:
    int data;
    Node* next;
};

/* Function to get the counts of node in a linked list */
int getCount(Node* head);

/* function to get the intersection point of two linked
lists head1 and head2 where head1 has d more nodes than
head2 */
int _getIntesectionNode(int d, Node* head1, Node* head2);
```

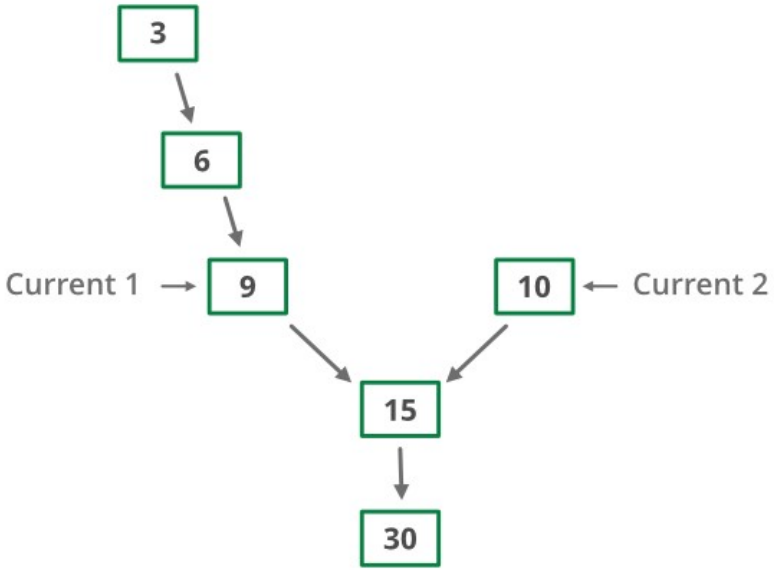
```
/* function to get the intersection point of two linked
lists head1 and head2 */
int getIntesectionNode(Node* head1, Node* head2)
{

    // Count the number of nodes in
    // both the linked list
    int c1 = getCount(head1);
    int c2 = getCount(head2);
    int d;

    // If first is greater
    if (c1 > c2) {
        d = c1 - c2;
        return _getIntesectionNode(d, head1, head2);
    }
    else {
        d = c2 - c1;
        return _getIntesectionNode(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 _getIntesectionNode(int d, Node* head1, Node* head2)
{
    // Stand at the starting of the bigger list
    Node* current1 = head1;
    Node* current2 = head2;

    // Move the pointer forward
    for (int i = 0; i < d; i++) {
        if (current1 == NULL) {
            return -1;
        }
        current1 = current1->next;
    }

    // Move both pointers of both list till they
    // intersect with each other
    while (current1 != NULL && current2 != NULL) {
        if (current1 == current2)
            return current1->data;

        // Move both the pointers forward
        current1 = current1->next;
        current2 = current2->next;
```

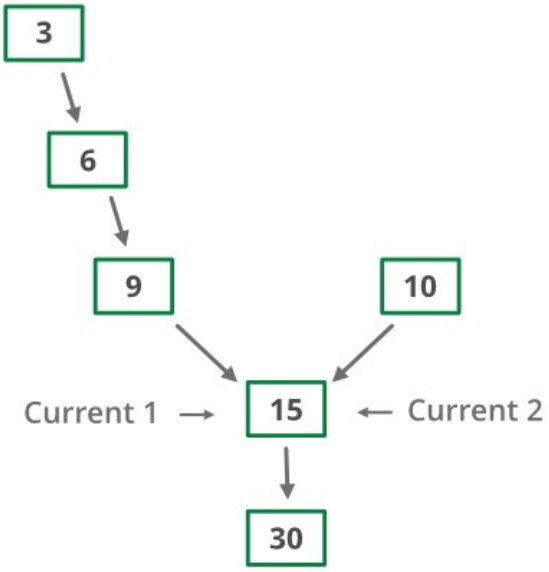


C1 = 5

C2 = 3

Traverse till 3rd node
in the bigger list





Intersection node = 15

```
    }

    return -1;
}

/* Takes head pointer of the linked list and
returns the count of nodes in the list */
int getCount(Node* head)
{
    Node* current = head;

    // Counter to store count of nodes
    int count = 0;

    // Iterate till NULL
    while (current != NULL) {

        // Increase the counter
        count++;

        // Move the Node ahead
        current = current->next;
    }

    return count;
}

// Driver Code
int main()
{
    /*
        Create two linked lists

        1st 3->6->9->15->30
        2nd 10->15->30

        15 is the intersection point
    */

    Node* newNode;

    // Addition of new nodes
    Node* head1 = new Node();
    head1->data = 10;

    Node* head2 = new Node();
    head2->data = 3;

    newNode = new Node();
```

```
    newNode->data = 6;
    head2->next = newNode;

    newNode = new Node();
    newNode->data = 9;
    head2->next->next = newNode;

    newNode = new Node();
    newNode->data = 15;
    head1->next = newNode;
    head2->next->next->next = newNode;

    newNode = new Node();
    newNode->data = 30;
    head1->next->next = newNode;

    head1->next->next->next = NULL;

    cout << "The node of intersection is " <<
getIntesectionNode(head1, head2);
}

// This code is contributed by rathbhupendra
```

Output:

```
The node of intersection is 15
```

Time Complexity: O(m+n)

Auxiliary Space: O(1)

Method 4(Make circle in first list)

Thanks to Saravanan Man for providing below solution.

1. Traverse the first linked list(count the elements) and make a circular linked list. (Remember the last node so that we can break the circle later on).

2. Now view the problem as finding the loop in the second linked list. So the problem is solved.

3. Since we already know the length of the loop(size of the first linked list) we can traverse those many numbers of nodes in the second list, and then start another pointer from the beginning of the second list. we have to traverse until they are equal, and that is the required intersection point.

4. remove the circle from the linked list.

Time Complexity: O(m+n)

Auxiliary Space: O(1)

Method 5 (Reverse the first list and make equations)

Thanks to Saravanan Mani for providing this method.

```
1) Let X be the length of the first linked list until intersection
point.

  Let Y be the length of the second linked list until the intersection
point.

  Let Z be the length of the linked list from the intersection point
to End of

  the linked list including the intersection node.

  We Have

         X + Z = C1;

         Y + Z = C2;

2) Reverse first linked list.

3) Traverse Second linked list. Let C3 be the length of second list -
1.

    Now we have

       X + Y = C3

    We have 3 linear equations. By solving them, we get

      X = (C1 + C3 – C2)/2;

      Y = (C2 + C3 – C1)/2;

      Z = (C1 + C2 – C3)/2;

     WE GOT THE INTERSECTION POINT.

4)  Reverse first linked list.
```

Advantage: No Comparison of pointers.

Disadvantage : Modifying linked list(Reversing list).

Time complexity: O(m+n)

Auxiliary Space: O(1)

Method 6 (Traverse both lists and compare addresses of last nodes) This method is only to detect if there is an intersection point or not. (Thanks to NeoTheSaviour for suggesting this)

```
1) Traverse the list 1, store the last node address

2) Traverse the list 2, store the last node address.

3) If nodes stored in 1 and 2 are same then they are intersecting.
```

The time complexity of this method is O(m+n) and used Auxiliary space is O(1)

Method 7 (Use Hashing)

Basically, we need to find a common node of two linked lists. So we hash all nodes of the first list and then check the second list.

1) Create an empty hash set.

2) Traverse the first linked list and insert all nodes' addresses in the hash set.

3) Traverse the second list. For every node check if it is present in the hash set. If we find a node in the hash set, return the node.

```
// Java program to get intersection point of two linked list
import java.util.*;
class Node {
    int data;
    Node next;
    Node(int d)
    {
        data = d;
        next = null;
    }
}
class LinkedListIntersect {
    public static void main(String[] args)
    {
        // list 1
        Node n1 = new Node(1);
        n1.next = new Node(2);
        n1.next.next = new Node(3);
        n1.next.next.next = new Node(4);
```

```
        n1.next.next.next.next = new Node(5);
        n1.next.next.next.next.next = new Node(6);
        n1.next.next.next.next.next.next = new Node(7);
        // list 2
        Node n2 = new Node(10);
        n2.next = new Node(9);
        n2.next.next = new Node(8);
        n2.next.next.next = n1.next.next.next;
        Print(n1);
        Print(n2);
        System.out.println(MegeNode(n1, n2).data);
    }

    // function to print the list
    public static void Print(Node n)
    {
        Node cur = n;
        while (cur != null) {
            System.out.print(cur.data + "  ");
            cur = cur.next;
        }
        System.out.println();
    }

    // function to find the intersection of two node
    public static Node MegeNode(Node n1, Node n2)
    {
        // define hashset
        HashSet<Node> hs = new HashSet<Node>();
        while (n1 != null) {
            hs.add(n1);
            n1 = n1.next;
        }
        while (n2 != null) {
            if (hs.contains(n2)) {
                return n2;
            }
            n2 = n2.next;
        }
        return null;
    }
}
```

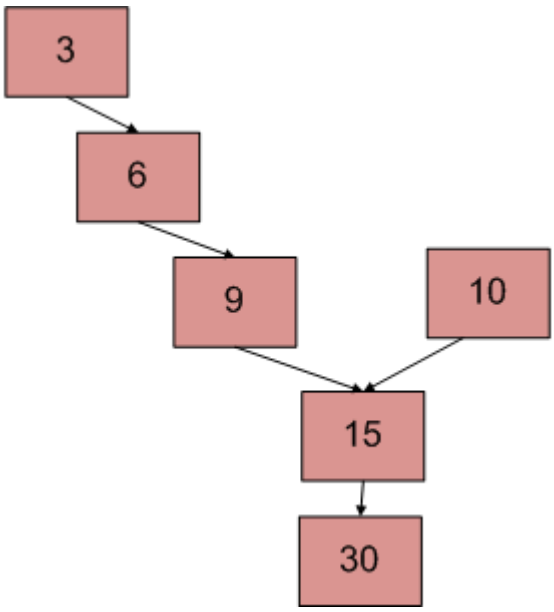
Output:

```
1 2 3 4 5 6 7

10 9 8 4 5 6 7

4
```

This method required O(n) additional space and not very efficient if one list is large.

7. Write a function to get the intersection point of two Linked Lists | Set 2

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.

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

Approach: It can be observed that the number of nodes in traversing the first linked list and then from the head of the second linked list to intersection point is equal to the number of nodes involved in traversing the second linked list and then from head of the first list to the intersection point. Considering the example given above, start traversing the two linked lists with two pointers **curr1** and **curr2** pointing to the heads of the given linked lists respectively.

1. If **curr1 != null** then update it to point to the next node, else it is updated to point to the first node of the second list.
2. If **curr2 != null** then update it to point to the next node, else it is updated to point to the first node of the first list.
3. Repeat the above steps while **curr1** is not equal to **curr2**.

The two pointers **curr1** and **curr2** will be pointing to the same node now i.e. the merging point.

Below is the implementation of the above approach:

```
// C++ implementation of the approach
#include <iostream>
using namespace std;

// Link list node
struct Node {
    int data;
    Node* next;
};

// Function to get the intersection point
// of the given linked lists
int getIntersectionNode(Node* head1, Node* head2)
{
    Node *curr1 = head1, *curr2 = head2;

    // While both the pointers are not equal
    while (curr1 != curr2) {

        // If the first pointer is null then
        // set it to point to the head of
        // the second linked list
        if (curr1 == NULL) {
            curr1 = head2;
        }

        // Else point it to the next node
        else {
            curr1 = curr1->next;
        }

