# Linked List

## 126. Write a Program to reverse the Linked List. (Both Iterative and recursive)

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 the 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*

### Solution:

**Iterative Method**

1. *Initialize three pointers prev as NULL, curr as head and next as NULL.*
2. *Iterate through 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 code\*/

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

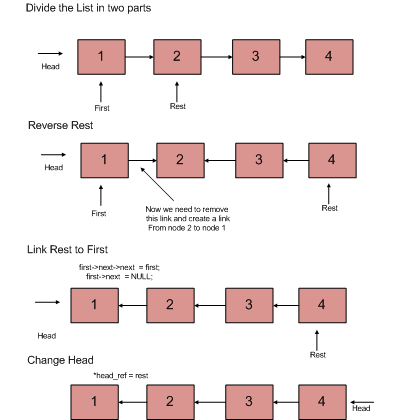
rest of the linked list.

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

3) Link rest to first.

4) Fix head pointer

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;

}

**Output:**

Given linked list

85 15 4 20

Reversed Linked list

20 4 15 85

**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 code

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

**Using Stack:**

* Store the nodes(values and address) in the stack until all the values are entered.
* Once all entries are done, Update the Head pointer to the last location(i.e the last value).
* Start popping the nodes(value and address) and store them in the same order until the stack is empty.
* Update the next pointer of last Node in the stack by NULL.

Below is the implementation of the above approach:

// C++ program for above approach

#include <bits/stdc++.h>

#include <iostream>

using namespace std;

// Create a class Node to enter

// values and address in the list

class Node

{

public:

int data;

Node\* next;

};

// Function to reverse the

// linked list

void reverseLL(Node\*\* head)

{

// Create a stack "s"

// of Node type

stack<Node\*> s;

Node\* temp = \*head;

while (temp->next != NULL)

{

// Push all the nodes

// in to stack

s.push(temp);

temp = temp->next;

}

\*head = temp;

while (!s.empty())

{

// Store the top value of

// stack in list

temp->next = s.top();

// Pop the value from stack

s.pop();

// update the next pointer in the

// in the list

temp = temp->next;

}

temp->next = NULL;

}

// Function to Display

// the elements in List

void printlist(Node\* temp)

{

while (temp != NULL)

{

cout << temp->data << " ";

temp = temp->next;

}

}

// Program to insert back of the

// linked list

void insert\_back(Node\*\* head, int value)

{

// we have used insertion at back method

// to enter values in the list.(eg:

// head->1->2->3->4->Null)

Node\* temp = new Node();

temp->data = value;

temp->next = NULL;

// If \*head equals to NULL

if (\*head == NULL)

{

\*head = temp;

return;

}

else

{

Node\* last\_node = \*head;

while (last\_node->next != NULL)

{

last\_node = last\_node->next;

}

last\_node->next = temp;

return;

}

}

// Driver Code

int main()

{

Node\* head = NULL;

insert\_back(&head, 1);

insert\_back(&head, 2);

insert\_back(&head, 3);

insert\_back(&head, 4);

cout << "Given linked list\n";

printlist(head);

reverseLL(&head);

cout << "\nReversed linked list\n";

printlist(head);

return 0;

}

**Output**

Given linked list

1 2 3 4

Reversed linked list

4 3 2 1

**Using array:**

**1.**Create a linked list.

**2.**Then, make a count(head) function to count the number of nodes.

**3.**Initialize an array with the size of the count.

**4.**and start a while(p->next!=NULL) loop and store all the node’s data into the array.

**5.**and then print the array from the last index to the first.

#include <iostream>

#include<cstdlib>

using namespace std;

typedef struct node

{

int val;

struct node\* next;

}node;

node\* head=NULL;

int count(node\* head) // code to count the no. of nodes

{

node\* p=head;

int k=1;

while(p!=NULL)

{

p=p->next;

k++;

}

return k;

}

node \*ll\_reverse(node\* head) // to reverse the linked list

{

node\* p=head;

long int i=count(head),j=1;

long int arr[i];

while(i && p!=NULL)

{

arr[j++]=p->val;

p=p->next;

i--;

}

j--;

while(j) // loop will break as soon as j=0

{

cout<<arr[j--]<<" ";

}

return head;

}

node\* insert\_end(node\* head,int data) //code to insert at end of ll

{

node\* q=head,\*p=(node\*)malloc(sizeof(node));

p->val=data;

while(q->next!=NULL)

{

q=q->next;

}

q->next=p;

p->next=NULL;

return head;

}

node \*create\_ll(node\* head,int data) //create ll

{

node\* p=(node\*)malloc(sizeof(node));

p->val=data;

if(head==NULL)

{

head=p;

p->next=NULL;

return head;

}

else

{

head=insert\_end(head,data);

return head;

}

}

//Driver code

int main()

{

int i=5,j=1;

while(i--)

{

head=create\_ll(head,j++);

}

head=ll\_reverse(head);

return 0;

}

**Input : 1->2->3->4->5**

**Output: 5->4->3->2->1**

**Time complexity: O(n)**

**Space complexity: O(n)**

## 127. Reverse a Linked List in group of Given Size. [Very Imp]

Given a linked list of size **N**. The task is to reverse every **k** nodes (where k is an input to the function) in the linked list. If the number of nodes is not a multiple of *k* then left-out nodes, in the end, should be considered as a group and must be reversed (See Example 2 for clarification).

**Example 1:**

**Input:**

LinkedList: 1->2->2->4->5->6->7->8

K = 4

**Output:** 4 2 2 1 8 7 6 5

**Explanation:**

The first 4 elements 1,2,2,4 are reversed first

and then the next 4 elements 5,6,7,8. Hence, the

resultant linked list is 4->2->2->1->8->7->6->5.

**Example 2:**

**Input:**

LinkedList: 1->2->3->4->5

K = 3

**Output:** 3 2 1 5 4

**Explanation:**

The first 3 elements are 1,2,3 are reversed

first and then elements 4,5 are reversed.Hence,

the resultant linked list is 3->2->1->5->4.

**Your Task:**  
You don't need to read input or print anything. Your task is to complete the function **reverse**() which should reverse the linked list in group of size **k**and return the head of the modified linked list.

**Expected Time Complexity**: O(N)  
**Expected Auxilliary Space**: O(1)

**Constraints:**

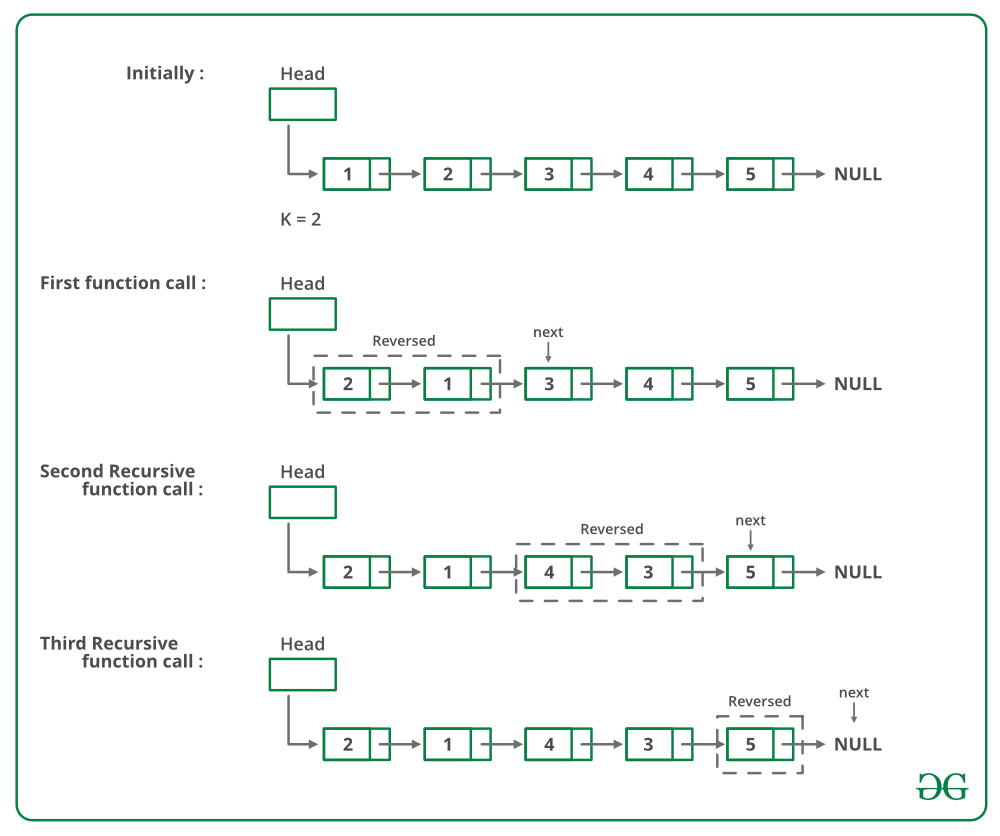
1 <= N <= 104  
1 <= k <= N

### Solution:

**Algorithm**: [*reverse*](https://www.geeksforgeeks.org/reverse-a-linked-list/)*(head, k)*

* Reverse the first sub-list of size k. While reversing keep track of the next node and previous node. Let the pointer to the next node be *next*and pointer to the previous node be *prev*. See [this post](https://www.geeksforgeeks.org/reverse-a-linked-list/) for reversing a linked list.
* *head->next = reverse(next, k)* ( Recursively call for rest of the list and link the two sub-lists )
* Return *prev*( *prev*becomes the new head of the list (see the diagrams of an iterative method of [this post](https://www.geeksforgeeks.org/reverse-a-linked-list/))

Below is image shows how the reverse function works:



Below is the implementation of the above approach:

// CPP program to reverse a linked list

// in groups of given size

#include <bits/stdc++.h>

using namespace std;

/\* Link list node \*/

class Node {

public:

int data;

Node\* next;

};

/\* Reverses the linked list in groups

of size k and returns the pointer

to the new head node. \*/

Node\* reverse(Node\* head, int k)

{

// base case

if (!head)

return NULL;

Node\* current = head;

Node\* next = NULL;

Node\* prev = NULL;

int count = 0;

/\*reverse first k nodes of the linked list \*/

while (current != NULL && count < k) {

next = current->next;

current->next = prev;

prev = current;

current = next;

count++;

}

/\* next is now a pointer to (k+1)th node

Recursively call for the list starting from current.

And make rest of the list as next of first node \*/

if (next != NULL)

head->next = reverse(next, k);

/\* prev is new head of the input list \*/

return prev;

}

/\* UTILITY FUNCTIONS \*/

/\* 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;

}

}

/\* Driver code\*/

int main()

{

/\* Start with the empty list \*/

Node\* head = NULL;

/\* Created Linked list

is 1->2->3->4->5->6->7->8->9 \*/

push(&head, 9);

push(&head, 8);

push(&head, 7);

push(&head, 6);

push(&head, 5);

push(&head, 4);

push(&head, 3);

push(&head, 2);

push(&head, 1);

cout << "Given linked list \n";

printList(head);

head = reverse(head, 3);

cout << "\nReversed Linked list \n";

printList(head);

return (0);

}

**Output:**

Given Linked List

1 2 3 4 5 6 7 8 9

Reversed list

3 2 1 6 5 4 9 8 7

**Complexity Analysis:**

* **Time Complexity:** O(n).   
  Traversal of list is done only once and it has ‘n’ elements.
* **Auxiliary Space:** O(n/k).   
  For each Linked List of size n, n/k or (n/k)+1 calls will be made during the recursion.

**Approach 2**

 we have used a stack which will store the nodes of the given linked list. Firstly, push the k elements of the linked list in the stack. Now pop elements one by one and keep track of the previously popped node. Point the next pointer of prev node to top element of stack. Repeat this process, until NULL is reached.  
This algorithm uses O(k) extra space.

// C++ program to reverse a linked list in groups

// of given size

#include <bits/stdc++.h>

using namespace std;

/\* Link list node \*/

struct Node {

int data;

struct Node\* next;

};

/\* Reverses the linked list in groups of size k

and returns the pointer to the new head node. \*/

struct Node\* Reverse(struct Node\* head, int k)

{

// Create a stack of Node\*

stack<Node\*> mystack;

struct Node\* current = head;

struct Node\* prev = NULL;

while (current != NULL) {

// Terminate the loop whichever comes first

// either current == NULL or count >= k

int count = 0;

while (current != NULL && count < k) {

mystack.push(current);

current = current->next;

count++;

}

// Now pop the elements of stack one by one

while (mystack.size() > 0) {

// If final list has not been started yet.

if (prev == NULL) {

prev = mystack.top();

head = prev;

mystack.pop();

} else {

prev->next = mystack.top();

prev = prev->next;

mystack.pop();

}

}

}

// Next of last element will point to NULL.

prev->next = NULL;

return head;

}

/\* UTILITY FUNCTIONS \*/

/\* Function to push a node \*/

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;

}

/\* Function to print 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(void)

{

/\* Start with the empty list \*/

struct Node\* head = NULL;

/\* Created Linked list is 1->2->3->4->5->6->7->8->9 \*/

push(&head, 9);

push(&head, 8);

push(&head, 7);

push(&head, 6);

push(&head, 5);

push(&head, 4);

push(&head, 3);

push(&head, 2);

push(&head, 1);

printf("\nGiven linked list \n");

printList(head);

head = Reverse(head, 3);

printf("\nReversed Linked list \n");

printList(head);

return 0;

}

**Output:**

Given Linked List

1 2 3 4 5 6 7 8 9

Reversed list

3 2 1 6 5 4 9 8 7

## 128. Write a program to Detect loop in a linked list.

Given a linked list of **N** nodes. The task is to check if the linked list has a loop. Linked list can contain self loop.

**Example 1:**

**Input:**

N = 3

value[] = {1,3,4}

x = 2

**Output:** True

**Explanation:** In above test case N = 3.

The linked list with nodes N = 3 is

given. Then value of x=2 is given which

means last node is connected with xth

node of linked list. Therefore, there

exists a loop.

**Example 2:**

**Input:**

N = 4

value[] = {1,8,3,4}

x = 0

**Output:** False

**Explanation:** For N = 4 ,x = 0 means

then lastNode->next = NULL, then

the Linked list does not contains

any loop.

**Your Task:**  
The task is to complete the function **detectloop**() which contains reference to the head as only argument. This function should return **true** if linked list contains loop, else return **false**.

**Expected Time Complexity:** O(N)  
**Expected Auxiliary Space:** O(1)

**Constraints:**  
1 ≤ N ≤ 104  
1 ≤ Data on Node ≤ 103

### Solution:

**Solution 1: Hashing Approach:**

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 the next of the current nodes 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;

}

**Output**

Loop found

**Complexity Analysis:**

* **Time complexity:** O(n).   
  Only one traversal of the loop is needed.
* **Auxiliary Space:** O(n).   
  n is the space required to store the value in hashmap.

**Solution 2:**This problem can be solved without hashmap by modifying the linked list data structure.   
**Approach:** 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.

// 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;

int flag;

};

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;

new\_node->flag = 0;

/\* 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)

{

while (h != NULL) {

// If this node is already traverse

// it means there is a cycle

// (Because you we encountering the

// node for the second time).

if (h->flag == 1)

return true;

// If we are seeing the node for

// the first time, mark its flag as 1

h->flag = 1;

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;

}

**Output**

Loop found

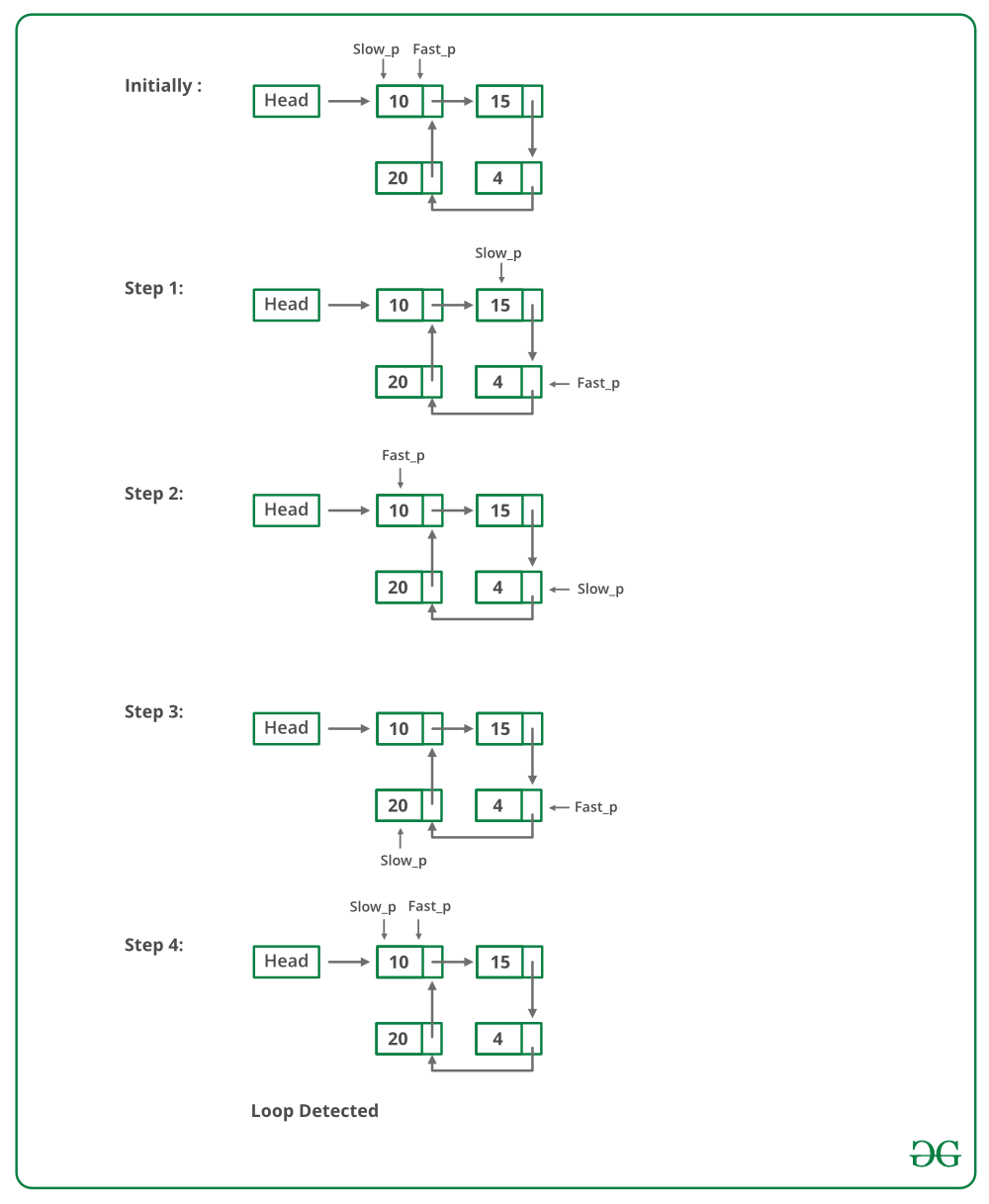
**Complexity Analysis:**

* **Time complexity:**O(n).   
  Only one traversal of the loop is needed.
* **Auxiliary Space:**O(1).   
  No extra space is needed.

**Solution 3: 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 detect loop 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) {

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

**Complexity Analysis:**

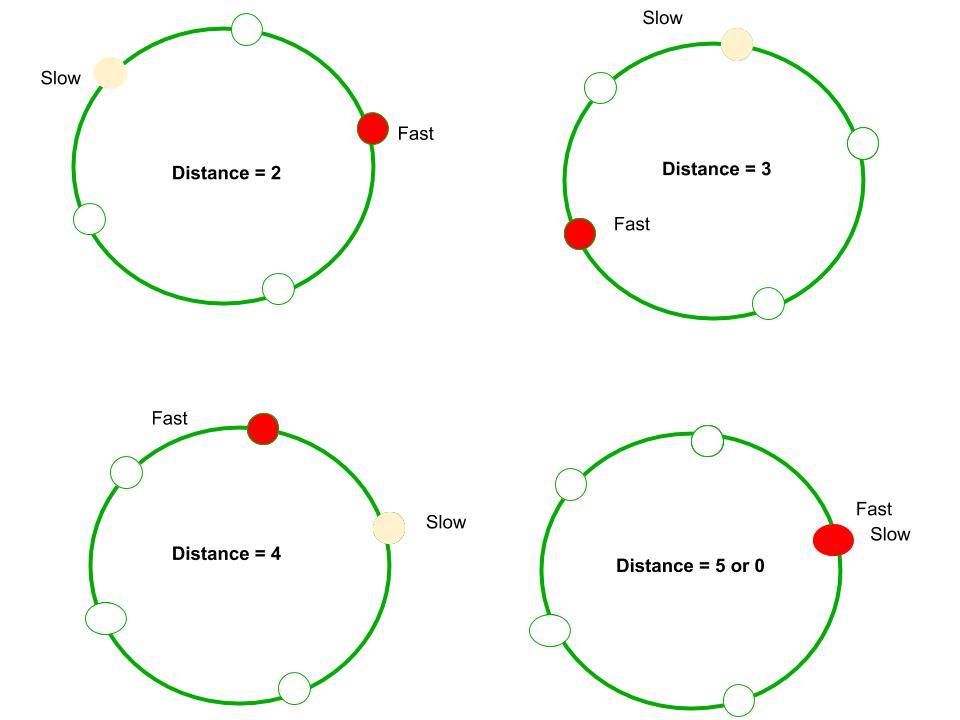
* **Time complexity:** O(n).   
  Only one traversal of the loop is needed.
* **Auxiliary Space:**O(1).   
  There is no space required.

**How does Floyd’s slow and fast pointers approach work?**

The algorithm is to start two pointers, slow and fast from head of linked list. We move slow one node at a time and fast two nodes at a time. If there is a loop, then they will definitely meet. This approach works because of the following facts.

1) When slow pointer enters the loop, the fast pointer must be inside the loop. Let fast pointer be distance k from slow.

2) Now if consider movements of slow and fast pointers, we can notice that distance between them (from slow to fast) increase by one after every iteration. After one iteration (of slow = next of slow and fast = next of next of fast), distance between slow and fast becomes k+1, after two iterations, k+2, and so on. When distance becomes n, they meet because they are moving in a cycle of length n.

For example, we can see in below diagram, initial distance is 2. After one iteration, distance becomes 3, after 2 iterations, it becomes 4. After 3 iterations, it becomes 5 which is distance 0. And they meet.  


**Solution 4: 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 the loop doesn’t exist.

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

**Complexity Analysis:**

* **Time complexity:** O(n).   
  Only one traversal of the loop is needed.
* **Auxiliary Space:** O(1).   
  There is no space required.

**Solution 5: Store length**

In this method, two pointers are created, first (always points to head) and last. Each time the last pointer moves we calculate no of nodes in between first and last and check whether the current no of nodes > previous no of nodes, if yes we proceed by moving last pointer else it means we've reached the end of the loop, so we return output accordingly.

// 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;

}

/\*returns distance between first and last node every time

\* last node moves forwards\*/

int distance(Node\* first, Node\* last)

{

/\*counts no of nodes between first and last\*/

int counter = 0;

Node\* curr;

curr = first;

while (curr != last) {

counter += 1;

curr = curr->next;

}

return counter + 1;

}

// 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;

Node \*first, \*last;

/\*first always points to head\*/

first = head;

/\*last pointer initially points to head\*/

last = head;

/\*current\_length stores no of nodes between current

\* position of first and last\*/

int current\_length = 0;

/\*current\_length stores no of nodes between previous

\* position of first and last\*/

int prev\_length = -1;

while (current\_length > prev\_length && last != NULL) {

// set prev\_length to current length then update the

// current length

prev\_length = current\_length;

// distance is calculated

current\_length = distance(first, last);

// last node points the next node

last = last->next;

}

if (last == NULL) {

return false;

}

else {

return true;

}

}

/\* 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 Found";

return 0;

}

**Output**

Loop Found

 **Complexity Analysis:**

* **Time complexity:** O(n2)
* **Auxiliary Space:** O(1)

**Another Approach:**

1. This is the simplest approach of the given problem, the only thing we have to do is to assign a new value to each data of node in the linked list which is not in the range given.
2. Examplesuppose (1 <= Data on Node <= 10^3) then after visiting node assign the data as -1 as it is out of the given range.

Follow the code given below for a better understanding:

// 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;

}

// Function to detect first node of loop

// in a linked list that may contain loop

bool detectLoop(Node\* head)

{

// If the head is null we will return false

if (!head)

return 0;

else {

// Traversing the linked list

// for detecting loop

while (head) {

// If loop found

if (head->key == -1) {

return true;

}

// Changing the data of visited node to any

// value which is outside th given range here it

// is supposed the given range is (1 <= Data on

// Node <= 10^3)

else {

head->key = -1;

head = head->next;

}

}

// If loop not found return false

return 0;

}

}

/\* 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);

cout << found << endl;

return 0;

}

**Output**

1

**Time Complexity:** O(N)

**Auxiliary Space:**O(1)

## 129. Write a program to Delete loop in a linked list.

Write a function *detectAndRemoveLoop()* that checks whether a given Linked List contains loop and if loop is present then removes the loop and returns true. If the list doesn’t contain loop then it returns false. Below diagram shows a linked list with a loop. *detectAndRemoveLoop()* must change the below list to 1->2->3->4->5->NULL.



### Solution:

**Method 1 (Check one by one)**We know that Floyd’s Cycle detection algorithm terminates when fast and slow pointers meet at a common point. We also know that this common point is one of the loop nodes (2 or 3 or 4 or 5 in the above diagram). Store the address of this in a pointer variable say ptr2. After that start from the head of the Linked List and check for nodes one by one if they are reachable from ptr2. Whenever we find a node that is reachable, we know that this node is the starting node of the loop in Linked List and we can get the pointer to the previous of this node.

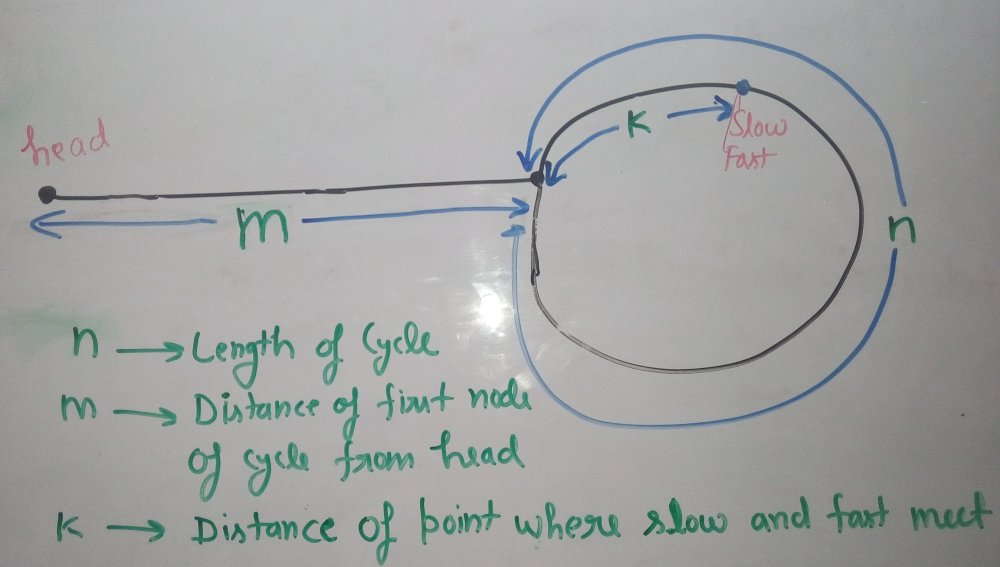
**Output:**

Linked List after removing loop

50 20 15 4 10

**Method 2**  
We do not need to count number of nodes in Loop. After detecting the loop, if we start slow pointer from head and move both slow and fast pointers at same speed until fast don’t meet, they would meet at the beginning of the loop.

