[Date]

Complete placement course

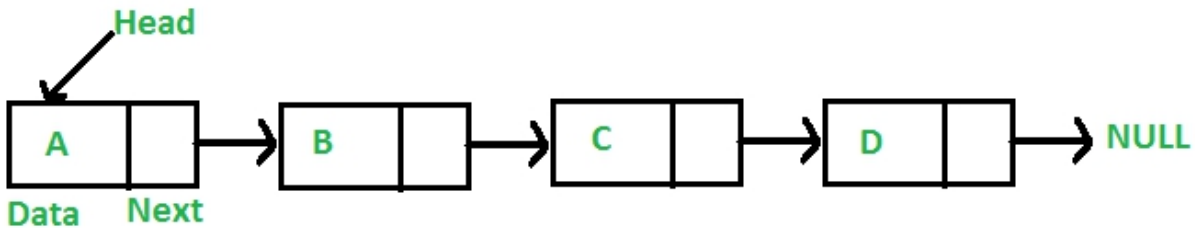
Singly linked list

ANIL KUMAR DWIVEDI
BADA VIDYALAY

LINKED LIST

Singly Linked List :

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:



In simple words, a linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.

Why Linked List?

Arrays can be used to store linear data of similar types, but arrays have the following limitations.

1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.

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

For example, in a system, if we maintain a sorted list of IDs in an array

id[].
id[] = [1000, 1010, 1050, 2000, 2040].
And if we want to insert a new ID 1005, then to maintain the sorted order we have to move all the elements after 1000 (excluding 1000).
Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved.

Advantages over arrays

1) Dynamic size
2) Ease of insertion/deletion

Drawbacks:

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

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

3) Not cache friendly. Since array elements are contiguous locations, there is locality of reference which is not there in case of linked lists.

Representation:

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

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

1) data
2) Pointer (Or Reference) to the next node
In C, we can represent a node using structures. Below is an example of a linked list node with integer data.

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

First Simple Linked List in C++ Let us create a simple linked list with 3 nodes.

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

class Node {
public:
    int data;
    Node* next;
};

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

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

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

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

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

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

    head          second          third
        |             |              |
        |             |              |
    +---+---+     +----+----+     +-----+----+
    | 1 | o----->| # | # |     | # | # |
    +---+---+     +----+----+     +-----+----+
*/

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

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

    /* data has been assigned to the data part of the second
    block (block pointed by second). And next
    pointer of the second block points to the third
    block. So all three blocks are linked.

    head          second          third
        |             |              |
        |             |              |
    +---+---+     +---+---+     +----+----+
    | 1 | o----->| 2 | o-----> | # | # |
    +---+---+     +---+---+     +----+----+     */

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

    /* data has been assigned to the data part of the third
    block (block pointed by third). And next pointer
    of the third block is made NULL to indicate
    that the linked list is terminated here.

    We have the linked list ready.

        head
            |
            |
        +---+---+     +---+---+     +----+------+
        | 1 | o----->| 2 | o-----> | 3 | NULL |
        +---+---+     +---+---+     +----+------+


    Note that only the head is sufficient to represent
    the whole list. We can traverse the complete
    list by following the next pointers. */

    return 0;
}
```

Linked List Traversal

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

```cpp
// A simple C++ program for traversal of a linked list
#include <bits/stdc++.h>
using namespace std;

class Node {
public:
    int data;
    Node* next;
};

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

// Driver code
int main()
{
    Node* head = NULL;
    Node* second = NULL;
    Node* third = NULL;

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

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

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

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

    printList(head);

    return 0;
}
```

Output:

```
1  2  3
```

Methods to insert a new node in linked list.

A node can be added in three ways

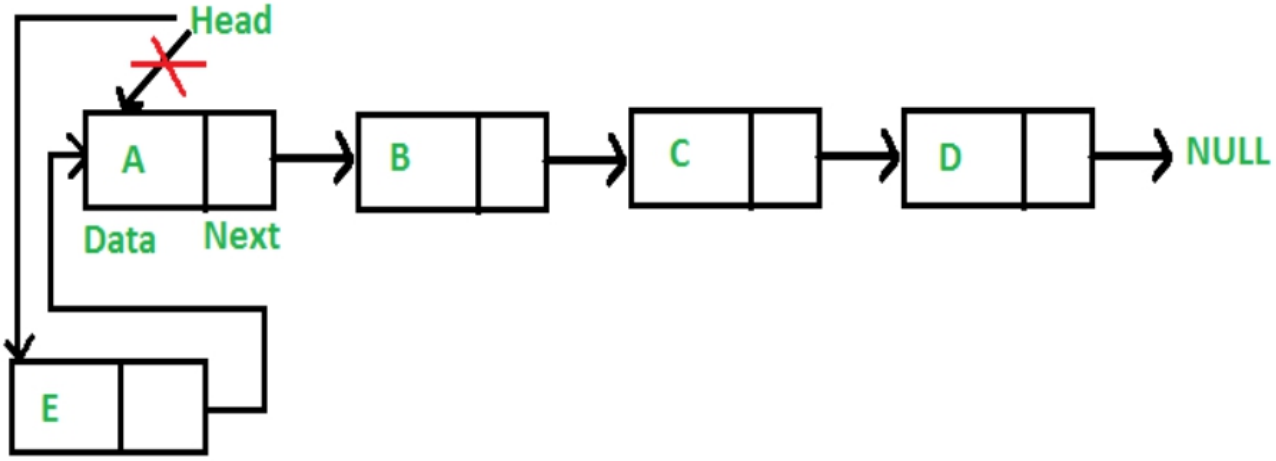
1) At the front of the linked list
2) After a given node.
3) At the end of the linked list.

1) Add a node at the front: (4 steps process)

The new node is always added before the head of the given Linked List. And newly added node becomes the new head of the Linked List.

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

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



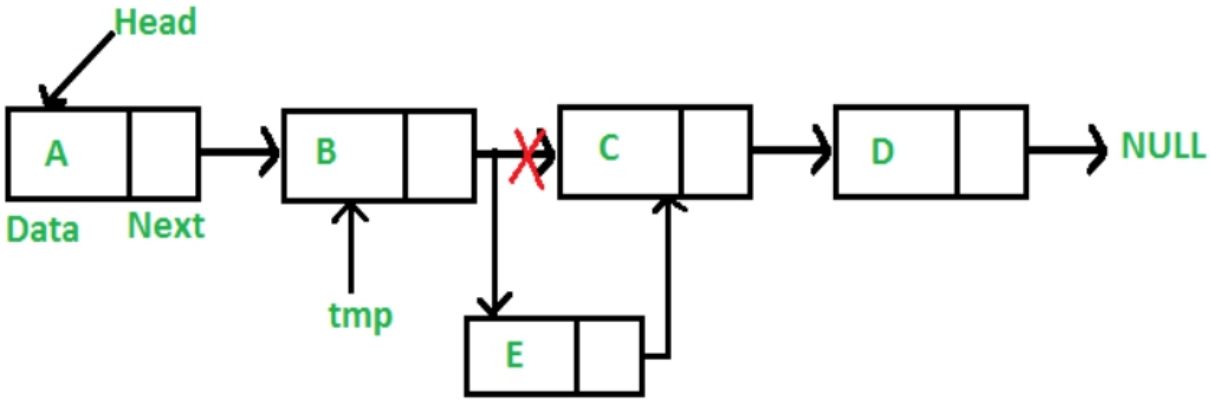
Following are the 4 steps to add a node at the front.

```
/* Given a reference (pointer to pointer)
to the head of a list and an int,
inserts a new node on the front of the list. */
void push(Node** head_ref, int new_data)
{
    /* 1. allocate node */
    Node* new_node = new Node();

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

    /* 3. Make next of new node as head */
    new_node->next = (*head_ref);

    /* 4. move the head to point to the new node */
    (*head_ref) = new_node;
}

// This code is contributed by rathbhupendra

```

Time complexity of push() is O(1) as it does a constant amount of work.

2)Add a node after a given node: (5 steps process)

We are given a pointer to a node, and the new node is inserted after the given node.



Time complexity of insertAfter() is O(1) as it does a constant amount of work.

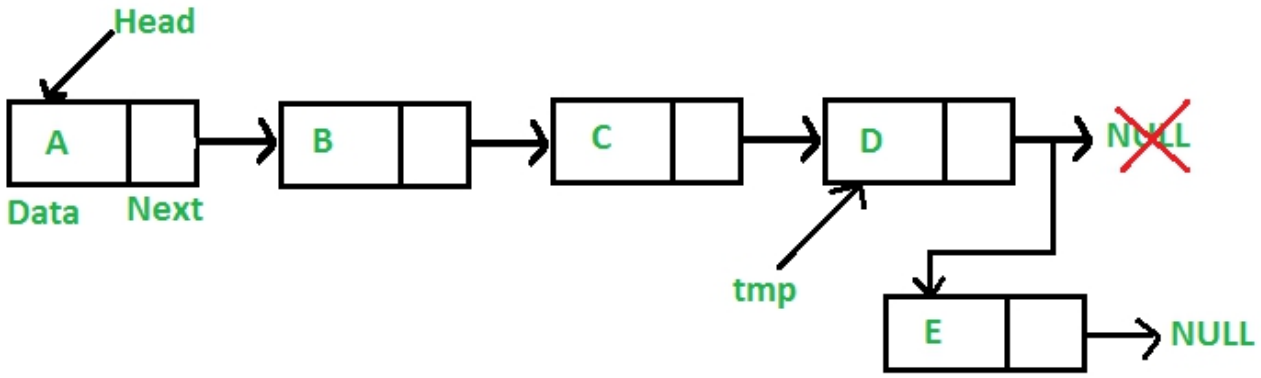
```cpp
// Given a node prev_node, insert a
// new node after the given
// prev_node
void insertAfter(Node* prev_node, int new_data)
{

    // 1. Check if the given prev_node is NULL
    if (prev_node == NULL)
    {
        cout << "the given previous node cannot be NULL";
        return;
    }

    // 2. Allocate new node
    Node* new_node = new Node();

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

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

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

3)Add a node at the end: (6 steps process)

The new node is always added after the last node of the given Linked List.