        // If the second pointer is null then
        // set it to point to the head of
        // the first linked list
        if (curr2 == NULL) {
```

```
            curr2 = head1;
        }

        // Else point it to the next node
        else {
            curr2 = curr2->next;
        }
    }

    // Return the intersection node
    return curr1->data;
}

// Driver code
int main()
{
    /*
    Create two linked lists

    1st Linked list is 3->6->9->15->30
    2nd Linked list is 10->15->30

    15 is the intersection point
    */

    Node* newNode;
    Node* head1 = new Node();
    head1->data = 10;
    Node* head2 = new Node();
    head2->data = 3;
    newNode = new Node();
    newNode->data = 6;
    head2->next = newNode;
    newNode = new Node();
    newNode->data = 9;
    head2->next->next = newNode;
    newNode = new Node();
    newNode->data = 15;
    head1->next = newNode;
    head2->next->next->next = newNode;
    newNode = new Node();
    newNode->data = 30;
    head1->next->next = newNode;
    head1->next->next->next = NULL;

    // Print the intersection node
    cout << getIntersectionNode(head1, head2);

    return 0;
```

```
}
```

Output:

```
15
```

8. Alternating split of a given Singly Linked List | Set 1

Write a function AlternatingSplit() that takes one list and divides up its nodes to make two smaller lists ‘a’ and ‘b’. The sublists should be made from alternating elements in the original list. So if the original list is 0->1->0->1->0->1 then one sublist should be 0->0->0 and the other should be 1->1->1.

Recommended: Please solve it on “*PRACTICE*” first, before moving on to the solution.

Method 1(Simple)

The simplest approach iterates over the source list and pull nodes off the source and alternately put them at the front (or beginning) of ‘a’ and b’. The only strange part is that the nodes will be in the reverse order that they occurred in the source list. Method 2 inserts the node at the end by keeping track of last node in sublists.

```
/* C++ Program to alternatively split
a linked list into two halves */
#include <bits/stdc++.h>
using namespace std;

/* Link list node */
class Node
{
    public:
    int data;
    Node* next;
};

/* pull off the front node of
the source and put it in dest */
void MoveNode(Node** destRef, Node** sourceRef) ;

/* Given the source list, split its
nodes into two shorter lists. If we number
the elements 0, 1, 2, ... then all the even
elements should go in the first list, and
all the odd elements in the second. The
elements in the new lists may be in any order. */
void AlternatingSplit(Node* source, Node** aRef,
                              Node** bRef)
```

```
{
    /* split the nodes of source
    to these 'a' and 'b' lists */
    Node* a = NULL;
    Node* b = NULL;

    Node* current = source;
    while (current != NULL)
    {
        MoveNode(&a, ¤t); /* Move a node to list 'a' */
        if (current != NULL)
        {
            MoveNode(&b, ¤t); /* Move a node to list 'b' */
        }
    }
    *aRef = a;
    *bRef = b;
}

/* Take the node from the front of
the source, and move it to the front
of the dest. It is an error to call
this with the source list empty.

Before calling MoveNode():
source == {1, 2, 3}
dest == {1, 2, 3}

Affter calling MoveNode():
source == {2, 3}
dest == {1, 1, 2, 3}
*/
void MoveNode(Node** destRef, Node** sourceRef)
{
    /* the front source node */
    Node* newNode = *sourceRef;
    assert(newNode != NULL);

    /* Advance the source pointer */
    *sourceRef = newNode->next;

    /* Link the old dest off the new node */
    newNode->next = *destRef;

    /* Move dest to point to the new node */
    *destRef = newNode;
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at
```

```
the beginging of the linked list */
void push(Node** head_ref, int new_data)
{
    /* allocate node */
    Node* new_node = new Node();

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

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Function to print nodes
in a given linked list */
void printList(Node *node)
{
    while(node!=NULL)
    {
    cout<<node->data<<" ";
    node = node->next;
    }
}

/* Driver code*/
int main()
{
    /* Start with the empty list */
    Node* head = NULL;
    Node* a = NULL;
    Node* b = NULL;

    /* Let us create a sorted linked list to test the functions
    Created linked list will be 0->1->2->3->4->5 */
    push(&head, 5);
    push(&head, 4);
    push(&head, 3);
    push(&head, 2);
    push(&head, 1);
    push(&head, 0);

    cout<<"Original linked List: ";
    printList(head);

    /* Remove duplicates from linked list */
    AlternatingSplit(head, &a, &b);
```

```
    cout<<"\nResultant Linked List 'a' : ";
    printList(a);

    cout<<"\nResultant Linked List 'b' : ";
    printList(b);

    return 0;
}

// This code is contributed by rathbhupendra
```

Output:

```
Original linked List: 0 1 2 3 4 5

Resultant Linked List 'a' : 4 2 0

Resultant Linked List 'b' : 5 3 1
```

Time Complexity: O(n) where n is number of node in the given linked list.

Method 2(Using Dummy Nodes)

Here is an alternative approach which builds the sub-lists in the same order as the source list. The code uses a temporary dummy header nodes for the ‘a’ and ‘b’ lists as they are being built. Each sublist has a “tail” pointer which points to its current last node — that way new nodes can be appended to the end of each list easily. The dummy nodes give the tail pointers something to point to initially. The dummy nodes are efficient in this case because they are temporary and allocated in the stack. Alternately, local “reference pointers” (which always points to the last pointer in the list instead of to the last node) could be used to avoid Dummy nodes.

```
void AlternatingSplit(Node* source,
                      Node** aRef, Node** bRef)
{
    Node aDummy;

    /* points to the last node in 'a' */
    Node* aTail = &aDummy;
```

```
    Node bDummy;

    /* points to the last node in 'b' */
    Node* bTail = &bDummy;
    Node* current = source;
    aDummy.next = NULL;
    bDummy.next = NULL;
    while (current != NULL)
    {
        MoveNode(&(aTail->next), ¤t); /* add at 'a' tail */
        aTail = aTail->next; /* advance the 'a' tail */
        if (current != NULL)
        {
            MoveNode(&(bTail->next), ¤t);
            bTail = bTail->next;
        }
    }
    *aRef = aDummy.next;
    *bRef = bDummy.next;
}

// This code is contributed
// by rathbhupendra
```

Time Complexity: O(n) where n is number of node in the given linked list

9. Clone a linked list with next and random pointer | Set 1

You are given a Double Link List with one pointer of each node pointing to the next node just like in a single link list. The second pointer however CAN point to any node in the list and not just the previous node. Now write a program in O(n) time to duplicate this list. That is, write a program which will create a copy of this list.

Let us call the second pointer as arbit pointer as it can point to any arbitrary node in the linked list.

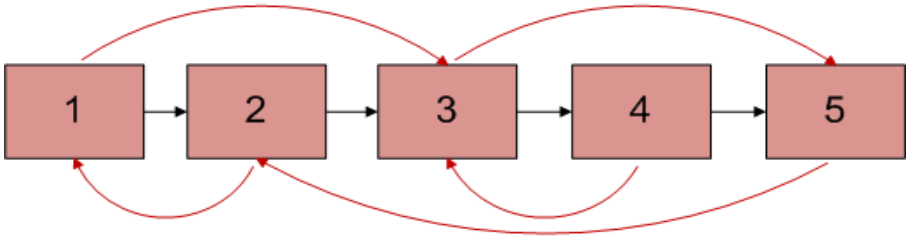


Figure 1

Arbitrary pointers are shown in red and next pointers in black

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

Method 1 (Uses O(n) extra space)

This method stores the next and arbitrary mappings (of original list) in an array first, then modifies the original Linked List (to create copy), creates a copy. And finally restores the original list.

1) Create all nodes in copy linked list using next pointers.

2) Store the node and its next pointer mappings of original linked list.

3) Change next pointer of all nodes in original linked list to point to the corresponding node in copy linked list.

Following diagram shows status of both Linked Lists after above 3 steps. The red arrow shows arbit pointers and black arrow shows next pointers.

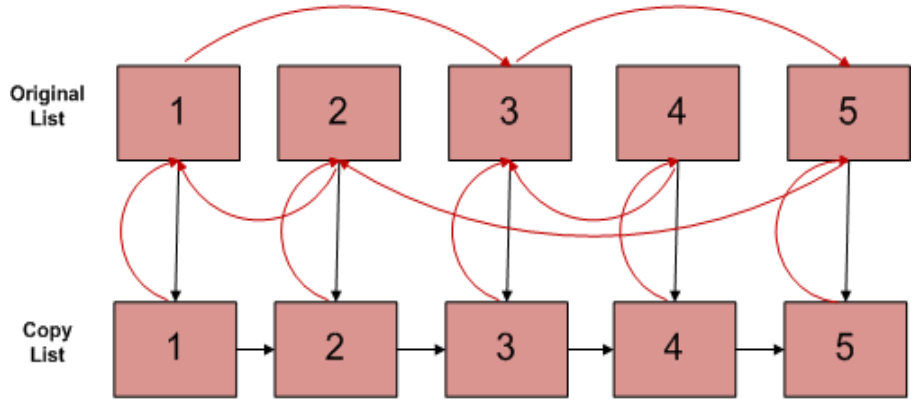


Figure 2

4) Change the arbit pointer of all nodes in copy linked list to point to corresponding node in original linked list.

5) Now construct the arbit pointer in copy linked list as below and restore the next pointer of nodes in the original linked list.

```
copy_list_node->arbit =

            copy_list_node->arbit->arbit->next;

copy_list_node = copy_list_node->next;
```

6) Restore the next pointers in original linked list from the stored mappings(in step 2).

Time Complexity: O(n)

Auxiliary Space: O(n)

Method 2 (Uses Constant Extra Space)

Thanks to Saravanan Mani for providing this solution. This solution works using constant space.

1) Create the copy of node 1 and insert it between node 1 & node 2 in original

Linked List, create the copy of 2 and insert it between 2 & 3.. Continue in this fashion, add the copy of N afte the Nth node

2) Now copy the arbitrary link in this fashion

```
    original->next->arbitrary = original->arbitrary->next;  /*TRAVERSE TWO NODES*/
```

This works because original->next is nothing but copy of original and Original->arbitrary->next is nothing but copy of arbitrary.

3) Now restore the original and copy linked lists in this fashion in a single loop.

```
    original->next = original->next->next;

    copy->next = copy->next->next;
```

4) Make sure that last element of original->next is NULL.

Refer below post for implementation of this method.

Clone a linked list with next and random pointer in O(1) space

Time Complexity: O(n)

Auxiliary Space: O(1)

10. Clone a linked list with next and random pointer | Set 2

We have already discussed 2 different ways to clone a linked list. In this post, one more simple method to clone a linked list is discussed.

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

The idea is to use Hashing. Below is algorithm.

1. Traverse the original linked list and make a copy in terms of data.

2. Make a hash map of key value pair with original linked list node and copied linked list node.

3. Traverse the original linked list again and using the hash map adjust the next and random reference of cloned linked list nodes.

Below is the implementation of above approach.