**How does this 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**.

Thus we can write, m + k = i\*n or **m = i\*n - k**.

Hence, distance moved by slow pointer: m, is equal to distance moved by fast pointer:

i\*n - k or (i-1)\*n + n - k (cover the loop completely i-1 times and start from n-k).

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.

// C++ program to detect and remove 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

void detectAndRemoveLoop(Node\* head)

{

// If list is empty or has only one node

// without loop

if (head == NULL || head->next == NULL)

return;

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 exists \*/

if (slow == fast)

{

slow = head;

// this check is needed when slow

// and fast both meet at the head of the LL

// eg: 1->2->3->4->5 and then

// 5->next = 1 i.e the head of the LL

if(slow == fast) {

while(fast->next != slow) fast = fast->next;

}

else {

while (slow->next != fast->next) {

slow = slow->next;

fast = fast->next;

}

}

/\* since fast->next is the looping point \*/

fast->next = NULL; /\* remove loop \*/

}

}

/\* Driver program to test above function\*/

int main()

{

Node\* head = newNode(50);

head->next = head;

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;

detectAndRemoveLoop(head);

printf("Linked List after removing loop \n");

printList(head);

return 0;

}

**Output:**

Linked List after removing loop

50 20 15 4 10

**Method 4 Hashing: Hash the address of the linked list nodes**   
We can hash the addresses of the linked list nodes in an unordered map and just check if the element already exists in the map. If it exists, we have reached a node which already exists by a cycle, hence we need to make the last node’s next pointer NULL.

// C++ program to detect and remove 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

void hashAndRemove(Node\* head)

{

// hash map to hash addresses of the linked list nodes

unordered\_map<Node\*, int> node\_map;

// pointer to last node

Node\* last = NULL;

while (head != NULL) {

// if node not present in the map, insert it in the map

if (node\_map.find(head) == node\_map.end()) {

node\_map[head]++;

last = head;

head = head->next;

}

// if present, it is a cycle, make the last node's next pointer NULL

else {

last->next = NULL;

break;

}

}

}

/\* Driver program to test above function\*/

int main()

{

Node\* head = newNode(50);

head->next = head;

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;

// printList(head);

hashAndRemove(head);

printf("Linked List after removing loop \n");

printList(head);

return 0;

}

**Output**

Linked List after removing loop

50 20 15 4 10

## 130. Find the starting point of the loop.

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

**Example :**

Input : Head of below linked list



Output : Pointer to node 2

### Solution:

We have discussed [Floyd’s loop detection algorithm](https://www.geeksforgeeks.org/detect-loop-in-a-linked-list/). Below are steps to find the first node of the loop.  
1. If a loop is found, initialize a 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;

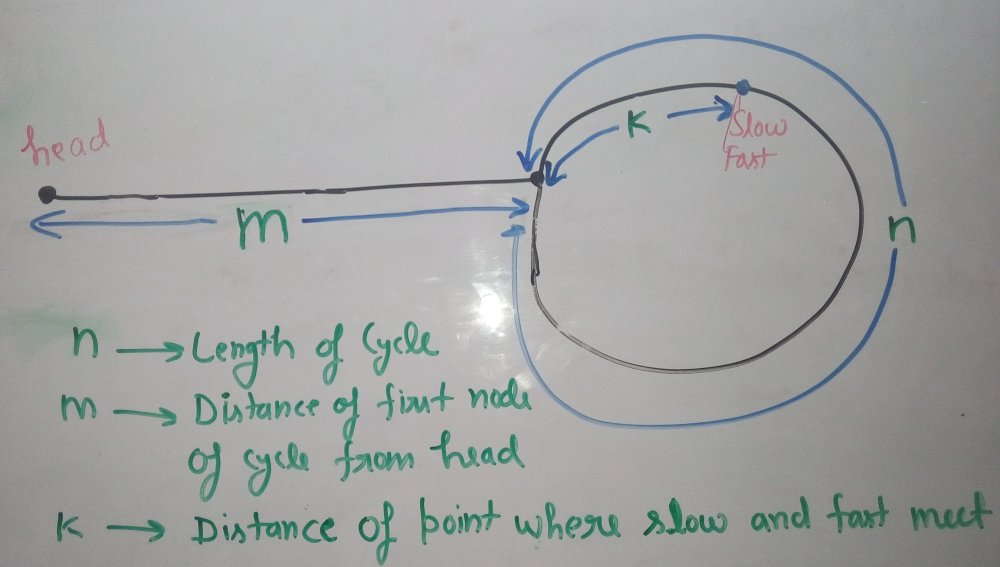
return 0;

}

**Output:**

Loop starting node is 15

**How does this approach work?**   
Let slow and fast meet at some point after Floyd’s Cycle finding algorithm. The below diagram shows the situation when the cycle is found.



We can conclude below from the 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 the 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 the **same speed** such that one pointer (say slow) begins from the head node of the linked list and other pointers (say fast) begins from the meeting point. When the slow pointer reaches the beginning of the loop (has made m steps), the fast pointer would have made also moved m steps as they are now moving the 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 the 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 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 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

**Method 3:**   
We can also use the concept of **hashing** in order to detect the first node of the loop. The idea is simple just iterate over the entire linked list and store node addresses in a set(**C++ STL**) one by one, while adding the node address into the set check if it already contains that particular node address if not then add node address to set if it is already present in the set then the current node is the first node of the loop.

// The below function take head of Linked List as

// input and return Address of first node in

// the loop if present else return NULL.

/\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode(int x) : val(x), next(NULL) {}

\* };\*/

ListNode\* detectCycle(ListNode\* A)

{

// declaring map to store node address

unordered\_set<ListNode\*> uset;

ListNode \*ptr = A;

// Default consider that no cycle is present

while (ptr != NULL) {

// checking if address is already present in map

if (uset.find(ptr) != uset.end())

return ptr;

// if address not present then insert into the set

else

uset.insert(ptr);

ptr = ptr->next;

}

return NULL;

}

## 131. Remove Duplicates in a sorted Linked List.

Given a singly linked list consisting of **N** nodes. The task is to remove duplicates (nodes with duplicate values) from the given list (if exists).  
**Note:** Try not to use extra space. Expected time complexity is **O(N)**. The nodes are arranged in a **sorted**way.

**Example 1:**

**Input:**

LinkedList: 2->2->4->5

**Output:** 2 4 5

**Explanation:** In the given linked list

2 ->2 -> 4-> 5, only 2 occurs more

than 1 time.

**Example 2:**

**Input:**

LinkedList: 2->2->2->2->2

**Output:** 2

**Explanation:** In the given linked list

2 ->2 ->2 ->2 ->2, 2 is the only element

and is repeated 5 times.

**Your Task:**  
The task is to complete the function **removeDuplicates**() which should remove the duplicates from linked list and return the head of the linkedlist.

**Expected Time Complexity** : O(N)  
**Expected Auxilliary Space** : O(1)

**Constraints:**  
1 <= Number of nodes <= 104

### Solution:

**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().

/\* 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 beginning 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.

**Recursive Approach :**

/\* 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 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);

}

}

}

/\* 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 = 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;

/\* 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

**Another Approach:** Create a pointer that will point towards the first occurrence of every element and another pointer temp which will iterate to every element and when the value of the previous pointer is not equal to the temp pointer, we will set the pointer of the previous pointer to the first occurrence of another node.

Below is the implementation of the above approach:

// C++ program to remove duplicates

// from a sorted linked list

#include <bits/stdc++.h>

using namespace std;

// Linked list Node

struct Node

{

int data;

Node \*next;

Node(int d)

{

data = d;

next = NULL;

}

};

// Function to remove duplicates

// from the given linked list

Node \*removeDuplicates(Node \*head)

{

// Two references to head

// temp will iterate to the

// whole Linked List

// prev will point towards

// the first occurrence of every element

Node \*temp = head,\*prev=head;

// Traverse list till the last node

while (temp != NULL)

{

// Compare values of both pointers

if(temp->data != prev->data)

{

/\* if the value of prev is

not equal to the value of

temp that means there are no

more occurrences of the prev data->

So we can set the next of

prev to the temp node->\*/

prev->next = temp;

prev = temp;

}

/\*Set the temp to the next node\*/

temp = temp->next;

}

/\*This is the edge case if there

are more than one occurrences

of the last element\*/

if(prev != temp)

{

prev->next = NULL;

}

return head;

}

Node \*push(Node \*head, int new\_data)

{

/\* 1 & 2: Allocate the Node &

Put in the data\*/

Node \*new\_node = new Node(new\_data);

/\* 3. Make next of new Node as head \*/

new\_node->next = head;

/\* 4. Move the head to point to new Node \*/

head = new\_node;

return head;

}

/\* Function to print linked list \*/

void printList(Node \*head)

{

Node \*temp = head;

while (temp != NULL)

{

cout << temp->data << " ";

temp = temp->next;

}

cout << endl;

}

/\* Driver code \*/

int main()

{

Node \*llist = NULL;

llist = push(llist,20);

llist = push(llist,13);

llist = push(llist,13);

llist = push(llist,11);

llist = push(llist,11);

llist = push(llist,11);

cout << ("List before removal of duplicates\n");

printList(llist);

cout << ("List after removal of elements\n");

llist = removeDuplicates(llist);

printList(llist);

}

**Output**

List before removal of duplicates

11 11 11 13 13 20

List after removal of elements

11 13 20

**Another Approach: Using Maps**

The idea is to push all the values in a map and printing its keys.

Below is the implementation of the above approach:

// CPP program for the above approach

#include <bits/stdc++.h>

using namespace std;

/\* Link list node \*/

struct Node {

int data;

Node\* next;

Node()

{

data = 0;

next = NULL;

}

};

/\* 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 = 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;

}

}

// Function to remove duplicates

void removeDuplicates(Node\* head)

{

unordered\_map<int, bool> track;

Node\* temp = head;

while (temp) {

if (track.find(temp->data) == track.end()) {

cout << temp->data << " ";

}

track[temp->data] = true;

temp = temp->next;

}

}

// Driver Code

int main()

{

Node\* head = NULL;

/\* 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);

cout << "\nLinked list after duplicate removal ";

removeDuplicates(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(Number of Nodes)

**Space Complexity:**O(Number of Nodes)

## 132. Remove Duplicates in a Un-sorted Linked List.

Given an unsorted linked list of **N** nodes. The task is to remove duplicate elements from this unsorted Linked List. When a value appears in multiple nodes, the node which appeared first should be kept, all others duplicates are to be removed.

**Example 1:**

**Input:**

N = 4

value[] = {5,2,2,4}

**Output:** 5 2 4

**Explanation:**Given linked list elements are

5->2->2->4, in which 2 is repeated only.

So, we will delete the extra repeated

elements 2 from the linked list and the

resultant linked list will contain 5->2->4

**Example 2:**

**Input:**

N = 5

value[] = {2,2,2,2,2}

**Output:** 2

**Explanation:**Given linked list elements are

2->2->2->2->2, in which 2 is repeated. So,

we will delete the extra repeated elements

2 from the linked list and the resultant

linked list will contain only 2.

**Your Task:**  
You have to complete the method **removeDuplicates**() which takes **1** argument: the **head** of the linked list.  Your function should return a pointer to a linked list with no duplicate element.

**Expected Time Complexity:** O(N)  
**Expected Auxiliary Space:** O(N)

**Constraints:**  
1 <= size of linked lists <= 106  
0 <= numbers in list <= 104

### Solution:

**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.

/\* 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 \*/

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

Time Complexity: O(n^2)

**METHOD 2 (Use Sorting)**   
In general, Merge Sort is the best-suited sorting algorithm for sorting linked lists efficiently.   
1) Sort the elements using Merge Sort. We will soon be writing a post about sorting a linked list. O(nLogn)   
2) Remove duplicates in linear time using the [algorithm for removing duplicates in sorted Linked List. O(n)](https://www.geeksforgeeks.org/remove-duplicates-from-a-sorted-linked-list/)  
Please note that this method doesn’t preserve the original order of elements.  
Time Complexity: O(nLogn)

**METHOD 3 (Use Hashing)**   
We traverse the link list from head to end. For every newly encountered element, we check whether it is in the hash table: if yes, we remove it; otherwise we put it in the hash table.

/\* 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)

{

// Hash to store seen values

unordered\_set<int> seen;

/\* Pick elements one by one \*/

struct Node\* curr = start;

struct Node\* prev = NULL;

while (curr != NULL) {

// If current value is seen before

if (seen.find(curr->data) != seen.end()) {

prev->next = curr->next;

delete (curr);

}

else {

seen.insert(curr->data);

prev = curr;

}

curr = prev->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 : \n");

printList(start);

removeDuplicates(start);

printf("\nLinked list after removing duplicates : \n");

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

Time Complexity: O(n) on average (assuming that hash table access time is O(1) on average).

## 133. Write a Program to Move the last element to Front in a 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.

### Solution:

**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)

/\* CPP Program to move last element

to front in a given linked list \*/

#include <bits/stdc++.h>

using namespace std;

/\* A linked list node \*/

class Node

{

public:

int data;

Node \*next;

};

/\* We are using a double pointer

head\_ref here because we change

head of the linked list inside

this function.\*/

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;

}

/\* UTILITY FUNCTIONS \*/

/\* Function to add a node

at the beginning of 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()

{

Node \*start = NULL;

/\* The constructed linked list is:

1->2->3->4->5 \*/

push(&start, 5);

push(&start, 4);

push(&start, 3);

push(&start, 2);

push(&start, 1);

cout<<"Linked list before moving last to front\n";

printList(start);

moveToFront(&start);

cout<<"\nLinked list after removing last to front\n";

printList(start);

return 0;

}

**Output:**

Linked list before moving last to front

1 2 3 4 5

Linked list after removing last to front

5 1 2 3 4

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

## 134. Add “1” to a number represented as a Linked List.

A number **N** is represented in Linked List such that each digit corresponds to a node in linked list. You need to add 1 to it.

**Example 1:**

**Input:**

LinkedList: 4->5->6

**Output:** 457

**Example 2:**

**Input:**

LinkedList: 1->2->3

**Output:** 124

**Your Task:**  
Your task is to complete the function addOne() which takes the head of the linked list as the only argument and returns the head of the modified linked list. The driver code prints the number.  
**Note:**The head represents the left-most digit of the number.

**Expected Time Complexity:**O(N).  
**Expected Auxiliary Space:**O(1).

**Constraints:**  
1 <= N <= 101000

### 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 calculation

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;

}

**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 code \*/

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

**Simple approach and easy implementation:**The idea is to store the last non 9 digit pointer so that if the last pointer is zero we can replace all the nodes after stored node(which contains the location of last digit before 9) to 0 and add the value of the stored node by 1

// 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;

}

Node\* addOne(Node\* head)

{

// Your Code here

// return head of list after adding one

Node\* ln = head;

if (head->next == NULL) {

head->data += 1;

return head;

}

Node\* t = head;

int prev;

while (t->next) {

if (t->data != 9) {

ln = t;

}

t = t->next;

}

if (t->data == 9 && ln != NULL) {

if (ln->data == 9 && ln == head) {

Node\* temp = newNode(1);

temp->next = head;

head = temp;

t = ln;

}

else {

t = ln;

t->data += 1;

t = t->next;

}

while (t) {

t->data = 0;

t = t->next;

}

}

else {

t->data += 1;

}

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("NULL");

printf("\n");

}

/\* Driver code \*/

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

The main focus in this question is on the digit 9 which creates all the changes otherwise for other digits we have to just increment their value by 1 but if we change the node’s value with the value 9 it makes a carry which then has to be passed through the linked list.  
Find the last node in the linked list which is not equal to 9. Now there are three cases:

1. If there is no such node i.e. the value of every node is 9 then the new linked list will contain all 0s and a single 1 inserted at the head of the linked list.
2. If the rightmost node which is not equal to 9 is the last node in the linked list then add 1 to this node and return the head of the linked list.
3. If the node is other than the last node i.e. every node after it is equal to 9 then add 1 to the current node and change all the nodes after it to 0.

Below is the implementation of the above approach:

// C++ implementation of the approach

#include <bits/stdc++.h>

using namespace std;

// Node of the linked list

struct Node {

int data;

Node\* next;

};

// Function to create a new node

Node\* create\_Node(int data)

{

Node\* temp = new Node();

temp->data = data;

temp->next = NULL;

return temp;

}

// Function to print the linked list

void print(Node\* head)

{

Node\* temp = head;

while (temp != NULL) {

cout << temp->data << " ";

temp = temp->next;

}

cout << endl;

}

// Function to add one to a number

// represented as linked list

Node\* addOne(Node\* head)

{

// To store the last node in the linked

// list which is not equal to 9

Node\* last = NULL;

Node\* cur = head;

// Iterate till the last node

while (cur->next != NULL) {

if (cur->data != 9) {

last = cur;

}

cur = cur->next;

}

// If last node is not equal to 9

// add 1 to it and return the head

if (cur->data != 9) {

cur->data++;

return head;

}

// If list is of the type 9 -> 9 -> 9 ...

if (last == NULL) {

last = new Node();

last->data = 0;

last->next = head;

head = last;

}

// For cases when the rightmost node which

// is not equal to 9 is not the last

// node in the linked list

last->data++;

last = last->next;

while (last != NULL) {

last->data = 0;

last = last->next;

}

return head;

}

// Driver code

int main()

{

Node\* head = create\_Node(1);

head->next = create\_Node(2);

head->next->next = create\_Node(9);

head->next->next->next = create\_Node(9);

cout << "Original list is : ";

print(head);

head = addOne(head);

cout << "Resultant list is : ";

print(head);

return 0;

}

**Output:**

Original list is : 1 2 9 9

Resultant list is : 1 3 0 0

**Time Complexity:** O(N)

**Space Complexity:** O(1)

## 135. Add two numbers represented by linked lists.

Given two numbers represented by two linked lists of size **N** and **M**. The task is to return a sum list.

The sum list is a linked list representation of the addition of two input numbers**from the last.**

**Example 1:**

**Input:**

N = 2

valueN[] = {4,5}

M = 3

valueM[] = {3,4,5}

**Output:** 3 9 0

**Explanation:** For the given two linked

list (4 5) and (3 4 5), after adding

the two linked list resultant linked

list will be (3 9 0).

**Example 2:**

**Input:**

N = 2

valueN[] = {6,3}

M = 1

valueM[] = {7}

**Output:** 7 0

**Explanation:** For the given two linked

list (6 3) and (7), after adding the

two linked list resultant linked list

will be (7 0).

**Your Task:**  
The task is to complete the function **addTwoLists**() which has node reference of both the linked lists and returns the head of the sum list.

**Expected Time Complexity:**O(N+M)  
**Expected Auxiliary Space:**O(Max(N,M)) for the resultant list.

**Constraints:**  
1 <= N, M <= 5000

### Solution:

**Approach**: Traverse both lists to the end and add preceding zeros in the list with lesser digits. Then call a recursive function on the start nodes of both lists which calls itself for the next nodes of both lists till it gets to the end. This functions creates a node for current digits sum and returns the carry.

**The steps are:**

1. Traverse the two linked lists in order to add preceding zeros in case a list is having lesser digits than the other one.
2. Start from the head node of both lists and call a recursive function for the next nodes.
3. Continue it till end of the lists.
4. Creates a node for current digits sum and return the carry.

Below is the implementation of this approach.

// C++ program to add two numbers

// represented by 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 insert a node at the

beginning of the Singly Linked List \*/

void push(Node\*\* head\_ref, int new\_data)

{

/\* allocate node \*/

Node\* new\_node = newNode(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;

}

/\* Adds contents of two linked lists and

return the head node of resultant list \*/

Node\* addTwoLists(Node\* first, Node\* second)

{

// res is head node of the resultant list

Node\* res = NULL;

Node \*temp, \*prev = NULL;

int carry = 0, sum;

// while both lists exist

while (first != NULL

|| second != NULL) {

// Calculate value of next

// digit in resultant list.

// The next digit is sum of

// following things

// (i) Carry

// (ii) Next digit of first

// list (if there is a next digit)

// (ii) Next digit of second

// list (if there is a next digit)

sum = carry + (first ? first->data : 0)

+ (second ? second->data : 0);

// update carry for next calculation

carry = (sum >= 10) ? 1 : 0;

// update sum if it is greater than 10

sum = sum % 10;

// Create a new node with sum as data

temp = newNode(sum);

// if this is the first node then

// set it as head of the resultant list

if (res == NULL)

res = temp;

// If this is not the first

// node then connect it to the rest.

else

prev->next = temp;

// Set prev for next insertion

prev = temp;

// Move first and second

// pointers to next nodes

if (first)

first = first->next;

if (second)

second = second->next;

}

if (carry > 0)

temp->next = newNode(carry);

// return head of the resultant list

return res;

}

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;

head->next = NULL;

/\* fix the head pointer \*/

return rest;

}

// A utility function to print a linked list

void printList(Node\* node)

{

while (node != NULL) {

cout << node->data << " ";

node = node->next;

}

cout << endl;

}

/\* Driver code \*/

int main(void)

{

Node\* res = NULL;

Node\* first = NULL;

Node\* second = NULL;

// create first list 7->5->9->4->6

push(&first, 6);

push(&first, 4);

push(&first, 9);

push(&first, 5);

push(&first, 7);

printf("First List is ");

printList(first);

// create second list 8->4

push(&second, 4);

push(&second, 8);

cout << "Second List is ";

printList(second);

// reverse both the lists

first = reverse(first);

second = reverse(second);

// Add the two lists

res = addTwoLists(first, second);

// reverse the res to get the sum

res = reverse(res);

cout << "Resultant list is ";

printList(res);

return 0;

}

**Output**

First List is 7 5 9 4 6

Second List is 8 4

Resultant list is 5 0 0 5 6

**Complexity Analysis:**

* **Time Complexity:** O(m + n), where m and n are numbers of nodes in first and second lists respectively.   
  The lists need to be traversed only once.
* **Space Complexity:** O(m + n).   
  A temporary linked list is needed to store the output number

**Method 2(Using STL):**Using stack data structure

**Approach :**

* *Create 3 stacks namely s1,s2,s3.*
* *Fill s1 with Nodes of list1 and fill s2 with nodes of list2.*
* *Fill s3 by creating new nodes and setting the data of new nodes to the sum of s1.top(), s2.top() and carry until list1 and list2 are empty .*
* *If the sum >9*
  + *set carry 1*
* *else*
  + *set carry 0*
* *Create a Node(say prev) that will contain the head of the sum List.*
* *Link all the elements of s3 from top to bottom*
* *return prev*

**Code:**

// C++ program to add two numbers represented by Linked

// Lists using Stack

#include <bits/stdc++.h>

using namespace std;

class Node {

public:

int data;

Node\* next;

};

Node\* newnode(int data)

{

Node\* x = new Node();

x->data = data;

return x;

}

// function that returns the sum of two numbers represented

// by linked lists

Node\* addTwoNumbers(Node\* l1, Node\* l2)

{

Node\* prev = NULL;

// Create 3 stacks

stack<Node\*> s1, s2, s3;

// Fill first stack with first List Elements

while (l1 != NULL) {

s1.push(l1);

l1 = l1->next;

}

// Fill second stack with second List Elements

while (l2 != NULL) {

s2.push(l2);

l2 = l2->next;

}

int carry = 0;

// Fill the third stack with the sum of first and second

// stack

while (!s1.empty() && !s2.empty()) {

int sum = s1.top()->data + s2.top()->data + carry;

Node\* temp = newnode(sum % 10);

s3.push(temp);

if (sum > 9) {

carry = 1;

}

else {

carry = 0;

}

s1.pop();

s2.pop();

}

while (!s1.empty()) {

int sum = carry + s1.top()->data;

Node\* temp = newnode(sum % 10);

s3.push(temp);

if (sum > 9) {

carry = 1;

}

else {

carry = 0;

}

s1.pop();

}

while (!s2.empty()) {

int sum = carry + s2.top()->data;

Node\* temp = newnode(sum % 10);

s3.push(temp);

if (sum > 9) {

carry = 1;

}

else {

carry = 0;

}

s2.pop();

}

// If carry is still present create a new node with

// value 1 and push it to the third stack

if (carry == 1) {

Node\* temp = newnode(1);

s3.push(temp);

}

// Link all the elements inside third stack with each

// other

if (!s3.empty())

prev = s3.top();

while (!s3.empty()) {

Node\* temp = s3.top();

s3.pop();

if (s3.size() == 0) {

temp->next = NULL;

}

else {

temp->next = s3.top();

}

}

return prev;

}

// utility functions

// Function that displays the List

void Display(Node\* head)

{

if (head == NULL) {

return;

}

while (head->next != NULL) {

cout << head->data << " -> ";

head = head->next;

}

cout << head->data << endl;

}

// Function that adds element at the end of the Linked List

void push(Node\*\* head\_ref, int d)

{

Node\* new\_node = newnode(d);

new\_node->next = NULL;

if (\*head\_ref == NULL) {

new\_node->next = \*head\_ref;

\*head\_ref = new\_node;

return;

}

Node\* last = \*head\_ref;

while (last->next != NULL && last != NULL) {

last = last->next;

}

last->next = new\_node;

return;

}

// Driver Program for above Functions

int main()

{

// Creating two lists

// first list = 9 -> 5 -> 0

// second List = 6 -> 7

Node\* first = NULL;

Node\* second = NULL;

Node\* sum = NULL;

push(&first, 9);

push(&first, 5);

push(&first, 0);

push(&second, 6);

push(&second, 7);

cout << "First List : ";

Display(first);

cout << "Second List : ";

Display(second);

sum = addTwoNumbers(first, second);

cout << "Sum List : ";

Display(sum);

return 0;

}

**Output**

First List : 9 -> 5 -> 0

Second List : 6 -> 7

Sum List : 1 -> 0 -> 1 -> 7

**Another Approach with time complexity O(N):**

 The given approach works as following steps:

1. First, we calculate sizes of both the linked lists, size1 and size2, respectively.
2. Then we traverse the bigger linked list, if any, and decrement till size of both become same.
3. Now we traverse both linked lists till end.
4. Now the backtracking occurs while performing addition.
5. Finally, the head node is returned of the linked list containing the answer.