For example if the given Linked List is 5->10->15->20->25 and we add an item 30 at the end, then the Linked List becomes 5->10->15->20->25->30.

Since a Linked List is typically represented by the head of it, we have to traverse the list till the end and then change the next to last node to a new node.



Following are the 6 steps to add node at the end.

```
// Given a reference (pointer to pointer) to the head
// of a list and an int, appends a new node at the end
void append(Node** head_ref, int new_data)
{

    // 1. allocate node
    Node* new_node = new Node();

    // Used in step 5
    Node *last = *head_ref;

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

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

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

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

     // 6. Change the next of last node
     last->next = new_node;
     return;
}

// This code is contributed by anmolgautam818
```

Time complexity of append is O(n) where n is the number of nodes in linked list. Since there is a loop from head to end, the function does O(n) work.
This method can also be optimized to work in O(1) by keeping an extra pointer to the tail of linked list/

Following is a complete program that uses all of the above methods to create a linked list.

```cpp
// A complete working C++ program to demonstrate
// all insertion methods on Linked List
#include <bits/stdc++.h>
using namespace std;

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

/* Given a reference (pointer to pointer)
to the head of a list and an int, inserts
a new node on the front of the list. */
void push(Node** head_ref, int new_data)
{
    /* 1. allocate node */
    Node* new_node = new Node();

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

    /* 3. Make next of new node as head */
    new_node->next = (*head_ref);

    /* 4. move the head to point to the new node */
    (*head_ref) = new_node;
}

/* Given a node prev_node, insert a new node after the given
prev_node */
void insertAfter(Node* prev_node, int new_data)
{
    /*1. check if the given prev_node is NULL */
    if (prev_node == NULL)
    {
        cout<<"the given previous node cannot be NULL";
        return;
    }

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

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

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

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

```
/* Given a reference (pointer to pointer) to the head
of a list and an int, appends a new node at the end */
void append(Node** head_ref, int new_data)
{
    /* 1. allocate node */
    Node* new_node = new Node();

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

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

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

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

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

    /* 6. Change the next of last node */
    last->next = new_node;
    return;
}

// This function prints contents of
// linked list starting from head
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;

```

```
107     // Insert 6. So linked list becomes 6->NULL
108     append(&head, 6);
109
110     // Insert 7 at the beginning.
111     // So linked list becomes 7->6->NULL
112     push(&head, 7);
113
114     // Insert 1 at the beginning.
115     // So linked list becomes 1->7->6->NULL
116     push(&head, 1);
117
118     // Insert 4 at the end. So
119     // linked list becomes 1->7->6->4->NULL
120     append(&head, 4);
121
122     // Insert 8, after 7. So linked
123     // list becomes 1->7->8->6->4->NULL
124     insertAfter(head->next, 8);
125
126     cout<<"Created Linked list is: ";
127     printList(head);
128
129     return 0;
130 }
```

Output:

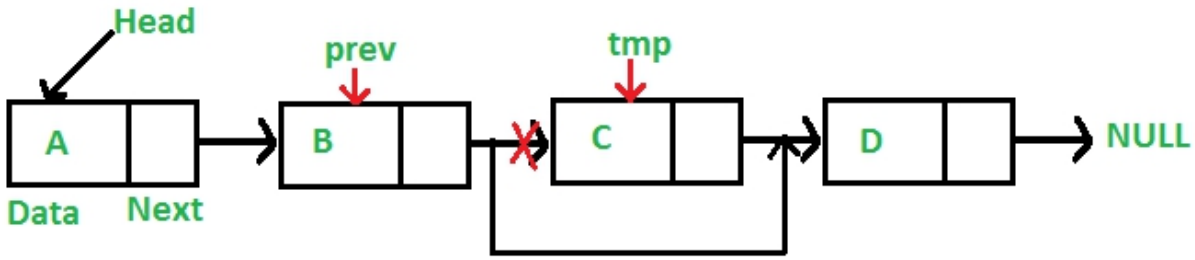
```
Created Linked list is:  1  7  8  6  4
```

Deleting a node

Given a 'key', delete the first occurrence of this key in the linked list.

Iterative Method:

To delete a node from the linked list, we need to do the following steps.
1) Find the previous node of the node to be deleted.
2) Change the next of the previous node.
3) Free memory for the node to be deleted.




```cpp
// A complete working C++ program to
// demonstrate deletion in singly
// linked list with class
#include <bits/stdc++.h>
using namespace std;

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

// Given a reference (pointer to pointer)
// to the head of a list and an int,
// inserts a new node on the front of the
// list.
void push(Node** head_ref, int new_data)
{
    Node* new_node = new Node();
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

// Given a reference (pointer to pointer)
// to the head of a list and a key, deletes
// the first occurrence of key in linked list
void deleteNode(Node** head_ref, int key)
{

    // Store head node
    Node* temp = *head_ref;
    Node* prev = NULL;

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

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

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

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

    // Free memory
    delete temp;
    }
}

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

// Driver code
int main()
{

    // Start with the empty list
    Node* head = NULL;

    // Add elements in linked list
    push(&head, 7);
    push(&head, 1);
    push(&head, 3);
    push(&head, 2);

    puts("Created Linked List: ");
    printList(head);

    deleteNode(&head, 1);
    puts("\nLinked List after Deletion of 1: ");

```

```
 99      printList(head);
100
101      return 0;
102  }
103
```

Output

```
Created Linked List:
 2  3  1  7
Linked List after Deletion of 1:
 2  3  7
```

Delete a Linked List node at a given position

Given a singly linked list and a position, delete a linked list node at the given position.

Example:

```
Input: position = 1, Linked List = 8->2->3->1->7
Output: Linked List =  8->3->1->7

Input: position = 0, Linked List = 8->2->3->1->7
Output: Linked List = 2->3->1->7
```

If the node to be deleted is the root, simply delete it. To delete a middle node, we must have a pointer to the node previous to the node to be deleted. So if positions are not zero, we run a loop position-1 times and get a pointer to the previous node.

Below is the implementation of the above idea.

```cpp
// A complete working C++ program to delete
// a node in a linked list at a given position
#include <iostream>
using namespace std;

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

// Given a reference (pointer to pointer) to
// the head of a list and an int inserts a
// new node on the front of the list.
void push(Node** head_ref, int new_data)
{
    Node* new_node = new Node();
    new_node->data = new_data;
    new_node->next = (*head_ref);
    (*head_ref) = new_node;
}

// Given a reference (pointer to pointer) to
// the head of a list and a position, deletes
// the node at the given position
void deleteNode(Node **head_ref, int position)
{

    // If linked list is empty
    if (*head_ref == NULL)
        return;

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

    // If head needs to be removed
    if (position == 0)
    {

        // Change head
        *head_ref = temp->next;

        // Free old head
        free(temp);
        return;
    }

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

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

```
57
58      // Node temp->next is the node to be deleted
59      // Store pointer to the next of node to be deleted
60      Node *next = temp->next->next;
61
62      // Unlink the node from linked list
63      free(temp->next); // Free memory
64
65      // Unlink the deleted node from list
66      temp->next = next;
67  }
68
69  // This function prints contents of linked
70  // list starting from the given node
71  void printList( Node *node)
72  {
73      while (node != NULL)
74      {
75          cout << node->data << " ";
76          node = node->next;
77      }
78  }
79
80  // Driver code
81  int main()
82  {
83
84      // Start with the empty list
85      Node* head = NULL;
86
87      push(&head, 7);
88      push(&head, 1);
89      push(&head, 3);
90      push(&head, 2);
91      push(&head, 8);
92
93      cout << "Created Linked List: ";
94      printList(head);
95      deleteNode(&head, 4);
96      cout << "\nLinked List after Deletion at position 4: ";
97      printList(head);
98      return 0;
99  }
100
```

Output

```
Created Linked List:
 8  2  3  1  7
Linked List after Deletion at position 4:
 8  2  3  1
```

Write a function

to delete a Linked List

Algorithm For C/C++: Iterate through the linked list and delete all the nodes one by one. The main point here is not to access the next of the current pointer if the current pointer is deleted.

```
/* Function to delete the entire linked list */
void deleteList(Node** head_ref)
{

    /* deref head_ref to get the real head */
    Node* current = *head_ref;
    Node* next = NULL;

    while (current != NULL)
    {
        next = current->next;
        free(current);
        current = next;
    }

    /* deref head_ref to affect the real head back
        in the caller. */
    *head_ref = NULL;
}
```

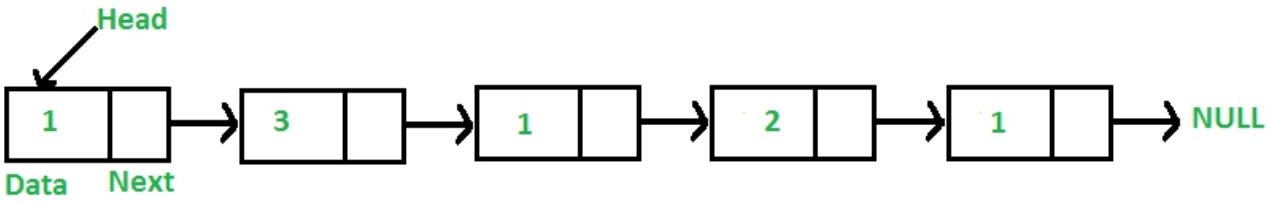
```
/* Driver code*/
int main()
{
    /* Start with the empty list */
    Node* head = NULL;

    /* Use push() to construct below list
    1->12->1->4->1 */
    push(&head, 1);
    push(&head, 4);
    push(&head, 1);
    push(&head, 12);
    push(&head, 1);

    cout << "Deleting linked list";
    deleteList(&head);

    cout << "\nLinked list deleted";
}
```

Time Complexity: O(n)

Auxiliary Space: $O_{(1)}$

Find Length of a Linked List (Iterative and Recursive)

Write a function to count the number of nodes in a given singly linked list.