```
// C++ program to clone a linked list with
// random pointers
#include<bits/stdc++.h>
using namespace std;


// Linked List Node
class Node
{
    public:
    int data;//Node data

    // Next and random reference
    Node *next, *random;

    Node(int data)
    {
        this->data = data;
        this->next = this->random = NULL;
    }
};

// linked list class
class LinkedList
```

```
{
    public:
    Node *head;// Linked list head reference

    LinkedList(Node *head)
    {
        this->head = head;
    }

    // push method to put data always at
    // the head in the linked list.
    void push(int data)
    {
        Node *node = new Node(data);
        node->next = head;
        head = node;
    }

    // Method to print the list.
    void print()
    {
        Node *temp = head;
        while (temp != NULL)
        {
            Node *random = temp->random;
            int randomData = (random != NULL)?
                        random->data: -1;
            cout << "Data = " << temp->data
                 << ", ";
            cout << "Random Data = " <<
                 randomData << endl;
            temp = temp->next;
        }
        cout << endl;
    }

    // Actual clone method which returns
    // head reference of cloned linked
    // list.
    LinkedList* clone()
    {
        // Initialize two references,
        // one with original list's head.
        Node *origCurr = head;
        Node *cloneCurr = NULL;

        // Hash map which contains node
        // to node mapping of original
        // and clone linked list.
```

```
        unordered_map<Node*, Node*> mymap;

        // Traverse the original list and
        // make a copy of that in the
        // clone linked list.
        while (origCurr != NULL)
        {
            cloneCurr = new Node(origCurr->data);
            mymap[origCurr] = cloneCurr;
            origCurr = origCurr->next;
        }

        // Adjusting the original list
        // reference again.
        origCurr = head;

        // Traversal of original list again
        // to adjust the next and random
        // references of clone list using
        // hash map.
        while (origCurr != NULL)
        {
            cloneCurr = mymap[origCurr];
            cloneCurr->next = mymap[origCurr->next];
            cloneCurr->random = mymap[origCurr->random];
            origCurr = origCurr->next;
        }

        // return the head reference of
        // the clone list.
        return new LinkedList(mymap[head]);
    }
};

// driver code
int main()
{
    // Pushing data in the linked list.
    LinkedList *mylist = new LinkedList(new Node(5));
    mylist->push(4);
    mylist->push(3);
    mylist->push(2);
    mylist->push(1);

    // Setting up random references.
    mylist->head->random = mylist->head->next->next;

    mylist->head->next->random =
        mylist->head->next->next->next;
```

```
    mylist->head->next->next->random =
        mylist->head->next->next->next->next;

    mylist->head->next->next->next->random =
        mylist->head->next->next->next->next->next;

    mylist->head->next->next->next->next->random =
        mylist->head->next;

    // Making a clone of the original
    // linked list.
    LinkedList *clone = mylist->clone();

    // Print the original and cloned
    // linked list.
    cout << "Original linked list\n";
    mylist->print();
    cout << "\nCloned linked list\n";
    clone->print();
}
// This code is contributed by Chhavi
```

Output:

```
Original linked list

Data = 1, Random data = 3

Data = 2, Random data = 4

Data = 3, Random data = 5

Data = 4, Random data = -1

Data = 5, Random data = 2



Cloned linked list

Data = 1, Random data = 3

Data = 2, Random data = 4

Data = 3, Random data = 5

Data = 4, Random data = -1

Data = 5, Random data = 2
```

Time complexity : O(n)

Auxiliary space : O(n)

11. Merge two sorted linked lists

Write a SortedMerge() function that takes two lists, each of which is sorted in increasing order, and merges the two together into one list which is in increasing order. SortedMerge() should return the new list. The new list should be made by splicing
together the nodes of the first two lists.

For example if the first linked list a is 5->10->15 and the other linked list b is 2->3->20, then SortedMerge() should return a pointer to the head node of the merged list 2->3->5->10->15->20.

There are many cases to deal with: either ‘a’ or ‘b’ may be empty, during processing either ‘a’ or ‘b’ may run out first, and finally there’s the problem of starting the result list empty, and building it up while going through ‘a’ and ‘b’.

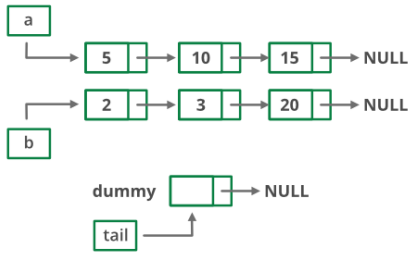
Recommended: Please solve it on “*PRACTICE*” first, before moving on to the solution.

Method 1 (Using Dummy Nodes)
The strategy here uses a temporary dummy node as the start of the result list. The pointer Tail always points to the last node in the result list, so appending new nodes is easy.
The dummy node gives tail something to point to initially when the result list is empty. This dummy node is efficient, since it is only temporary, and it is allocated in the stack. The loop proceeds, removing one node from either ‘a’ or ‘b’, and adding it to tail. When
we are done, the result is in dummy.next.
Below image is a dry run of the above approach:

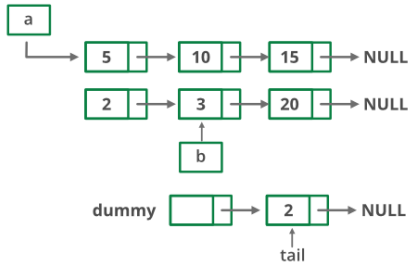
Below is the implemenation of the above approach:

```cpp
/* C++ program to merge two sorted linked lists */
#include <bits/stdc++.h>
using namespace std;

/* Link list node */
class Node
{
    public:
    int data;
    Node* next;
};

/* pull off the front node of
the source and put it in dest */
void MoveNode(Node** destRef, Node** sourceRef);

/* Takes two lists sorted in increasing
order, and splices their nodes together
to make one big sorted list which
is returned. */
Node* SortedMerge(Node* a, Node* b)
{
    /* a dummy first node to hang the result on */
    Node dummy;

    /* tail points to the last result node */
    Node* tail = &dummy;

    /* so tail->next is the place to
    add new nodes to the result. */
    dummy.next = NULL;
    while (1)
    {
        if (a == NULL)
        {
            /* if either list runs out, use the
            other list */
            tail->next = b;
            break;
        }
        else if (b == NULL)
        {
            tail->next = a;
            break;
        }
        if (a->data <= b->data)
```

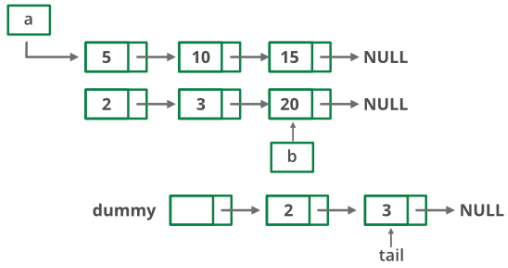
Initially :



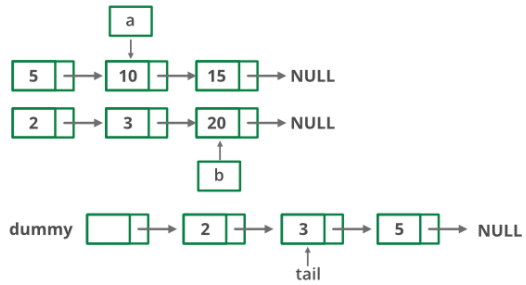
Step 1:



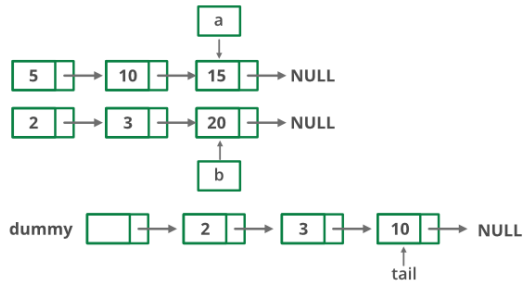
Step 2:



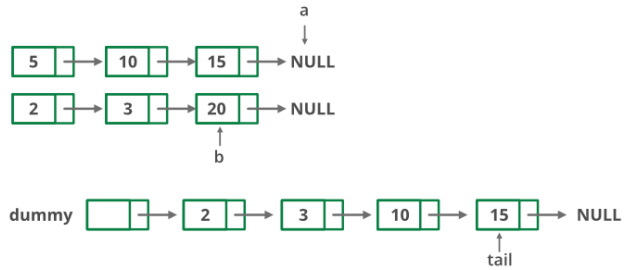
Step 3:



Step 4:

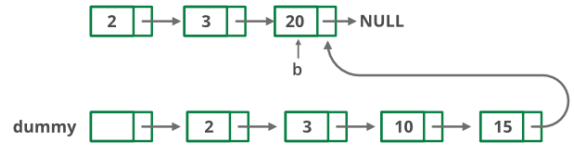


Step 5:



Step 6:

a points to NULL




```
MoveNode(&(t
ail->next),
&a);
        else

MoveNode(&(t
ail->next),
&b);

        tail
= tail-
>next;
    }
    return(d
ummy.next);
}

/* UTILITY
FUNCTIONS */
/*
MoveNode()
function
takes the
node from
the front of
the source,
and move it
to the front
of the
dest.
It is an
error to
call this
with the
source list
empty.

Before
calling
MoveNode():
source ==
{1, 2, 3}
dest == {1,
2, 3}

Affter
calling
MoveNode():
source ==
{2, 3}
```

```
dest == {1, 1, 2, 3} */
void MoveNode(Node** destRef, Node** sourceRef)
{
    /* the front source node */
    Node* newNode = *sourceRef;
    assert(newNode != NULL);

    /* Advance the source pointer */
    *sourceRef = newNode->next;

    /* Link the old dest off the new node */
    newNode->next = *destRef;

    /* Move dest to point to the new node */
    *destRef = newNode;
}


/* Function to insert a node at
the beginging of the linked list */
void push(Node** head_ref, int new_data)
{
    /* allocate node */
    Node* new_node = new Node();

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

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Function to print nodes in a given linked list */
void printList(Node *node)
{
    while (node!=NULL)
    {
        cout<<node->data<<" ";
        node = node->next;
    }
}

/* Driver code*/
int main()
{
    /* Start with the empty list */
    Node* res = NULL;
```

```
    Node* a = NULL;
    Node* b = NULL;

    /* Let us create two sorted linked lists
    to test the functions
    Created lists, a: 5->10->15, b: 2->3->20 */
    push(&a, 15);
    push(&a, 10);
    push(&a, 5);

    push(&b, 20);
    push(&b, 3);
    push(&b, 2);

    /* Remove duplicates from linked list */
    res = SortedMerge(a, b);

    cout << "Merged Linked List is: \n";
    printList(res);

    return 0;
}

// This code is contributed by rathbhupendra
```

Output :

```
Merged Linked List is:

2 3 5 10 15 20
```

Method 2 (Using Local References)

This solution is structurally very similar to the above, but it avoids using a dummy node. Instead, it maintains a struct node** pointer, lastPtrRef, that always points to the last pointer of the result list. This solves the same case that the dummy node did — dealing with the result list when it is empty. If you are trying to build up a list at its tail, either the dummy node or the struct node** “reference” strategy can be used (see Section 1 for details).

```
Node* SortedMerge(Node* a, Node* b)
{
Node* result = NULL;

/* point to the last result pointer */
```

```
Node** lastPtrRef = &result;

while(1)
{
    if (a == NULL)
    {
    *lastPtrRef = b;
    break;
    }
    else if (b==NULL)
    {
    *lastPtrRef = a;
    break;
    }
    if(a->data <= b->data)
    {
    MoveNode(lastPtrRef, &a);
    }
    else
    {
    MoveNode(lastPtrRef, &b);
    }

    /* tricky: advance to point to the next ".next" field */
    lastPtrRef = &((*lastPtrRef)->next);
}
return(result);
}

//This code is contributed by rathbhupendra
```

Method 3 (Using Recursion)

Merge is one of those nice recursive problems where the recursive solution code is much cleaner than the iterative code. You probably wouldn’t want to use the recursive version for production code however, because it will use stack space which is proportional to the length of the lists.