#include <iostream>

using namespace std;

struct Node {

int data;

struct Node\* next;

};

// recursive function

Node\* addition(Node\* temp1, Node\* temp2, int size1,

int size2)

{

// creating a new Node

Node\* newNode

= (struct Node\*)malloc(sizeof(struct Node));

// base case

if (temp1->next == NULL && temp2->next == NULL) {

// addition of current nodes which is the last nodes

// of both linked lists

newNode->data = (temp1->data + temp2->data);

// set this current node's link null

newNode->next = NULL;

// return the current node

return newNode;

}

// creating a node that contains sum of previously added

// number

Node\* returnedNode

= (struct Node\*)malloc(sizeof(struct Node));

// if sizes are same then we move in both linked list

if (size2 == size1) {

// recursively call the function

// move ahead in both linked list

returnedNode = addition(temp1->next, temp2->next,

size1 - 1, size2 - 1);

// add the current nodes and append the carry

newNode->data = (temp1->data + temp2->data)

+ ((returnedNode->data) / 10);

}

// or else we just move in big linked list

else {

// recursively call the function

// move ahead in big linked list

returnedNode = addition(temp1, temp2->next, size1,

size2 - 1);

// add the current node and carry

newNode->data

= (temp2->data) + ((returnedNode->data) / 10);

}

// this node contains previously added numbers

// so we need to set only rightmost digit of it

returnedNode->data = (returnedNode->data) % 10;

// set the returned node to the current node

newNode->next = returnedNode;

// return the current node

return newNode;

}

// Function to add two numbers represented by nexted list.

struct Node\* addTwoLists(struct Node\* head1,

struct Node\* head2)

{

struct Node \*temp1, \*temp2, \*ans = NULL;

temp1 = head1;

temp2 = head2;

int size1 = 0, size2 = 0;

// calculating the size of first linked list

while (temp1 != NULL) {

temp1 = temp1->next;

size1++;

}

// calculating the size of second linked list

while (temp2 != NULL) {

temp2 = temp2->next;

size2++;

}

Node\* returnedNode

= (struct Node\*)malloc(sizeof(struct Node));

// traverse the bigger linked list

if (size2 > size1) {

returnedNode = addition(head1, head2, size1, size2);

}

else {

returnedNode = addition(head2, head1, size2, size1);

}

// creating new node if head node is >10

if (returnedNode->data >= 10) {

ans = (struct Node\*)malloc(sizeof(struct Node));

ans->data = (returnedNode->data) / 10;

returnedNode->data = returnedNode->data % 10;

ans->next = returnedNode;

}

else

ans = returnedNode;

// return the head node of linked list that contains

// answer

return ans;

}

void Display(Node\* head)

{

if (head == NULL) {

return;

}

while (head->next != NULL) {

cout << head->data << " -> ";

head = head->next;

}

cout << head->data << endl;

}

// Function that adds element at the end of the Linked List

void push(Node\*\* head\_ref, int d)

{

Node\* new\_node

= (struct Node\*)malloc(sizeof(struct Node));

new\_node->data = d;

new\_node->next = NULL;

if (\*head\_ref == NULL) {

new\_node->next = \*head\_ref;

\*head\_ref = new\_node;

return;

}

Node\* last = \*head\_ref;

while (last->next != NULL && last != NULL) {

last = last->next;

}

last->next = new\_node;

return;

}

// Driver Program for above Functions

int main()

{

// Creating two lists

Node\* first = NULL;

Node\* second = NULL;

Node\* sum = NULL;

push(&first, 5);

push(&first, 6);

push(&first, 3);

push(&second, 8);

push(&second, 4);

push(&second, 2);

cout << "First List : ";

Display(first);

cout << "Second List : ";

Display(second);

sum = addTwoLists(first, second);

cout << "Sum List : ";

Display(sum);

return 0;

}

**Output**

First List : 5 -> 6 -> 3

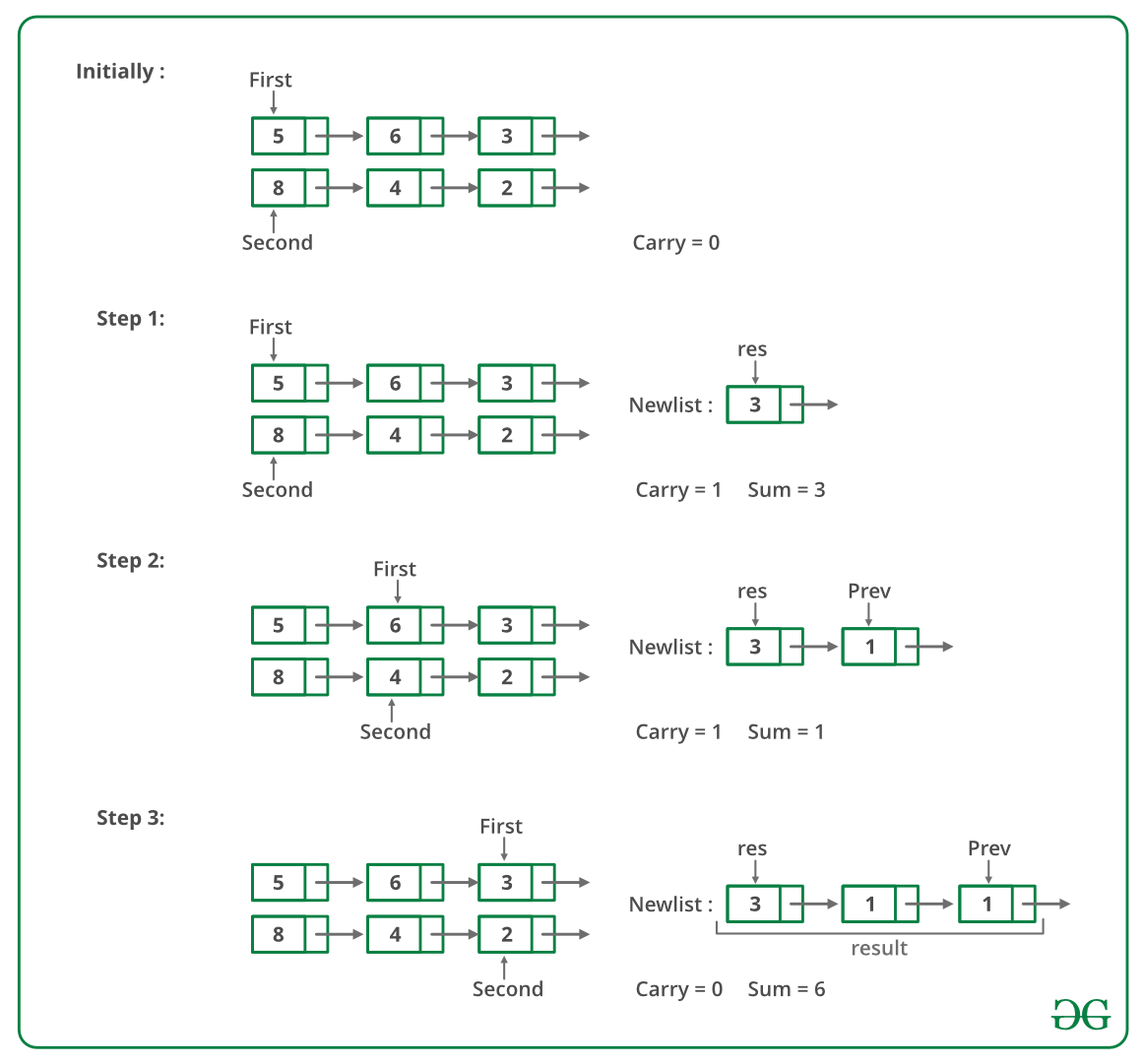
Second List : 8 -> 4 -> 2

Sum List : 1 -> 4 -> 0 -> 5

**Another Approach: (If it is not allowed to modify the list)**

Following are the steps.   
**1)** Calculate sizes of given two linked lists.   
**2)** If sizes are same, then calculate sum using recursion. Hold all nodes in recursion call stack till the rightmost node, calculate the sum of rightmost nodes and forward carry to the left side.   
**3)** If size is not same, then follow below steps:   
….**a)** Calculate difference of sizes of two linked lists. Let the difference be *diff*   
….**b)** Move *diff*nodes ahead in the bigger linked list. Now use step 2 to calculate the sum of the smaller list and right sub-list (of the same size) of a larger list. Also, store the carry of this sum.   
….**c)** Calculate the sum of the carry (calculated in the previous step) with the remaining left sub-list of a larger list. Nodes of this sum are added at the beginning of the sum list obtained the previous step.

Below is a dry run of the above approach:



Below image is the implementation of the above approach.

// A C++ recursive program to add two linked lists

#include <bits/stdc++.h>

using namespace std;

// A linked List Node

class Node {

public:

int data;

Node\* next;

};

typedef Node node;

/\* A utility function to insert

a node at the beginning of linked list \*/

void push(Node\*\* head\_ref, int new\_data)

{

/\* allocate node \*/

Node\* new\_node = new Node[(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;

}

/\* A utility function to print linked list \*/

void printList(Node\* node)

{

while (node != NULL) {

cout << node->data << " ";

node = node->next;

}

cout << endl;

}

// A utility function to swap two pointers

void swapPointer(Node\*\* a, Node\*\* b)

{

node\* t = \*a;

\*a = \*b;

\*b = t;

}

/\* A utility function to get size of linked list \*/

int getSize(Node\* node)

{

int size = 0;

while (node != NULL) {

node = node->next;

size++;

}

return size;

}

// Adds two linked lists of same size

// represented by head1 and head2 and returns

// head of the resultant linked list. Carry

// is propagated while returning from the recursion

node\* addSameSize(Node\* head1, Node\* head2, int\* carry)

{

// Since the function assumes linked lists are of same

// size, check any of the two head pointers

if (head1 == NULL)

return NULL;

int sum;

// Allocate memory for sum node of current two nodes

Node\* result = new Node[(sizeof(Node))];

// Recursively add remaining nodes and get the carry

result->next

= addSameSize(head1->next, head2->next, carry);

// add digits of current nodes and propagated carry

sum = head1->data + head2->data + \*carry;

\*carry = sum / 10;

sum = sum % 10;

// Assigne the sum to current node of resultant list

result->data = sum;

return result;

}

// This function is called after the

// smaller list is added to the bigger

// lists's sublist of same size. Once the

// right sublist is added, the carry

// must be added toe left side of larger

// list to get the final result.

void addCarryToRemaining(Node\* head1, Node\* cur, int\* carry,

Node\*\* result)

{

int sum;

// If diff. number of nodes are not traversed, add carry

if (head1 != cur) {

addCarryToRemaining(head1->next, cur, carry,

result);

sum = head1->data + \*carry;

\*carry = sum / 10;

sum %= 10;

// add this node to the front of the result

push(result, sum);

}

}

// The main function that adds two linked lists

// represented by head1 and head2. The sum of

// two lists is stored in a list referred by result

void addList(Node\* head1, Node\* head2, Node\*\* result)

{

Node\* cur;

// first list is empty

if (head1 == NULL) {

\*result = head2;

return;

}

// second list is empty

else if (head2 == NULL) {

\*result = head1;

return;

}

int size1 = getSize(head1);

int size2 = getSize(head2);

int carry = 0;

// Add same size lists

if (size1 == size2)

\*result = addSameSize(head1, head2, &carry);

else {

int diff = abs(size1 - size2);

// First list should always be larger than second

// list. If not, swap pointers

if (size1 < size2)

swapPointer(&head1, &head2);

// move diff. number of nodes in first list

for (cur = head1; diff--; cur = cur->next)

;

// get addition of same size lists

\*result = addSameSize(cur, head2, &carry);

// get addition of remaining first list and carry

addCarryToRemaining(head1, cur, &carry, result);

}

// if some carry is still there, add a new node to the

// front of the result list. e.g. 999 and 87

if (carry)

push(result, carry);

}

// Driver code

int main()

{

Node \*head1 = NULL, \*head2 = NULL, \*result = NULL;

int arr1[] = { 9, 9, 9 };

int arr2[] = { 1, 8 };

int size1 = sizeof(arr1) / sizeof(arr1[0]);

int size2 = sizeof(arr2) / sizeof(arr2[0]);

// Create first list as 9->9->9

int i;

for (i = size1 - 1; i >= 0; --i)

push(&head1, arr1[i]);

// Create second list as 1->8

for (i = size2 - 1; i >= 0; --i)

push(&head2, arr2[i]);

addList(head1, head2, &result);

printList(result);

return 0;

}

**Output**

1 0 1 7

Time Complexity: O(m+n) where m and n are the sizes of given two linked lists.

**Iterative Approach:**

This implementation does not have any recursion call overhead, which means it is an iterative solution.

Since we need to start adding numbers from the last of the two linked lists. So, here we will use the stack data structure to implement this.

* We will firstly make two stacks from the given two linked lists.
* Then, we will run a loop till both the stack become empty.
* in every iteration, we keep the track of the carry.
* In the end, if carry>0, that means we need extra node at the start of the resultant list to accommodate this carry.

// C++ Iterative program to add two linked lists

#include <bits/stdc++.h>

using namespace std;

// A linked List Node

class Node

{

public:

int data;

Node\* next;

};

// to push a new node to linked list

void push(Node\*\* head\_ref, int new\_data)

{

/\* allocate node \*/

Node\* new\_node = new Node[(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;

}

// to add two new numbers

Node\* addTwoNumList(Node\* l1, Node\* l2) {

stack<int> s1,s2;

while(l1!=NULL){

s1.push(l1->data);

l1=l1->next;

}

while(l2!=NULL){

s2.push(l2->data);

l2=l2->next;

}

int carry=0;

Node\* result=NULL;

while(s1.empty()==false || s2.empty()==false){

int a=0,b=0;

if(s1.empty()==false){

a=s1.top();s1.pop();

}

if(s2.empty()==false){

b=s2.top();s2.pop();

}

int total=a+b+carry;

Node\* temp=new Node();

temp->data=total%10;

carry=total/10;

if(result==NULL){

result=temp;

}else{

temp->next=result;

result=temp;

}

}

if(carry!=0){

Node\* temp=new Node();

temp->data=carry;

temp->next=result;

result=temp;

}

return result;

}

// to print a linked list

void printList(Node \*node)

{

while (node != NULL)

{

cout<<node->data<<" ";

node = node->next;

}

cout<<endl;

}

// Driver Code

int main()

{

Node \*head1 = NULL, \*head2 = NULL;

int arr1[] = {5, 6, 7};

int arr2[] = {1, 8};

int size1 = sizeof(arr1) / sizeof(arr1[0]);

int size2 = sizeof(arr2) / sizeof(arr2[0]);

// Create first list as 5->6->7

int i;

for (i = size1-1; i >= 0; --i)

push(&head1, arr1[i]);

// Create second list as 1->8

for (i = size2-1; i >= 0; --i)

push(&head2, arr2[i]);

Node\* result=addTwoNumList(head1, head2);

printList(result);

return 0;

}

**Output**

5 8 5

**Approach:**

Following are the steps:

1. Reverse List L1.
2. Reverse List L2.
3. Add the nodes of both the lists iteratively.
4. Reverse the resultant list and return its head.

Below is the implementation of the above approach:

// C++ implementation of the approach

#include <iostream>

using namespace std;

class LinkedList;

// Node class for the linked list

class Node {

int data;

Node\* next;

friend LinkedList;

public:

Node();

Node(int x);

};

Node::Node()

{

data = 0;

next = NULL;

}

// Function to initialise

// a node with value x

Node::Node(int x)

{

data = x;

next = NULL;

}

// Linkedlist class with helper functions

class LinkedList {

public:

Node\* head;

LinkedList();

void insert(int x);

void reverse();

void traverse();

void sum(LinkedList\*);

};

LinkedList::LinkedList()

{

head = NULL;

}

// Function to insert a node at

// the head of the list

void LinkedList::insert(int x)

{

Node\* node = new Node();

node->data = x;

if (head == NULL)

head = node;

else {

node->next = head;

head = node;

}

}

// Function to reverse the linked list

void LinkedList::reverse()

{

Node \*prev = NULL, \*curr = head;

while (curr) {

Node\* temp = curr->next;

curr->next = prev;

prev = curr;

curr = temp;

}

head = prev;

}

// Function to traverse and print the list

void LinkedList::traverse()

{

Node\* temp = head;

while (temp) {

cout << temp->data << " -> ";

temp = temp->next;

}

cout << "NULL";

}

// Function to add two numbers

// represented as linked lists

void LinkedList::sum(LinkedList\* l2)

{

reverse();

l2->reverse();

Node \*start1 = head, \*start2 = l2->head;

Node\* prev = NULL;

int carry = 0;

// While both lists exist

while (start1 && start2) {

// Current sum

int temp = start1->data + start2->data + carry;

// Handle carry

start1->data = temp % 10;

carry = temp / 10;

prev = start1;

// Get to next nodes

start1 = start1->next;

start2 = start2->next;

}

// If there are remaining digits

// in any one of the lists

if (start1 || start2) {

if (start2)

prev->next = start2;

start1 = prev->next;

// While first list has digits remaining

while (start1) {

int temp = start1->data + carry;

start1->data = temp % 10;

carry = temp / 10;

prev = start1;

start1 = start1->next;

}

}

// If a new node needs to be

// created due to carry

if (carry > 0) {

prev->next = new Node(carry);

}

// Reverse the resultant list

reverse();

}

// Driver code

int main()

{

// Create first list

LinkedList\* l1 = new LinkedList();

l1->insert(3);

l1->insert(6);

l1->insert(5);

// Create second list

LinkedList\* l2 = new LinkedList();

l2->insert(2);

l2->insert(4);

l2->insert(8);

// Add the lists

l1->sum(l2);

// Print the resultant list

l1->traverse();

return 0;

}

**Output:**

1 -> 4 -> 0 -> 5 -> NULL

**Time Complexity:** O(max(m, n)) where m and n are number of nodes in list l1 and list l2 respectively.   
**Space Complexity:** O(1)

## 136. Intersection of two Sorted Linked List.

Given two lists sorted in increasing order, create 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.  
**Note:** The list elements are not necessarily distinct.

**Example 1:**

**Input:**

L1 = 1->2->3->4->6

L2 = 2->4->6->8

**Output:** 2 4 6

**Explanation:** For the given first two

linked list, 2, 4 and 6 are the elements

in the intersection.

**Example 2:**

**Input:**

L1 = 10->20->40->50

L2 = 15->40

**Output:** 40

**Your Task:**  
The task is to complete the function **intersection**() which should find the intersection of two linked list and add all the elements in intersection to the third linked list and return the head of the third linked list.

**Expected Time Complexity** : O(n+m)  
**Expected Auxilliary Space** : O(n+m)  
**Note:** n,m are the size of the linked lists.

**Constraints:**  
1 <= size of linked lists <= 5000  
1 <= Data in linked list nodes <= 1000

### Solution:

**My Implementation:**

Node\* findIntersection(Node\* head1, Node\* head2)

{

Node \*head = NULL, \*curr=NULL;

while(head1!=NULL && head2!=NULL){

if(head1->data == head2->data){

Node \*temp = new Node(head1->data);

if(head==NULL){

head = temp;

curr = head;

}

else{

curr->next = temp;

curr = curr->next;

}

head1 = head1->next;

head2 = head2->next;

}

else if(head1->data < head2->data)

head1 = head1->next;

else

head2 = head2->next;

}

return head;

}

**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 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:

#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:** Using Local References.   
**Approach:** 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 the list is built at its tail, either the dummy node or the struct node\*\* “reference” strategy can be used.

Below is the implementation of the above approach:

#include <stdio.h>

#include <stdlib.h>

/\* Link list node \*/

struct Node {

int data;

struct Node\* next;

};

void push(struct Node\*\* head\_ref,

int new\_data);

/\* This solution uses the local reference \*/

struct Node\* sortedIntersect(

struct Node\* a,

struct Node\* b)

{

struct Node\* result = NULL;

struct Node\*\* lastPtrRef = &result;

/\* Advance comparing the first

nodes in both lists.

When one or the other list runs

out, we're done. \*/

while (a != NULL && b != NULL) {

if (a->data == b->data) {

/\* found a node for the intersection \*/

push(lastPtrRef, a->data);

lastPtrRef = &((\*lastPtrRef)->next);

a = a->next;

b = b->next;

}

else if (a->data < b->data)

a = a->next; /\* advance the smaller list \*/

else

b = b->next;

}

return (result);

}

/\* UTILITY FUNCTIONS \*/

/\* Function to insert a node at the

beginning of the linked list \*/

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;

}

/\* 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 functions\*/

int main()

{

/\* Start with the empty lists \*/

struct Node\* a = NULL;

struct Node\* b = NULL;

struct 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);

printf("\n Linked list containing common items of a & b \n ");

printList(intersect);

getchar();

}

**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.

**Method 3:** 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:

#include <bits/stdc++.h>

using namespace std;

// Link list node

struct Node

{

int data;

struct Node\* next;

};

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;

}

/\* UTILITY FUNCTIONS \*/

/\* Function to insert a node at

the beginning of the linked list \*/

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;

}

/\* Function to print nodes in

a given linked list \*/

void printList(struct Node\* node)

{

while (node != NULL)

{

cout << " " << node->data;

node = node->next;

}

}

// Driver code

int main()

{

/\* Start with the empty lists \*/

struct Node\* a = NULL;

struct Node\* b = NULL;

struct 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 << "\n Linked list containing "

<< "common items of a & b \n ";

printList(intersect);

return 0;

}

**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.

**Method 4:**Use Hashing

import java.util.\*;

// This code is contributed by ayyuce demirbas

public class LinkedList {

Node head;

static class Node {

int data;

Node next;

Node(int d) {

data = d;

next=null;

}

}

public void printList() {

Node n= head;

while(n!=null) {

System.out.println(n.data+ " ");

n=n.next;

}

}

public void append(int d) {

Node n= new Node(d);

if(head== null) {

head= new Node(d);

return;

}

n.next=null;

Node last= head;

while(last.next !=null) {

last=last.next;

}

last.next=n;

return;

}

static int[] intersection(Node tmp1, Node tmp2, int k) {

int[] res = new int[k];

HashSet<Integer> set = new HashSet<Integer>();

while(tmp1 != null) {

set.add(tmp1.data);

tmp1=tmp1.next;

}

int cnt=0;

while(tmp2 != null) {

if(set.contains(tmp2.data)) {

res[cnt]=tmp2.data;

cnt++;

}

tmp2=tmp2.next;

}

return res;

}

public static void main(String[] args) {

LinkedList ll = new LinkedList();

LinkedList ll1 = new LinkedList();

ll.append(0);

ll.append(1);

ll.append(2);

ll.append(3);

ll.append(4);

ll.append(5);

ll.append(6);

ll.append(7);

ll1.append(9);

ll1.append(0);

ll1.append(12);

ll1.append(3);

ll1.append(4);

ll1.append(5);

ll1.append(6);

ll1.append(7);

int[] arr= intersection(ll.head, ll1.head,6);

for(int i : arr) {

System.out.println(i);

}

}

}

**Output**

0

3

4

5

6

7

**Complexity Analysis:**

* Time Complexity: O(n)

## 137. Intersection Point of two Linked Lists.

Given two singly linked lists of size **N** and **M,**write a program to get the point where two linked lists intersect each other.

**Example 1:**

**Input:**

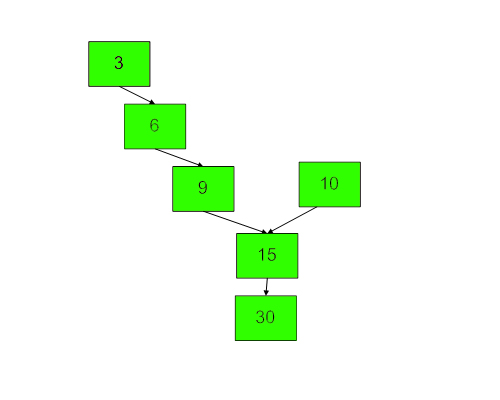
LinkList1 = 3->6->9->common

LinkList2 = 10->common

common = 15->30->NULL

**Output: 1**5

**Explanation:**



**Example 2:**

**Input:**

Linked List 1 = 4->1->common

Linked List 2 = 5->6->1->common

common = 8->4->5->NULL

**Output:** 8

**Explanation:**

**4 5**

**| |**

**1 6**

**\ /**

**8 ----- 1**

**|**

**4**

**|**

**5**

**|**

**NULL**

**Your Task:**  
You don't need to read input or print anything. The task is to complete the function **intersetPoint**() which takes the pointer to the head of linklist1(**head1**) and linklist2(**head2**) as input parameters and returns data value of a node where two linked lists intersect. If linked list do not merge at any point, then it should return **-1**.  
**Challenge**: Try to solve the problem without using any extra space.