Output: 5

For example, the function should return 5 for linked list 1->3->1->2->1.

Iterative Solution

```
1) Initialize count as 0
2) Initialize a node pointer, current = head.
3) Do following while current is not NULL
     a) current = current -> next
     b) count++;
4) Return count
```

Following are the Iterative implementations of the above algorithm to find the count of nodes in a given singly linked list.

```
/* Counts no. of nodes in linked list */
int getCount(Node* head)
{
    int count = 0; // Initialize count
    Node* current = head; // Initialize current
    while (current != NULL)
    {
        count++;
        current = current->next;
    }
    return count;
}
/* Driver program to test count function*/
int main()
{
    /* Start with the empty list */
    Node* head = NULL;

    /* Use push() to construct below list
    1->2->1->3->1 */
    push(&head, 1);
    push(&head, 3);
    push(&head, 1);
    push(&head, 2);
    push(&head, 1);

    /* Check the count function */
    cout<<"count of nodes is "<< getCount(head);
    return 0;
}
```

Output:

```
count of nodes is 5
```

Recursive Solution

```
int getCount(head)
1) If head is NULL, return 0.
2) Else return 1 + getCount(head->next)
```

Following are the Recursive implementations of the above algorithm to find the count of nodes in a given singly linked list.

```
/* Recursively count number of nodes in linked list */
int getCount(Node* head)
{
    // Base Case
    if (head == NULL) {
        return 0;
    }
    // Count this node plus the rest of the list
    else {
        return 1 + getCount(head->next);
    }
}

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

    /* Use push() to construct below list
    1->2->1->3->1 */
    push(&head, 1);
    push(&head, 3);
    push(&head, 1);
    push(&head, 2);
    push(&head, 1);

    /* Check the count function */
    cout << "Count of nodes is " << getCount(head);
    return 0;
}
```

Output

Output:

```
Count of nodes is 5
```

Search an element in a

Linked List (Iterative and Recursive)

Write a function that searches a given key ‘x’ in a given singly linked list. The function should return true if x is present in linked list and false otherwise.

Iterative Solution

```
2) Initialize a node pointer, current = head.
3) Do following while current is not NULL
    a) current->key is equal to the key being searched return true.
    b) current = current->next
4) Return false
```

Following is iterative implementation of above algorithm to search a given key.

```cpp
// Iterative C++ program to search
// an element in linked list
#include <bits/stdc++.h>
using namespace std;

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

/* Given a reference (pointer to pointer) to the head
of a list and an int, push a new node on the front
of the list. */
void push(Node** head_ref, int new_key)
{
```

```
    /* allocate node */
    Node* new_node = new Node();

    /* put in the key */
    new_node->key = new_key;

    /* 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;
}

/* Checks whether the value x is present in linked list */
bool search(Node* head, int x)
{
    Node* current = head; // Initialize current
    while (current != NULL)
    {
        if (current->key == x)
            return true;
        current = current->next;
    }
    return false;
}

/* Driver program to test count function*/
int main()
{
    /* Start with the empty list */
    Node* head = NULL;
    int x = 21;

    /* Use push() to construct below list
    14->21->11->30->10 */
    push(&head, 10);
    push(&head, 30);
    push(&head, 11);
    push(&head, 21);
    push(&head, 14);

    search(head, 21)? cout<<"Yes" : cout<<"No";
    return 0;
}
```

Output

```
Yes
```

Recursive Solution

```
bool search(head, x)
1) If head is NULL, return false.
2) If head's key is same as x, return true;
2) Else return search(head->next, x)
```

```c
/* Checks whether the value x is present in linked list */
bool search(struct Node* head, int x)
{
    // Base case
    if (head == NULL)
        return false;

    // If key is present in current node, return true
    if (head->key == x)
        return true;

    // Recur for remaining list
    return search(head->next, x);
}
```

Write a function to get Nth node in a Linked List

Write a GetNth() function that takes a linked list and an integer index and returns the data value stored in the node at that index position.

Example:

```
Input:  1->10->30->14,  index = 2
Output: 30
The node at index 2 is 30
```

Algorithm:

```
1. Initialize count = 0
2. Loop through the link list
     a. if count is equal to the passed index then return current
         node
     b. Increment count
     c. change current to point to next of the current.
```

```
// Takes head pointer of
// the linked list and index
// as arguments and return
// data at index
int GetNth(Node* head, int index)
{

    Node* current = head;

    // the index of the
    // node we're currently
    // looking at
    int count = 0;
    while (current != NULL) {
        if (count == index)
            return (current->data);
        count++;
        current = current->next;
    }

    /* if we get to this line,
    the caller was asking
    for a non-existent element
    so we assert fail */
    assert(0);
}
// Driver Code
int main()
{

    // Start with the
    // empty list
    Node* head = NULL;

    // Use push() to construct
    // below list
    // 1->12->1->4->1
    push(&head, 1);
    push(&head, 4);
    push(&head, 1);
    push(&head, 12);
    push(&head, 1);

    // Check the count
    // function
    cout << "Element at index 3 is " << GetNth(head, 3);
    return 0;
}
```

Output

```
Element at index 3 is 4
```

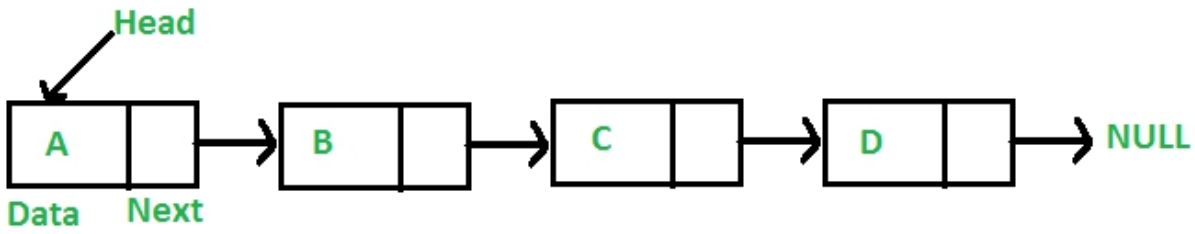
Time Complexity: O(n)

Method 2- With Recursion

```
/* Takes head pointer of the linked list and index
    as arguments and return data at index. (Don't use
another variable)*/
int GetNth(struct Node* head, int n)
{
    // if length of the list is less
    // than the given index, return -1
    if (head == NULL)
        return -1;

    // if n equal to 0 return node->data
    if (n == 0)
        return head->data;

    // increase head to next pointer
    // n - 1: decrease the number of recursions until n = 0
    return GetNth(head->next, n - 1);
}
```

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"



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.

Double pointer concept : First pointer is used to store the address of the variable and second pointer used to store the address of the first pointer. If we wish to change the value of a variable by a function, we pass pointer to it. And if we wish to change value of a pointer (i. e., it should start pointing to something else), we pass pointer to a pointer.

Below is the implementation of the above approach:

```cpp
// 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;

    // 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;
}

```

```
39  void push(struct Node** head_ref, int new_data)
40  {
41      /* allocate node */
42      struct Node* new_node = new Node();
43
44      /* put in the data */
45      new_node->data = new_data;
46
47      /* link the old list off the new node */
48      new_node->next = (*head_ref);
49
50      /* move the head to point to the new node */
51      (*head_ref) = new_node;
52  }
53
54  // Driver Code
55  int main()
56  {
57      /* Start with the empty list */
58      struct Node* head = NULL;
59
60      // create linked 35->15->4->20
61      push(&head, 20);
62      push(&head, 4);
63      push(&head, 15);
64      push(&head, 35);
65
66      printNthFromLast(head, 4);
67      return 0;
68  }
```

Output

```
35
```

Following is a recursive C code for the same method

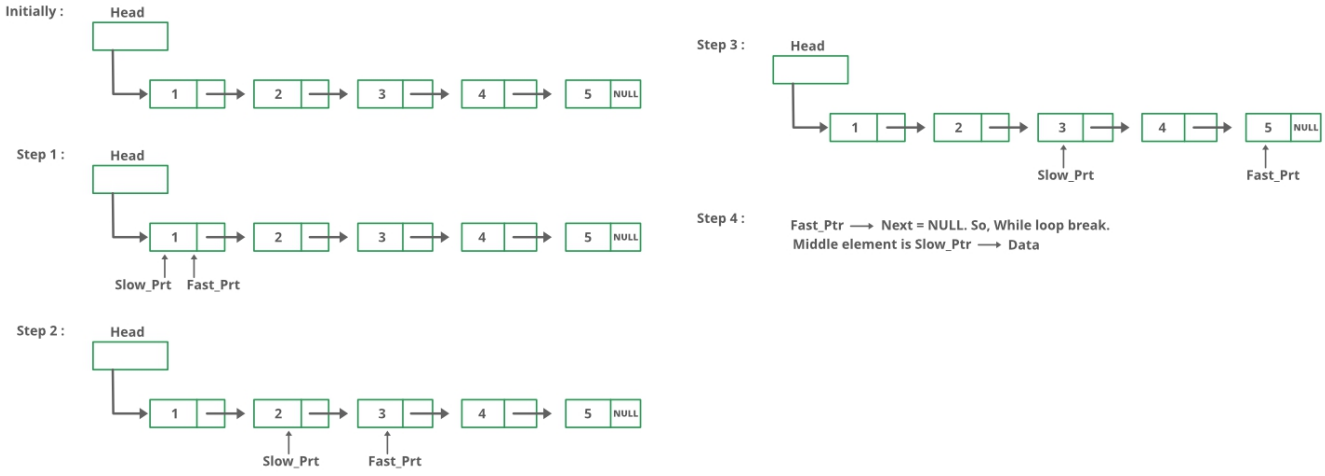
```
void printNthFromLast(struct Node* head, int n)
{
    static int i = 0;
    if (head == NULL)
        return;
    printNthFromLast(head->next, n);
    if (++i == n)
        printf("%d", head->data);
}
```

Find the middle of a given linked list

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

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

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 the other pointers by two. When the fast pointer reaches the end slow pointer will reach the middle of the linked list.
Below image shows how printMiddle function works in the code :




```
void printMiddle(class 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;
        }
        cout << "The middle element is [" << slow_ptr->data << "]" << endl;
    }
}
```

Write a function

that counts the number of times a given int occurs in a Linked List

Given a singly linked list and a key, count the number of occurrences of the given key in the linked list. For example, if the given linked list is 1->2->1->2->1->3->1 and the given key is 1, then the output should be 4.