```
Node* SortedMerge(Node* a, Node* b)
{
    Node* result = NULL;

    /* Base cases */
    if (a == NULL)
```

```
        return(b);
    else if (b == NULL)
        return(a);

    /* Pick either a or b, and recur */
    if (a->data <= b->data)
    {
        result = a;
        result->next = SortedMerge(a->next, b);
    }
    else
    {
        result = b;
        result->next = SortedMerge(a, b->next);
    }
    return(result);
}

// This code is contributed by rathbhupendra
```

Please refer below post for simpler implementations :

Merge two sorted lists (in-place)

12. Merge two sorted lists (in-place)

Given two sorted lists, merge them so as to produce a combined sorted list (without using extra space).

Examples:

```
Input : head1: 5->7->9
        head2: 4->6->8
Output : 4->5->6->7->8->9


Input : head1: 1->3->5->7
        head2: 2->4
Output : 1->2->3->4->5->7
```

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

We have discussed different solutions in below post.

Merge two sorted linked lists

In this post, a new simpler solutions are discussed. The idea is to one by one compare nodes and form the result list.

Method 1 (Recursive)

```
// C program to merge two sorted linked lists
// in-place.
#include <bits/stdc++.h>
using namespace std;

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

// Function to create newNode in a linkedlist
Node* newNode(int key)
{
    struct Node* temp = new Node;
```

```
    temp->data = key;
    temp->next = NULL;
    return temp;
}

// A utility function to print linked list
void printList(Node* node)
{
    while (node != NULL) {
        printf("%d  ", node->data);
        node = node->next;
    }
}

// Merges two given lists in-place. This function
// mainly compares head nodes and calls mergeUtil()
Node* merge(Node* h1, Node* h2)
{
    if (!h1)
        return h2;
    if (!h2)
        return h1;

    // start with the linked list
    // whose head data is the least
    if (h1->data < h2->data) {
        h1->next = merge(h1->next, h2);
        return h1;
    }
    else {
        h2->next = merge(h1, h2->next);
        return h2;
    }
}

// Driver program
int main()
{
    Node* head1 = newNode(1);
    head1->next = newNode(3);
    head1->next->next = newNode(5);

    // 1->3->5 LinkedList created

    Node* head2 = newNode(0);
    head2->next = newNode(2);
    head2->next->next = newNode(4);

    // 0->2->4 LinkedList created
```

```
    Node* mergedhead = merge(head1, head2);

    printList(mergedhead);
    return 0;
}
```

Output:

```
0 1 2 3 4 5
```

Method 2 (Iterative)

```
// C++ program to merge two sorted linked lists
// in-place.
#include <bits/stdc++.h>
using namespace std;

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

// Function to create newNode in a linkedlist
struct Node* newNode(int key)
{
    struct Node* temp = new Node;
    temp->data = key;
    temp->next = NULL;
    return temp;
}

// A utility function to print linked list
void printList(struct Node* node)
{
    while (node != NULL) {
        printf("%d  ", node->data);
        node = node->next;
    }
}

// Merges two lists with headers as h1 and h2.
// It assumes that h1's data is smaller than
// or equal to h2's data.
struct Node* mergeUtil(struct Node* h1,
                       struct Node* h2)
{
    // if only one node in first list
    // simply point its head to second list
    if (!h1->next) {
```

```
        h1->next = h2;
        return h1;
    }

    // Initialize current and next pointers of
    // both lists
    struct Node *curr1 = h1, *next1 = h1->next;
    struct Node *curr2 = h2, *next2 = h2->next;

    while (next1 && curr2) {
        // if curr2 lies in between curr1 and next1
        // then do curr1->curr2->next1
        if ((curr2->data) >= (curr1->data) && (curr2->data) <=
(next1->data)) {
            next2 = curr2->next;
            curr1->next = curr2;
            curr2->next = next1;

            // now let curr1 and curr2 to point
            // to their immediate next pointers
            curr1 = curr2;
            curr2 = next2;
        }
        else {
            // if more nodes in first list
            if (next1->next) {
                next1 = next1->next;
                curr1 = curr1->next;
            }

            // else point the last node of first list
            // to the remaining nodes of second list
            else {
                next1->next = curr2;
                return h1;
            }
        }
    }
    return h1;
}

// Merges two given lists in-place. This function
// mainly compares head nodes and calls mergeUtil()
struct Node* merge(struct Node* h1,
                   struct Node* h2)
{
    if (!h1)
        return h2;
    if (!h2)
```

```
        return h1;

    // start with the linked list
    // whose head data is the least
    if (h1->data < h2->data)
        return mergeUtil(h1, h2);
    else
        return mergeUtil(h2, h1);
}

// Driver program
int main()
{
    struct Node* head1 = newNode(1);
    head1->next = newNode(3);
    head1->next->next = newNode(5);

    // 1->3->5 LinkedList created

    struct Node* head2 = newNode(0);
    head2->next = newNode(2);
    head2->next->next = newNode(4);

    // 0->2->4 LinkedList created

    struct Node* mergedhead = merge(head1, head2);

    printList(mergedhead);
    return 0;
}
```

Output:

```
0 1 2 3 4 5
```

13. Merge K sorted linked lists | Set 1

Given K sorted linked lists of size N each, merge them and print the sorted output.

Example:

```
Input: k = 3, n =  4

list1 = 1->3->5->7->NULL

list2 = 2->4->6->8->NULL

list3 = 0->9->10->11


Output:

0->1->2->3->4->5->6->7->8->9->10->11
```

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

Method 1 (Simple)

A Simple Solution is to initialize result as first list. Now traverse all lists starting from second list. Insert every node of currently traversed list into result in a sorted way. Time complexity of this solution is O(N2) where N is total number of nodes, i.e., N = kn.

Method 2 (Using Min Heap)

A Better solution is to use Min Heap based solution which is discussed here for arrays. Time complexity of this solution would be O(nk Log k)

Method 3 (Using Divide and Conquer))

In this post, Divide and Conquer approach is discussed. This approach doesn't require extra space for heap and works in O(nk Log k)

We already know that merging of two linked lists can be done in O(n) time and O(1) space (For arrays O(n) space is required). The idea is to pair up K lists and merge each pair in linear time using O(1) space. After first cycle, K/2 lists are left each of size 2*N. After second cycle, K/4 lists are left each of size 4*N and so on. We repeat the procedure until we have only one list left.

Below is implementation of the above idea.

```
// C++ program to merge k sorted arrays of size n each
#include <bits/stdc++.h>
using namespace std;

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

/* Function to print nodes in a given linked list */
void printList(Node* node)
{
    while (node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
}

/* Takes two lists sorted in increasing order, and merge
   their nodes together to make one big sorted list. Below
   function takes O(Log n) extra space for recursive calls,
   but it can be easily modified to work with same time and
   O(1) extra space  */
Node* SortedMerge(Node* a, Node* b)
{
    Node* result = NULL;

    /* Base cases */
    if (a == NULL)
        return (b);
    else if(b == NULL)
        return (a);

    /* Pick either a or b, and recur */
```

```
    if(a->data <= b->data)
    {
        result = a;
        result->next = SortedMerge(a->next, b);
    }
    else
    {
        result = b;
        result->next = SortedMerge(a, b->next);
    }

    return result;
}

// The main function that takes an array of lists
// arr[0..last] and generates the sorted output
Node* mergeKLists(Node* arr[], int last)
{
    // repeat until only one list is left
    while (last != 0)
    {
        int i = 0, j = last;

        // (i, j) forms a pair
        while (i < j)
        {
            // merge List i with List j and store
            // merged list in List i
            arr[i] = SortedMerge(arr[i], arr[j]);

            // consider next pair
            i++, j--;

            // If all pairs are merged, update last
            if (i >= j)
                last = j;
        }
    }

    return arr[0];
}

// Utility function to create a new node.
Node *newNode(int data)
{
    struct Node *temp = new Node;
    temp->data = data;
    temp->next = NULL;
    return temp;
```

```
}

// Driver program to test above functions
int main()
{
    int k = 3; // Number of linked lists
    int n = 4; // Number of elements in each list

    // an array of pointers storing the head nodes
    // of the linked lists
    Node* arr[k];

    arr[0] = newNode(1);
    arr[0]->next = newNode(3);
    arr[0]->next->next = newNode(5);
    arr[0]->next->next->next = newNode(7);

    arr[1] = newNode(2);
    arr[1]->next = newNode(4);
    arr[1]->next->next = newNode(6);
    arr[1]->next->next->next = newNode(8);

    arr[2] = newNode(0);
    arr[2]->next = newNode(9);
    arr[2]->next->next = newNode(10);
    arr[2]->next->next->next = newNode(11);

    // Merge all lists
    Node* head = mergeKLists(arr, k - 1);

    printList(head);

    return 0;
}
```

Output :

```
0 1 2 3 4 5 6 7 8 9 10 11
```

Time Complexity of above algorithm is O(nk logk) as outer while loop in function mergeKLists() runs log k times and every time we are processing nk elements.

14. Merge k sorted linked lists | Set 2 (Using Min Heap)

Given **k** sorted linked lists each of size **n**, merge them and print the sorted output.

Examples:

```
Input: k = 3, n =  4

list1 = 1->3->5->7->NULL

list2 = 2->4->6->8->NULL

list3 = 0->9->10->11



Output:

0->1->2->3->4->5->6->7->8->9->10->11
```

Source: Merge K sorted Linked Lists | Method 2

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

Approach: An efficient solution for the problem has been dicussed in **Method 3** of this post. Here another solution has been provided which the uses the **MIN HEAP** data structure. This solution is based on the min heap approach used to solve the problem 'merge k sorted arrays' which is discussed here.

```
// C++ implementation to merge k sorted linked lists
// | Using MIN HEAP method
#include <bits/stdc++.h>
using namespace std;

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

// 'compare' function used to build up the
// priority queue
struct compare {
    bool operator()(struct Node* a, struct Node* b)
    {
        return a->data > b->data;
```

```
    }
};

// function to merge k sorted linked lists
struct Node* mergeKSortedLists(struct Node* arr[], int k)
{
    struct Node *head = NULL, *last;

    // priority_queue 'pq' implemeted as min heap with the
    // help of 'compare' function
    priority_queue<Node*, vector<Node*>, compare> pq;

    // push the head nodes of all the k lists in 'pq'
    for (int i = 0; i < k; i++)
        if (arr[i] != NULL)
            pq.push(arr[i]);

    // loop till 'pq' is not empty
    while (!pq.empty()) {

        // get the top element of 'pq'
        struct Node* top = pq.top();
        pq.pop();

        // check if there is a node next to the 'top' node
        // in the list of which 'top' node is a member
        if (top->next != NULL)
            // push the next node in 'pq'
            pq.push(top->next);

        // if final merged list is empty
        if (head == NULL) {
            head = top;

            // points to the last node so far of
            // the final merged list
            last = top;
        }

        else {
            // insert 'top' at the end of the merged list so far
            last->next = top;

            // update the 'last' pointer
            last = top;
        }
    }

    // head node of the required merged list
```

```
    return head;
}

// function to print the singly linked list
void printList(struct Node* head)
{
    while (head != NULL) {
        cout << head->data << " ";
        head = head->next;
    }
}

// Utility function to create a new node
struct Node* newNode(int data)
{
    // allocate node
    struct Node* new_node = new Node();

    // put in the data
    new_node->data = data;
    new_node->next = NULL;

    return new_node;
}

// Driver program to test above
int main()
{
    int k = 3; // Number of linked lists
    int n = 4; // Number of elements in each list