**Expected Time Complexity:** O(N+M)  
**Expected Auxiliary Space:** O(1)

**Constraints:**  
1 ≤ N + M ≤ 2\*105  
-1000 ≤ value ≤ 1000

### Solution:

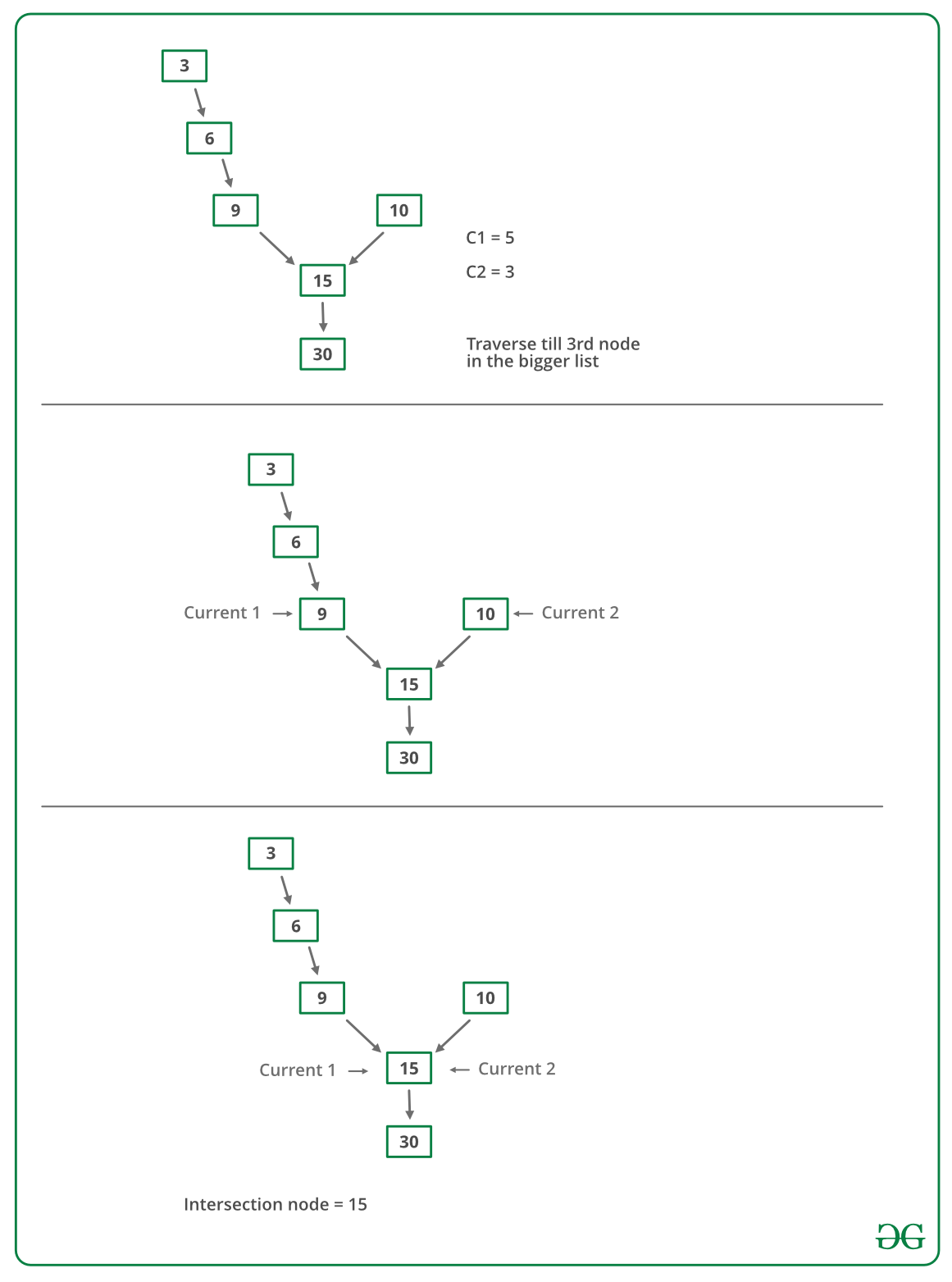
**Method 1(Simply use two loops)**   
Use 2 nested for loops. The outer loop will be for each node of the 1st list and inner loop will be for 2nd list. In the inner loop, check if any of nodes of the 2nd list is same as the current node of the first linked list. The time complexity of this method will 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;

}

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);

}

**Output**

The node of intersection is 15

**Time Complexity:** O(m+n)   
**Auxiliary Space:** O(1)

**My code using same approach:**

int intersectPoint(Node\* head1, Node\* head2)

{

int count1=0, count2=0;

Node \*temp = head1;

while(temp!=NULL){

count1++;

temp = temp->next;

}

temp = head2;

while(temp!=NULL){

count2++;

temp = temp->next;

}

Node \*curr1 = head1, \*curr2 = head2;

if(count1>count2){

while(count1!=count2){

curr1 = curr1->next;

count1--;

}

}

else if(count2>count1){

while(count2!=count1){

curr2 = curr2->next;

count2--;

}

}

while(curr1!=curr2 && curr1){

curr1 = curr1->next;

curr2 = curr2->next;

}

if(curr1==NULL)

return -1;

return curr1->data;

}

**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.

**Method 8( 2-pointer technique ):**

Using Two pointers :

* 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, they 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 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;

}

// Function to print intersection nodes

// in a given linked list

void print(Node\* node)

{

if (node == NULL)

cout << "NULL";

while (node->next != NULL) {

cout << node->data << "->";

node = node->next;

}

cout << node->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 30 are elements in the intersection list

\*/

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;

Node\* intersect\_node = NULL;

// Find the intersection node of two linked lists

intersect\_node = intersectPoint(head1, head2);

cout << "INTERSEPOINT LIST :";

print(intersect\_node);

return 0;

// This code is contributed by bolliranadheer

}

**Output**

INTERSEPOINT LIST :15->30

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

## 138. Merge Sort For Linked lists.[Very Important]

Given Pointer/Reference to the head of the linked list, the task is to **Sort the given linked list using Merge Sort**.  
**Note:** If the length of linked list is odd, then the extra node should go in the first list while splitting.

**Example 1:**

**Input:**

N = 5

value[] = {3,5,2,4,1}

**Output:** 1 2 3 4 5

**Explanation:** After sorting the given

linked list, the resultant matrix

will be 1->2->3->4->5.

**Example 2:**

**Input:**

N = 3

value[] = {9,15,0}

**Output:** 0 9 15

**Explanation:** After sorting the given

linked list , resultant will be

0->9->15.

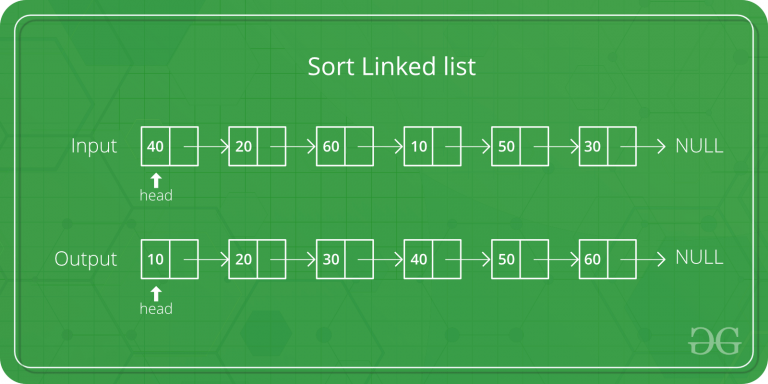
**Your Task:**  
**For C++ and Python:** The task is to complete the function **mergeSort**() which sort the linked list using merge sort function.  
**For Java:**The task is to complete the function **mergeSort**() and return the node which can be used to print the sorted linked list.

**Expected Time Complexity:** O(N\*Log(N))  
**Expected Auxiliary Space:** O(N)

**Constraints:**  
1 <= T <= 100  
1 <= N <= 105

### Solution:

[Merge sort](http://en.wikipedia.org/wiki/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 the 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 the 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](https://www.geeksforgeeks.org/merge-two-sorted-linked-lists/))

and update the head pointer using headRef.

\*headRef = SortedMerge(a, b);

// 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 beginning 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;

}

**Output:**

Sorted Linked List is:

2 3 5 10 15 20

**Time Complexity:**O(n\*log n)

**Space Complexity:**O(n\*log n)

**Approach 2:** This approach is simpler and uses log n space.

mergeSort():

1. If the size of the linked list is 1 then return the head
2. Find mid using The Tortoise and The Hare Approach
3. Store the next of mid in head2 i.e. the right sub-linked list.
4. Now Make the next midpoint null.
5. Recursively call mergeSort() on both left and right sub-linked list and store the new head of the left and right linked list.
6. Call merge() given the arguments new heads of left and right sub-linked lists and store the final head returned after merging.
7. Return the final head of the merged linkedlist.

merge(head1, head2):

1. Take a pointer say merged to store the merged list in it and store a dummy node in it.
2. Take a pointer temp and assign merge to it.
3. If the data of head1 is less than the data of head2, then, store head1 in next of temp & move head1 to the next of head1.
4. Else store head2 in next of temp & move head2 to the next of head2.
5. Move temp to the next of temp.
6. Repeat steps 3, 4 & 5 until head1 is not equal to null and head2 is not equal to null.
7. Now add any remaining nodes of the first or the second linked list to the merged linked list.
8. Return the next of merged(that will ignore the dummy and return the head of the final merged linked list)

#include<iostream>

using namespace std;

//Node structure

struct Node{

int data;

Node \*next;

};

//function to insert in list

void insert(int x,Node \*\*head) {

if(\*head == NULL){

\*head = new Node;

(\*head)->data = x;

(\*head)->next = NULL;

return;

}

Node \*temp = new Node;

temp->data = (\*head)->data;

temp->next = (\*head)->next;

(\*head)->data=x;

(\*head)->next=temp;

}

//function to print the list

void print(Node \*head) {

Node \*temp=head;

while(temp!=NULL) {

cout<<temp->data<<" ";

temp = temp->next;

}

}

Node \*merge(Node \*firstNode,Node \*secondNode) {

Node \*merged = new Node;

Node \*temp = new Node;

//merged is equal to temp so in the end we have the top Node.

merged = temp;

//while either firstNode or secondNode becomes NULL

while(firstNode != NULL && secondNode != NULL) {

if(firstNode->data<=secondNode->data) {

temp->next = firstNode;

firstNode = firstNode->next;

}

else {

temp->next = secondNode;

secondNode = secondNode->next;

}

temp = temp->next;

}

//any remaining Node in firstNode or secondNode gets inserted in the temp List

while(firstNode!=NULL) {

temp->next = firstNode;

firstNode = firstNode->next;

temp = temp->next;

}

while(secondNode!=NULL) {

temp->next = secondNode;

secondNode = secondNode->next;

temp = temp->next;

}

//return the head of the sorted list

return merged->next;

}

//function to calculate the middle Element

Node \*middle(Node \*head) {

Node \*slow = head;

Node \*fast = head->next;

while(slow->next != NULL && (fast!=NULL && fast->next!=NULL)) {

slow = slow->next;

fast = fast->next->next;

}

return slow;

}

//function to sort the given list

Node \*sort(Node \*head){

if(head->next == NULL) {

return head;

}

Node \*mid = new Node;

Node \*head2 = new Node;

mid = middle(head);

head2 = mid->next;

mid->next = NULL;

//recursive call to sort() hence diving our problem, and then merging the solution

Node \*finalhead = merge(sort(head),sort(head2));

return finalhead;

}

int main(void) {

Node \*head = NULL;

int n[]={7,10,5,20,3,2};

for(int i=0;i<6;i++) {

insert(n[i],&head); //inserting in the list

}

cout<<"\nSorted Linked List is: \n";

print(sort(head)); //printing the sorted list returned by sort()

return 0;

}

**Output:**

Sorted Linked List is:

2 3 5 7 10 20

**Time Complexity**: O(n\*log n)

**Space Complexity:** O(log n)

## 139. Quicksort for Linked Lists.[Very Important]

Sort the given **L**inked **L**ist using quicksort. which takes **O(n^2)** time in worst case and **O(nLogn)** in average and best cases, otherwise you may get TLE.

**Input:**  
In this problem, method takes 1 argument: address of the **head** of the linked list. The function should not read any input from stdin/console.  
The struct Node has a data part which stores the **data** and a next pointer which points to the **next** element of the linked list.  
There are multiple test cases. For each test case, this method will be called individually.

**Output:**  
Set **\*headRef** to head of resultant linked list.

**User Task:**  
The task is to complete the function **quickSort**() which should set the \*headRef to head of the resultant linked list.

**Constraints:**  
1<=**T**<=100  
1<=**N**<=200

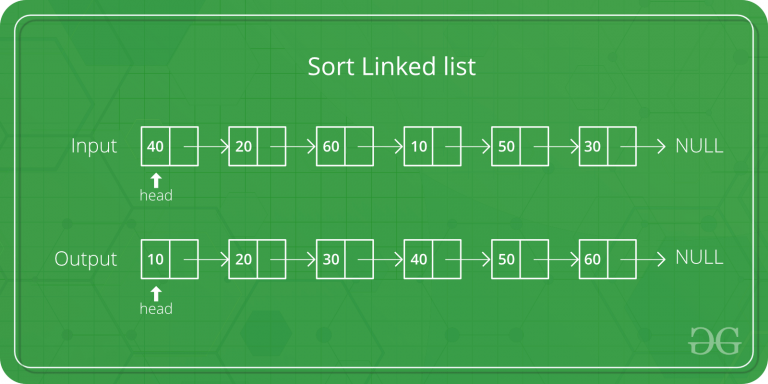
**Note:**If you use "Test" or "Expected Output Button" use below example format  
  
**Example:  
Input:**  
2  
3  
1 6 2  
4  
1 9 3 8

**Output:**  
1 2 6  
1 3 8 9

**Explanation:  
Testcase 1:** After sorting the nodes, we have 1, 2 and 6.  
**Testcase 2:** After sorting the nodes, we have 1, 3, 8 and 9.

### Solution:

Following is C++ implementation for same. 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.

// 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;

}

// here the sorting happens exclusive of the end node

struct Node\* quickSortRecur(struct Node\* head,

struct Node\* end)

{

// base condition

if (!head || head == end)

return head;

Node \*newHead = NULL, \*newEnd = NULL;

// Partition the list, newHead and newEnd will be

// updated by the partition function

struct Node\* pivot

= partition(head, end, &newHead, &newEnd);

// If pivot is the smallest element - no need to recur

// for the left part.

if (newHead != pivot) {

// Set the node before the pivot node as NULL

struct Node\* tmp = newHead;

while (tmp->next != pivot)

tmp = tmp->next;

tmp->next = NULL;

// Recur for the list before pivot

newHead = quickSortRecur(newHead, tmp);

// Change next of last node of the left half to

// pivot

tmp = getTail(newHead);

tmp->next = pivot;

}

// Recur for the list after the pivot element

pivot->next = quickSortRecur(pivot->next, newEnd);

return newHead;

}

// The main function for quick sort. This is a wrapper over

// recursive function quickSortRecur()

void quickSort(struct Node\*\* headRef)

{

(\*headRef)

= quickSortRecur(\*headRef, getTail(\*headRef));

return;

}

// Driver code

int main()

{

struct Node\* a = NULL;

push(&a, 5);

push(&a, 20);

push(&a, 4);

push(&a, 3);

push(&a, 30);

cout << "Linked List before sorting \n";

printList(a);

quickSort(&a);

cout << "Linked List after sorting \n";

printList(a);

return 0;

}

**Output**

Linked List before sorting

30 3 4 20 5

Linked List after sorting

3 4 5 20 30

## 140. Find the middle Element of a linked list.

Given a singly linked list of **N** nodes.  
The task is to find the **middle** of the linked list. For example, if the linked list is  
**1-> 2->3->4->5**,then the middle node of the list is **3**.  
If there are two middle nodes(in case, when **N** is even), print the **second middle** element.  
For example, if the linked list given is **1->2->3->4->5->6**, then the middle node of the list is **4**.

**Example 1:**

**Input:**

LinkedList: 1->2->3->4->5

**Output:** 3

**Explanation:**

Middle of linked list is 3.

**Example 2:**

**Input:**

LinkedList: 2->4->6->7->5->1

**Output:** 7

**Explanation:**

Middle of linked list is 7.

**Your Task:**  
The task is to complete the function **getMiddle**() which takes a head reference as the only argument and should return the data at the middle node of the linked list.

**Expected Time Complexity:**O(N).  
**Expected Auxiliary Space:**O(1).

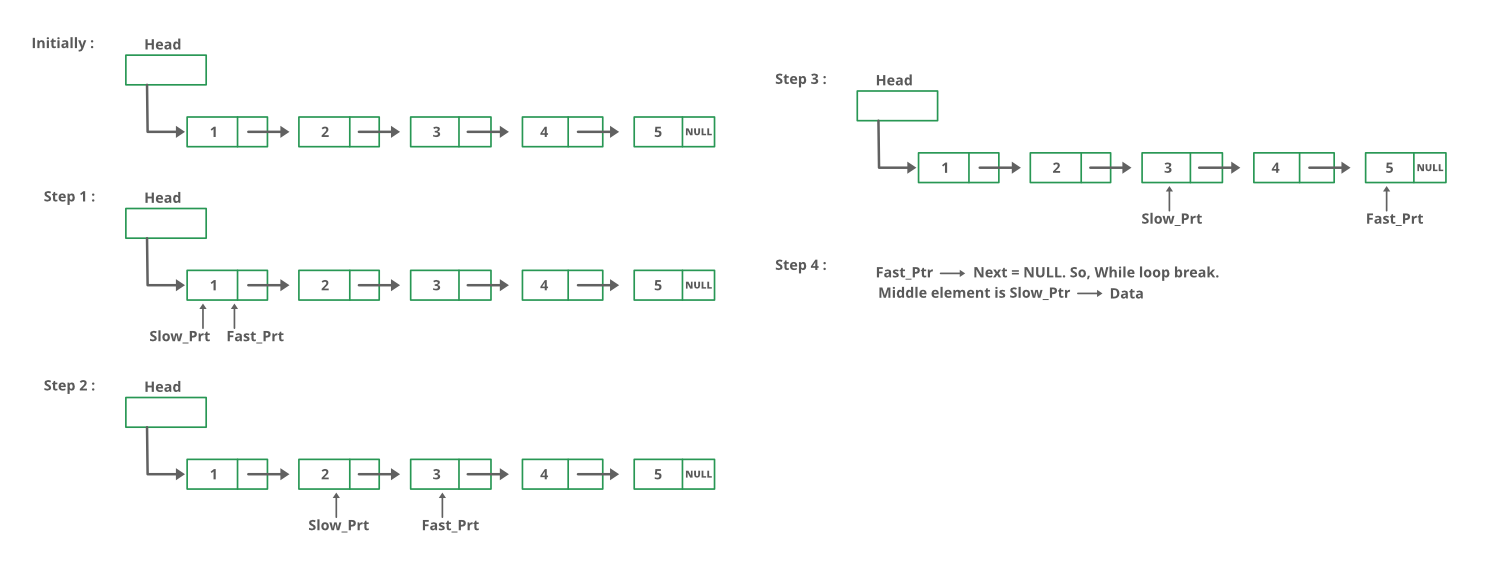
**Constraints:**  
1 <= N <= 5000

### 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 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 :



// C++ program for the above approach

#include <iostream>

using namespace std;

class Node{

public:

int data;

Node \*next;

};

class NodeOperation{

public:

// Function to add a new node

void pushNode(class Node\*\* head\_ref,int data\_val){

// Allocate node

class Node \*new\_node = new Node();

// Put in the data

new\_node->data = data\_val;

// 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 printNode(class Node \*head){

while(head != NULL){

cout <<head->data << "->";

head = head->next;

}

cout << "NULL" << endl;

}

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;

}

}

};

// Driver Code

int main(){

class Node \*head = NULL;

class NodeOperation \*temp = new NodeOperation();

for(int i=5; i>0; i--){

temp->pushNode(&head, i);

temp->printNode(head);

temp->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.

#include <bits/stdc++.h>

using namespace std;

// 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");

}

// Driver code

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]

## 141. Check if a linked list is a circular linked list.

Given **head**, the head of a singly linked list, find if the linked list is circular or not. A linked list is called circular if it not NULL terminated and all nodes are connected in the form of a cycle. An empty linked list is considered as circular.

**Example 1:**

**Input:**

LinkedList: 1->2->3->4->5

(the first and last node is connected,

i.e. 5 --> 1)

**Output:** 1

**Example 2:**

**Input:**

LinkedList: 2->4->6->7->5->1

**Output:** 0

**Your Task:**  
The task is to complete the function isCircular() which checks if the given linked list is circular or not. It should return true or false accordingly. (the driver code prints 1 if the returned values is true, otherwise 0)

**Expected Time Complexity:**O(N).  
**Expected Auxiliary Space:**O(1).

**Constraints:**  
1 <=Number of nodes<= 100

### Solution:

The idea is to store head of the linked list and traverse it. If we reach NULL, linked list is not circular. If reach head again, linked list is circular.

// C++ program to check if linked list is circular

#include<bits/stdc++.h>

using namespace std;

/\* Link list Node \*/

struct Node

{

int data;

struct Node\* next;

};

/\* This function returns true if given linked

list is circular, else false. \*/

bool isCircular(struct Node \*head)

{

// An empty linked list is circular

if (head == NULL)

return true;

// Next of head

struct Node \*node = head->next;

// This loop would stop in both cases (1) If

// Circular (2) Not circular

while (node != NULL && node != head)

node = node->next;

// If loop stopped because of circular

// condition

return (node == head);

}

// 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 function\*/

int main()

{

/\* Start with the empty list \*/

struct Node\* head = newNode(1);

head->next = newNode(2);

head->next->next = newNode(3);

head->next->next->next = newNode(4);

isCircular(head)? cout << "Yes\n" :

cout << "No\n" ;

// Making linked list circular

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

isCircular(head)? cout << "Yes\n" :

cout << "No\n" ;

return 0;

}

**Output :**

No

Yes

**Time Complexity:** O(n)

**Space Complexity:** O(1)

## 142. Split a Circular linked list into two halves.

Given a **C**irular **L**inked **L**ist of size **N,** split it into two halves circular lists. If there are odd number of nodes in the given circular linked list then out of the resulting two halved lists, first list should have one node more than the second list. The resultant lists should also be circular lists and not linear lists.

**Example 1:**

**Input:**

Circular LinkedList: 1->5->7

**Output:**

1 5

7

**Example 2:**

**Input:**

Circular LinkedList: 2->6->1->5

**Output:**

2 6

1 5

**Your Task:**  
Your task is to complete the given function **splitList**(), which takes 3 input parameters: The address of the head of the linked list, addresses of the head of the first and second halved resultant lists and Set the **head1\_ref**and **head2\_ref** to the first resultant list and second resultant list respectively.

**Expected Time Complexity**: O(N)  
**Expected Auxilliary Space**: O(1)

**Constraints:**  
1 <= N <= 100

### Solution:

1) Store the mid and last pointers of the circular linked list using tortoise and hare algorithm.   
2) Make the second half circular.   
3) Make the first half circular.   
4) Set head (or start) pointers of the two linked lists.  
In the below implementation, if there are odd nodes in the given circular linked list then the first result list has 1 more node than the second result list.

// Program to split a circular linked list

// into two halves

#include <bits/stdc++.h>

using namespace std;

/\* structure for a node \*/

class Node

{

public:

int data;

Node \*next;

};

/\* Function to split a list (starting with head)

into two lists. head1\_ref and head2\_ref are

references to head nodes of the two resultant

linked lists \*/

void splitList(Node \*head, Node \*\*head1\_ref,

Node \*\*head2\_ref)

{

Node \*slow\_ptr = head;

Node \*fast\_ptr = head;

if(head == NULL)

return;

/\* If there are odd nodes in the circular list then

fast\_ptr->next becomes head and for even nodes

fast\_ptr->next->next becomes head \*/

while(fast\_ptr->next != head &&

fast\_ptr->next->next != head)

{

fast\_ptr = fast\_ptr->next->next;

slow\_ptr = slow\_ptr->next;

}

/\* If there are even elements in list

then move fast\_ptr \*/

if(fast\_ptr->next->next == head)

fast\_ptr = fast\_ptr->next;

/\* Set the head pointer of first half \*/

\*head1\_ref = head;

/\* Set the head pointer of second half \*/

if(head->next != head)

\*head2\_ref = slow\_ptr->next;

/\* Make second half circular \*/

fast\_ptr->next = slow\_ptr->next;

/\* Make first half circular \*/

slow\_ptr->next = head;

}

/\* UTILITY FUNCTIONS \*/

/\* Function to insert a node at

the beginning of a Circular linked list \*/

void push(Node \*\*head\_ref, int data)

{

Node \*ptr1 = new Node();

Node \*temp = \*head\_ref;

ptr1->data = data;

ptr1->next = \*head\_ref;

/\* If linked list is not NULL then

set the next of last node \*/

if(\*head\_ref != NULL)

{

while(temp->next != \*head\_ref)

temp = temp->next;

temp->next = ptr1;

}

else

ptr1->next = ptr1; /\*For the first node \*/

\*head\_ref = ptr1;

}

/\* Function to print nodes in

a given Circular linked list \*/

void printList(Node \*head)

{

Node \*temp = head;

if(head != NULL)

{

cout << endl;

do {

cout << temp->data << " ";

temp = temp->next;

} while(temp != head);

}

}

// Driver Code

int main()

{

int list\_size, i;

/\* Initialize lists as empty \*/

Node \*head = NULL;

Node \*head1 = NULL;

Node \*head2 = NULL;

/\* Created linked list will be 12->56->2->11 \*/

push(&head, 12);

push(&head, 56);

push(&head, 2);

push(&head, 11);

cout << "Original Circular Linked List";

printList(head);

/\* Split the list \*/

splitList(head, &head1, &head2);

cout << "\nFirst Circular Linked List";

printList(head1);

cout << "\nSecond Circular Linked List";

printList(head2);

return 0;

}

**Output:** 

Original Circular Linked List

11 2 56 12

First Circular Linked List

11 2

Second Circular Linked List

56 12

Time Complexity: **O(n)**

## 143. Write a Program to check whether the Singly Linked list is a palindrome or not.

Given a singly linked list of size **N** of integers. The task is to check if the given linked list is palindrome or not.

**Example 1:**

**Input:**

N = 3

value[] = {1,2,1}

**Output:** 1

**Explanation:** The given linked list is

1 2 1 , which is a palindrome and

Hence, the output is 1.

**Example 2:**

**Input:**

N = 4

value[] = {1,2,3,4}

**Output:** 0

**Explanation:** The given linked list

is 1 2 3 4 , which is not a palindrome

and Hence, the output is 0.

**Your Task:**  
The task is to complete the function **isPalindrome**() which takes head as reference as the only parameter and returns true or false if linked list is palindrome or not respectively.