Method 1- Without Recursion

Algorithm:

1. Initialize count as zero.
2. Loop through each element of linked list:
   a) If element data is equal to the passed number then increment the count.
3. Return count.

```
/* Counts the no. of occurrences of a node
(search_for) in a linked list (head)*/
int count(Node* head, int search_for)
{
    Node* current = head;
    int count = 0;
    while (current != NULL) {
        if (current->data == search_for)
            count++;
        current = current->next;
    }
    return count;
}
```

Time Complexity: O(n)
Auxiliary Space: O(1)

Method 2- With Recursion

Algorithm:

```
Algorithm
count(head, key);
if head is NULL
return frequency
if(head->data==key)
increase frequency by 1
count(head->next, key)
```

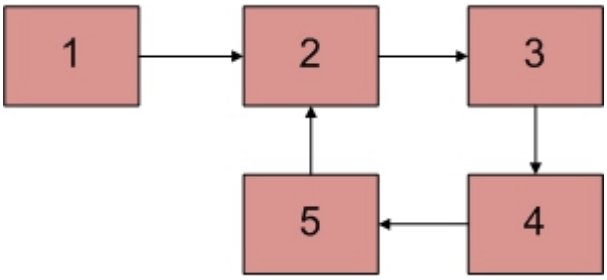
```c
/* Counts the no. of occurrences of a node
(search_for) in a linked list (head)*/
int count(struct Node* head, int key)
{
    if (head == NULL)
        return frequency;
    if (head->data == key)
        frequency++;
    return count(head->next, key);
}
```

Below method can be used to avoid Global variable 'frequency'

```c
// method can be used to avoid
// Global variable 'frequency'

/* Counts the no. of occurrences of a node
(search_for) in a linked list (head)*/
int count(struct Node* head, int key)
{
    if (head == NULL)
        return 0;
    if (head->data == key)
        return 1 + count(head->next, key);
    return count(head->next, key);
}
```

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.



Floyd's Cycle-Finding Algorithm

Approach: 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.

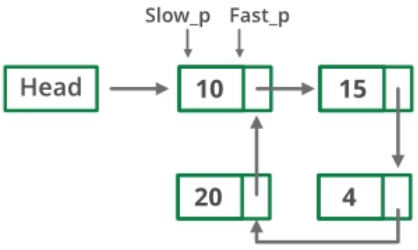
The below image shows how the detectloop function works in the code:

Complexity Analysis:

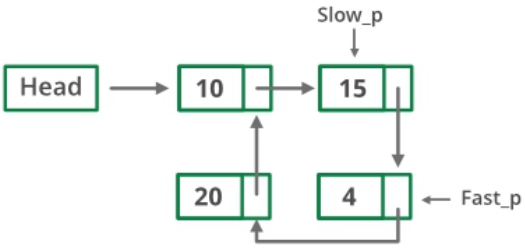
- **Time complexity:** O(n).
  Only one traversal of the loop is needed.

- **Auxiliary Space:**O(1).
  There is no space required.

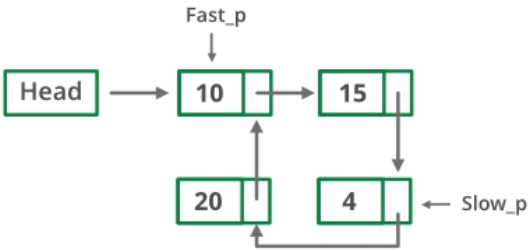
Initially :



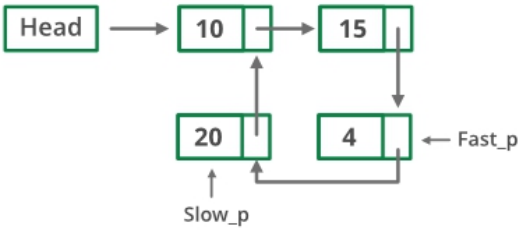
Step 1:



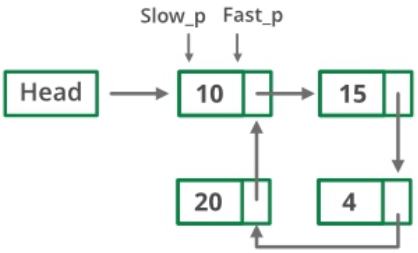
Step 2:



Step 3:



Step 4:



Loop Detected

```cpp
// 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) {
            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;
    if (detectLoop(head))
        cout << "Loop found";
    else
        cout << "No Loop";
    return 0;
}
```

Output

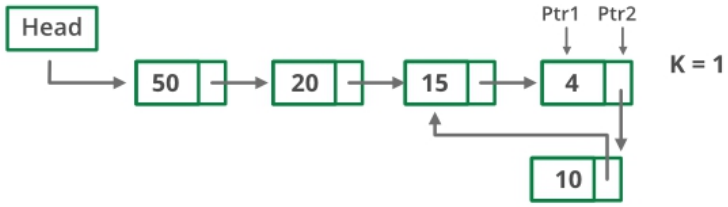
```
Loop found
```

Detect and Remove Loop in a Linked List

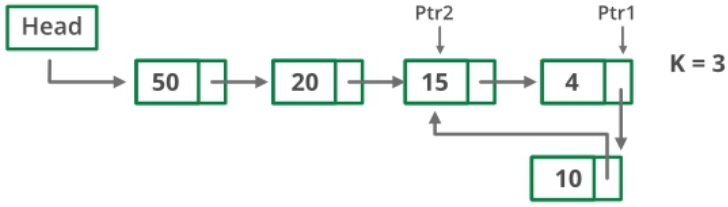
1. This method is also dependent on Floyd's Cycle detection algorithm.
2. Detect Loop using Floyd's Cycle detection algorithm and get the pointer to a loop node.
3. Count the number of nodes in loop. Let the count be k.
4. Fix one pointer to the head and another to a kth node from the head.
5. Move both pointers at the same pace, they will meet at loop starting node.
6. Get a pointer to the last node of the loop and make next of it as NULL.

Below image is a dry run of 'remove loop' function in the code :

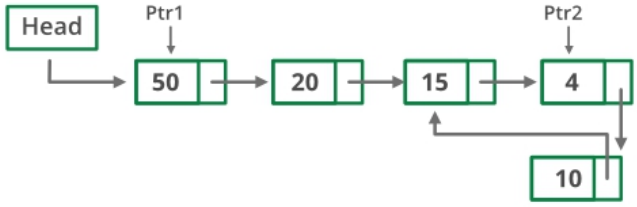
Initially : Detect loop and call 'remove loop' function.
Loop found at node 4



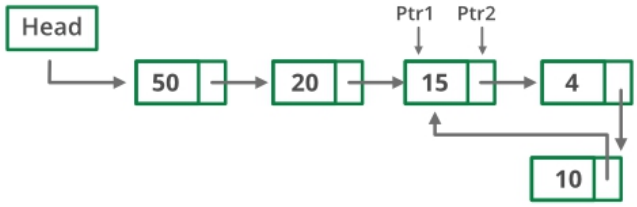
Step 1: Find number of nodes involved in loop



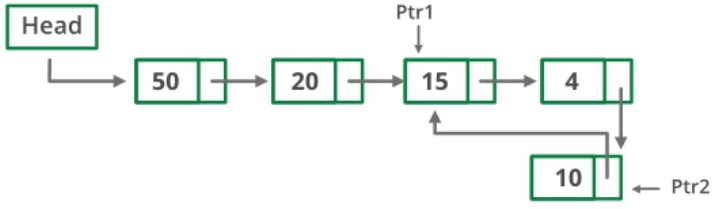
Step 2: Fix one pointer to head and other pointer to K nodes after head



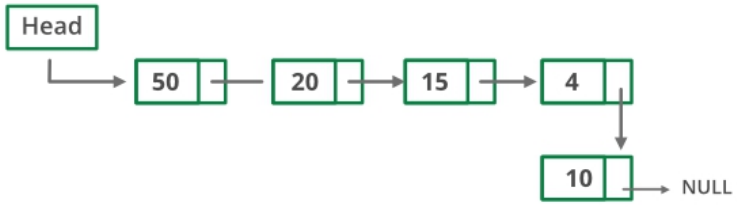
Step 3: Find starting node of loop



Step 4: Find last node of the loop



Step 5: Remove loop



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

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

/* Function to remove loop. */
void removeLoop(struct Node*, struct Node*);

/* This function detects and removes loop in the list
If loop was there in the list then it returns 1,
otherwise returns 0 */
int detectAndRemoveLoop(struct Node* list)
{
    struct Node *slow_p = list, *fast_p = list;

    // Iterate and find if loop exists or not
    while (slow_p && fast_p && fast_p->next) {
        slow_p = slow_p->next;
        fast_p = fast_p->next->next;

        /* If slow_p and fast_p meet at some point then there
        is a loop */
        if (slow_p == fast_p) {
            removeLoop(slow_p, list);

            /* Return 1 to indicate that loop is found */
            return 1;
        }
    }