    // an array of pointers storing the head nodes
    // of the linked lists
    Node* arr[k];

    // creating k = 3 sorted lists
    arr[0] = newNode(1);
    arr[0]->next = newNode(3);
    arr[0]->next->next = newNode(5);
    arr[0]->next->next->next = newNode(7);

    arr[1] = newNode(2);
    arr[1]->next = newNode(4);
    arr[1]->next->next = newNode(6);
    arr[1]->next->next->next = newNode(8);

    arr[2] = newNode(0);
    arr[2]->next = newNode(9);
    arr[2]->next->next = newNode(10);
```

```
    arr[2]->next->next->next = newNode(11);

    // merge the k sorted lists
    struct Node* head = mergeKSortedLists(arr, k);

    // print the merged list
    printList(head);

    return 0;
}
```

Output:

```
0 1 2 3 4 5 6 7 8 9 10 11
```

Time Complexity: O(nk Logk)

Auxiliary Space: O(k)

15. Merge Sort for Linked Lists

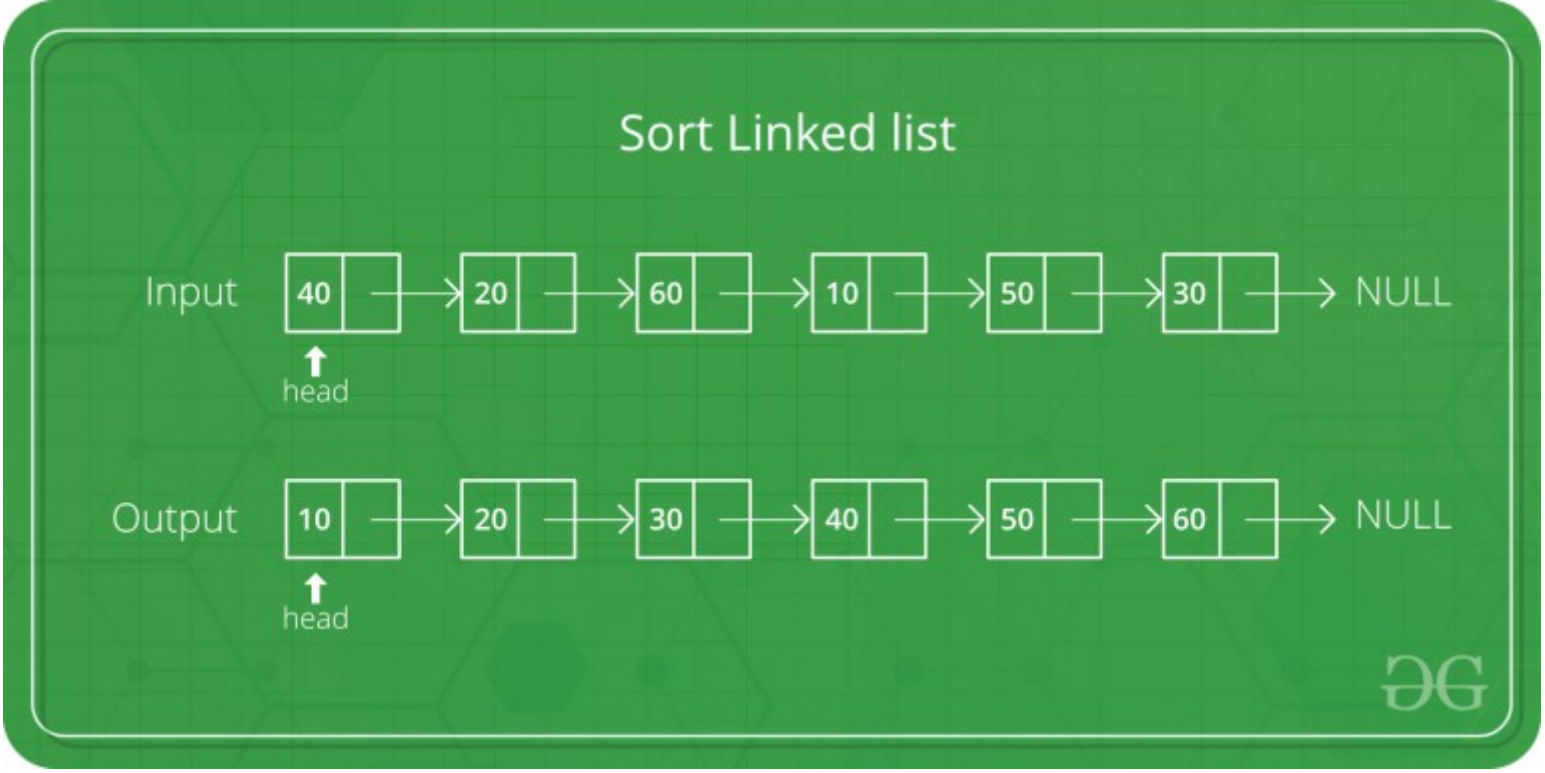
Merge sort is often preferred for sorting a linked list. The slow random-access performance of a linked list makes some other algorithms (such as quicksort) perform poorly, and others (such as heapsort) completely impossible.

Let head be the first node of the linked list to be sorted and headRef be the pointer to head. Note that we need a reference to head in MergeSort() as the below implementation changes next links to sort the linked lists (not data at the nodes), so head node has to be changed if the data at the original head is not the smallest value in the linked list.

```
MergeSort(headRef)
1) If the head is NULL or there is only one element in the Linked List
   then return.
2) Else divide the linked list into two halves.
     FrontBackSplit(head, &a, &b); /* a and b are two halves */
3) Sort the two halves a and b.
     MergeSort(a);
     MergeSort(b);
4) Merge the sorted a and b (using SortedMerge() discussed here)
   and update the head pointer using headRef.
    *headRef = SortedMerge(a, b);
```

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

```
// C++ code for linked list merged sort
#include <bits/stdc++.h>
using namespace std;

/* Link list node */
class Node {
```




```
public:
    int data;
    Node* next;
};

/* function prototypes */
Node* SortedMerge(Node* a, Node* b);
void FrontBackSplit(Node* source,
                    Node** frontRef, Node** backRef);

/* sorts the linked list by changing next pointers (not data) */
void MergeSort(Node** headRef)
{
    Node* head = *headRef;
    Node* a;
    Node* b;

    /* Base case -- length 0 or 1 */
    if ((head == NULL) || (head->next == NULL)) {
        return;
    }

    /* Split head into 'a' and 'b' sublists */
    FrontBackSplit(head, &a, &b);

    /* Recursively sort the sublists */
    MergeSort(&a);
    MergeSort(&b);

    /* answer = merge the two sorted lists together */
    *headRef = SortedMerge(a, b);
}
```

```
/* See https:// www.geeksforgeeks.org/?p=3622 for details of this
function */
Node* SortedMerge(Node* a, Node* b)
{
    Node* result = NULL;

    /* Base cases */
    if (a == NULL)
        return (b);
    else if (b == NULL)
        return (a);

    /* Pick either a or b, and recur */
    if (a->data <= b->data) {
        result = a;
        result->next = SortedMerge(a->next, b);
    }
    else {
        result = b;
        result->next = SortedMerge(a, b->next);
    }
    return (result);
}

/* UTILITY FUNCTIONS */
/* Split the nodes of the given list into front and back halves,
    and return the two lists using the reference parameters.
    If the length is odd, the extra node should go in the front
list.
    Uses the fast/slow pointer strategy. */
void FrontBackSplit(Node* source,
                    Node** frontRef, Node** backRef)
{
    Node* fast;
    Node* slow;
    slow = source;
    fast = source->next;

    /* Advance 'fast' two nodes, and advance 'slow' one node */
    while (fast != NULL) {
        fast = fast->next;
        if (fast != NULL) {
            slow = slow->next;
            fast = fast->next;
        }
    }

    /* 'slow' is before the midpoint in the list, so split it in
two
```

```
    at that point. */
    *frontRef = source;
    *backRef = slow->next;
    slow->next = NULL;
}

/* Function to print nodes in a given linked list */
void printList(Node* node)
{
    while (node != NULL) {
        cout << node->data << " ";
        node = node->next;
    }
}

/* Function to insert a node at the beginging of the linked list
*/
void push(Node** head_ref, int new_data)
{
    /* allocate node */
    Node* new_node = new Node();

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

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Driver program to test above functions*/
int main()
{
    /* Start with the empty list */
    Node* res = NULL;
    Node* a = NULL;

    /* Let us create a unsorted linked lists to test the
functions
Created lists shall be a: 2->3->20->5->10->15 */
    push(&a, 15);
    push(&a, 10);
    push(&a, 5);
    push(&a, 20);
    push(&a, 3);
    push(&a, 2);

    /* Sort the above created Linked List */
```

```
    MergeSort(&a);

    cout << "Sorted Linked List is: \n";
    printList(a);

    return 0;
}

// This is code is contributed by rathbhupendra
```

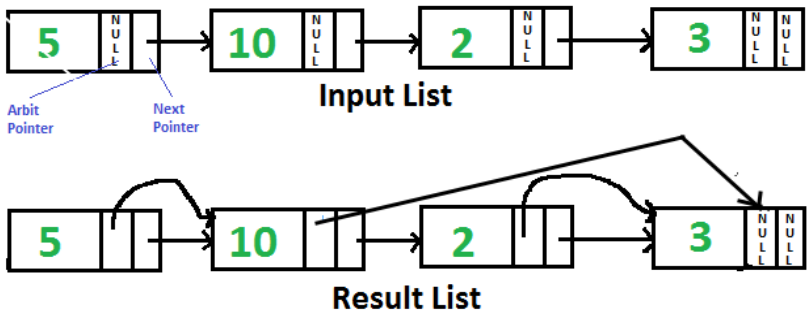
Output:

```
Sorted Linked List is:

2 3 5 10 15 20
```

Time Complexity: O(n Log n)

16. Point arbit pointer to greatest value right side node in a linked list

Given singly linked list with every node having an additional “arbitrary” pointer that currently points to NULL. We need to make the “arbitrary” pointer to greatest value node in a linked list on its right side.



Recommended: Please try your approach on *{IDE}* first, before moving on to the solution.

A Simple Solution is to traverse all nodes one by one. For every node, find the node which has greatest value on right side and change the next pointer. Time Complexity of this solution is O(n2).

An Efficient Solution can work in O(n) time. Below are steps.

1. Reverse given linked list.
2. Start traversing linked list and store maximum value node encountered so far. Make arbit of every node to point to max. If the data in current node is more than max node so far, update max.
3. Reverse modified linked list and return head.

Following is the implementation of above steps.