**Expected Time Complexity**: O(N)  
**Expected Auxialliary Space Usage**: O(1)  (ie, you should not use the recursive stack space as well)

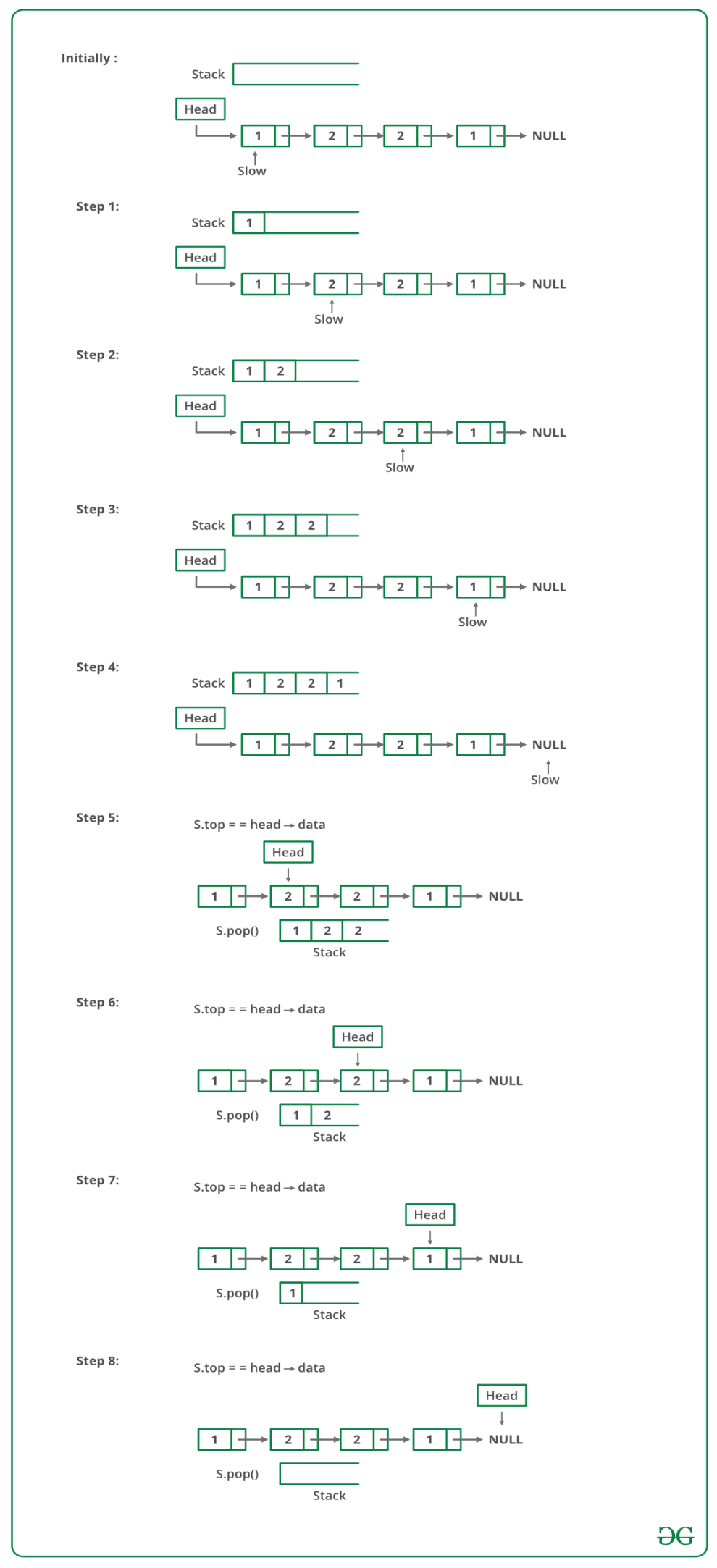
**Constraints:**  
1 <= N <= 105

### Solution:

**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 the stack and compare data of popped node with the 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**

**// 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;**

**}**

**Output**

isPalindrome: true

**Time complexity:** 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 into two halves, method 2 of [this](https://www.geeksforgeeks.org/write-a-c-function-to-print-the-middle-of-the-linked-list/)post is used.

When a 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 the number of nodes is odd. We don’t want the middle node as part of the lists as we are going to compare them for equality. For odd cases, we use a separate variable ‘midnode’.

**// C++ program to check if a linked list is palindrome**

**#include <bits/stdc++.h>**

**using namespace std;**

**// 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;**

**// To handle odd size list**

**struct Node\* midnode = NULL;**

**// initialize result**

**bool res = true;**

**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;**

**// NULL terminate first half**

**prev\_of\_slow\_ptr->next = NULL;**

**// Reverse the second half**

**reverse(&second\_half);**

**// compare**

**res = compareLists(head, second\_half);**

**// 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 return 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)**

**{**

**cout << ptr->data << "->";**

**ptr = ptr->next;**

**}**

**cout << "NULL" << "\n";**

**}**

**// Driver code**

**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;**

**}**

**Output:**

a->NULL

Is 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 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.

// 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;

}

**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).

## 145. Reverse a Doubly Linked list.

Given a doubly linked list of n elements. The task is to **reverse**the doubly linked list.

**Example 1:**

**Input:**

LinkedList: 3 <--> 4 <--> 5

**Output:** 5 4 3

**Example 2:**

**Input:**

LinkedList: 75 <--> 122 <--> 59 <--> 196

**Output:** 196 59 122 75

**Your Task:**  
Your task is to complete the given function **reverseDLL**(), which takes **head**reference as argument and should **reverse**the elements so that the tail becomes the new head and all pointers are correctly pointed. You need to **return**the **new head** of the reversed list. The **printing**and **verification**is done by the **driver**code.

**Expected Time Complexity:**O(n).  
**Expected Auxiliary Space:**O(1).

**Constraints:**  
1 <= number of nodes <= 103  
0 <= value of nodes <= 103

### Solution:

Here is a simple method for reversing a Doubly Linked List. All we need to do is swap prev and next pointers for all nodes, change prev of the head (or start) and change the head pointer in the end.

/\* C++ program to reverse a doubly linked list \*/

#include <bits/stdc++.h>

using namespace std;

/\* a node of the doubly linked list \*/

class Node

{

public:

int data;

Node \*next;

Node \*prev;

};

/\* Function to reverse a Doubly Linked List \*/

void reverse(Node \*\*head\_ref)

{

Node \*temp = NULL;

Node \*current = \*head\_ref;

/\* swap next and prev for all nodes of

doubly linked list \*/

while (current != NULL)

{

temp = current->prev;

current->prev = current->next;

current->next = temp;

current = current->prev;

}

/\* Before changing the head, check for the cases like empty

list and list with only one node \*/

if(temp != NULL )

\*head\_ref = temp->prev;

}

/\* UTILITY FUNCTIONS \*/

/\* Function to insert a node at the

beginning of the Doubly 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;

/\* since we are adding at the beginning,

prev is always NULL \*/

new\_node->prev = NULL;

/\* link the old list off the new node \*/

new\_node->next = (\*head\_ref);

/\* change prev of head node to new node \*/

if((\*head\_ref) != NULL)

(\*head\_ref)->prev = new\_node ;

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

(\*head\_ref) = new\_node;

}

/\* Function to print nodes in a given doubly linked list

This function is same as printList() of singly 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;

/\* Let us create a sorted linked list to test the functions

Created linked list will be 10->8->4->2 \*/

push(&head, 2);

push(&head, 4);

push(&head, 8);

push(&head, 10);

cout << "Original Linked list" << endl;

printList(head);

/\* Reverse doubly linked list \*/

reverse(&head);

cout << "\nReversed Linked list" << endl;

printList(head);

return 0;

}

**Output:**

Original linked list

10 8 4 2

The reversed Linked List is

2 4 8 10

***Time Complexity:****O(N), where N denotes the number of nodes in the doubly linked list.*  
*Auxiliary Space: O(1)*

**Method 2:**

The same question can also be done by using Stacks.

Steps:

1. Keep pushing the node’s data in the stack. -> O(n)
2. The keep popping the elements out and updating the Doubly Linked List

// C++ program to reverse a doubly linked list

#include <bits/stdc++.h>

using namespace std;

struct LinkedList {

struct Node {

int data;

Node \*next, \*prev;

Node(int d)

{

data = d;

next = prev = NULL;

}

};

Node\* head = NULL;

/\* Function to reverse a Doubly Linked List using Stacks

\*/

void reverse()

{

stack<int> st;

Node\* temp = head;

while (temp != NULL) {

st.push(temp->data);

temp = temp->next;

}

// added all the elements sequence wise in the

// st

temp = head;

while (temp != NULL) {

temp->data = st.top();

st.pop();

temp = temp->next;

}

// popped all the elements and the added in the

// linked list,

// which are in the reversed order->

}

/\* UTILITY FUNCTIONS \*/

/\* Function to insert a node at the beginning of the

\* Doubly Linked List \*/

void Push(int new\_data)

{

/\* allocate node \*/

Node\* new\_node = new Node(new\_data);

/\* since we are adding at the beginning,

prev is always NULL \*/

new\_node->prev = NULL;

/\* link the old list off the new node \*/

new\_node->next = head;

/\* change prev of head node to new node \*/

if (head != NULL) {

head->prev = new\_node;

}

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

head = new\_node;

}

/\* Function to print nodes in a given doubly linked list

This function is same as printList() of singly linked

list \*/

void printList(Node\* node)

{

while (node) {

cout << node->data << " ";

node = node->next;

}

}

};

// Driver Code

int main()

{

LinkedList list;

/\* Let us create a sorted linked list to test the

functions Created linked list will be 10->8->4->2

\*/

list.Push(2);

list.Push(4);

list.Push(8);

list.Push(10);

cout << "Original linked list " << endl;

list.printList(list.head);

list.reverse();

cout << endl;

cout << "The reversed Linked List is " << endl;

list.printList(list.head);

}

**Output**

Original linked list

10 8 4 2

The reversed Linked List is

2 4 8 10

***Time Complexity:****O(N)*  
***Auxiliary Space:****O(N)*

In this method, we traverse the linked list once and add elements to the stack, and again traverse the whole for updating all the elements. The whole takes 2n time, which is the time complexity of O(n).

## 146. Find pairs with a given sum in a DLL.

Given a sorted doubly linked list of positive distinct elements, the task is to find pairs in a doubly-linked list whose sum is equal to given value x, without using any extra space?

**Example:**

Input : head : 1 <-> 2 <-> 4 <-> 5 <-> 6 <-> 8 <-> 9

x = 7

Output: (6, 1), (5,2)

The expected time complexity is O(n) and auxiliary space is O(1).

### Solution:

A **simple approach** for this problem is to one by one pick each node and find a second element whose sum is equal to x in the remaining list by traversing in the forward direction. The time complexity for this problem will be O(n^2), n is the total number of nodes in the doubly linked list.

An **efficient solution** for this problem is the same as [this](https://www.geeksforgeeks.org/write-a-c-program-that-given-a-set-a-of-n-numbers-and-another-number-x-determines-whether-or-not-there-exist-two-elements-in-s-whose-sum-is-exactly-x/) article. Here is the algorithm :

* Initialize two pointer variables to find the candidate elements in the sorted doubly linked list. Initialize **first** with the start of the doubly linked list i.e; **first=head** and initialize **second** with the last node of the doubly linked list i.e; **second=last\_node**.
* We initialize **first** and **second** pointers as first and last nodes. Here we don’t have random access, so to find the second pointer, we traverse the list to initialize the second.
* If current sum of **first** and **second** is less than x, then we move **first** in forward direction. If current sum of **first** and **second** element is greater than x, then we move **second** in backward direction.
* Loop termination conditions are also different from arrays. The loop terminates when two pointers cross each other (second->next = first), or they become the same (first == second).
* The case when no pairs are present will be handled by the condition “first==second”

// C++ program to find a pair with given sum x.

#include<bits/stdc++.h>

using namespace std;

// structure of node of doubly linked list

struct Node

{

int data;

struct Node \*next, \*prev;

};

// Function to find pair whose sum equal to given value x.

void pairSum(struct Node \*head, int x)

{

// Set two pointers, first to the beginning of DLL

// and second to the end of DLL.

struct Node \*first = head;

struct Node \*second = head;

while (second->next != NULL)

second = second->next;

// To track if we find a pair or not

bool found = false;

// The loop terminates when two pointers

// cross each other (second->next

// == first), or they become same (first == second)

while (first != second && second->next != first)

{

// pair found

if ((first->data + second->data) == x)

{

found = true;

cout << "(" << first->data<< ", "

<< second->data << ")" << endl;

// move first in forward direction

first = first->next;

// move second in backward direction

second = second->prev;

}

else

{

if ((first->data + second->data) < x)

first = first->next;

else

second = second->prev;

}

}

// if pair is not present

if (found == false)

cout << "No pair found";

}

// A utility function to insert a new node at the

// beginning of doubly linked list

void insert(struct Node \*\*head, int data)

{

struct Node \*temp = new Node;

temp->data = data;

temp->next = temp->prev = NULL;

if (!(\*head))

(\*head) = temp;

else

{

temp->next = \*head;

(\*head)->prev = temp;

(\*head) = temp;

}

}

// Driver program

int main()

{

struct Node \*head = NULL;

insert(&head, 9);

insert(&head, 8);

insert(&head, 6);

insert(&head, 5);

insert(&head, 4);

insert(&head, 2);

insert(&head, 1);

int x = 7;

pairSum(head, x);

return 0;

}

**Output:**

(1,6)

(2,5)

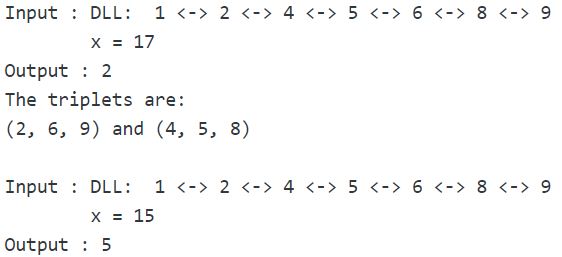
**Time complexity :** O(n)   
**Auxiliary space :** O(1)

If linked list is not sorted, then we can sort the list as a first step. But in that case overall time complexity would become O(n Log n). We can use Hashing in such cases if extra space is not a constraint.

## 147. Count triplets in a sorted DLL whose sum is equal to given value “X”.

Given a sorted doubly linked list of distinct nodes(no two nodes have the same data) and a value **x**. Count triplets in the list that sum up to a given value **x**.

Examples:



### Solution:

**Method 1 (Naive Approach):**   
Using three nested loops generate all triplets and check whether elements in the triplet sum up to **x** or not.

// C++ implementation to count triplets in a sorted doubly linked list

// whose sum is equal to a given value 'x'

#include <bits/stdc++.h>

using namespace std;

// structure of node of doubly linked list

struct Node {

int data;

struct Node\* next, \*prev;

};

// function to count triplets in a sorted doubly linked list

// whose sum is equal to a given value 'x'

int countTriplets(struct Node\* head, int x)

{

struct Node\* ptr1, \*ptr2, \*ptr3;

int count = 0;

// generate all possible triplets

for (ptr1 = head; ptr1 != NULL; ptr1 = ptr1->next)

for (ptr2 = ptr1->next; ptr2 != NULL; ptr2 = ptr2->next)

for (ptr3 = ptr2->next; ptr3 != NULL; ptr3 = ptr3->next)

// if elements in the current triplet sum up to 'x'

if ((ptr1->data + ptr2->data + ptr3->data) == x)

// increment count

count++;

// required count of triplets

return count;

}

// A utility function to insert a new node at the

// beginning of doubly linked list

void insert(struct Node\*\* head, int data)

{

// allocate node

struct Node\* temp = new Node();

// put in the data

temp->data = data;

temp->next = temp->prev = NULL;

if ((\*head) == NULL)

(\*head) = temp;

else {

temp->next = \*head;

(\*head)->prev = temp;

(\*head) = temp;

}

}

// Driver program to test above

int main()

{

// start with an empty doubly linked list

struct Node\* head = NULL;

// insert values in sorted order

insert(&head, 9);

insert(&head, 8);

insert(&head, 6);

insert(&head, 5);

insert(&head, 4);

insert(&head, 2);

insert(&head, 1);

int x = 17;

cout << "Count = "

<< countTriplets(head, x);

return 0;

}

**Output:**

Count = 2

**Time Complexity:** O(n3)   
**Auxiliary Space:** O(1)

**Method 2 (Hashing):**   
Create a hash table with **(key, value)** tuples represented as **(node data, node pointer) tuples**. Traverse the doubly linked list and store each node’s data and its pointer pair(tuple) in the hash table. Now, generate each possible pair of nodes. For each pair of nodes, calculate the **p\_sum**(sum of data in the two nodes) and check whether **(x-p\_sum)** exists in the hash table or not. If it exists, then also verify that the two nodes in the pair are not same to the node associated with **(x-p\_sum)** in the hash table and finally increment **count**. Return **(count / 3)** as each triplet is counted 3 times in the above process.

// C++ implementation to count triplets in a sorted doubly linked list

// whose sum is equal to a given value 'x'

#include <bits/stdc++.h>

using namespace std;

// structure of node of doubly linked list

struct Node {

int data;

struct Node\* next, \*prev;

};

// function to count triplets in a sorted doubly linked list

// whose sum is equal to a given value 'x'

int countTriplets(struct Node\* head, int x)

{

struct Node\* ptr, \*ptr1, \*ptr2;

int count = 0;

// unordered\_map 'um' implemented as hash table

unordered\_map<int, Node\*> um;

// insert the <node data, node pointer> tuple in 'um'

for (ptr = head; ptr != NULL; ptr = ptr->next)

um[ptr->data] = ptr;

// generate all possible pairs

for (ptr1 = head; ptr1 != NULL; ptr1 = ptr1->next)

for (ptr2 = ptr1->next; ptr2 != NULL; ptr2 = ptr2->next) {

// p\_sum - sum of elements in the current pair

int p\_sum = ptr1->data + ptr2->data;

// if 'x-p\_sum' is present in 'um' and either of the two nodes

// are not equal to the 'um[x-p\_sum]' node

if (um.find(x - p\_sum) != um.end() && um[x - p\_sum] != ptr1

&& um[x - p\_sum] != ptr2)

// increment count

count++;

}

// required count of triplets

// division by 3 as each triplet is counted 3 times

return (count / 3);

}

// A utility function to insert a new node at the

// beginning of doubly linked list

void insert(struct Node\*\* head, int data)

{

// allocate node

struct Node\* temp = new Node();

// put in the data

temp->data = data;

temp->next = temp->prev = NULL;

if ((\*head) == NULL)

(\*head) = temp;

else {

temp->next = \*head;

(\*head)->prev = temp;

(\*head) = temp;

}

}

// Driver program to test above

int main()

{

// start with an empty doubly linked list

struct Node\* head = NULL;

// insert values in sorted order

insert(&head, 9);

insert(&head, 8);

insert(&head, 6);

insert(&head, 5);

insert(&head, 4);

insert(&head, 2);

insert(&head, 1);

int x = 17;

cout << "Count = "

<< countTriplets(head, x);

return 0;

}

**Output:**

Count = 2

**Time Complexity:** O(n2)   
**Auxiliary Space:** O(n)

**Method 3 Efficient Approach(Use of two pointers):**   
Traverse the doubly linked list from left to right. For each **current** node during the traversal, initialize two pointers **first** = pointer to the node next to the **current** node and **last** = pointer to the last node of the list. Now, count pairs in the list from **first** to **last** pointer that sum up to value **(x – current node’s data)** (algorithm described in [this](https://www.geeksforgeeks.org/find-pairs-given-sum-doubly-linked-list/) post). Add this count to the **total\_count** of triplets. Pointer to the **last** node can be found only once in the beginning.

// C++ implementation to count triplets in a sorted doubly linked list

// whose sum is equal to a given value 'x'

#include <bits/stdc++.h>

using namespace std;

// structure of node of doubly linked list

struct Node {

int data;

struct Node\* next, \*prev;

};

// function to count pairs whose sum equal to given 'value'

int countPairs(struct Node\* first, struct Node\* second, int value)

{

int count = 0;

// The loop terminates when either of two pointers

// become NULL, or they cross each other (second->next

// == first), or they become same (first == second)

while (first != NULL && second != NULL &&

first != second && second->next != first) {

// pair found

if ((first->data + second->data) == value) {

// increment count

count++;

// move first in forward direction

first = first->next;

// move second in backward direction

second = second->prev;

}

// if sum is greater than 'value'

// move second in backward direction

else if ((first->data + second->data) > value)

second = second->prev;

// else move first in forward direction

else

first = first->next;

}

// required count of pairs

return count;

}

// function to count triplets in a sorted doubly linked list

// whose sum is equal to a given value 'x'

int countTriplets(struct Node\* head, int x)

{

// if list is empty

if (head == NULL)

return 0;

struct Node\* current, \*first, \*last;

int count = 0;

// get pointer to the last node of

// the doubly linked list

last = head;

while (last->next != NULL)

last = last->next;

// traversing the doubly linked list

for (current = head; current != NULL; current = current->next) {

// for each current node

first = current->next;

// count pairs with sum(x - current->data) in the range

// first to last and add it to the 'count' of triplets

count += countPairs(first, last, x - current->data);

}

// required count of triplets

return count;

}

// A utility function to insert a new node at the

// beginning of doubly linked list

void insert(struct Node\*\* head, int data)

{

// allocate node

struct Node\* temp = new Node();

// put in the data

temp->data = data;

temp->next = temp->prev = NULL;

if ((\*head) == NULL)

(\*head) = temp;

else {

temp->next = \*head;

(\*head)->prev = temp;

(\*head) = temp;

}

}

// Driver program to test above

int main()

{

// start with an empty doubly linked list

struct Node\* head = NULL;

// insert values in sorted order

insert(&head, 9);

insert(&head, 8);

insert(&head, 6);

insert(&head, 5);

insert(&head, 4);

insert(&head, 2);

insert(&head, 1);

int x = 17;

cout << "Count = "

<< countTriplets(head, x);

return 0;

}

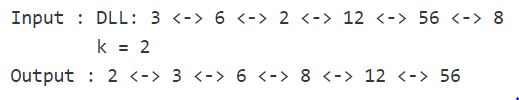
**Output:**

Count = 2

**Time Complexity:** O(n2)   
**Auxiliary Space:** O(1)

## 148. Sort a “k”sorted Doubly Linked list.[Very IMP]

Given a doubly linked list containing **n** nodes, where each node is at most **k** away from its target position in the list. The problem is to sort the given doubly linked list.   
For example, let us consider **k** is 2, a node at position 7 in the sorted doubly linked list, can be at positions 5, 6, 7, 8, 9 in the given doubly linked list.  
Examples:



### Solution:

**Naive Approach:** Sort the given doubly linked list using the insertion sort technique. While inserting each element in the sorted part of the list, there will be at most k swaps to place the element to its correct position since it is at most k steps away from its correct position.

// C++ implementation to sort a k sorted doubly

// linked list

#include<bits/stdc++.h>

using namespace std;

// a node of the doubly linked list

struct Node {

int data;

struct Node\* next;

struct Node\* prev;

};

// function to sort a k sorted doubly linked list

struct Node\* sortAKSortedDLL(struct Node\* head, int k)

{

if(head == NULL || head->next == NULL)

return head;

// perform on all the nodes in list

for(Node \*i = head->next; i != NULL; i = i->next) {

Node \*j = i;

// There will be atmost k swaps for each element in the list

// since each node is k steps away from its correct position

while(j->prev != NULL && j->data < j->prev->data) {

// swap j and j.prev node

Node\* temp = j->prev->prev;

Node\* temp2 = j->prev;

Node \*temp3 = j->next;

j->prev->next = temp3;

j->prev->prev = j;

j->prev = temp;

j->next = temp2;

if(temp != NULL)

temp->next = j;

if(temp3 != NULL)

temp3->prev = temp2;

}

// if j is now the new head

// then reset head

if(j->prev == NULL)

head = j;

}

return head;

}

// Function to insert a node at the beginning

// of the Doubly Linked List

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;

// since we are adding at the beginning,

// prev is always NULL

new\_node->prev = NULL;

// link the old list off the new node

new\_node->next = (\*head\_ref);

// change prev of head node to new node

if ((\*head\_ref) != NULL)

(\*head\_ref)->prev = new\_node;

// move the head to point to the new node

(\*head\_ref) = new\_node;

}

// Function to print nodes in a given doubly linked list

void printList(struct Node\* head)

{

// if list is empty

if (head == NULL)

cout << "Doubly Linked list empty";

while (head != NULL) {

cout << head->data << " ";

head = head->next;

}

}

// Driver program to test above

int main()

{

struct Node\* head = NULL;

// Create the doubly linked list:

// 3<->6<->2<->12<->56<->8

push(&head, 8);

push(&head, 56);

push(&head, 12);

push(&head, 2);

push(&head, 6);

push(&head, 3);

int k = 2;

cout << "Original Doubly linked list:\n";

printList(head);

// sort the biotonic DLL

head = sortAKSortedDLL(head, k);

cout << "\nDoubly linked list after sorting:\n";

printList(head);

return 0;

}

**Output**

Original Doubly linked list:

3 6 2 12 56 8

Doubly Linked List after sorting:

2 3 6 8 12 56

**Time Complexity:** O(n\*k)

**Auxiliary Space:** O(1)

**Efficient Approach:** We can sort the list using the MIN HEAP data structure.

1) Create a Min Heap of size k+1 with first k+1 elements. This will take O(k) time   
2) One by one remove min element from heap, put it in result array, and add a new element to heap from remaining elements.  
Removing an element and adding a new element to min heap will take log k time. So overall complexity will be O(k) + O((n-k) \* log(k)).