    /* Return 0 to indicate that there is no loop*/
    return 0;
}

/* Function to remove loop.
loop_node --> Pointer to one of the loop nodes
head --> Pointer to the start node of the linked list */
void removeLoop(struct Node* loop_node, struct Node* head)
{
    struct Node* ptr1 = loop_node;
    struct Node* ptr2 = loop_node;

    // Count the number of nodes in loop
    unsigned int k = 1, i;
    while (ptr1->next != ptr2) {
        ptr1 = ptr1->next;
        k++;
    }

    // Fix one pointer to head
    ptr1 = head;

```

```
    // And the other pointer to k nodes after head
    ptr2 = head;
    for (i = 0; i < k; i++)
        ptr2 = ptr2->next;

    /* Move both pointers at the same pace,
    they will meet at loop starting node */
    while (ptr2 != ptr1) {
        ptr1 = ptr1->next;
        ptr2 = ptr2->next;
    }

    // Get pointer to the last node
    while (ptr2->next != ptr1)
        ptr2 = ptr2->next;

    /* Set the next node of the loop ending node
    to fix the loop */
    ptr2->next = NULL;
}

/* Function to print linked list */
void printList(struct Node* node)
{
    // Print the list after loop removal
    while (node != NULL) {
        cout << node->data << " ";
        node = node->next;
    }
}

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

// Driver Code
int main()
{
    struct 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;

    detectAndRemoveLoop(head);

    cout << "Linked List after removing loop \n";
    printList(head);
    return 0;
}
```

Output:

```
Linked List after removing loop
50 20 15 4 10
```

Find length of loop in linked list

Approach: It is known that Floyd's Cycle detection algorithm terminates when fast and slow pointers meet at a common point. It is also known that this common point is one of the loop nodes. Store the address of this common point in a pointer variable say (ptr). Then initialize a counter with 1 and start from the common point and keeps on visiting the next node and increasing the counter till the common pointer is reached again.
At that point, the value of the counter will be equal to the length of the loop.

Algorithm:

1. Find the common point in the loop by using the Floyd's Cycle detection algorithm
2. Store the pointer in a temporary variable and keep a *count = 0*
3. Traverse the linked list until the same node is reached again and increase the count while moving to next node.
4. Print the count as length of loop

```cpp
// C++ program to count number of nodes
// in loop in a linked list if loop is
// present
#include<bits/stdc++.h>
using namespace std;

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

```

```
// Returns count of nodes present in loop.
int countNodes(struct Node *n)
{
    int res = 1;
    struct Node *temp = n;
    while (temp->next != n)
    {
        res++;
        temp = temp->next;
    }
    return res;
}

/* This function detects and counts loop
nodes in the list. If loop is not there
in then returns 0 */
int countNodesinLoop(struct Node *list)
{
    struct 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 and fast_p meet at
        some point then there is a loop */
        if (slow_p == fast_p)
            return countNodes(slow_p);
    }

    /* Return 0 to indeciate that
    their is no loop*/
    return 0;
}

struct Node *newNode(int key)
{
    struct Node *temp =
(struct Node*)malloc(sizeof(struct Node));
    temp->data = key;
    temp->next = NULL;
    return temp;
}

// Driver Code
int main()
{
    struct 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 */
    head->next->next->next->next->next = head->next;

    cout << countNodesinLoop(head) << endl;

    return 0;
}
```

Output :

```
4
```

Complexity Analysis:

- **Time complexity:** O(n).
  Only one traversal of the linked list is needed.
- **Auxiliary Space:** O(1).
  As no extra space is required.

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.

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 a 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 a container. Recursively traverse till the end of list. When we return from last NULL, we will be at the 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 the head. We advance the head pointer in the previous call, to refer to the next node in the list.

However, the trick is identifying a double-pointer. Passing a 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 the head pointer for reflecting the changes in parent recursive calls.

```cpp
// Recursive program to check if a given linked list is palindrome
#include <bits/stdc++.h>
using namespace std;

/* 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) {
        cout << ptr->data << "->";
        ptr = ptr->next;
    }
    cout << "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) ? cout << "Is Palindrome\n\n" : cout << "Not Palindrome\n\n";
    }

    return 0;
}
```

Time Complexity: O(n)

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

Output

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
```

Remove duplicates from a sorted linked list

Write a function that takes a list sorted in non-decreasing order and deletes any duplicate nodes from the list. The list should only be traversed once.
For example if the linked list is 11->11->11->21->43->43->60 then removeDuplicates() should convert the list to 11->21->43->60.

Algorithm:
Traverse the list from the head (or start) node. While traversing, compare each node with its next node. If the data of the next node is the

same as the current node then delete the next node. Before we delete a node, we need to store the next pointer of the node

Implementation:

Functions other than removeDuplicates() are just to create a linked list and test removeDuplicates().

```cpp
/* C++ Program to remove duplicates from a sorted linked list */
#include <bits/stdc++.h>
using namespace std;

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

/* The function removes duplicates from a sorted list */
void removeDuplicates(Node* head)
{
    /* Pointer to traverse the linked list */
    Node* current = head;

    /* Pointer to store the next pointer of a node to be deleted*/
    Node* next_next;

    /* do nothing if the list is empty */
    if (current == NULL)
    return;

    /* Traverse the list till last node */
    while (current->next != NULL)
    {
    /* Compare current node with next node */
    if (current->data == current->next->data)
    {
        /* The sequence of steps is important*/
        next_next = current->next->next;
        free(current->next);
        current->next = next_next;
    }
    else /* This is tricky: only advance if no deletion */
    {
        current = current->next;
    }
    }
}

```

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

    /* Let us create a sorted linked list to test the functions
    Created linked list will be 11->11->11->13->13->20 */
    push(&head, 20);
    push(&head, 13);
    push(&head, 13);
    push(&head, 11);
    push(&head, 11);
    push(&head, 11);

    cout<<"Linked list before duplicate removal ";
    printList(head);

    /* Remove duplicates from linked list */
    removeDuplicates(head);

    cout<<"\nLinked list after duplicate removal ";
    printList(head);

    return 0;
}
```

Output

```
Linked list before duplicate removal  11 11 11 13 13 20
Linked list after duplicate removal  11 13 20
```

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

```
/* The function removes duplicates
from a sorted list */
void removeDuplicates(Node* head)
{
    /* Pointer to store the pointer of a node to be deleted*/
    Node* to_free;

    /* do nothing if the list is empty */
    if (head == NULL)
        return;

    /* Traverse the list till last node */
    if (head->next != NULL)
    {

        /* Compare head node with next node */
        if (head->data == head->next->data)
        {
            /* The sequence of steps is important.
              to_free pointer stores the next of head
             pointer which is to be deleted.*/
            to_free = head->next;
        head->next = head->next->next;
        free(to_free);
        removeDuplicates(head);
        }
        else /* This is tricky: only
        advance if no deletion */
        {
            removeDuplicates(head->next);
        }
    }
}
```

Remove duplicates from an unsorted linked list

Write a removeDuplicates() function that takes a list and deletes any duplicate nodes from the list. The list is not sorted.
For example if the linked list is 12->11->12->21->41->43->21 then removeDuplicates() should convert the list to 12->11->21->41->43.

METHOD 1 (Using two loops)
This is the simple way where two loops are used. Outer loop is used to pick the elements one by one and the inner loop compares the picked element with the rest of the elements.
Thanks to Gaurav Saxena for his help in writing this code.

```cpp
/* C++ Program to remove duplicates in an unsorted
linked list */
#include <bits/stdc++.h>
using namespace std;

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

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

/* Function to remove duplicates from a
unsorted linked list */
void removeDuplicates(struct Node* start)
{
    struct Node *ptr1, *ptr2, *dup;
    ptr1 = start;

```

```
    /* Pick elements one by one */
    while (ptr1 != NULL && ptr1->next != NULL) {
        ptr2 = ptr1;

        /* Compare the picked element with rest
        of the elements */
        while (ptr2->next != NULL) {
            /* If duplicate then delete it */
            if (ptr1->data == ptr2->next->data) {
                /* sequence of steps is important here */
                dup = ptr2->next;
                ptr2->next = ptr2->next->next;
                delete (dup);
            }
            else /* This is tricky */
                ptr2 = ptr2->next;
        }
        ptr1 = ptr1->next;
    }
}

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

/* Driver program to test above function */
int main()
{
    /* The constructed linked list is:
    10->12->11->11->12->11->10*/
    struct Node* start = newNode(10);
    start->next = newNode(12);
    start->next->next = newNode(11);
    start->next->next->next = newNode(11);
    start->next->next->next->next = newNode(12);
    start->next->next->next->next->next = newNode(11);
    start->next->next->next->next->next->next = newNode(10);

    printf("Linked list before removing duplicates ");
    printList(start);

    removeDuplicates(start);

    printf("\nLinked list after removing duplicates ");
    printList(start);

    return 0;
}
```

Output:

```
Linked list before removing duplicates:
 10 12 11 11 12 11 10
Linked list after removing duplicates:
 10 12 11
```

Swap nodes in a linked list without swapping data

Given a linked list and two keys in it, swap nodes for two given keys. Nodes should be swapped by changing links. Swapping data of nodes may be expensive in many situations when data contains many fields.

It may be assumed that all keys in the linked list are distinct.

Examples:

```
Input : 10->15->12->13->20->14,  x = 12, y = 20
Output: 10->15->20->13->12->14

Input : 10->15->12->13->20->14,  x = 10, y = 20
Output: 20->15->12->13->10->14

Input : 10->15->12->13->20->14,  x = 12, y = 13
Output: 10->15->13->12->20->14
```

This may look a simple problem, but is an interesting question as it has the following cases to be handled.

1. x and y may or may not be adjacent.
2. Either x or y may be a head node.
3. Either x or y may be the last node.
4. x and/or y may not be present in the linked list.