```
// C++ program to point arbit pointers to highest
// value on its right
#include<bits/stdc++.h>
using namespace std;

/* Link list node */
struct Node
```

```
{
    int data;
    Node* next, *arbit;
};

/* Function to reverse the linked list */
Node* reverse(Node *head)
{
    Node *prev = NULL, *current = head, *next;
    while (current != NULL)
    {
        next  = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    return prev;
}

// This function populates arbit pointer in every
// node to the greatest value to its right.
Node* populateArbit(Node *head)
{
    // Reverse given linked list
    head = reverse(head);

    // Initialize pointer to maximum value node
    Node *max = head;

    // Traverse the reversed list
    Node *temp = head->next;
    while (temp != NULL)
    {
        // Connect max through arbit pointer
        temp->arbit = max;

        // Update max if required
        if (max->data < temp->data)
            max = temp;

        // Move ahead in reversed list
        temp = temp->next;
    }

    // Reverse modified linked list and return
    // head.
    return reverse(head);
}
```

```
// Utility function to print result linked list
void printNextArbitPointers(Node *node)
{
    printf("Node\tNext Pointer\tArbit Pointer\n");
    while (node!=NULL)
    {
        cout << node->data << "\t\t";

        if (node->next)
            cout << node->next->data << "\t\t";
        else cout << "NULL" << "\t\t";

        if (node->arbit)
            cout << node->arbit->data;
        else cout << "NULL";

        cout << endl;
        node = node->next;
    }
}

/* Function to create a new node with given data */
Node *newNode(int data)
{
    Node *new_node = new Node;
    new_node->data = data;
    new_node->next = NULL;
    return new_node;
}

/* Driver program to test above functions*/
int main()
{
    Node *head = newNode(5);
    head->next = newNode(10);
    head->next->next = newNode(2);
    head->next->next->next = newNode(3);

    head = populateArbit(head);

    printf("Resultant Linked List is: \n");
    printNextArbitPointers(head);

    return 0;
}
```

Output:

```
Resultant Linked List is:

Node    Next Pointer    Arbit Pointer
```

```
5               10              10

10              2               3

2               3               3

3               NULL            NULL
```

Recursive Solution:

We can recursively reach the last node and traverse the linked list from end. Recursive solution doesn't require reversing of linked list. We can also use a stack in place of recursion to temporarily hold nodes. Thanks to Santosh Kumar Mishra for providing this solution.

```
// C++ program to point arbit pointers to highest
// value on its right
#include<bits/stdc++.h>
using namespace std;

/* Link list node */
struct Node
{
    int data;
    Node* next, *arbit;
};

// This function populates arbit pointer in every
// node to the greatest value to its right.
void populateArbit(Node *head)
{
    // using static maxNode to keep track of maximum
    // orbit node address on right side
    static Node *maxNode;

    // if head is null simply return the list
    if (head == NULL)
        return;

    /* if head->next is null it means we reached at
       the last node just update the max and maxNode */
    if (head->next == NULL)
    {
        maxNode = head;
        return;
```

```
    }

    /* Calling the populateArbit to the next node */
    populateArbit(head->next);

    /* updating the arbit node of the current
     node with the maximum value on the right side */
    head->arbit = maxNode;

    /* if current Node value id greater then
     the previous right node then  update it */
    if (head->data > maxNode->data)
        maxNode = head;

   return;
}

// Utility function to print result linked list
void printNextArbitPointers(Node *node)
{
    printf("Node\tNext Pointer\tArbit Pointer\n");
    while (node!=NULL)
    {
        cout << node->data << "\t\t";

        if(node->next)
            cout << node->next->data << "\t\t";
        else cout << "NULL" << "\t\t";

        if(node->arbit)
            cout << node->arbit->data;
        else cout << "NULL";

        cout << endl;
        node = node->next;
    }
}

/* Function to create a new node with given data */
Node *newNode(int data)
{
    Node *new_node = new Node;
    new_node->data = data;
    new_node->next = NULL;
    return new_node;
}

/* Driver program to test above functions*/
int main()
```

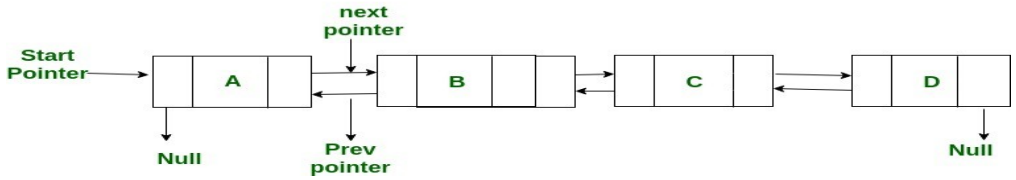
```
{
    Node *head = newNode(5);
    head->next = newNode(10);
    head->next->next = newNode(2);
    head->next->next->next = newNode(3);

    populateArbit(head);

    printf("Resultant Linked List is: \n");
    printNextArbitPointers(head);

    return 0;
}
```

Output:

```
Resultant Linked List is:

Node    Next Pointer    Arbit Pointer

5               10              10

10              2               3

2               3               3

3               NULL            NULL
```

17. XOR Linked List – A Memory Efficient Doubly Linked List | Set 1

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 address field with every node. This memory efficient Doubly Linked List is called XOR Linked List or Memory Efficient 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 Effiecient) 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 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 direction. 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 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 next node. Similarly we can traverse the list in backward direction.

We have covered more on XOR Linked List in the following post.

XOR Linked List – A Memory Efficient Doubly Linked List | Set 2

18. XOR Linked List – A Memory Efficient Doubly Linked List | Set 2

In the previous post, we discussed how a Doubly Linked can be created using only one space for address field with every node. In this post, we will discuss the implementation of memory efficient doubly linked list. We will mainly discuss the following two simple functions.

1) A function to insert a new node at the beginning.

2) A function to traverse the list in forward direction.

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

In the following code, insert() function inserts a new node at the beginning. We need to change the head pointer of Linked List, that is why a double pointer is used (See this). Let us first discuss few things again that have been discussed in the previous post. We store XOR of next and previous nodes with every node and we call it npx, which is the only address member we have with every node. When we insert a new node at the beginning, npx of new node will always be XOR of NULL and current head. And npx of the current head must be changed to XOR of new node and node next to the current head.

printList() traverses the list in forward direction. It prints data values from every node. To traverse the list, we need to get pointer to the next node at every point. We can get the address of next node by keeping track of current node and previous node. If we do XOR of curr->npx and prev, we get the address of next node.

```
/* C++ Implementation of Memory
efficient Doubly Linked List */
#include <bits/stdc++.h>
#include <inttypes.h>
using namespace std;

// Node structure of a memory
// efficient doubly linked list
class Node
```

```
{
    public:
    int data;
    Node* npx; /* XOR of next and previous node */
};

/* returns XORed value of the node addresses */
Node* XOR (Node *a, Node *b)
{
    return (Node*) ((uintptr_t) (a) ^ (uintptr_t) (b));
}

/* Insert a node at the beginning of the
XORed linked list and makes the newly
inserted node as head */
void insert(Node **head_ref, int data)
{
    // Allocate memory for new node
    Node *new_node = new Node();
    new_node->data = data;

    /* Since new node is being inserted at the
    beginning, npx of new node will always be
    XOR of current head and NULL */
    new_node->npx = XOR(*head_ref, NULL);

    /* If linked list is not empty, then npx of
    current head node will be XOR of new node
    and node next to current head */
    if (*head_ref != NULL)
    {
        // *(head_ref)->npx is XOR of NULL and next.
        // So if we do XOR of it with NULL, we get next
        Node* next = XOR((*head_ref)->npx, NULL);
        (*head_ref)->npx = XOR(new_node, next);
    }

    // Change head
    *head_ref = new_node;
}

// prints contents of doubly linked
// list in forward direction
void printList (Node *head)
{
    Node *curr = head;
    Node *prev = NULL;
    Node *next;

    cout << "Following are the nodes of Linked List: \n";
```

```
    while (curr != NULL)
    {
        // print current node
        cout<<curr->data<<" ";

        // get address of next node: curr->npx is
        // next^prev, so curr->npx^prev will be
        // next^prev^prev which is next
        next = XOR (prev, curr->npx);

        // update prev and curr for next iteration
        prev = curr;
        curr = next;
    }
}

// Driver code
int main ()
{
    /* Create following Doubly Linked List
    head-->40<-->30<-->20<-->10 */
    Node *head = NULL;
    insert(&head, 10);
    insert(&head, 20);
    insert(&head, 30);
    insert(&head, 40);

    // print the created list
    printList (head);

    return (0);
}

// This code is contributed by rathbhupendra
```

Output:

```
Following are the nodes of Linked List:

40 30 20 10
```

Note that XOR of pointers is not defined by C/C++ standard. So the above implementation may not work on all platforms.

19. Sort a linked list of 0s, 1s and 2s

Given a linked list of 0s, 1s and 2s, sort it.

Examples:

Input: 1 -> 1 -> 2 -> 0 -> 2 -> 0 -> 1 -> NULL

Output: 0 -> 0 -> 1 -> 1 -> 1 -> 2 -> 2 -> NULL

Input: 1 -> 1 -> 2 -> 1 -> 0 -> NULL

Output: 0 -> 1 -> 1 -> 1 -> 2 -> NULL

Source: Microsoft Interview | Set 1

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

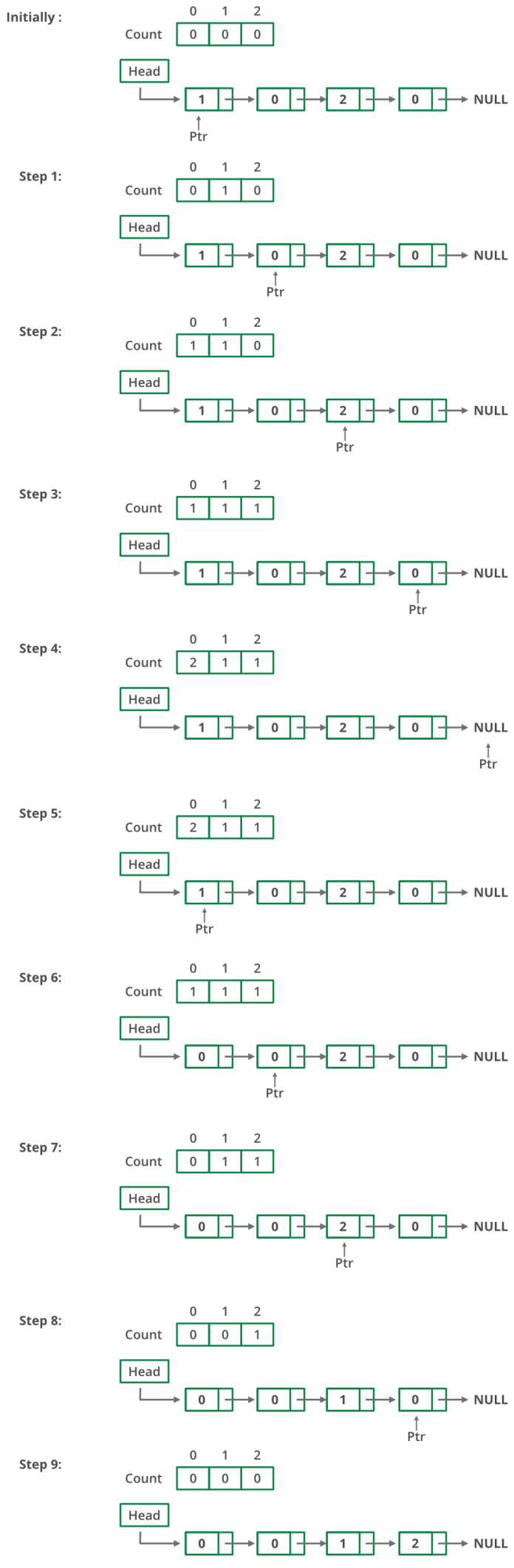
Following steps can be used to sort the given linked list.

- Traverse the list and count the number of 0s, 1s and 2s. Let the counts be n1, n2 and n3 respectively.
- Traverse the list again, fill the first n1 nodes with 0, then n2 nodes with 1 and finally n3 nodes with 2.

Below image is a dry run of the above approach:

Below is the implementation of the above approach:

```
// C++ Program to sort a linked list 0s, 1s or 2s
#include <bits/stdc++.h>
using namespace std;

/* Link list node */
class Node
{
    public:
    int data;
    Node* next;
};

// Function to sort a linked list of 0s, 1s and 2s
void sortList(Node *head)
{
```