// C++ implementation to sort a k sorted doubly

// linked list

#include <bits/stdc++.h>

using namespace std;

// a node of the doubly linked list

struct Node {

int data;

struct Node\* next;

struct Node\* prev;

};

// 'compare' function used to build up the

// priority queue

struct compare {

bool operator()(struct Node\* p1, struct Node\* p2)

{

return p1->data > p2->data;

}

};

// function to sort a k sorted doubly linked list

struct Node\* sortAKSortedDLL(struct Node\* head, int k)

{

// if list is empty

if (head == NULL)

return head;

// priority\_queue 'pq' implemented as min heap with the

// help of 'compare' function

priority\_queue<Node\*, vector<Node\*>, compare> pq;

struct Node\* newHead = NULL, \*last;

// Create a Min Heap of first (k+1) elements from

// input doubly linked list

for (int i = 0; head != NULL && i <= k; i++) {

// push the node on to 'pq'

pq.push(head);

// move to the next node

head = head->next;

}

// loop till there are elements in 'pq'

while (!pq.empty()) {

// place root or top of 'pq' at the end of the

// result sorted list so far having the first node

// pointed to by 'newHead'

// and adjust the required links

if (newHead == NULL) {

newHead = pq.top();

newHead->prev = NULL;

// 'last' points to the last node

// of the result sorted list so far

last = newHead;

}

else {

last->next = pq.top();

pq.top()->prev = last;

last = pq.top();

}

// remove element from 'pq'

pq.pop();

// if there are more nodes left in the input list

if (head != NULL) {

// push the node on to 'pq'

pq.push(head);

// move to the next node

head = head->next;

}

}

// making 'next' of last node point to NULL

last->next = NULL;

// new head of the required sorted DLL

return newHead;

}

// Function to insert a node at the beginning

// of the Doubly Linked List

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;

// since we are adding at the beginning,

// prev is always NULL

new\_node->prev = NULL;

// link the old list off the new node

new\_node->next = (\*head\_ref);

// change prev of head node to new node

if ((\*head\_ref) != NULL)

(\*head\_ref)->prev = new\_node;

// move the head to point to the new node

(\*head\_ref) = new\_node;

}

// Function to print nodes in a given doubly linked list

void printList(struct Node\* head)

{

// if list is empty

if (head == NULL)

cout << "Doubly Linked list empty";

while (head != NULL) {

cout << head->data << " ";

head = head->next;

}

}

// Driver program to test above

int main()

{

struct Node\* head = NULL;

// Create the doubly linked list:

// 3<->6<->2<->12<->56<->8

push(&head, 8);

push(&head, 56);

push(&head, 12);

push(&head, 2);

push(&head, 6);

push(&head, 3);

int k = 2;

cout << "Original Doubly linked list:\n";

printList(head);

// sort the biotonic DLL

head = sortAKSortedDLL(head, k);

cout << "\nDoubly linked list after sorting:\n";

printList(head);

return 0;

}

**Output**

Original Doubly linked list:

3 6 2 12 56 8

Doubly linked list after sorting:

2 3 6 8 12 56

**Time Complexity:** O(n\*log k)  
**Auxiliary Space:**O(k)

**Approach:**[Shell sort](https://www.geeksforgeeks.org/shellsort/), which is a variation of Insertion sort can be used to solve this problem as well, by initializing the gap with **K** instead of **N,**as the list is already **K-sorted.**

Below is an implementation of the above approach:

// Java implementation to sort a k sorted doubly

import java.util.\*;

class DoublyLinkedList {

static Node head;

static class Node {

int data;

Node next, prev;

Node(int d)

{

data = d;

next = prev = null;

}

}

// this method returns

// the ceiling value of gap/2

int nextGap(double gap)

{

if (gap < 2)

return 0;

return (int)Math.ceil(gap / 2);

}

// Sort a k sorted Doubly Linked List

// Time Complexity: O(n\*log k)

// Space Complexity: O(1)

Node sortAKSortedDLL(Node head, int k)

{

if (head == null || head.next == null)

return head;

// for all gaps

for (int gap = k; gap > 0; gap = nextGap(gap)) {

Node i = head, j = head;

int count = gap;

while (count-- > 0)

j = j.next;

for (; j != null; i = i.next, j = j.next) {

// if data at jth node is less than

// data at ith node

if (i.data > j.data) {

// if i is head

// then replace head with j

if (i == head)

head = j;

// swap i & j pointers

Node iTemp = i;

i = j;

j = iTemp;

// i & j pointers are swapped because

// below code only swaps nodes by

// swapping their associated

// pointers(i.e. prev & next pointers)

// Now, swap both the

// nodes in linked list

Node iPrev = i.prev, iNext = i.next;

if (iPrev != null)

iPrev.next = j;

if (iNext != null)

iNext.prev = j;

i.prev = j.prev;

i.next = j.next;

if (j.prev != null)

j.prev.next = i;

if (j.next != null)

j.next.prev = i;

j.prev = iPrev;

j.next = iNext;

}

}

}

return head;

}

/\* UTILITY FUNCTIONS \*/

// Function to insert a node

// at the beginning of the

// Doubly Linked List

void push(int new\_data)

{

// allocate node

Node new\_node = new Node(new\_data);

// since we are adding at the beginning,

// prev is always NULL

new\_node.prev = null;

// link the old list off the new node

new\_node.next = head;

// change prev of head node to new node

if (head != null) {

head.prev = new\_node;

}

// move the head to point

// to the new node

head = new\_node;

}

// Function to print nodes

// in a given doubly linked list

// This function is same as

// printList() of singly linked list

void printList(Node node)

{

while (node != null) {

System.out.print(node.data + " ");

node = node.next;

}

}

// Driver code

public static void main(String[] args)

{

DoublyLinkedList list = new DoublyLinkedList();

// 3<->6<->2<->12<->56<->8

list.push(8);

list.push(56);

list.push(12);

list.push(2);

list.push(6);

list.push(3);

int k = 2;

System.out.println("Original Doubly linked list:");

list.printList(head);

Node sortedDLL = list.sortAKSortedDLL(head, k);

System.out.println("");

System.out.println(

"Doubly Linked List after sorting:");

list.printList(sortedDLL);

}

}

**Output:**

Original Doubly linked list:

3 6 2 12 56 8

Doubly Linked List after sorting:

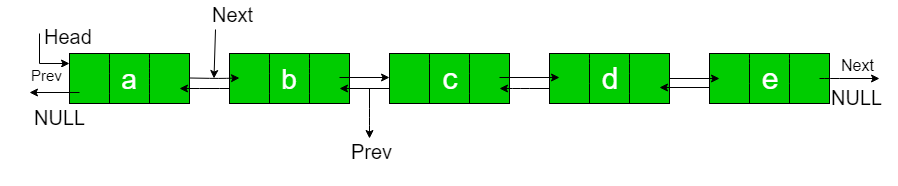
2 3 6 8 12 56

**Time Complexity:** O(N\*log K)  
gap is initialized with k and reduced to the ceiling value of gap/2 after each iteration. Hence, log k gaps will be calculated, and for each gap, the list will be iterated.

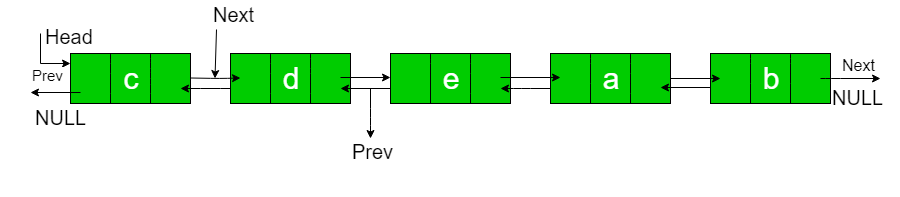
**Space Complexity:** O(1)

## 149. Rotate DoublyLinked list by k nodes.

Given a doubly-linked list, rotate the linked list counter-clockwise by N nodes. Here N is a given positive integer and is smaller than the count of nodes in linked list.



N = 2  
Rotated List:



**Examples:**

Input : **a b c d e**  N = 2

Output : **c d e a b**

Input : **a b c d e f g h**  N = 4

Output : **e f g h a b c d**

### Solution:

1. To rotate the Doubly linked list, first, we need to traverse through the linked list and find the address of the last node.

2. Then make it a circular linked list.

3. Then move the head as well as a temp by n nodes.

4. Then make the linked list as un-circular.

**Solution 1:**

#include<iostream>

using namespace std;

class Node

{

public:

char data;

Node\* next;

Node\* pre;

Node(int data)

{

this->data=data;

pre=NULL;

next=NULL;

}

};

void insertAtHead(Node\* &head, int data)

{

Node\* n = new Node(data);

if(head==NULL)

{

head=n;

return;

}

n->next=head;

head->pre=n;

head=n;

return;

}

void insertAtTail(Node\* &head, int data)

{

if(head==NULL)

{

insertAtHead(head,data);

return;

}

Node\* temp=head;

while(temp->next!=NULL)

{

temp=temp->next;

}

Node\* n=new Node(data);

temp->next=n;

n->pre=temp;

return;

}

void display(Node\* head)

{

while(head!=NULL)

{

cout << head->data << "-->";

head=head->next;

}

cout << "NULL\n";

}

void rotateByN(Node\* &head, int pos)

{

// return without any changes if position is 0.

if(pos==0) return;

// Finding last node.

Node\* temp=head;

while(temp->next!=NULL)

{

temp=temp->next;

}

// making the list circular.

temp->next=head;

head->pre=temp;

// move head and temp by the given position.

int count=1;

while(count<=pos)

{

head=head->next;

temp=temp->next;

count++;

}

// now again make list un-circular.

temp->next=NULL;

head->pre=NULL;

}

int main()

{

Node\* head=NULL;

insertAtTail(head,'a');

insertAtTail(head,'b');

insertAtTail(head,'c');

insertAtTail(head,'d');

insertAtTail(head,'e');

int n=2;

cout << "\nBefore Rotation : \n";

display(head);

rotateByN(head,n);

cout << "\nAfter Rotation : \n";

display(head);

cout << "\n\n";

return 0;

}

Output:

Before Rotation :

a-->b-->c-->d-->e-->NULL

After Rotation :

c-->d-->e-->a-->b-->NULL

Time Complexity: O(N)

Space Complexity: O(1)

**Solution 2:**

#include<iostream>

using namespace std;

class Node

{

public:

char data;

Node\* next;

Node\* pre;

Node(int data)

{

this->data=data;

pre=NULL;

next=NULL;

}

};

void insertAtHead(Node\* &head, int data)

{

Node\* n = new Node(data);

if(head==NULL)

{

head=n;

return;

}

n->next=head;

head->pre=n;

head=n;

return;

}

void insertAtTail(Node\* &head, int data)

{

if(head==NULL)

{

insertAtHead(head,data);

return;

}

Node\* temp=head;

while(temp->next!=NULL)

{

temp=temp->next;

}

Node\* n=new Node(data);

temp->next=n;

n->pre=temp;

return;

}

void display(Node\* head)

{

while(head!=NULL)

{

cout << head->data << "-->";

head=head->next;

}

cout << "NULL\n";

}

void rotateByN(Node \*&head, int pos)

{

if (pos == 0)

return;

Node \*curr = head;

while (pos)

{

curr = curr->next;

pos--;

}

Node \*tail = curr->pre;

Node \*NewHead = curr;

tail->next = NULL;

curr->pre = NULL;

while (curr->next != NULL)

{

curr = curr->next;

}

curr->next = head;

head->pre = curr;

head = NewHead;

}

int main()

{

Node\* head=NULL;

insertAtTail(head,'a');

insertAtTail(head,'b');

insertAtTail(head,'c');

insertAtTail(head,'d');

insertAtTail(head,'e');

int n=2;

cout << "\nBefore Rotation : \n";

display(head);

rotateByN(head,n);

cout << "\nAfter Rotation : \n";

display(head);

cout << "\n\n";

return 0;

}

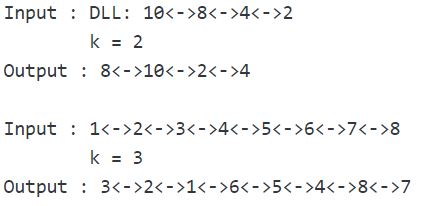
Time Complexity: O(N)

Space Complexity: O(1)

## 150. Reverse a Doubly Linked list in group of Given Size.[Very IMP]

Given a doubly linked list containing **n** nodes. The problem is to reverse every group of **k** nodes in the list.

**Examples:**



### Solution:

**Approach:** Create a recursive function say **reverse(head, k)**. This function receives the head or the first node of each group of **k** nodes. It reverses those group of **k** nodes by applying the approach discussed in [Reverse a doubly linked list | Set-2.](https://www.geeksforgeeks.org/reverse-doubly-linked-list-set-2/) After reversing the group of **k** nodes the function checks whether next group of nodes exists in the list or not. If group exists then it makes a recursive call to itself with the first node of the next group and makes the necessary adjustments with the next and previous links of that group. Finally it returns the new head node of the reversed group.

// C++ implementation to reverse a doubly linked list

// in groups of given size

#include <bits/stdc++.h>

using namespace std;

// a node of the doubly linked list

struct Node {

int data;

Node \*next, \*prev;

};

// function to get a new node

Node\* getNode(int data)

{

// allocate space

Node\* new\_node = (Node\*)malloc(sizeof(Node));

// put in the data

new\_node->data = data;

new\_node->next = new\_node->prev = NULL;

return new\_node;

}

// function to insert a node at the beginging

// of the Doubly Linked List

void push(Node\*\* head\_ref, Node\* new\_node)

{

// since we are adding at the beginning,

// prev is always NULL

new\_node->prev = NULL;

// link the old list off the new node

new\_node->next = (\*head\_ref);

// change prev of head node to new node

if ((\*head\_ref) != NULL)

(\*head\_ref)->prev = new\_node;

// move the head to point to the new node

(\*head\_ref) = new\_node;

}

// function to reverse a doubly linked list

// in groups of given size

Node\* revListInGroupOfGivenSize(Node\* head, int k)

{

Node \*current = head;

Node\* next = NULL;

Node\* newHead = NULL;

int count = 0;

// reversing the current group of k

// or less than k nodes by adding

// them at the beginning of list

// 'newHead'

while (current != NULL && count < k)

{

next = current->next;

push(&newHead, current);

current = next;

count++;

}

// if next group exists then making the desired

// adjustments in the link

if (next != NULL)

{

head->next = revListInGroupOfGivenSize(next, k);

head->next->prev = head;

}

// pointer to the new head of the

// reversed group

return newHead;

}

// Function to print nodes in a

// given doubly linked list

void printList(Node\* head)

{

while (head != NULL) {

cout << head->data << " ";

head = head->next;

}

}

// Driver program to test above

int main()

{

// Start with the empty list

Node\* head = NULL;

// Create doubly linked: 10<->8<->4<->2

push(&head, getNode(2));

push(&head, getNode(4));

push(&head, getNode(8));

push(&head, getNode(10));

int k = 2;

cout << "Original list: ";

printList(head);

// Reverse doubly linked list in groups of

// size 'k'

head = revListInGroupOfGivenSize(head, k);

cout << "\nModified list: ";

printList(head);

return 0;

}

**Output**

Original list: 10 8 4 2

Modified list: 8 10 2 4

**Time Complexity:**O(n).

We can further simplify the implementation of this algorithm using the same idea with recursion in just one function.

#include <iostream>

using namespace std;

struct Node {

int data;

Node \*next, \*prev;

};

// function to add Node at the end of a Doubly LinkedList

Node\* insertAtEnd(Node\* head, int data)

{

Node\* new\_node = new Node();

new\_node->data = data;

new\_node->next = NULL;

Node\* temp = head;

if (head == NULL) {

new\_node->prev = NULL;

head = new\_node;

return head;

}

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = new\_node;

new\_node->prev = temp;

return head;

}

// function to print Doubly LinkedList

void printDLL(Node\* head)

{

while (head != NULL) {

cout << head->data << " ";

head = head->next;

}

cout << endl;

}

// function to Reverse a doubly linked list

// in groups of given size

Node\* reverseByN(Node\* head, int k)

{

if (!head)

return NULL;

head->prev = NULL;

Node \*temp, \*curr = head, \*newHead;

int count = 0;

while (curr != NULL && count < k) {

newHead = curr;

temp = curr->prev;

curr->prev = curr->next;

curr->next = temp;

curr = curr->prev;

count++;

}

// checking if the reversed LinkedList size is

// equal to K or not

// if it is not equal to k that means we have reversed

// the last set of size K and we don't need to call the

// recursive function

if (count >= k) {

Node\* rest = reverseByN(curr, k);

head->next = rest;

if (rest != NULL)

// it is required for prev link otherwise u wont

// be backtrack list due to broken links

rest->prev = head;

}

return newHead;

}

int main()

{

Node\* head;

for (int i = 1; i <= 10; i++) {

head = insertAtEnd(head, i);

}

printDLL(head);

int n = 4;

head = reverseByN(head, n);

printDLL(head);

}

**Output**

1 2 3 4 5 6 7 8 9 10

4 3 2 1 8 7 6 5 10 9

## 151. Can we reverse a linked list in less than O(n) ?

It doesn’t look possible to reverse a[simple singly linked list](https://www.geeksforgeeks.org/linked-list-set-1-introduction/) in less than O(n). A [simple singly linked list can only be reversed in O(n) time using recursive and iterative method](https://www.geeksforgeeks.org/write-a-function-to-reverse-the-nodes-of-a-linked-list/)s.

A memory-efficient[doubly linked list](https://www.geeksforgeeks.org/xor-linked-list-a-memory-efficient-doubly-linked-list-set-1/) with head and tail pointers can also be reversed in O(1) time by swapping head and tail pointers.

A [doubly linked list](https://www.geeksforgeeks.org/doubly-linked-list/) with head and tail pointers can also be reversed in O(1) time by swapping head and tail pointers, but we would have to traverse the list in forward direction using prev pointer and reverse direction using next pointer which may not be considered valid.

## 152. Why Quicksort is preferred for. Arrays and Merge Sort for LinkedLists ?

**Why is**[Quick Sort](http://geeksquiz.com/quick-sort/)**preferred for arrays?**

* Quick Sort in its general form is an in-place sort (i.e. it doesn’t require any extra storage) whereas merge sort requires O(N) extra storage, N denoting the array size which may be quite expensive. Allocating and de-allocating the extra space used for merge sort increases the running time of the algorithm.
* Comparing average complexity we find that both type of sorts have O(NlogN) average complexity but the constants differ. For arrays, merge sort loses due to the use of extra O(N) storage space.
* Most practical implementations of Quick Sort use randomized version. The randomized version has expected time complexity of O(nLogn). The worst case is possible in randomized version also, but worst case doesn’t occur for a particular pattern (like sorted array) and randomized Quick Sort works well in practice.
* Quick Sort is also a cache friendly sorting algorithm as it has good [locality of reference](http://en.wikipedia.org/wiki/Locality_of_reference) when used for arrays.
* Quick Sort is also [tail recursive](https://www.geeksforgeeks.org/tail-recursion/), therefore tail call optimizations is done.

**Why is**[Merge Sort](http://geeksquiz.com/merge-sort/)**preferred for Linked Lists?**

* In case of [linked lists](http://geeksquiz.com/linked-list-set-1-introduction/) the case is different mainly due to difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory.
* Unlike array, in [linked list](http://geeksquiz.com/linked-list-set-1-introduction/), we can insert items in the middle in O(1) extra space and O(1) time if we are given reference/pointer to the previous node. Therefore merge operation of merge sort can be implemented without extra space for linked lists.
* In arrays, we can do random access as elements are continuous in memory. Let us say we have an integer (4-byte) array A and let the address of A[0] be x then to access A[i], we can directly access the memory at (x + i\*4). Unlike arrays, we can not do random access in linked list.
* Quick Sort requires a lot of this kind of access. In linked list to access i’th index, we have to travel each and every node from the head to i’th node as we don’t have continuous block of memory. Therefore, the overhead increases for quick sort. Merge sort accesses data sequentially and the need of random access is low.

## 153. Flatten a Linked List

Given a Linked List of size N, where every node represents a sub-linked-list and contains two pointers:  
(i) a**next**pointer to the next node,  
(ii) a**bottom** pointer to a linked list where this node is head.  
Each of the sub-linked-list is in sorted order.  
Flatten the Link List such that all the nodes appear in a single level while maintaining the sorted order.   
**Note:** The flattened list will be printed using the bottom pointer instead of next pointer.

**Example 1:**

**Input:**

5 -> 10 -> 19 -> 28

| | | |

7 20 22 35

| | |

8 50 40

| |

30 45

**Output:**  5-> 7-> 8- > 10 -> 19-> 20->

22-> 28-> 30-> 35-> 40-> 45-> 50.

**Explanation**:

The resultant linked lists has every

node in a single level.

(**Note:** | represents the bottom pointer.)

**Example 2:**

**Input:**

5 -> 10 -> 19 -> 28

| |

7 22

| |

8 50

|

30

**Output:** 5->7->8->10->19->20->22->30->50

**Explanation:**

The resultant linked lists has every

node in a single level.

(**Note:** | represents the bottom pointer.)

**Your Task:**  
You do not need to read input or print anything. Complete the function **flatten()** that takes the **head**of the linked list as input parameterand returns the head of flattened link list.

**Expected Time Complexity:** O(N\*M)  
**Expected Auxiliary Space:** O(1)

**Constraints:**  
0 <= N <= 50  
1 <=**Mi**<= 20  
1 <= Element of linked list <= 103

### Solution:

The idea is to use the Merge() process of [merge sort for linked lists.](https://www.geeksforgeeks.org/merge-sort-for-linked-list/) We use merge() to merge lists one by one. We recursively merge() the current list with the already flattened list.   
The down pointer is used to link nodes of the flattened list.

Below is the implementation of the above approach:

// C++ program for flattening a Linked List

#include <bits/stdc++.h>

using namespace std;

// Link list node

class Node

{

public:

int data;

Node \*right, \*down;

};

Node\* head = NULL;

// An utility function to merge two sorted

// linked lists

Node\* merge(Node\* a, Node\* b)

{

// If first linked list is empty then second

// is the answer

if (a == NULL)

return b;

// If second linked list is empty then first

// is the result

if (b == NULL)

return a;

// Compare the data members of the two linked

// lists and put the larger one in the result

Node\* result;

if (a->data < b->data)

{

result = a;

result->down = merge(a->down, b);

}

else

{

result = b;

result->down = merge(a, b->down);

}

result->right = NULL;

return result;

}

Node\* flatten(Node\* root)

{

// Base Cases

if (root == NULL || root->right == NULL)

return root;

// Recur for list on right

root->right = flatten(root->right);

// Now merge

root = merge(root, root->right);

// Return the root

// it will be in turn merged with its left

return root;

}

// Utility function to insert a node at

// beginning of the linked list

Node\* push(Node\* head\_ref, int data)

{

// Allocate the Node & Put in the data

Node\* new\_node = new Node();

new\_node->data = data;

new\_node->right = NULL;

// Make next of new Node as head

new\_node->down = head\_ref;

// Move the head to point to new Node

head\_ref = new\_node;

return head\_ref;

}

void printList()

{

Node\* temp = head;

while (temp != NULL)

{

cout << temp->data << " ";

temp = temp->down;

}

cout << endl;

}

// Driver code

int main()

{

/\* Let us create the following linked list

5 -> 10 -> 19 -> 28

| | | |

V V V V

7 20 22 35

| | |

V V V

8 50 40

| |

V V

30 45

\*/

head = push(head, 30);

head = push(head, 8);

head = push(head, 7);

head = push(head, 5);

head->right = push(head->right, 20);

head->right = push(head->right, 10);

head->right->right = push(head->right->right, 50);

head->right->right = push(head->right->right, 22);

head->right->right = push(head->right->right, 19);

head->right->right->right = push(head->right->right->right, 45);

head->right->right->right = push(head->right->right->right, 40);

head->right->right->right = push(head->right->right->right, 35);

head->right->right->right = push(head->right->right->right, 20);

// Flatten the list

head = flatten(head);

printList();

return 0;

}

**Output:**

5 7 8 10 19 20 20 22 30 35 40 45 50

## 154. Sort a LL of 0's, 1's and 2's

Given a linked list of **N** nodes where nodes can contain values **0s**, **1s,** and **2s**only. The task is to segregate **0s**, **1s,** and **2s** linked list such that all zeros segregate to head side, 2s at the end of the linked list, and 1s in the mid of 0s and 2s.

**Example 1:**

**Input:**

N = 8

value[] = {1,2,2,1,2,0,2,2}

**Output:** 0 1 1 2 2 2 2 2

**Explanation:** All the 0s are segregated

to the left end of the linked list,

2s to the right end of the list, and

1s in between.

**Example 2:**

**Input:**

N = 4

value[] = {2,2,0,1}

**Output:** 0 1 2 2

**Explanation:** After arranging all the

0s,1s and 2s in the given format,

the output will be 0 1 2 2.

**Your Task:**  
The task is to complete the function **segregate**() which segregates the nodes in the linked list as asked in the problem statement and returns the head of the modified linked list. The **printing**is done **automatically**by the **driver code**.

**Expected Time Complexity:**O(N).  
**Expected Auxiliary Space:**O(1).

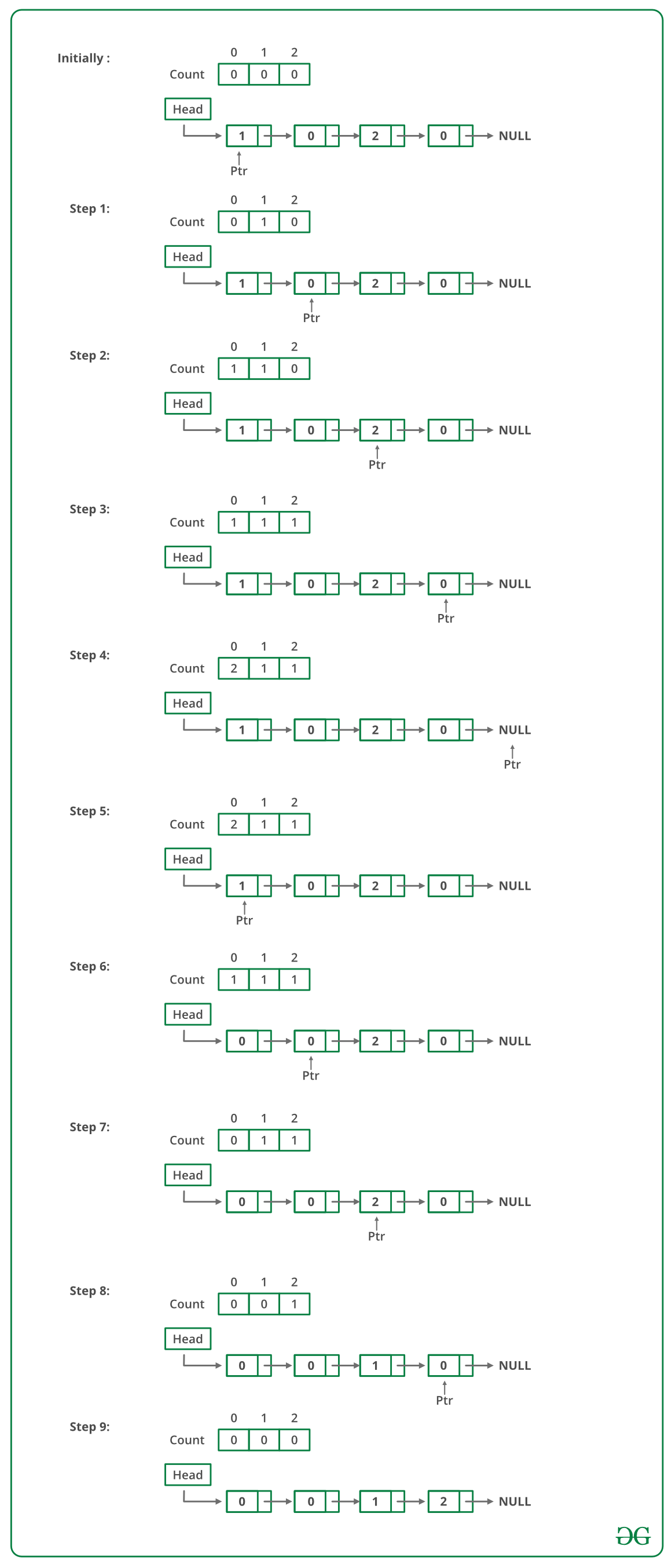
**Constraints:**  
1 <= N <= 103

### 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;

}

**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 the number of nodes in the linked list.   
**Auxiliary Space:** O(1)

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

**Method 2:** In this post, a new solution is discussed that works by changing links.  
**Approach:** 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 <bits/stdc++.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

**Complexity Analysis:**

* **Time Complexity:** O(n) where n is a number of nodes in linked list.   
  Only one traversal of the linked list is needed.
* **Auxiliary Space:**O(1).   
  As no extra space is required.

**My Implementation:**

class Solution

{

public:

//Function to sort a linked list of 0s, 1s and 2s.