```cpp
// C++ program to swap two given nodes of a linked list
#include <iostream>

using namespace std;

// A linked list node class
class Node {

public:
    int data;
    class Node* next;

    // constructor
    Node(int val, Node* next)
        : data(val)
        , next(next)
    {
    }

    // print list from this
    // to last till null
    void printList()
    {

        Node* node = this;

        while (node != NULL) {

            cout << node->data << " ";
            node = node->next;
        }

        cout << endl;
    }
};

// Function to add a node
// at the beginning of List
void push(Node** head_ref, int new_data)
{

    // allocate node
    (*head_ref) = new Node(new_data, *head_ref);
}

void swap(Node*& a, Node*& b)
{

    Node* temp = a;
    a = b;
    b = temp;
}
```

```
void swapNodes(Node** head_ref, int x, int y)
{

    // Nothing to do if x and y are same
    if (x == y)
        return;

    Node **a = NULL, **b = NULL;

    // search for x and y in the linked list
    // and store therir pointer in a and b
    while (*head_ref) {

        if ((*head_ref)->data == x) {
            a = head_ref;
        }

        else if ((*head_ref)->data == y) {
            b = head_ref;
        }

        head_ref = &((*head_ref)->next);
    }

    // if we have found both a and b
    // in the linked list swap current
    // pointer and next pointer of these
    if (a && b) {

        swap(*a, *b);
        swap(((*a)->next), ((*b)->next));
    }
}

// Driver code
int main()
{

    Node* start = NULL;

    // The constructed linked list is:
    // 1->2->3->4->5->6->7
    push(&start, 7);
    push(&start, 6);
    push(&start, 5);
    push(&start, 4);
    push(&start, 3);
    push(&start, 2);
    push(&start, 1);

    cout << "Linked list before calling swapNodes() ";
    start->printList();

    swapNodes(&start, 6, 1);

    cout << "Linked list after calling swapNodes() ";
    start->printList();
}
```

Output

```
Linked list before calling swapNodes() 1 2 3 4 5 6 7
Linked list after calling swapNodes() 6 2 3 4 5 1 7
```

Pairwise swap elements of a given linked list

Given a singly linked list, write a function to swap elements pairwise.

Input : 1->2->3->4->5->6->NULL
Output : 2->1->4->3->6->5->NULL
Input : 1->2->3->4->5->NULL
Output : 2->1->4->3->5->NULL
Input : 1->NULL
Output : 1->NULL

For example, if the linked list is 1->2->3->4->5 then the function should change it to 2->1->4->3->5, and if the linked list is then the function should change it to.

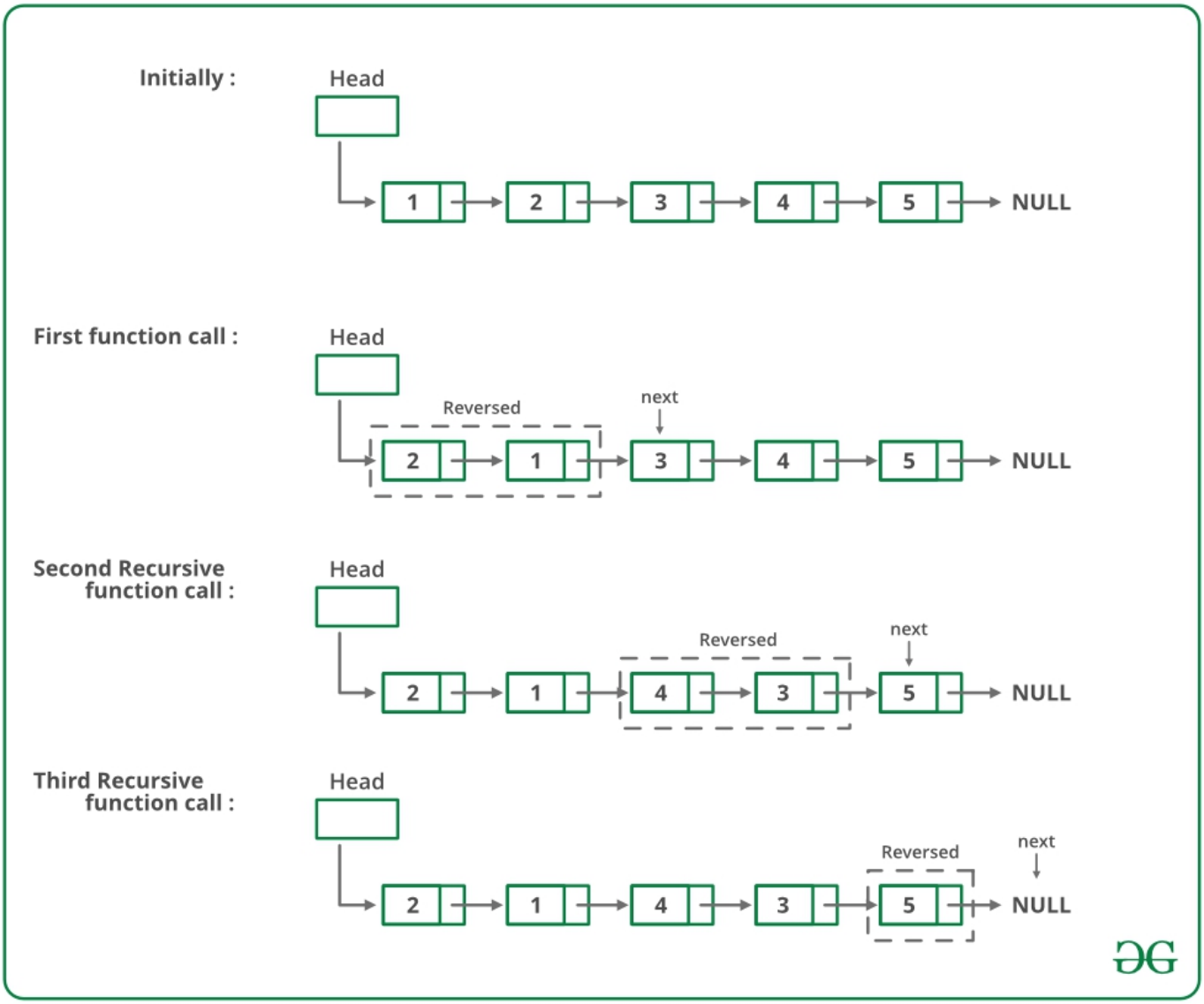
METHOD 1 (Iterative)
Start from the head node and traverse the list. While traversing swap data of each node with its next node's data.
Below is the implementation of the above approach:

Time complexity: O(n)

```
/* Function to pairwise swap elements
of a linked list */
void pairWiseSwap(Node* head)
{
    Node* temp = head;

    /* Traverse further only if
    there are at-least two nodes left */
    while (temp != NULL && temp->next != NULL) {
        /* Swap data of node with
        its next node's data */
        swap(temp->data,
             temp->next->data);

        /* Move temp by 2 for the next pair */
        temp = temp->next->next;
    }
}
```

METHOD 2 (Recursive)

If there are 2 or more than 2 nodes in Linked List then swap the first two nodes and recursively call for rest of the list.
Below image is a dry run of the above approach:




```c
/* Recursive function to pairwise swap elements
   of a linked list */
void pairWiseSwap(struct node* head)
{
    /* There must be at-least two nodes in the list */
    if (head != NULL && head->next != NULL) {

        /* Swap the node's data with data of next node */
        swap(head->data, head->next->data);

        /* Call pairWiseSwap() for rest of the list */
        pairWiseSwap(head->next->next);
    }
}
```

Time complexity: O(n)

Move last element to front of a given Linked List

Write a function that moves the last element to the front in a given Singly Linked List. For example, if the given Linked List is 1->2->3->4->5, then the function should change the list to 5->1->2->3->4.

Algorithm:
Traverse the list till last node. Use two pointers: one to store the address of last node and other for address of second last node. After the end of loop do following operations.
i) Make second last as last (secLast->next = NULL).
ii) Set next of last as head (last->next = *head_ref).
iii) Make last as head (*head_ref = last)

Time Complexity: O(n) where n is the number of nodes in the given Linked List.

```
void moveToFront(Node **head_ref)
{
    /* If linked list is empty, or
    it contains only one node,
    then nothing needs to be done,
    simply return */
    if (*head_ref == NULL || (*head_ref)->next == NULL)
        return;

    /* Initialize second last
    and last pointers */
    Node *secLast = NULL;
    Node *last = *head_ref;

    /*After this loop secLast contains
    address of second last node and
    last contains address of last node in Linked List */
    while (last->next != NULL)
    {
        secLast = last;
        last = last->next;
    }

    /* Set the next of second last as NULL */
    secLast->next = NULL;

    /* Set next of last as head node */
    last->next = *head_ref;

    /* Change the head pointer
    to point to last node now */
    *head_ref = last;
}
```

Intersection of two Sorted Linked Lists

Given two lists sorted in increasing order, create and return a new list representing the intersection of the two lists. The new list should be made with its own memory — the original lists should not be changed

Example:

```
Input:
First linked list: 1->2->3->4->6
Second linked list be 2->4->6->8,
Output: 2->4->6.
The elements 2, 4, 6 are common in
both the list so they appear in the
intersection list.