```
    int count[3] = {0,
0, 0}; // Initialize
count of '0', '1' and
'2' as 0
    Node *ptr = head;

    /* count total
number of '0', '1' and
'2'
    * count[0] will
store total number of
'0's
    * count[1] will
store total number of
'1's
    * count[2] will
store total number of
'2's */
    while (ptr != NULL)
    {
        count[ptr->data]
+= 1;
        ptr = ptr-
>next;
    }

    int i = 0;
    ptr = head;

    /* Let say count[0]
= n1, count[1] = n2 and
count[2] = n3
    * now start
traversing list from
head node,
    * 1) fill the list
with 0, till n1 > 0
    * 2) fill the list
with 1, till n2 > 0
    * 3) fill the list
with 2, till n3 > 0 */
    while (ptr != NULL)
    {
        if (count[i] ==
0)
            ++i;
        else
        {
            ptr->data =
i;
```

```
            --count[i];
            ptr = ptr->next;
        }
    }
}

/* Function to push a node */
void push (Node** head_ref, int new_data)
{
    /* allocate node */
    Node* new_node = new Node();

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

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Function to print linked list */
void printList(Node *node)
{
    while (node != NULL)
    {
        cout << node->data << " ";
        node = node->next;
    }
    cout << endl;
}

/* Driver code*/
int main(void)
{
    Node *head = NULL;
    push(&head, 0);
    push(&head, 1);
    push(&head, 0);
    push(&head, 2);
    push(&head, 1);
    push(&head, 1);
    push(&head, 2);
    push(&head, 1);
    push(&head, 2);

    cout << "Linked List Before Sorting\n";
    printList(head);
```

```
    sortList(head);

    cout << "Linked List After Sorting\n";
    printList(head);

    return 0;
}

// This code is contributed by rathbhupendra
```

Output:

```
Linked List Before Sorting

2  1  2  1  1  2  0  1  0

Linked List After Sorting

0  0  1  1  1  1  2  2  2
```

Time Complexity: O(n) where n is number of nodes in linked list.

Auxiliary Space: O(1

20. Sort a linked list of 0s, 1s and 2s by changing links

Given a linked list of 0s, 1s and 2s, sort it.

Examples:

```
Input  : 2->1->2->1->1->2->0->1->0

Output : 0->0->1->1->1->1->2->2->2


Input : 2->1->0

Output : 0->1->2
```

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

We have discussed a solution in below post that works by changing data of nodes.

Sort a linked list of 0s, 1s and 2s

The above solution does not work when these values have associated data with them. For example, these three represent three colors and different types of objects associated with the colors and we want to sort objects (connected with a linked list) based on colors.

In this post, a new solution is discussed that works by changing links.

Iterate through the linked list. Maintain 3 pointers named zero, one and two to point to current ending nodes of linked lists containing 0, 1, and 2 respectively. For every traversed node, we attach it to the end of its corresponding list. Finally we link all three lists. To avoid many null checks, we use three dummy pointers zeroD, oneD and twoD that work as dummy headers of three lists.

```
// CPP Program to sort a linked list 0s, 1s
// or 2s by changing links
#include <stdio.h>

/* Link list node */
struct Node {
```

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

Node* newNode(int data);

// Sort a linked list of 0s, 1s and 2s
// by changing pointers.
Node* sortList(Node* head)
{
    if (!head || !(head->next))
        return head;

    // Create three dummy nodes to point to
    // beginning of three linked lists. These
    // dummy nodes are created to avoid many
    // null checks.
    Node* zeroD = newNode(0);
    Node* oneD = newNode(0);
    Node* twoD = newNode(0);

    // Initialize current pointers for three
    // lists and whole list.
    Node* zero = zeroD, *one = oneD, *two = twoD;

    // Traverse list
    Node* curr = head;
    while (curr) {
        if (curr->data == 0) {
            zero->next = curr;
            zero = zero->next;
            curr = curr->next;
        } else if (curr->data == 1) {
            one->next = curr;
            one = one->next;
            curr = curr->next;
        } else {
            two->next = curr;
            two = two->next;
            curr = curr->next;
        }
    }

    // Attach three lists
    zero->next = (oneD->next) ? (oneD->next) : (twoD->next);
    one->next = twoD->next;
    two->next = NULL;

    // Updated head
```

```
    head = zeroD->next;

    // Delete dummy nodes
    delete zeroD;
    delete oneD;
    delete twoD;

    return head;
}

// function to create and return a node
Node* newNode(int data)
{
    // allocating space
    Node* newNode = new Node;

    // inserting the required data
    newNode->data = data;
    newNode->next = NULL;
}

/* Function to print linked list */
void printList(struct Node* node)
{
    while (node != NULL) {
        printf("%d  ", node->data);
        node = node->next;
    }
    printf("\n");
}

/* Driver program to test above function*/
int main(void)
{
    // Creating the list 1->2->4->5
    Node* head = newNode(1);
    head->next = newNode(2);
    head->next->next = newNode(0);
    head->next->next->next = newNode(1);

    printf("Linked List Before Sorting\n");
    printList(head);

    head = sortList(head);

    printf("Linked List After Sorting\n");
    printList(head);

    return 0;
}
```

Output :

```
Linked List Before Sorting

1  2  0  1

Linked List After Sorting

0  1  1  2
```

Thanks to Musarrat_123 for suggesting above solution in a comment here.

Time Complexity: O(n) where n is number of nodes in linked list.

Auxiliary Space: O(1)

21. Add 1 to a number represented as linked list

Number is represented in linked list such that each digit corresponds to a node in linked list. Add 1 to it. For example 1999 is represented as (1-> 9-> 9 -> 9) and adding 1 to it should change it to (2->0->0->0)

Recommended: Please solve it on "*PRACTICE*" first, before moving on to the solution.

Below are the steps :

1. Reverse given linked list. For example, 1-> 9-> 9 -> 9 is converted to 9-> 9 -> 9 ->1.
2. Start traversing linked list from leftmost node and add 1 to it. If there is a carry, move to the next node. Keep moving to the next node while there is a carry.
3. Reverse modified linked list and return head.

Below is the implementation of above steps.

```
// C++ program to add 1 to a linked list
#include <bits/stdc++.h>
using namespace std;

/* Linked list node */
class Node
{
    public:
    int data;
    Node* next;
};

/* Function to create a new node with given data */
Node *newNode(int data)
{
    Node *new_node = new Node;
    new_node->data = data;
    new_node->next = NULL;
    return new_node;
}

/* Function to reverse the linked list */
Node *reverse(Node *head)
{
    Node * prev = NULL;
```

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

/* Adds one to a linked lists and return the head
node of resultant list */
Node *addOneUtil(Node *head)
{
    // res is head node of the resultant list
    Node* res = head;
    Node *temp, *prev = NULL;

    int carry = 1, sum;

    while (head != NULL) //while both lists exist
    {
        // Calculate value of next digit in resultant list.
        // The next digit is sum of following things
        // (i) Carry
        // (ii) Next digit of head list (if there is a
        // next digit)
        sum = carry + head->data;

        // update carry for next calulation
        carry = (sum >= 10)? 1 : 0;

        // update sum if it is greater than 10
        sum = sum % 10;

        // Create a new node with sum as data
        head->data = sum;

        // Move head and second pointers to next nodes
        temp = head;
        head = head->next;
    }

    // if some carry is still there, add a new node to
    // result list.
    if (carry > 0)
        temp->next = newNode(carry);
```

```
    // return head of the resultant list
    return res;
}

// This function mainly uses addOneUtil().
Node* addOne(Node *head)
{
    // Reverse linked list
    head = reverse(head);

    // Add one from left to right of reversed
    // list
    head = addOneUtil(head);

    // Reverse the modified list
    return reverse(head);
}

// A utility function to print a linked list
void printList(Node *node)
{
    while (node != NULL)
    {
        cout << node->data;
        node = node->next;
    }
    cout<<endl;
}

/* Driver program to test above function */
int main(void)
{
    Node *head = newNode(1);
    head->next = newNode(9);
    head->next->next = newNode(9);
    head->next->next->next = newNode(9);

    cout << "List is ";
    printList(head);

    head = addOne(head);

    cout << "\nResultant list is ";
    printList(head);

    return 0;
}

// This is code is contributed by rathbhupendra
```

Output:

```
List is 1999



Resultant list is 2000
```

Recursive Implementation:

We can recursively reach the last node and forward carry to previous nodes.

Recursive solution doesn’t require reversing of linked list. We can also use a stack in place of recursion to temporarily hold nodes.

Below is the implementation of recursive solution.

```
// Recursive C++ program to add 1 to a linked list
#include<bits/stdc++.h>

/* Linked list node */
struct Node
{
    int data;
    Node* next;
};

/* Function to create a new node with given data */
Node *newNode(int data)
{
    Node *new_node = new Node;
    new_node->data = data;
    new_node->next = NULL;
    return new_node;
}

// Recursively add 1 from end to beginning and returns
// carry after all nodes are processed.
int addWithCarry(Node *head)
{
    // If linked list is empty, then
    // return carry
    if (head == NULL)
        return 1;

    // Add carry returned be next node call
    int res = head->data + addWithCarry(head->next);

    // Update data and return new carry
```

```
    head->data = (res) % 10;
    return (res) / 10;
}

// This function mainly uses addWithCarry().
Node* addOne(Node *head)
{
    // Add 1 to linked list from end to beginning
    int carry = addWithCarry(head);

    // If there is carry after processing all nodes,
    // then we need to add a new node to linked list
    if (carry)
    {
        Node *newNode = new Node;
        newNode->data = carry;
        newNode->next = head;
        return newNode; // New node becomes head now
    }

    return head;
}

// A utility function to print a linked list
void printList(Node *node)
{
    while (node != NULL)
    {
        printf("%d", node->data);
        node = node->next;
    }
    printf("\n");
}

/* Driver program to test above function */
int main(void)
{
    Node *head = newNode(1);
    head->next = newNode(9);
    head->next->next = newNode(9);
    head->next->next->next = newNode(9);

    printf("List is ");
    printList(head);

    head = addOne(head);

    printf("\nResultant list is ");
    printList(head);
```

```
    return 0;
}
```
Output:

```
List is 1999



Resultant list is 2000
```

22. Function to check if a singly linked list is palindrome

Given a singly linked list of characters, write a function that returns true if the given list is a palindrome, else false.

METHOD 1 (Use a Stack)

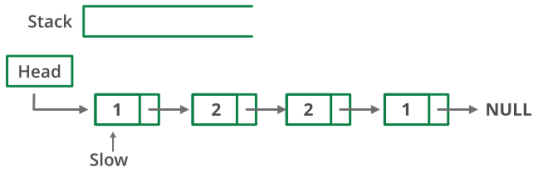
- A simple solution is to use a stack of list nodes. This mainly involves three steps.
- Traverse the given list from head to tail and push every visited node to stack.
- Traverse the list again. For every visited node, pop a node from stack and compare data of popped node with currently visited node.
- If all nodes matched, then return true, else false.

Below image is a dry run of the above approach:

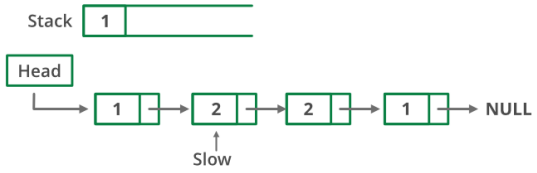
Below is the implementation of the above approach

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

class Node {
public:
        int data;
        Node(int d){
            data = d;
        }
        Node *ptr;
};

// Function to check if the linked list
```