Node\* segregate(Node \*head) {

Node \*end = head, \*curr = head, \*prev = NULL;

while(end->next)

end = end->next;

Node \*last = end;

while(curr!=end->next){

if(curr->data==0){

if(head!=curr){

prev->next = curr->next;

curr->next = head;

head = curr;

if(curr==end)

break;

curr = prev->next;

}

else{

prev = curr;

if(curr==end)

break;

curr = curr->next;

}

}

else if(curr->data==2){

if(head!=curr){

prev->next = curr->next;

last->next = curr;

curr->next = NULL;

last = curr;

if(curr==end)

break;

curr = prev->next;

}

else{

last->next = curr;

head = head->next;

curr->next = NULL;

last = curr;

if(curr==end)

break;

curr = head;

}

}

else{

prev = curr;

if(curr==end)

break;

curr = curr->next;

}

}

return head;

}

};

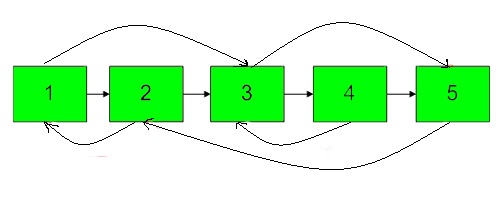
## 155. Clone a linked list with next and random pointer

You are given a special linked list with **N**nodes where each node has a next pointer pointing to its next node. You are also given **M** random pointers, where you will be given **M**number of pairs denoting two nodes **a** and **b**  **i.e. a->arb = b.**

Construct a copy of the given list. The copy should consist of exactly **N** new nodes, where each new node has its value set to the value of its corresponding original node. Both the next and random pointer of the new nodes should point to new nodes in the copied list such that the pointers in the original list and copied list represent the same list state. None of the pointers in the new list should point to nodes in the original list.

For example, if there are two nodes **X** and **Y** in the original list, where **X.random** **-->** **Y**, then for the corresponding two nodes **x** and **y** in the copied list, **x.random --> y.**

Return the head of the copied linked list.



**Example 1:**

**Input:**

N = 4, M = 2

value = {1,2,3,4}

pairs = {{1,2},{2,4}}

**Output:** 1

**Explanation:** In this test case, there

are 4 nodes in linked list.  Among these

4 nodes,  2 nodes have arbitrary pointer

set, rest two nodes have arbitrary pointer

as NULL. Second line tells us the value

of four nodes. The third line gives the

information about arbitrary pointers.

The first node arbitrary pointer is set to

node 2.  The second node arbitrary pointer

is set to node 4.

**Example 2:**

**Input:**

N = 4, M = 2

value[] = {1,3,5,9}

pairs[] = {{1,1},{3,4}}

**Output:** 1

**Explanation:** In the given testcase ,

applying the method as stated in the

above example, the output will be 1.

**Your Task:**  
The task is to complete the function **copyList**() which takes one argument the head of the linked list to be cloned and should **return** the head of the cloned linked list.

**NOTE :**  
1. If there is any node whose arbitrary pointer is not given then it's by default NULL.   
2. Your solution return an output **1** if your clone linked list is correct, else it returns **0**.

**Expected Time Complexity** : O(n)  
**Expected Auxilliary Space**: O(1)

**Constraints:**  
1 <= N <= 100  
1 <= M <= N  
1 <= a, b <= 100

### 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();

}

**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)

**My code using same approach:**

Node \*copyList(Node \*head)

{

if(head==NULL)

return head;

unordered\_map<Node\*, Node\*> mp;

Node \*newHead;

while(head!=NULL){

if(mp.find(head)==mp.end()){

Node \*t = new Node(head->data);

mp[head] = t;

if(head->next){

if(mp.find(head->next)!=mp.end())

t->next = mp[head->next];

else{

Node \*p = new Node(head->next->data);

mp[head->next] = p;

t->next = p;

}

}

if(head->arb){

if(mp.find(head->arb)!=mp.end())

t->arb = mp[head->arb];

else{

Node \*p = new Node(head->arb->data);

mp[head->arb] = p;

t->arb = p;

}

}

if(!newHead)

newHead = t;

}

else{

if(head->next){

if(mp.find(head->next)!=mp.end())

mp[head]->next = mp[head->next];

else{

Node \*p = new Node(head->next->data);

mp[head->next] = p;

mp[head]->next = p;

}

}

if(head->arb){

if(mp.find(head->arb)!=mp.end())

mp[head]->arb = mp[head->arb];

else{

Node \*p = new Node(head->arb->data);

mp[head->arb] = p;

mp[head]->arb = p;

}

}

}

head = head->next;

}

return newHead;

}

**Method 2 (O(1) space)**

* Create the copy of node 1 and insert it between node 1 & node 2 in the original Linked List, create a copy of 2 and insert it between 2 & 3. Continue in this fashion, add the copy of N after the Nth node
* Now copy the random link in this fashion

original->next->random= original->random->next; /\*TRAVERSE

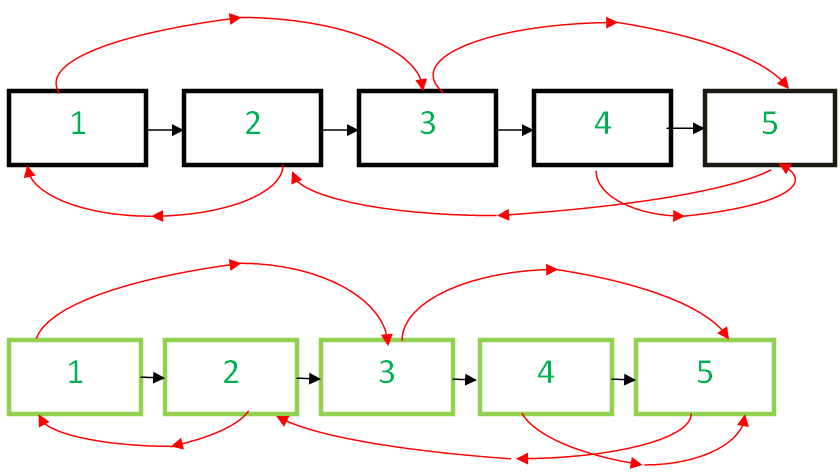
TWO NODES\*/

* This works because original->next is nothing but a copy of the original and Original->random->next is nothing but a copy of the random.
* 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;

* Ensure that original->next is NULL and return the cloned list



Below is the implementation.

// C++ program to clone a linked list with next

// and arbit pointers in O(n) time

#include <bits/stdc++.h>

using namespace std;

// Structure of linked list Node

struct Node

{

int data;

Node \*next,\*random;

Node(int x)

{

data = x;

next = random = NULL;

}

};

// Utility function to print the list.

void print(Node \*start)

{

Node \*ptr = start;

while (ptr)

{

cout << "Data = " << ptr->data << ", Random = "

<< ptr->random->data << endl;

ptr = ptr->next;

}

}

// This function clones a given linked list

// in O(1) space

Node\* clone(Node \*start)

{

Node\* curr = start, \*temp;

// insert additional node after

// every node of original list

while (curr)

{

temp = curr->next;

// Inserting node

curr->next = new Node(curr->data);

curr->next->next = temp;

curr = temp;

}

curr = start;

// adjust the random pointers of the

// newly added nodes

while (curr)

{

if(curr->next)

curr->next->random = curr->random ?

curr->random->next : curr->random;

// move to the next newly added node by

// skipping an original node

curr = curr->next?curr->next->next:curr->next;

}

Node\* original = start, \*copy = start->next;

// save the start of copied linked list

temp = copy;

// now separate the original list and copied list

while (original && copy)

{

original->next =

original->next? original->next->next : original->next;

copy->next = copy->next?copy->next->next:copy->next;

original = original->next;

copy = copy->next;

}

return temp;

}

// Driver code

int main()

{

Node\* start = new Node(1);

start->next = new Node(2);

start->next->next = new Node(3);

start->next->next->next = new Node(4);

start->next->next->next->next = new Node(5);

// 1's random points to 3

start->random = start->next->next;

// 2's random points to 1

start->next->random = start;

// 3's and 4's random points to 5

start->next->next->random =

start->next->next->next->next;

start->next->next->next->random =

start->next->next->next->next;

// 5's random points to 2

start->next->next->next->next->random =

start->next;

cout << "Original list : \n";

print(start);

cout << "\nCloned list : \n";

Node \*cloned\_list = clone(start);

print(cloned\_list);

return 0;

}

**Output**

Original list :

Data = 1, Random = 3

Data = 2, Random = 1

Data = 3, Random = 5

Data = 4, Random = 5

Data = 5, Random = 2

Cloned list :

Data = 1, Random = 3

Data = 2, Random = 1

Data = 3, Random = 5

Data = 4, Random = 5

Data = 5, Random = 2

**My code:**

Node \*copyList(Node \*head)

{

if(!head)

return head;

Node \*curr = head;

while(curr){

Node \*t = new Node(curr->data);

t->next = curr->next;

curr->next = t;

curr = curr->next->next;

}

curr = head;

while(curr){

if(curr->arb)

curr->next->arb = curr->arb->next;

curr = curr->next->next;

}

Node \*newHead = head->next;

curr = head;

while(curr){

Node \*nxt = curr->next;

if(nxt)

curr->next = curr->next->next;

curr = nxt;

}

return newHead;

}

## 156. Merge K sorted Linked list

Given **K** sorted linked lists of different sizes. The task is to merge them in such a way that after merging they will be a single sorted linked list.

**Example 1:**

**Input:**

K = 4

value = {{1,2,3},{4 5},{5 6},{7,8}}

**Output:** 1 2 3 4 5 5 6 7 8

**Explanation:**

The test case has 4 sorted linked

list of size 3, 2, 2, 2

1st    list     1 -> 2-> 3

2nd   list      4->5

3rd    list      5->6

4th    list      7->8

The merged list will be

1->2->3->4->5->5->6->7->8.

**Example 2:**

**Input:**

K = 3

value = {{1,3},{4,5,6},{8}}

**Output:** 1 3 4 5 6 8

**Explanation:**

The test case has 3 sorted linked

list of size 2, 3, 1.

1st list 1 -> 3

2nd list 4 -> 5 -> 6

3rd list 8

The merged list will be

1->3->4->5->6->8.

**Your Task:**  
The task is to complete the function **mergeKList**() which merges the K given lists into a sorted one. The **printing**is done **automatically**by the **driver code**.

**Expected Time Complexity:** O(nk Logk)  
**Expected Auxiliary Space:** O(k)  
**Note: n**is the maximum size of all the **k** link list

**Constraints**  
1 <= K <= 103

### Solution:

**Simple:**

Merge list one by one.

Node\* merge(Node \*a, Node \*b){

if(a==NULL)

return b;

if(b==NULL)

return a;

Node \*res = NULL;

if(a->data<=b->data){

res = a;

res->next = merge(a->next,b);

}

else{

res = b;

res->next = merge(a,b->next);

}

return res;

}

Node \* mergeKLists(Node \*arr[], int K)

{

if(K<1)

return NULL;

Node \*res = NULL;

for(int i=0;i<K;i++)

res = merge(res, arr[i]);

return res;

}

**Approach:**   
A Simple Solution is to initialize the result as the first list. Now traverse all lists starting from the second list. Insert every node of the currently traversed list into result in a sorted way.

// 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;

}

}

// The main function that

// takes an array of lists

// arr[0..last] and generates

// the sorted output

Node\* mergeKLists(Node\* arr[], int last)

{

// Traverse form second list to last

for (int i = 1; i <= last; i++) {

while (true) {

// head of both the lists,

// 0 and ith list.

Node \*head\_0 = arr[0], \*head\_i = arr[i];

// Break if list ended

if (head\_i == NULL)

break;

// Smaller than first element

if (head\_0->data >= head\_i->data) {

arr[i] = head\_i->next;

head\_i->next = head\_0;

arr[0] = head\_i;

}

else

// Traverse the first list

while (head\_0->next != NULL) {

// Smaller than next element

if (head\_0->next->data

>= head\_i->data) {

arr[i] = head\_i->next;

head\_i->next = head\_0->next;

head\_0->next = head\_i;

break;

}

// go to next node

head\_0 = head\_0->next;

// if last node

if (head\_0->next == NULL) {

arr[i] = head\_i->next;

head\_i->next = NULL;

head\_0->next = head\_i;

head\_0->next->next = NULL;

break;

}

}

}

}

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()

{

// Number of linked lists

int k = 3;

// Number of elements in each list

int n = 4;

// 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

**Complexity Analysis:**

* **Time complexity:** O(nk2)
* **Auxiliary Space:** O(1).   
  As no extra space is required.

**Approach:** This solution is based on the **MIN HEAP** approach used to solve the problem ‘merge k sorted arrays’ which is discussed [here](https://www.geeksforgeeks.org/merge-k-sorted-arrays/).  
**MinHeap:** A Min-Heap is a complete binary tree in which the value in each internal node is smaller than or equal to the values in the children of that node. Mapping the elements of a heap into an array is trivial: if a node is stored at index k, then its left child is stored at index 2k + 1 and its right child at index 2k + 2.

1. Create a min-heap and insert the first element of all the ‘k’ linked lists.
2. As long as the min-heap is not empty, perform the following steps:
   * Remove the top element of the min-heap (which is the current minimum among all the elements in the min-heap) and add it to the result list.
   * If there exists an element (in the same linked list) next to the element popped out in previous step, insert it into the min-heap.
3. Return the head node address of the merged list.

Below is the implementation of the above approach:

// 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;

};

// 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;

}

// '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)

{

// Priority\_queue 'pq' implemented

// 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]);

// Handles the case when k = 0

// or lists have no elements in them

if (pq.empty())

return NULL;

struct Node \*dummy = newNode(0);

struct Node \*last = dummy;

// Loop till 'pq' is not empty

while (!pq.empty())

{

// Get the top element of 'pq'

struct Node\* curr = pq.top();

pq.pop();

// Add the top element of 'pq'

// to the resultant merged list

last->next = curr;

last = last->next;

// Check if there is a node

// next to the 'top' node

// in the list of which 'top'

// node is a member

if (curr->next != NULL)

// Push the next node of top node

// in 'pq'

pq.push(curr->next);

}

// Address of head node of the required

// merged list

return dummy->next;

}

// Function to print the singly

// linked list

void printList(struct Node\* head)

{

while (head != NULL)

{

cout << head->data << " ";

head = head->next;

}

}

// Driver code

int main()

{

// Number of linked lists

int k = 3;

// Number of elements in each list

int n = 4;

// 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

**Complexity Analysis:**

* **Time Complexity:** O(N \* log k) or O(n \* k \* log k), where, ‘N’ is the total number of elements among all the linked lists, ‘k’ is the total number of lists, and ‘n’ is the size of each linked list.  
  Insertion and deletion operation will be performed in min-heap for all N nodes.  
  Insertion and deletion in a min-heap require log k time.
* **Auxiliary Space:** O(k).  
  The priority queue will have atmost ‘k’ number of elements at any point of time, hence the additional space required for our algorithm is O(k).

**Method 3:** [**Divide and Conquer**](https://www.geeksforgeeks.org/divide-and-conquer-algorithm-introduction/).   
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)  
It is known that [merging of two linked lists](https://www.geeksforgeeks.org/merge-two-sorted-linked-lists/) can be done in O(n) time and O(n) space.

1. The idea is to pair up K lists and merge each pair in linear time using O(n) space.
2. After the first cycle, K/2 lists are left each of size 2\*N. After the second cycle, K/4 lists are left each of size 4\*N and so on.
3. Repeat the procedure until we have only one list left.

Below is the 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(n) extra space for recursive calls,

\*/

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

**Complexity Analysis:**

     Assuming N(n\*k) is the total number of nodes, n is the size of each linked list, and k is the total number of linked lists.

* **Time Complexity:** O(N\*log k) or O(n\*k\*log k)  
  As outer while loop in function mergeKLists() runs log k times and every time it processes n\*k elements.
* **Auxiliary Space:** O(N) or O(n\*k)  
  Because recursion is used in SortedMerge() and to merge the final 2 linked lists of size N/2, N recursive calls will be made.

## 157. Multiply 2 no. represented by LL

Given elements as nodes of the two linked lists. The task is to multiply these two linked lists, say L1 and L2.

**Note:** The output could be large take modulo 109+7.

**Input:**  
The first line of input contains an integer T denoting the number of test cases. Then T test cases follow, the first line of each test case contains an integer **N** denoting the size of the first linked list (L1). In the next line are the space separated values of the first linked list. The third line of each test case contains an integer **M** denoting the size of the second linked list (L2). In the forth line are space separated values of the second linked list.

**Output:**  
For each test case output will be an integer denoting the product of the two linked list.

**User Task**:  
The task is to complete the function **multiplyTwoLists**() which should multiply the given two linked lists and return the result.

**Constraints:**  
1 <= T <= 100  
1 <= N, M <= 100

**Example:**  
**Input:**  
2  
2  
3 2  
1  
2  
3  
1 0 0  
2  
1 0

**Output:**  
64  
1000

**Explanation:  
Testcase 1:** 32\*2 = 64.

**Testcase 2:** 100\*10 = 1000.

### Solution:

**Solution**:   
Traverse both lists and generate the required numbers to be multiplied and then return the multiplied values of the two numbers.   
Algorithm to generate the number from linked list representation:

1) Initialize a variable to zero

2) Start traversing the linked list

3) Add the value of first node to this variable

4) From the second node, multiply the variable by 10

and also take modulus of this value by 10^9+7

and then add the value of the node to this

variable.

5) Repeat step 4 until we reach the last node of the list.

Use the above algorithm with both of linked lists to generate the numbers.

Below is the program for multiplying two numbers represented as linked lists:

// C++ program to Multiply two numbers

// represented as linked lists

#include<bits/stdc++.h>

#include<stdio.h>

using namespace std;

// Linked list node

struct Node

{

int data;

struct Node\* next;

};

// Function to create a new node

// with given data

struct Node \*newNode(int data)

{

struct Node \*new\_node = (struct Node \*) malloc(sizeof(struct Node));

new\_node->data = data;

new\_node->next = NULL;

return new\_node;

}

// Function to insert a node at the

// beginning of the Linked List

void push(struct Node\*\* head\_ref, int new\_data)

{

// allocate node

struct Node\* new\_node = newNode(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;

}

// Multiply contents of two linked lists

long long multiplyTwoLists (Node\* first, Node\* second)

{

long long N= 1000000007;

long long num1 = 0, num2 = 0;

while (first || second){

if(first){

num1 = ((num1)\*10)%N + first->data;

first = first->next;

}

if(second)

{

num2 = ((num2)\*10)%N + second->data;

second = second->next;

}

}

return ((num1%N)\*(num2%N))%N;

}

// A utility function to print a linked list

void printList(struct Node \*node)

{

while(node != NULL)

{

cout<<node->data;

if(node->next)

cout<<"->";

node = node->next;

}

cout<<"\n";

}

// Driver program to test above function

int main()

{

struct Node\* first = NULL;

struct Node\* second = NULL;

// create first list 9->4->6

push(&first, 6);

push(&first, 4);

push(&first, 9);

printf("First List is: ");

printList(first);

// create second list 8->4

push(&second, 4);

push(&second, 8);

printf("Second List is: ");

printList(second);

// Multiply the two lists and see result

cout<<"Result is: ";

cout<<multiplyTwoLists(first, second);

return 0;

}

**Output**

First List is: 9->4->6

Second List is: 8->4

Result is: 79464

## 158. Delete nodes which have a greater value on right side

Given a singly linked list, remove all the nodes which have a greater value on its following nodes.

**Example 1:**

**Input:**

LinkedList = 12->15->10->11->5->6->2->3

**Output:** 15 11 6 3

**Explanation:** Since, 12, 10, 5 and 2 are

the elements which have greater elements

on the following nodes. So, after deleting

them, the linked list would like be 15,

11, 6, 3.

**Example 2:**

**Input:**

LinkedList = 10->20->30->40->50->60

**Output:** 60

**Your Task:**  
The task is to complete the function **compute**() which should modify the list as required and return the head of the modified linked list. The **printing**is done by the **driver**code,

**Expected Time Complexity:** O(N)  
**Expected Auxiliary Space:** O(1)

**Constraints:**  
1 ≤ size of linked list ≤ 1000  
1 ≤ element of linked list ≤ 1000  
**Note**: Try to solve the problem without using any extra space.

### Solution:

**Method 1 (Simple)**   
Use two loops. In the outer loop, pick nodes of the linked list one by one. In the inner loop, check if there exists a node whose value is greater than the picked node. If there exists a node whose value is greater, then delete the picked node.   
**Time Complexity:** O(n^2)

**Method 2 (Use Reverse)**  
Thanks to Paras for providing the below algorithm.   
1. Reverse the list.   
2. Traverse the reversed list. Keep max till now. If the next node is less than max, then delete the next node, otherwise max = next node.   
3. Reverse the list again to retain the original order.   
**Time Complexity:** O(n)

// C++ program to delete nodes

// which have a greater value on

// right side

#include <bits/stdc++.h>

using namespace std;

/\* structure of a linked list node \*/

struct Node

{

int data;

struct Node\* next;

};

/\* prototype for utility functions \*/

void reverseList(struct Node\*\* headref);

void \_delLesserNodes(struct Node\* head);

/\* Deletes nodes which have a node with

greater value node on left side \*/

void delLesserNodes(struct Node\*\* head\_ref)

{

/\* 1) Reverse the linked list \*/

reverseList(head\_ref);

/\* 2) In the reversed list, delete nodes

which have a node with greater value node

on left side. Note that head node is never

deleted because it is the leftmost node.\*/

\_delLesserNodes(\*head\_ref);

/\* 3) Reverse the linked list again to

retain the original order \*/

reverseList(head\_ref);

}

/\* Deletes nodes which have

greater value node(s) on left side \*/

void \_delLesserNodes(struct Node\* head)

{

struct Node\* current = head;

/\* Initialize max \*/

struct Node\* maxnode = head;

struct Node\* temp;

while (current != NULL &&

current->next != NULL)

{

/\* If current is smaller than max,

then delete current \*/

if (current->next->data < maxnode->data)

{

temp = current->next;

current->next = temp->next;

free(temp);

}

/\* If current is greater than max,

then update max and move current \*/

else

{

current = current->next;

maxnode = current;

}

}

}

/\* Utility function to insert a node

at the beginning \*/

void push(struct Node\*\* head\_ref, int new\_data)

{

struct Node\* new\_node = (struct Node\*)malloc(sizeof(struct Node));

new\_node->data = new\_data;

new\_node->next = \*head\_ref;

\*head\_ref = new\_node;

}

/\* Utility function to reverse a linked list \*/

void reverseList(struct Node\*\* headref)

{

struct Node\* current = \*headref;

struct Node\* prev = NULL;

struct Node\* next;

while (current != NULL)

{

next = current->next;

current->next = prev;

prev = current;

current = next;

}

\*headref = prev;

}

/\* Utility function to print a linked list \*/

void printList(struct Node\* head)

{

while (head != NULL)

{

cout << " " << head->data ;

head = head->next;

}

cout << "\n" ;

}

/\* Driver program to test above functions \*/

int main()

{

struct Node\* head = NULL;

/\* Create following linked list

12->15->10->11->5->6->2->3 \*/

push(&head, 3);

push(&head, 2);

push(&head, 6);

push(&head, 5);

push(&head, 11);

push(&head, 10);

push(&head, 15);

push(&head, 12);

cout << "Given Linked List \n" ;

printList(head);

delLesserNodes(&head);

cout << "Modified Linked List \n" ;

printList(head);

return 0;

}

**Output**

Given Linked List

12 15 10 11 5 6 2 3

Modified Linked List

15 11 6 3

**Method 3:**

The other simpler method is to traverse the list from the start and delete the node when the current Node < next Node. To delete the current node, follow this approach.

Let us assume you have to delete current node X

1. Copy next node’s data into X i.e X.data = X.next.data

2. Copy next node’s next address i.e X.next = X.next.next;

Move forward in the List only when the current Node is > the next Node.

// Java program for above approach

import java.io.\*;

// This class represents a single node

// in a linked list

class Node {

int data;

Node next;

public Node(int data){

this.data = data;

this.next = null;

}

}

//This is a utility class for linked list

class LLUtil{

// This function creates a linked list from a

// given array and returns head

public Node createLL(int[] arr){

Node head = new Node(arr[0]);

Node temp = head;

Node newNode = null;

for(int i = 1; i < arr.length; i++){

newNode = new Node(arr[i]);

temp.next = newNode;

temp = temp.next;

}

return head;

}

//This function prints given linked list

public void printLL(Node head){

while(head != null){

System.out.print(head.data + " ");

head = head.next;

}

System.out.println();

}

}

class GFG {

public static void main (String[] args) {

int[] arr = {12,15,10,11,5,6,2,3};

LLUtil llu = new LLUtil();

Node head = llu.createLL(arr);

System.out.println("Given Linked List");

llu.printLL(head);

head = deleteNodesOnRightSide(head);

System.out.println("Modified Linked List");

llu.printLL(head);

}

//Main function

public static Node deleteNodesOnRightSide(Node head){

if(head == null || head.next == null) return head;

Node nextNode = deleteNodesOnRightSide(head.next);

if(nextNode.data > head.data) return nextNode;

head.next = nextNode;

return head;

}

}

**Output**

Given Linked List

12 15 10 11 5 6 2 3

Modified Linked List

15 11 6 3

## 159. Segregate even and odd nodes in a Linked List

Given a link list of size N, modify the list such that all the even numbers appear before all the odd numbers in the modified list. The order of appearance of numbers within each segregation should be same as that in the original list.

**NOTE:**Don't create a new linked list, instead rearrange the provided one.

**Example 1:**

**Input:**

N = 7

Link List:

17 -> 15 -> 8 -> 9 -> 2 -> 4 -> 6 -> NULL

**Output:** 8 2 4 6 17 15 9

**Explaination:** 8,2,4,6 are the even numbers

so they appear first and 17,15,9 are odd

numbers that appear later.

**Example 2:**

**Input:**

N = 4

Link List:

1 -> 3 -> 5 -> 7

**Output:** 1 3 5 7

**Explaination:** There is no even number.

So ne need for modification.

**Your Task:**  
You do not need to read input or print anything. Your task is to complete the function **divide()**which takes N and head of Link List as input parameters and returns the head of modified link list. Don't create a new linked list, instead rearrange the provided one.

**Expected Time Complexity:**O(N)  
**Expected Auxiliary Space:** O(1)

**Constraints:**  
1 ≤ N ≤ 105  
1 ≤ Each element of the list ≤ 105

### Solution:

Initialize two list even and odd with dummy node. Traverse the given list, if node is even then attach it to even list else attach it to odd list.

Node\* divide(int N, Node \*head){

Node\* dummyEven = new Node(0);

Node\* dummyOdd = new Node(0);

Node\* currEven = dummyEven, \*currOdd = dummyOdd, \*curr = head;

while(curr){

if(curr->data % 2 == 0){

currEven->next = curr;

currEven = currEven->next;

}

else{

currOdd->next = curr;

currOdd = currOdd->next;

}

curr = curr->next;

}

currOdd->next = NULL;

if(currEven==dummyEven)

return dummyOdd->next;

currEven->next = dummyOdd->next;

delete dummyOdd;

return dummyEven->next;

}

**Time Complexity:** O(n)

**Space Complexity:** O(1)

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

Given a linked list consisting of **L** nodes and given a number **N**. The task is to find the **N**th node from the end of the linked list.

**Example 1:**

**Input:**

N = 2

LinkedList: 1->2->3->4->5->6->7->8->9

**Output:** 8

**Explanation:** In the first example, there

are 9 nodes in linked list and we need

to find 2nd node from end. 2nd node

from end os 8.