Input:
First linked list: 1->2->3->4->5
Second linked list be 2->3->4,
Output: 2->3->4
The elements 2, 3, 4 are common in
both the list so they appear in the
intersection list.
```

Method 1: Using Dummy Node.

Approach:

The idea is to use a temporary dummy node at the start of the result list. The pointer tail always points to the last node in the result list, so new nodes can be added easily.
The dummy node initially gives the tail a memory space to point to. 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 the tail.
When the given lists are traversed the result is in dummy. next, as the values are allocated from next node of the dummy.
If both the elements are equal then remove both and insert the element to the tail. Else remove the smaller element among both the lists.

Below is the implementation of the above approach:

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

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

void push(Node** head_ref, int new_data);

/*This solution uses the temporary
dummy to build up the result list */
Node* sortedIntersect(Node* a, Node* b)
{
    Node dummy;
    Node* tail = &dummy;
    dummy.next = NULL;

```

```
    /* Once one or the other
    list runs out -- we're done */
    while (a != NULL && b != NULL) {
        if (a->data == b->data) {
            push((&tail->next), a->data);
            tail = tail->next;
            a = a->next;
            b = b->next;
        }
        /* advance the smaller list */
        else if (a->data < b->data)
            a = a->next;
        else
            b = b->next;
    }
    return (dummy.next);
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at
the beginning of the linked list */
void push(Node** head_ref, int new_data)
{
    /* allocate node */
    Node* new_node = (Node*)malloc(
        sizeof(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 program to test above functions*/
int main()
{
    /* Start with the empty lists */
    Node* a = NULL;
    Node* b = NULL;
    Node* intersect = NULL;

```

```
    /* Let us create the first sorted
    linked list to test the functions
    Created linked list will be
    1->2->3->4->5->6 */
    push(&a, 6);
    push(&a, 5);
    push(&a, 4);
    push(&a, 3);
    push(&a, 2);
    push(&a, 1);

    /* Let us create the second sorted linked list
Created linked list will be 2->4->6->8 */
    push(&b, 8);
    push(&b, 6);
    push(&b, 4);
    push(&b, 2);

    /* Find the intersection two linked lists */
    intersect = sortedIntersect(a, b);

    cout<<"Linked list containing common items of a & b \n";
    printList(intersect);
}
```

Output

```
Linked list containing common items of a & b
 2 4 6
```

Complexity Analysis:

- **Time Complexity:** O(m+n) where m and n are number of nodes in first and second linked lists respectively.
  Only one traversal of the lists are needed.
- **Auxiliary Space:** O(min(m, n)).
  The output list can store at most min(m,n) nodes .

Method 2: Recursive Solution.
Approach:
The recursive approach is very similar to the the above two approaches. Build a recursive function that takes two nodes and returns a linked list node. Compare the first element of both the lists.

- If they are similar then call the recursive function with the next node of both the lists. Create a node with the data of the current node and put the returned node from the recursive function to the next pointer of the node created. Return the node created.
- If the values are not equal then remove the smaller node of both the lists and call the recursive function.
-

Below is the implementation of the above approach:

```
struct Node* sortedIntersect(struct Node* a,
                             struct Node* b)
{
    // base case
    if (a == NULL || b == NULL)
        return NULL;
    /* If both lists are non-empty */
    /* Advance the smaller list and
    call recursively */
    if (a->data < b->data)
        return sortedIntersect(a->next, b);
    if (a->data > b->data)
        return sortedIntersect(a, b->next);
    // Below lines are executed only
    // when a->data == b->data
    struct Node* temp = (struct Node*)malloc(
                sizeof(struct Node));
    temp->data = a->data;
// Advance both lists and call recursively
    temp->next = sortedIntersect(a->next,
                                 b->next);
    return temp;
}
```

Output:

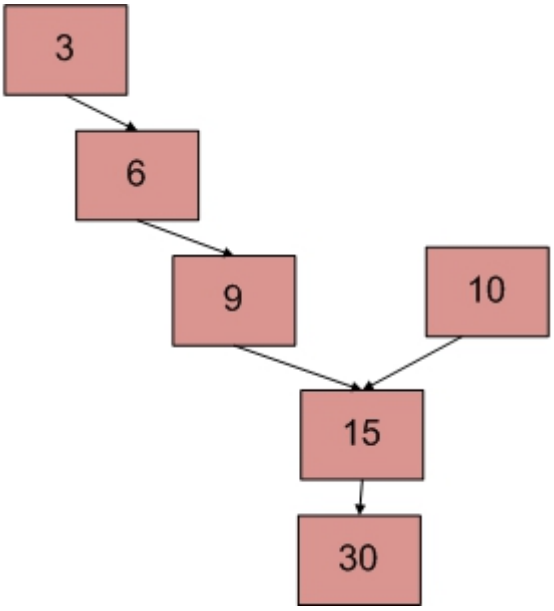
```
Linked list containing common items of a & b
 2 4 6
```

Complexity Analysis:

- **Time Complexity:** O(m+n) where m and n are number of nodes in first and second linked lists respectively.
  Only one traversal of the lists are needed.
- **Auxiliary Space:** O(max(m, n)).
  The output list can store at most m+n nodes.

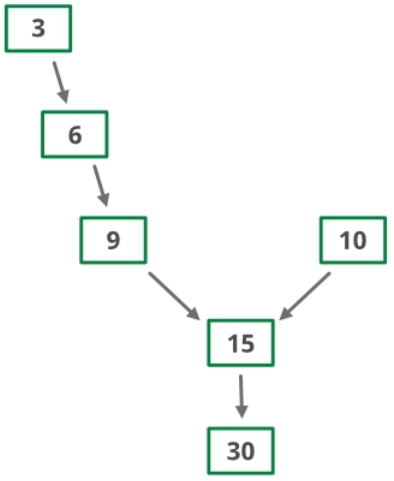
Write a function to get the intersection point of two Linked Lists

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.



Using difference of node counts

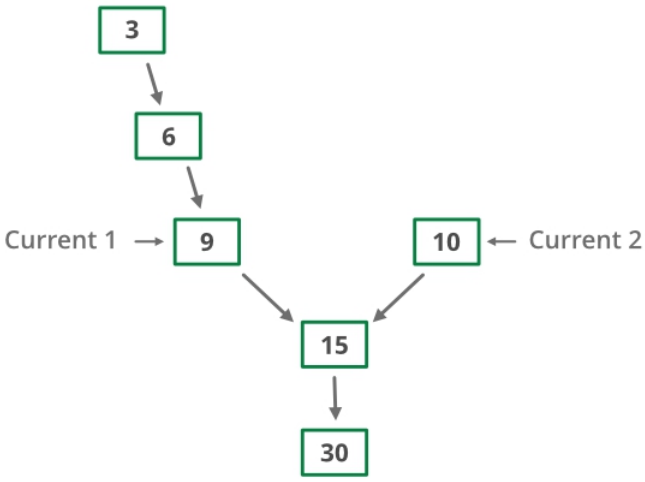
- Get count of the nodes in the first list, let count be c1.
- Get count of the nodes in the second list, let count be c2.
- Get the difference of counts **d = abs(c1 – c2)**
- Now traverse the bigger list from the first node till d nodes so that from here onwards both the lists have equal no of nodes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

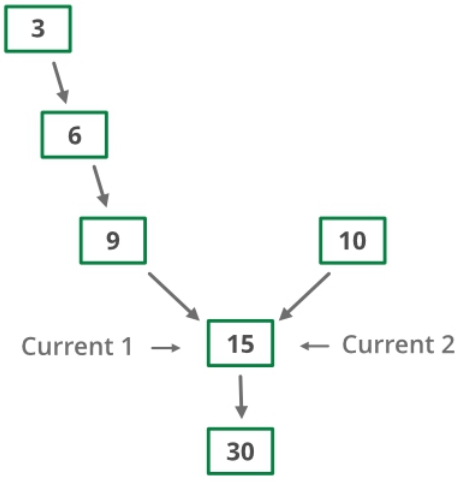


C1 = 5

C2 = 3

Traverse till 3rd node
in the bigger list





Intersection node = 15

```cpp
// 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;
    }
```

```
58
59      // Move both pointers of both list till they
60      // intersect with each other
61      while (current1 != NULL && current2 != NULL) {
62          if (current1 == current2)
63              return current1->data;
64
65          // Move both the pointers forward
66          current1 = current1->next;
67          current2 = current2->next;
68      }
69
70      return -1;
71  }
72
73  /* Takes head pointer of the linked list and
74  returns the count of nodes in the list */
75  int getCount(Node* head)
76  {
77      Node* current = head;
78
79      // Counter to store count of nodes
80      int count = 0;
81
82      // Iterate till NULL
83      while (current != NULL) {
84
85          // Increase the counter
86          count++;
87
88          // Move the Node ahead
89          current = current->next;
90      }
91
92      return count;
93  }
94
95  // Driver Code
96  int main()
97  {
98      /*
99          Create two linked lists
100
101         1st 3->6->9->15->30
102         2nd 10->15->30
103
104         15 is the intersection point
105     */
106
107     Node* newNode;
108
109     // Addition of new nodes
110     Node* head1 = new Node();
111     head1->data = 10;
112
113     Node* head2 = new Node();
114     head2->data = 3;
```

```
115
116     newNode = new Node();
117     newNode->data = 6;
118     head2->next = newNode;
119
120     newNode = new Node();
121     newNode->data = 9;
122     head2->next->next = newNode;
123
124     newNode = new Node();
125     newNode->data = 15;
126     head1->next = newNode;
127     head2->next->next->next = newNode;
128
129     newNode = new Node();
130     newNode->data = 30;
131     head1->next->next = newNode;
132
133     head1->next->next->next = NULL;
134
135     cout << "The node of intersection is " << getIntesectionNode(head1, head2);
136 }
```

Output

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

Time Complexity: O(m+n)

Auxiliary Space: O(1)