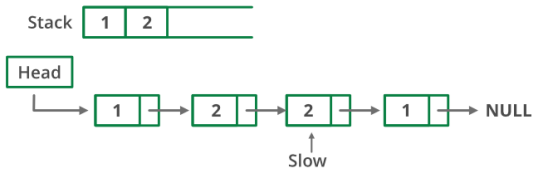
Initially :



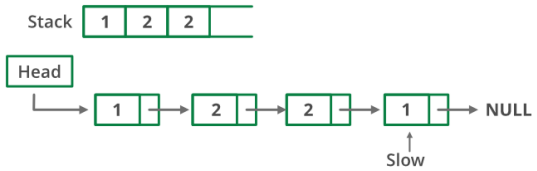
Step 1:



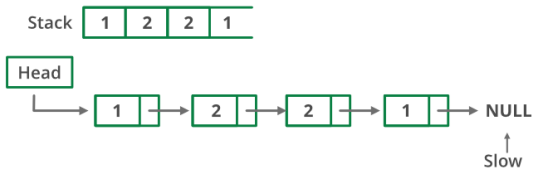
Step 2:



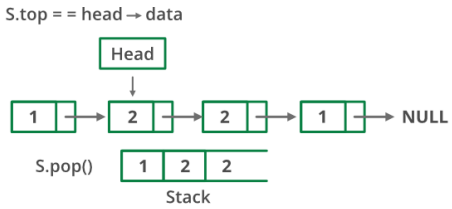
Step 3:



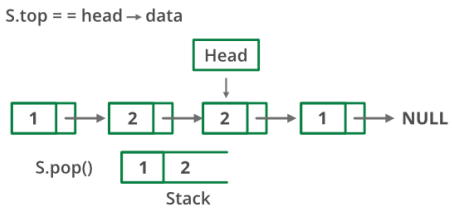
Step 4:



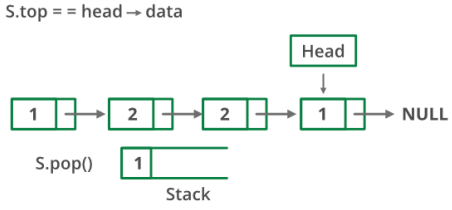
Step 5:



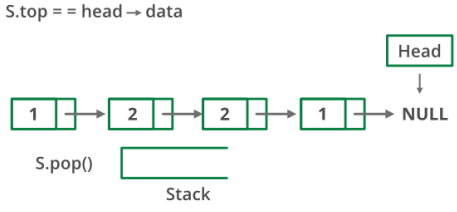
Step 6:



Step 7:



Step 8:



```
// is palindrome or not
bool isPalin(Node* head){

        // Temp pointer
        Node* slow= head;

        // Declare a stack
        stack <int> s;


        // Push all elements of the list
        // to the stack
        while(slow != NULL){
                s.push(slow->data);

                // Move ahead
                slow = slow->ptr;
        }

        // Iterate in the list again and
        // check by popping from the stack
        while(head != NULL ){

            // Get the top most element
             int i=s.top();

             // Pop the element
             s.pop();

             // Check if data is not
             // same as popped element
            if(head -> data != i){
                return false;
            }

             // Move ahead
           head=head->ptr;
        }

return true;
}

// Driver Code
int main(){

    // Addition of linked list
    Node one =  Node(1);
    Node two = Node(2);
    Node three = Node(3);
```

```
    Node four = Node(2);
    Node five = Node(1);

    // Initialize the next pointer
    // of every current pointer
    five.ptr = NULL;
    one.ptr = &two;
    two.ptr = &three;
    three.ptr = &four;
    four.ptr = &five;
    Node* temp = &one;


    // Call function to check palindrome or not
    int result = isPalin(&one);

    if(result == 1)
            cout<<"isPalindrome is true\n";
    else
        cout<<"isPalindrome is true\n";

return 0;
}

// This code has been contributed by Striver
```

Output

```
isPalindrome: true
```

The time complexity of the above method is O(n).

METHOD 2 (By reversing the list)

This method takes O(n) time and O(1) extra space.

1) Get the middle of the linked list.

2) Reverse the second half of the linked list.

3) Check if the first half and second half are identical.

4) Construct the original linked list by reversing the second half again and attaching it back to the first half

To divide the list in two halves, method 2 of this post is used.

When number of nodes are even, the first and second half contain exactly half nodes. The challenging thing in this method is to handle the case when number of nodes are odd. We don’t want the middle node as part of any of the lists as

we are going to compare them for equality. For odd case, we use a separate variable ‘midnode’.

```
/* Program to check if a linked list is palindrome */
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

/* Link list node */
struct Node {
    char data;
    struct Node* next;
};

void reverse(struct Node**);
bool compareLists(struct Node*, struct Node*);

/* Function to check if given linked list is
  palindrome or not */
bool isPalindrome(struct Node* head)
{
    struct Node *slow_ptr = head, *fast_ptr = head;
    struct Node *second_half, *prev_of_slow_ptr = head;
    struct Node* midnode = NULL; // To handle odd size list
    bool res = true; // initialize result

    if (head != NULL && head->next != NULL) {
        /* Get the middle of the list. Move slow_ptr by 1
          and fast_ptrr by 2, slow_ptr will have the middle
          node */
        while (fast_ptr != NULL && fast_ptr->next != NULL) {
            fast_ptr = fast_ptr->next->next;

            /*We need previous of the slow_ptr for
             linked lists  with odd elements */
            prev_of_slow_ptr = slow_ptr;
            slow_ptr = slow_ptr->next;
        }

        /* fast_ptr would become NULL when there are even elements
in list.
           And not NULL for odd elements. We need to skip the
middle node
           for odd case and store it somewhere so that we can
restore the
           original list*/
        if (fast_ptr != NULL) {
            midnode = slow_ptr;
            slow_ptr = slow_ptr->next;
```

```
        }

        // Now reverse the second half and compare it with first
half
        second_half = slow_ptr;
        prev_of_slow_ptr->next = NULL; // NULL terminate first
half
        reverse(&second_half); // Reverse the second half
        res = compareLists(head, second_half); // compare

        /* Construct the original list back */
        reverse(&second_half); // Reverse the second half again

        // If there was a mid node (odd size case) which
        // was not part of either first half or second half.
        if (midnode != NULL) {
            prev_of_slow_ptr->next = midnode;
            midnode->next = second_half;
        }
        else
            prev_of_slow_ptr->next = second_half;
    }
    return res;
}

/* Function to reverse the linked list  Note that this
    function may change the head */
void reverse(struct Node** head_ref)
{
    struct Node* prev = NULL;
    struct Node* current = *head_ref;
    struct Node* next;
    while (current != NULL) {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    *head_ref = prev;
}

/* Function to check if two input lists have same data*/
bool compareLists(struct Node* head1, struct Node* head2)
{
    struct Node* temp1 = head1;
    struct Node* temp2 = head2;

    while (temp1 && temp2) {
        if (temp1->data == temp2->data) {
```

```
            temp1 = temp1->next;
            temp2 = temp2->next;
        }
        else
            return 0;
    }

    /* Both are empty reurn 1*/
    if (temp1 == NULL && temp2 == NULL)
        return 1;

    /* Will reach here when one is NULL
      and other is not */
    return 0;
}

/* Push a node to linked list. Note that this function
  changes the head */
void push(struct Node** head_ref, char new_data)
{
    /* allocate node */
    struct Node* new_node = (struct Node*)malloc(sizeof(struct
Node));

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

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to pochar to the new node */
    (*head_ref) = new_node;
}

// A utility function to print a given linked list
void printList(struct node* ptr)
{
    while (ptr != NULL) {
        printf("%c->", ptr->data);
        ptr = ptr->next;
    }
    printf("NULL\n");
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct Node* head = NULL;
    char str[] = "abacaba";
```

```
    int i;

    for (i = 0; str[i] != '\0'; i++) {
        push(&head, str[i]);
        printList(head);
        isPalindrome(head) ? printf("Is Palindrome\n\n") :
printf("Not Palindrome\n\n");
    }

    return 0;
}
```

Output:

```
a->NULL

Palindrome



b->a->NULL

Not Palindrome



a->b->a->NULL

Is Palindrome



c->a->b->a->NULL

Not Palindrome



a->c->a->b->a->NULL

Not Palindrome



b->a->c->a->b->a->NULL

Not Palindrome



a->b->a->c->a->b->a->NULL

Is Palindrome
```

Time Complexity: O(n)

Auxiliary Space: O(1)

METHOD 3 (Using Recursion)

Use two pointers left and right. Move right and left using recursion and check for following in each recursive call.

1) Sub-list is palindrome.

2) Value at current left and right are matching.

If both above conditions are true then return true.

The idea is to use function call stack as container. Recursively traverse till the end of list. When we return from last NULL, we will be at last node. The last node to be compared with first node of list.

In order to access first node of list, we need list head to be available in the last call of recursion. Hence we pass head also to the recursive function. If they both match we need to compare (2, n-2) nodes. Again when recursion falls back to (n-2)nd node, we need reference to 2nd node from head. We advance the head pointer in previous call, to refer to next node in the list.

However, the trick in identifying double pointer. Passing single pointer is as good as pass-by-value, and we will pass the same pointer again and again. We need to pass the address of head pointer for reflecting the changes in parent recursive calls.

Thanks to Sharad Chandra for suggesting this approach.

```
// Recursive program to check if a given linked list is palindrome
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

/* Link list node */
struct node {
    char data;
    struct node* next;
};

// Initial parameters to this function are &head and head
```

```
bool isPalindromeUtil(struct node** left, struct node* right)
{
    /* stop recursion when right becomes NULL */
    if (right == NULL)
        return true;

    /* If sub-list is not palindrome then no need to
       check for current left and right, return false */
    bool isp = isPalindromeUtil(left, right->next);
    if (isp == false)
        return false;

    /* Check values at current left and right */
    bool isp1 = (right->data == (*left)->data);

    /* Move left to next node */
    *left = (*left)->next;

    return isp1;
}

// A wrapper over isPalindromeUtil()
bool isPalindrome(struct node* head)
{
    isPalindromeUtil(&head, head);
}

/* Push a node to linked list. Note that this function
  changes the head */
void push(struct node** head_ref, char new_data)
{
    /* allocate node */
    struct node* new_node = (struct node*)malloc(sizeof(struct
node));

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

    /* link the old list off the new node */
    new_node->next = (*head_ref);

    /* move the head to pochar to the new node */
    (*head_ref) = new_node;
}

// A utility function to print a given linked list
void printList(struct node* ptr)
{
    while (ptr != NULL) {
        printf("%c->", ptr->data);
```

```
        ptr = ptr->next;
    }
    printf("NULL\n");
}

/* Driver program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;
    char str[] = "abacaba";
    int i;

    for (i = 0; str[i] != '\0'; i++) {
        push(&head, str[i]);
        printList(head);
        isPalindrome(head) ? printf("Is Palindrome\n\n") :
printf("Not Palindrome\n\n");
    }

    return 0;
}
```

Output:

```
a->NULL

Not Palindrome



b->a->NULL

Not Palindrome



a->b->a->NULL

Is Palindrome



c->a->b->a->NULL

Not Palindrome



a->c->a->b->a->NULL

Not Palindrome
```

```
b->a->c->a->b->a->NULL

Not Palindrome


a->b->a->c->a->b->a->NULL

Is Palindrome
```

Time Complexity: O(n)

Auxiliary Space: O(n) if Function Call Stack size is considered, otherwise O(1).