**Example 2:**

**Input:**

N = 5

LinkedList: 10->5->100->5

**Output:** -1

**Explanation:** In the second example, there

are 4 nodes in the linked list and we

need to find 5th from the end. Since 'n'

is more than the number of nodes in the

linked list, the output is -1.

**Your Task:**  
The task is to complete the function **getNthFromLast**() which takes two **arguments**: **reference**to **head and N** and you need to**return Nth**from the end or -1 in case node doesn't exist..

**Note:**  
Try to solve in single traversal.

**Expected Time Complexity:**O(N).  
**Expected Auxiliary Space:**O(1).

**Constraints:**  
1 <= L <= 106  
1 <= N <= 106

### 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.   
**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:

// 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);

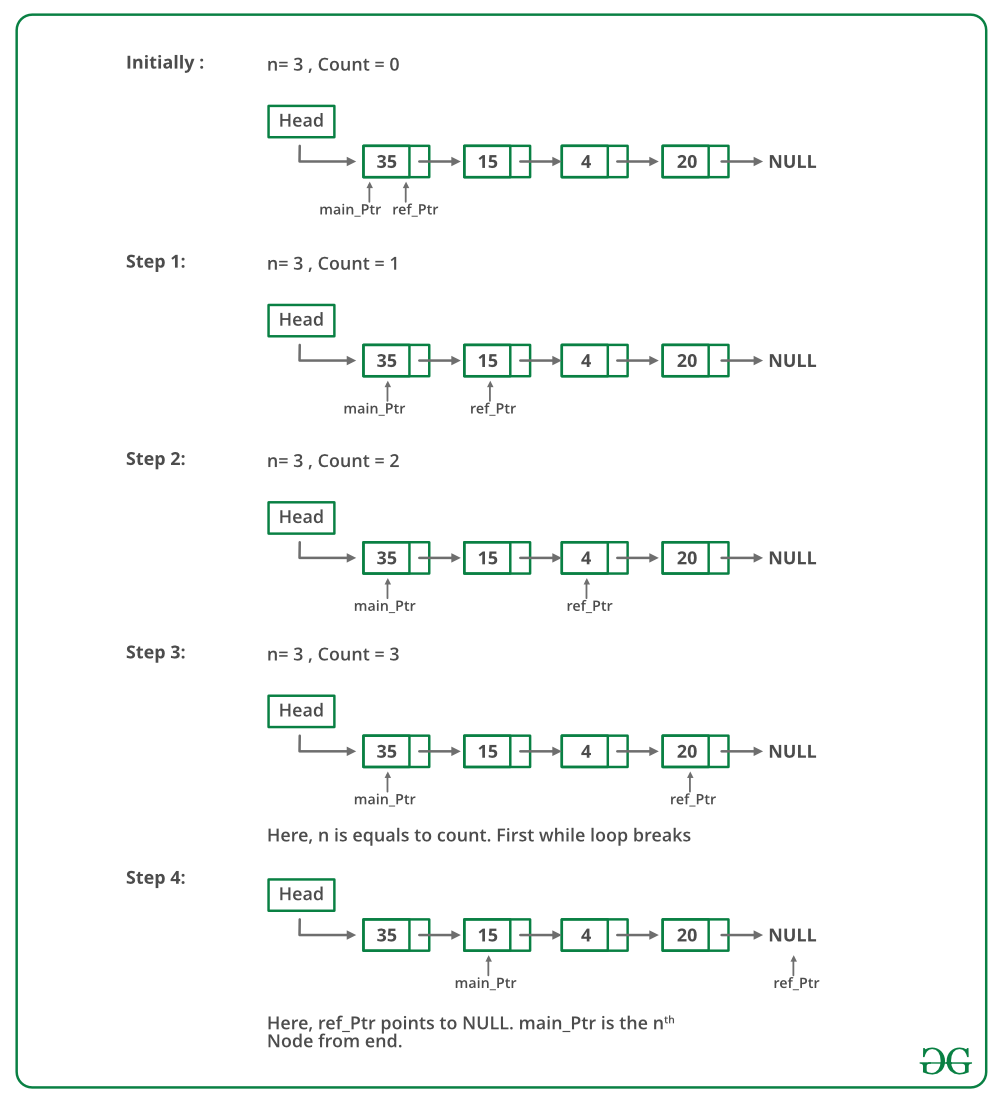
if (++i == n)

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 the 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)

{

printf("%d is greater than the no. of "

"nodes in list", n);

return;

}

ref\_ptr = ref\_ptr->next;

count++;

} /\* End of while\*/

if(ref\_ptr == NULL)

{

head = head->next;

if(head != NULL)

printf("Node no. %d from last is %d ", n, main\_ptr->data);

}

else

{

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);

}

}

}

// Function to push

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 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.

## 161. Find the first non-repeating character from a stream of characters

Given an input stream of **A**of **n** characters consisting only of lower case alphabets. The task is to find the first non repeating character, each time a character is inserted to the stream. If there is no such character then append '#' to the answer.

**Example 1:**

**Input:** A = "aabc"

**Output:** "a#bb"

**Explanation:** For every character first non

repeating character is as follow-

"a" - first non-repeating character is 'a'

"aa" - no non-repeating character so '#'

"aab" - first non-repeating character is 'b'

"aabc" - first non-repeating character is 'b'

**Example 2:**

**Input:** A = "zz"

**Output:** "z#"

**Explanation:** For every character first non

repeating character is as follow-

"z" - first non-repeating character is 'z'

"zz" - no non-repeating character so '#'

**Your Task:**  
You don't need to read or print anything. Your task is to complete the function **FirstNonRepeating()**which takes A as input parameter and returns a string after processing the input stream.

**Expected Time Complexity:**O(26 \* n)  
**Expected Space Complexity:**O(26)

**Constraints:**  
1 <= n <= 105

### Solution:

**Method 1:** [HashMap](http://www.geeksforgeeks.org/java-util-hashmap-in-java/) and Two-string method traversals.  
**Approach:** A character is said to be non-repeating if its frequency in the string is unit. Now for finding such characters, one needs to find the frequency of all characters in the string and check which character has **unit frequency**. This task could be done efficiently using a **hash\_map** which will map the character to there respective frequencies and in which we can simultaneously update the frequency of any character we come across in constant time. The maximum distinct characters in the **ASCII system are 256**. So **hash\_map** has a maximum size of **256**. Now read the string again and the first character which we find has a frequency as unity is the answer.   
**Algorithm:**

1. Make a **hash\_map** which will map the character to there respective frequencies.
2. Traverse the given string using a pointer.
3. Increase the count of current character in the **hash\_map**.
4. Now traverse the string again and check whether the current character has**frequency=1**.
5. If the **frequency>1** continue the traversal.
6. Else **break** the loop and print the current character as the answer.

**Pseudo Code :**

for ( i=0 to str.length())

hash\_map[str[i]]++;

for(i=0 to str.length())

if(hash\_map[str[i]]==1)

return str[i]

break

// C program to find first

// non-repeating character

#include <stdio.h>

#include <stdlib.h>

#define NO\_OF\_CHARS 256

/\* Returns an array of size 256 containing count

of characters in the passed char array \*/

int\* getCharCountArray(char\* str)

{

int\* count = (int\*)calloc(sizeof(int), NO\_OF\_CHARS);

int i;

for (i = 0; \*(str + i); i++)

count[\*(str + i)]++;

return count;

}

/\* The function returns index of first

non-repeating character in a string. If all

characters are repeating then returns -1 \*/

int firstNonRepeating(char\* str)

{

int\* count = getCharCountArray(str);

int index = -1, i;

for (i = 0; \*(str + i); i++) {

if (count[\*(str + i)] == 1) {

index = i;

break;

}

}

// To avoid memory leak

free(count);

return index;

}

/\* Driver program to test above function \*/

int main()

{

char str[] = "geeksforgeeks";

int index = firstNonRepeating(str);

if (index == -1)

printf("Either all characters are repeating or "

"string is empty");

else

printf("First non-repeating character is %c",

str[index]);

getchar();

return 0;

}

**Output**

First non-repeating character is f

**Can this be done by traversing the string only once?**   
The above approach takes **O(n)** time, but in practice, it can be improved. The first part of the algorithm runs through the string to construct the count array (in **O(n)** time). This is reasonable. But the second part about running through the string again just to find the first non-repeater is not a good practice.  
In real situations, the string is expected to be much larger than your alphabet. Take DNA sequences, for example, they could be millions of letters long with an alphabet of just 4 letters. What happens if the non-repeater is at the end of the string? Then we would have to scan for a long time (again). 

**Method 2:** [HashMap](http://www.geeksforgeeks.org/java-util-hashmap-in-java/) and single string traversal.  
**Approach:** Make a count array instead of hash\_map of maximum number of characters(256). We can augment the count array by storing not just counts but also the index of the first time you encountered the character **e.g. (3, 26) for ‘a’ meaning that ‘a’ got counted 3 times and the first time it was seen is at position 26.** So when it comes to finding the first non-repeater, we just have to scan the count array, instead of the string. Thanks to Ben for suggesting this approach.  
**Algorithm :**

1. Make a **count\_array** which will have two fields namely **frequency, first occurrence of a character**.
2. The size of **count\_array** is **‘256’**.
3. Traverse the given string using a pointer.
4. Increase the count of current character and update the occurrence.
5. Now here’s a catch, the array will contain valid first occurrence of the character which has frequency has unity otherwise the first occurrence keeps updating.
6. Now traverse the count\_array and find the character which has **least value of first occurrence** and frequency value as unity.
7. Return the character

**Pseudo Code :**

for ( i=0 to str.length())

count\_arr[str[i]].first++;

count\_arr[str[i]].second=i;

int res=INT\_MAX;

for(i=0 to count\_arr.size())

if(count\_arr[str[i]].first==1)

res=min(min, count\_arr[str[i]].second)

return res

#include <limits.h>

#include <stdio.h>

#include <stdlib.h>

#define NO\_OF\_CHARS 256

// Structure to store count of a

// character and index of the first

// occurrence in the input string

struct countIndex {

int count;

int index;

};

/\* Returns an array of above

structure type. The size of

array is NO\_OF\_CHARS \*/

struct countIndex\* getCharCountArray(char\* str)

{

struct countIndex\* count = (struct countIndex\*)calloc(

sizeof(struct countIndex), NO\_OF\_CHARS);

int i;

for (i = 0; \*(str + i); i++) {

(count[\*(str + i)].count)++;

// If it's first occurrence,

// then store the index

if (count[\*(str + i)].count == 1)

count[\*(str + i)].index = i;

}

return count;

}

/\* The function returns index of the

first non-repeating character in

a string. If all characters are

repeating then returns INT\_MAX \*/

int firstNonRepeating(char\* str)

{

struct countIndex\* count

= getCharCountArray(str);

int result = INT\_MAX, i;

for (i = 0; i < NO\_OF\_CHARS; i++) {

// If this character occurs

// only once and appears

// before the current result,

// then update the result

if (count[i].count == 1

&& result > count[i].index)

result = count[i].index;

}

// To avoid memory leak

free(count);

return result;

}

/\* Driver program to test above function \*/

int main()

{

char str[] = "geeksforgeeks";

int index = firstNonRepeating(str);

if (index == INT\_MAX)

printf("Either all characters are"

" repeating or string is empty");

else

printf("First non-repeating character"

" is %c",

str[index]);

getchar();

return 0;

}

**Output**

First non-repeating character is f

**Complexity Analysis:**

* **Time Complexity:** O(n).   
  As the string need to be traversed at-least once.
* **Auxiliary Space:** O(n).   
  Space is occupied by the use of **count\_array/hash\_map** to keep track of frequency.

**Method #3: Count array and single string traversal:**

**Approach:**

Make a count array of maximum number of characters(256). We can initialize all the elements in this array to -1. And then loop through our string character by character and check if the array element with this character as index is -1 or not. If it is -1 then change it to i and if it not -1 then this means that this character already appeared before, so change it to -2.

In the end all the repeating characters will be changed to -2 and all non-repeating characters will contain the index where they occur. Now we can just loop through all the non-repeating characters and find the minimum index or the first index.

// CPP program to find first non-repeating

// character

# include<iostream>

# include<climits>

using namespace std;

// this function return the index of first non-repeating

// character if found, or else it returns -1

int firstNonRepeating(string str) {

int fi[256]; // array to store First Index

// initializing all elements to -1

for(int i = 0; i<256; i++)

fi[i] = -1;

// sets all repeating characters to -2 and non-repeating characters

// contain the index where they occur

for(int i = 0; i<str.length(); i++) {

if(fi[str[i]] == -1) {

fi[str[i]] = i;

}else {

fi[str[i]] = -2;

}

}

int res = INT\_MAX;

for(int i = 0; i<256; i++) {

// If this character is not -1 or -2 then it

// means that this character occurred only once

// so find the min index of all characters that

// occur only once, that's our first index

if(fi[i] >= 0)

res = min(res, fi[i]);

}

// if res remains INT\_MAX, it means there are no

// characters that repeat only once or the string is empty

if(res == INT\_MAX) return -1;

else return res;

}

int main(){

string str;

str = "geeksforgeeks";

int firstIndex = firstNonRepeating(str);

if (firstIndex == -1)

cout<<"Either all characters are repeating or string is empty";

else

cout<<"First non-repeating character is "<< str[firstIndex];

return 0;

}

**Output**

First non-repeating character is f

**Complexity Analysis:**

* **Time Complexity**: O(n).

                As the string need to be traversed once

* **Auxiliary Space:** O(256).

                Space is occupied by the use of **count-array** to keep track of frequency.

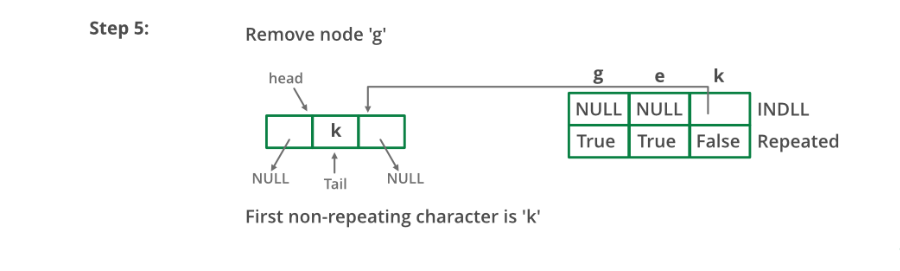
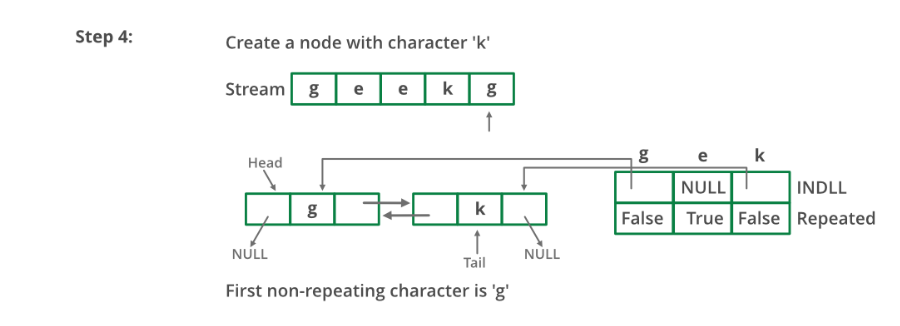
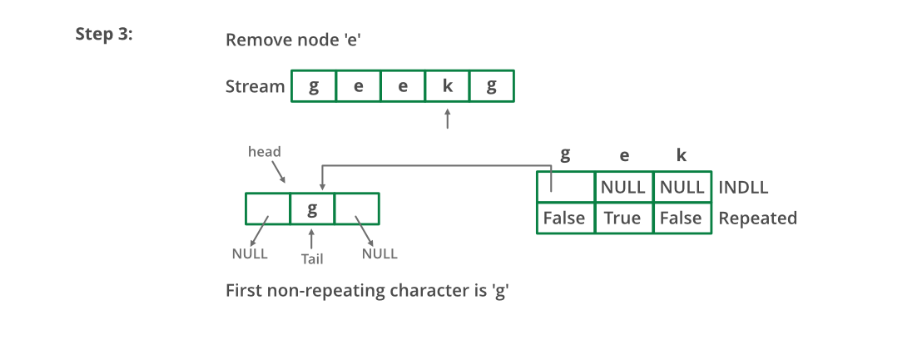
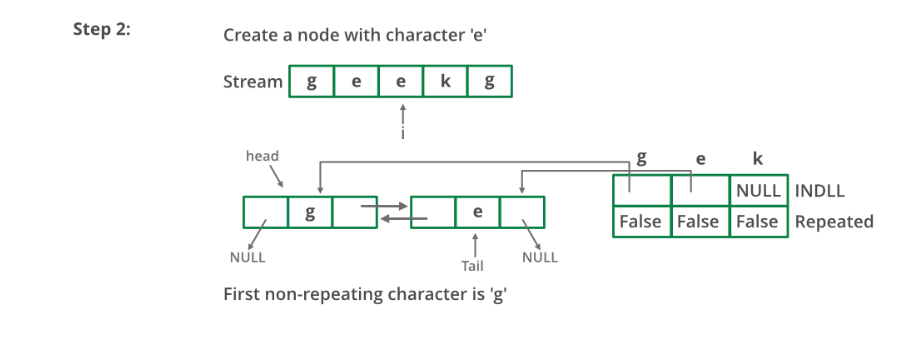
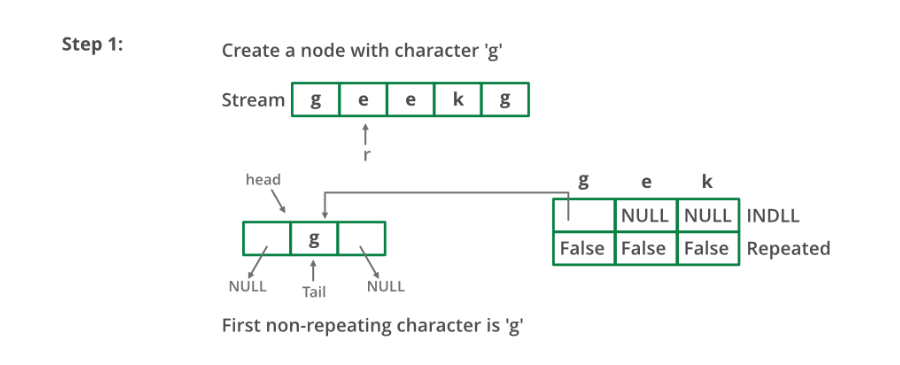
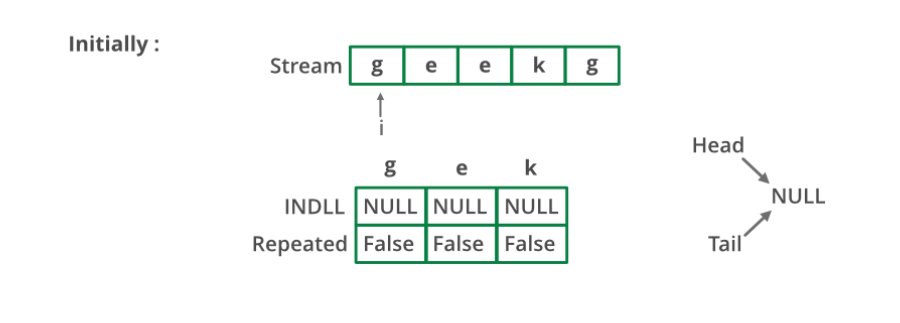
The idea is to use a [DLL](https://www.geeksforgeeks.org/doubly-linked-list/) (**D**oubly **L**inked **L**ist) to efficiently get the first non-repeating character from a stream. The DLL contains all non-repeating characters in order, i.e., the head of DLL contains first non-repeating character, the second node contains the second non-repeating, and so on.

We also maintain two arrays: one array is to maintain characters that are already visited two or more times, we call it repeated[], the other array is an array of pointers to linked list nodes, we call it inDLL[]. The size of both arrays is equal to alphabet size which is typically 256.

1. Create an empty DLL. Also, create two arrays inDLL[] and repeated[] of size 256. In DLL is an array of pointers to DLL nodes. repeated[] is a boolean array, repeated[x] is true if x is repeated two or more times, otherwise false. inDLL[x] contains a pointer to a DLL node if character x is present in DLL, otherwise NULL.
2. Initialize all entries of inDLL[] as NULL and repeated[] as false.
3. To get the first non-repeating character, return character at the head of DLL.
4. Following are steps to process a new character ‘x’ in a stream.
   * If repeated[x] is true, ignore this character (x is already repeated two or more times in the stream)
   * If repeated[x] is false and inDLL[x] is NULL (x is seen the first time). Append x to DLL and store address of new DLL node in inDLL[x].
   * If repeated[x] is false and inDLL[x] is not NULL (x is seen a second time). Get DLL node of x using inDLL[x] and remove the node. Also, mark inDLL[x] as NULL and repeated[x] as true.

Note that appending a new node to DLL is **O(1)** operation if we maintain a tail pointer. Removing a node from DLL is also **O(1)**. So both operations, addition of new character and finding first non-repeating character take **O(1)** time.

Below image is a dry run of the above approach:



Below is the implementation of the above approach:

// A C++ program to find first

// non-repeating character

// from a stream of characters

#include <iostream>

#define MAX\_CHAR 256

using namespace std;

// A linked list node

struct node {

char a;

struct node \*next, \*prev;

};

// A utility function to append a character x at the end

// of DLL. Note that the function may change head and tail

// pointers, that is why pointers to these pointers are

// passed.

void appendNode(struct node\*\* head\_ref,

struct node\*\* tail\_ref, char x)

{

struct node\* temp = new node;

temp->a = x;

temp->prev = temp->next = NULL;

if (\*head\_ref == NULL) {

\*head\_ref = \*tail\_ref = temp;

return;

}

(\*tail\_ref)->next = temp;

temp->prev = \*tail\_ref;

\*tail\_ref = temp;

}

// A utility function to remove a node 'temp' from DLL.

// Note that the function may change the head and tail pointers,

// that is why pointers to these pointers are passed.

void removeNode(struct node\*\* head\_ref,

struct node\*\* tail\_ref, struct node\* temp)

{

if (\*head\_ref == NULL)

return;

if (\*head\_ref == temp)

\*head\_ref = (\*head\_ref)->next;

if (\*tail\_ref == temp)

\*tail\_ref = (\*tail\_ref)->prev;

if (temp->next != NULL)

temp->next->prev = temp->prev;

if (temp->prev != NULL)

temp->prev->next = temp->next;

delete (temp);

}

void findFirstNonRepeating()

{

// inDLL[x] contains pointer to

// a DLL node if x is present

// in DLL. If x is not present, then inDLL[x] is NULL

struct node\* inDLL[MAX\_CHAR];

// repeated[x] is true if x is repeated two or more

// times. If x is not seen so far or x is seen only

// once. then repeated[x] is false

bool repeated[MAX\_CHAR];

// Initialize the above two arrays

struct node \*head = NULL, \*tail = NULL;

for (int i = 0; i < MAX\_CHAR; i++) {

inDLL[i] = NULL;

repeated[i] = false;

}

// Let us consider following stream and see the process

char stream[] = "geeksforgeeksandgeeksquizfor";

for (int i = 0; stream[i]; i++) {

char x = stream[i];

cout << "Reading " << x << " from stream \n";

// We process this character only if it has not

// occurred or occurred only once. repeated[x] is

// true if x is repeated twice or more.s

if (!repeated[x]) {

// If the character is not in DLL, then add this

// at the end of DLL.

if (inDLL[x] == NULL) {

appendNode(&head, &tail, stream[i]);

inDLL[x] = tail;

}

else // Otherwise remove this character from DLL

{

removeNode(&head, &tail, inDLL[x]);

inDLL[x] = NULL;

repeated[x]

= true; // Also mark it as repeated

}

}

// Print the current first non-repeating character

// from stream

if (head != NULL)

cout << "First non-repeating character so far "

"is "

<< head->a << endl;

}

}

/\* Driver code \*/

int main()

{

findFirstNonRepeating();

return 0;

}

**Output:**

Reading g from stream

First non-repeating character so far is g

Reading e from stream

First non-repeating character so far is g

Reading e from stream

First non-repeating character so far is g

Reading k from stream

First non-repeating character so far is g

Reading s from stream

First non-repeating character so far is g

Reading f from stream

First non-repeating character so far is g

Reading o from stream

First non-repeating character so far is g

Reading r from stream

First non-repeating character so far is g

Reading g from stream

First non-repeating character so far is k

Reading e from stream

First non-repeating character so far is k

Reading e from stream

First non-repeating character so far is k

Reading k from stream

First non-repeating character so far is s

Reading s from stream

First non-repeating character so far is f

Reading a from stream

First non-repeating character so far is f

Reading n from stream

First non-repeating character so far is f

Reading d from stream

First non-repeating character so far is f

Reading g from stream

First non-repeating character so far is f

Reading e from stream

First non-repeating character so far is f

Reading e from stream

First non-repeating character so far is f

Reading k from stream

First non-repeating character so far is f

Reading s from stream

First non-repeating character so far is f

Reading q from stream

First non-repeating character so far is f

Reading u from stream

First non-repeating character so far is f

Reading i from stream

First non-repeating character so far is f

Reading z from stream

First non-repeating character so far is f

Reading f from stream

First non-repeating character so far is o

Reading o from stream

First non-repeating character so far is r

Reading r from stream

First non-repeating character so far is a

**Queue based approach for first non-repeating character in a stream**

**Approach-** 

1. Create a count array of size 26(assuming only lower case characters are present) and initialize it with zero.
2. Create a queue of char datatype.
3. Store each character in queue and increase its frequency in the hash array.
4. For every character of stream, we check front of the queue.
5. If the frequency of character at the front of queue is one, then that will be the first non-repeating character.
6. Else if frequency is more than 1, then we pop that element.
7. If queue became empty that means there are no non-repeating characters so we will print -1.

Below is the implementation of above approach:

// C++ program for a Queue based approach

// to find first non-repeating character

#include <bits/stdc++.h>

using namespace std;

const int MAX\_CHAR = 26;

// function to find first non repeating

// character of sa stream

void firstnonrepeating(char str[])

{

queue<char> q;

int charCount[MAX\_CHAR] = { 0 };

// traverse whole stream

for (int i = 0; str[i]; i++) {

// push each character in queue

q.push(str[i]);

// increment the frequency count

charCount[str[i] - 'a']++;

// check for the non pepeating character

while (!q.empty()) {

if (charCount[q.front() - 'a'] > 1)

q.pop();

else {

cout << q.front() << " ";

break;

}

}

if (q.empty())

cout << -1 << " ";

}

cout << endl;

}

// Driver function

int main()

{

char str[] = "aabc";

firstnonrepeating(str);

return 0;

}

**Output:**

a -1 b b