2-pointer technique

- Initialize two pointers ptr1 and ptr2 at the head1 and head2.
- Traverse through the lists,one node at a time.
- When ptr1 reaches the end of a list, then redirect it to the head2.
- similarly when ptr2 reaches the end of a list, redirect it the head1.
- Once both of them go through reassigning,hey will be equidistant from the collision point
- If at any node ptr1 meets ptr2, then it is the intersection node.
- After second iteration if there is no intersection node it returns NULL.

```cpp
// CPP program to print intersection of lists
#include <bits/stdc++.h>
using namespace std;

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

// A utility function to return intersection node
Node* intersectPoint(Node* head1, Node* head2)
{
    // Maintaining two pointers ptr1 and ptr2
    // at the head of A and B,
    Node* ptr1 = head1;
    Node* ptr2 = head2;

    // If any one of head is NULL i.e
    // no Intersection Point
    if (ptr1 == NULL || ptr2 == NULL) {

        return NULL;
    }

    // Traverse through the lists until they
    // reach Intersection node
    while (ptr1 != ptr2) {

        ptr1 = ptr1->next;
        ptr2 = ptr2->next;

        // If at any node ptr1 meets ptr2, then it is
        // intersection node.Return intersection node.

        if (ptr1 == ptr2) {

            return ptr1;
        }
        /* Once both of them go through reassigning,
        they will be equidistant from the collision point.*/

        // When ptr1 reaches the end of a list, then
        // reassign it to the head2.
        if (ptr1 == NULL) {

            ptr1 = head2;
        }
        // When ptr2 reaches the end of a list, then
        // redirect it to the head1.
        if (ptr2 == NULL) {

            ptr2 = head1;
        }
    }

    return ptr1;
}
```

```
60
61  // Function to print intersection nodes
62  // in a given linked list
63  void print(Node* node)
64  {
65      if (node == NULL)
66          cout << "NULL";
67      while (node->next != NULL) {
68          cout << node->data << "->";
69          node = node->next;
70      }
71      cout << node->data;
72  }
73  // Driver code
74  int main()
75  {
76      /*
77      Create two linked lists
78
79      1st Linked list is 3->6->9->15->30
80      2nd Linked list is 10->15->30
81
82      15 30 are elements in the intersection list
83      */
84
85      Node* newNode;
86      Node* head1 = new Node();
87      head1->data = 10;
88      Node* head2 = new Node();
89      head2->data = 3;
90      newNode = new Node();
91      newNode->data = 6;
92      head2->next = newNode;
93      newNode = new Node();
94      newNode->data = 9;
95      head2->next->next = newNode;
96      newNode = new Node();
97      newNode->data = 15;
98      head1->next = newNode;
99      head2->next->next->next = newNode;
100     newNode = new Node();
101     newNode->data = 30;
102     head1->next->next = newNode;
103     head1->next->next->next = NULL;
104     Node* intersect_node = NULL;
105
106     // Find the intersection node of two linked lists
107     intersect_node = intersectPoint(head1, head2);
108
109      cout << "INTERSEPOINT LIST :";
110
111      print(intersect_node);
112
113      return 0;
114  // This code is contributed by bolliranadheer
115  }
```

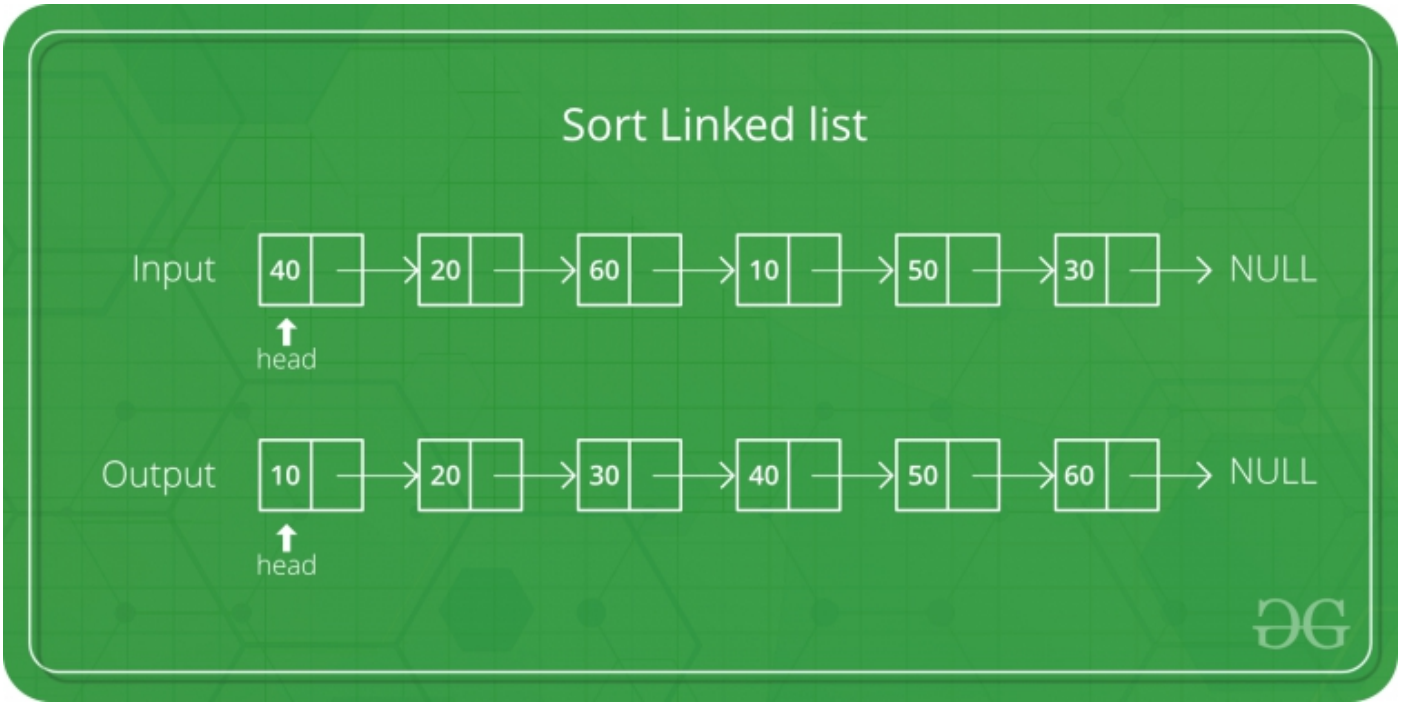
Output

```
INTERSEPOINT LIST :15->30
```

Time complexity : O(m + n)
Auxiliary Space: O(1)

QuickSort on Singly Linked List

The important things about implementation are, it changes pointers rather swapping data and time complexity is same as the implementation for Doubly Linked List.



In **partition()**, we consider last element as pivot. We traverse through the current list and if a node has value greater than pivot, we move it after tail. If the node has smaller value, we keep it at its current position.

In **QuickSortRecur()**, we first call partition() which places pivot at correct position and returns pivot. After pivot is placed at correct position we find tail node of left side (list before pivot) and recur for left list.

Finally, we recur for right list.

```cpp
// C++ program for Quick Sort on Singly Linled List
#include <cstdio>
#include <iostream>
using namespace std;

/* a node of the singly linked list */
struct Node {
    int data;
    struct Node* next;
};

/* A utility function to insert a node at the beginning of
* linked list */
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;
}

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

// Returns the last node of the list
struct Node* getTail(struct Node* cur)
{
    while (cur != NULL && cur->next != NULL)
        cur = cur->next;
    return cur;
}

```

```
// Partitions the list taking the last element as the pivot
struct Node* partition(struct Node* head, struct Node* end,
                       struct Node** newHead,
                       struct Node** newEnd)
{
    struct Node* pivot = end;
    struct Node *prev = NULL, *cur = head, *tail = pivot;

    // During partition, both the head and end of the list
    // might change which is updated in the newHead and
    // newEnd variables
    while (cur != pivot) {
        if (cur->data < pivot->data) {
            // First node that has a value less than the
            // pivot - becomes the new head
            if ((*newHead) == NULL)
                (*newHead) = cur;

            prev = cur;
            cur = cur->next;
        }
        else // If cur node is greater than pivot
        {
            // Move cur node to next of tail, and change
            // tail
            if (prev)
                prev->next = cur->next;
            struct Node* tmp = cur->next;
            cur->next = NULL;
            tail->next = cur;
            tail = cur;
            cur = tmp;
        }
    }

    // If the pivot data is the smallest element in the
    // current list, pivot becomes the head
    if ((*newHead) == NULL)
        (*newHead) = pivot;

    // Update newEnd to the current last node
    (*newEnd) = tail;

    // Return the pivot node
    return pivot;
}
```

```
93
94  // here the sorting happens exclusive of the end node
95  struct Node* quickSortRecur(struct Node* head,
96                              struct Node* end)
97  {
98      // base condition
99      if (!head || head == end)
100         return head;
101
102     Node *newHead = NULL, *newEnd = NULL;
103
104     // Partition the list, newHead and newEnd will be
105     // updated by the partition function
106     struct Node* pivot
107         = partition(head, end, &newHead, &newEnd);
108
109     // If pivot is the smallest element - no need to recur
110     // for the left part.
111     if (newHead != pivot) {
112         // Set the node before the pivot node as NULL
113         struct Node* tmp = newHead;
114         while (tmp->next != pivot)
115             tmp = tmp->next;
116         tmp->next = NULL;
117
118         // Recur for the list before pivot
119         newHead = quickSortRecur(newHead, tmp);
120
121         // Change next of last node of the left half to
122         // pivot
123         tmp = getTail(newHead);
124         tmp->next = pivot;
125     }
126
127     // Recur for the list after the pivot element
128     pivot->next = quickSortRecur(pivot->next, newEnd);
129
130     return newHead;
131 }
132
```

```
133 // The main function for quick sort. This is a wrapper over
134 // recursive function quickSortRecur()
135 void quickSort(struct Node** headRef)
136 {
137     (*headRef)
138         = quickSortRecur(*headRef, getTail(*headRef));
139     return;
140 }
141
142 // Driver code
143 int main()
144 {
145     struct Node* a = NULL;
146     push(&a, 5);
147     push(&a, 20);
148     push(&a, 4);
149     push(&a, 3);
150     push(&a, 30);
151
152     cout << "Linked List before sorting \n";
153     printList(a);
154
155     quickSort(&a);
156
157     cout << "Linked List after sorting \n";
158     printList(a);
159
160     return 0;
161 }
```

Output

```
Linked List before sorting
30 3 4 20 5
Linked List after sorting
3 4 5 20 30
```

To be continued….

More examples will be added on later.

You will get updated one!!!