Linked List (Sequential Data Structure)

Disadvantages of array

* types of array in C++ :-

- int arr[100]; → Fixed size
- int arr[n]; → user defined size

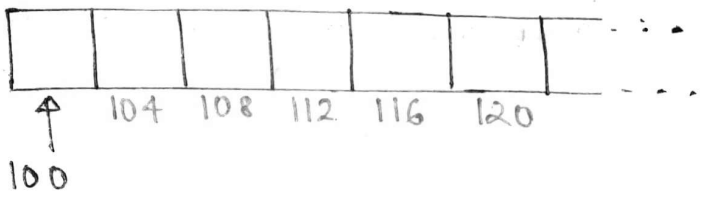
↳ allocated on stack frame of function

- int *arr = new int[n]; → allocated dynamically on heap
- vector <int> v; → dynamic arrays.

The main problem is we have fixed size for arrays.

Suppose we've an array of size 6.

int arr[6];



Now if I want to add one more element, we have to insert it at $(124)^{th}$ address. But we are not aware whether this position is free or not.

Also in vectors, when the initial size is finished, and we have to add one more element that operation is too costly.

There will be one operation of complexity O(n)

As the average complexity is O(1) but in runtime we can't allow any operation to be costly.

- Also the insertion in the middle is costly, deletion too
- Implementation of Data Structures like queue, deque is complex with linked list

Insert 5 at i = 1

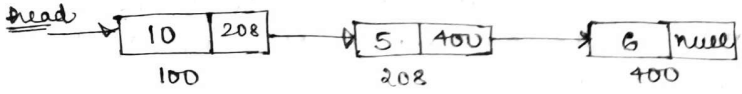
we have to shift all the element after i=0 to one position ahead.

suppose ~~we're~~ we've an array of 1000 elements and we have to insert at i=2, we have to shift the remaining 998 element one step forward.

* If my memory is fragmented, then it's is really difficult to allocate large space using array.

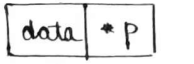
Introduction

* Linear data structure
* contiguous memory requirement is dropped.
* store pointers of next node.
* No need to pre-allocate space



head = 100

Every node contains → own data
→ Reference to next node.



→ This type of storage is not achieved by primitive data types → we will create our own data type → This is achieved by classes & structures

Now, what are the requirement

① int data / char data / bool data --- etc

② ~~int *p~~ address ← address of next node

→ * next

type = node type (Node *next)

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

* last node contains NULL to show, that there is no node after that.
* we can also implement this using structures

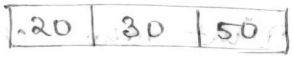
```
struct Node {
    int data;
    Node * next;
    Node (int x) {
        data = x;
        next = NULL;
    }
}
```

→ pointer type is same as type of structure (Self Referencing structure)

→ constructor.

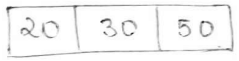
Implementation

```
int main(){
    Node * head = new Node(20);
    Node * tear temp1 = new Node(30);
    Node * temp2 = new Node(50);
    head -> next = temp1;
    temp1 -> next = temp2;
    return 0
}
```



simpler implementation

```
Node * head = new Node(20);
head -> next = new Node(30);
head -> next -> next = new Node(50);
return 0;
```



Node * head = new Node(20)

address of n1 is saved in *head

This will create a node having value 20 (dynamically)

int *p = new int

dynamically memory allocated

Statically

Node n1; creating object of class Node

Node n1(20); Address of n1 is (&n1)

Creating Node Statically

```
Node n1(10);
Node * head = &n1;
```

Creating Node dynamically

```
Node * head = new
              Node(10);
```

Traversal of Linked list

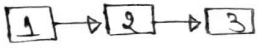
O/P → 10, 20, 40, 50

```
void printList(Node *head){
    Node * curr = head;
    while(curr != NULL){
        cout << curr -> data;
        curr = curr -> next;
    }
}
```

we can directly use the head pointer in c++ when we pass pointer to a function it is passed by value not reference

Recursive function to print linked list

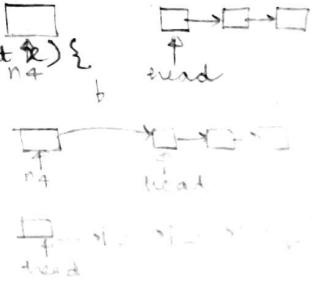
```
void print(Node *head){
    if(head == NULL)
        return;
    cout << (head -> data) << " ";
    print(head -> next);
}
```



→ 1 2 3

Insertion of a Node in Linked list

⊙ In beginning -




```
Node * insertBegin(Node * head, int x){
    Node * temp = new Node(20);
    temp -> next = head;
    head = temp; } or
    return head; } return temp
}
```

* Insertion at End

I/P → 10 → 20 → 30 , 5

O/P → 10 → 20 → 30 → 5

```
void insertEnd(Node *head, int data){
    Node *temp = new Node(data);
    temp->next = NULL
    while(temp->next != NULL){        // temp = head
        head = head->next;
    }                                 } Traverse upto last node
    head->next = temp;
}
```

⇓

```
Node *insertEnd(Node *head, int x){
    Node *tmp = new Node(x);
    if(head == NULL)                  } If head is null, temp becomes head
        return temp;
    Node *curr = head;
    while(curr->next != NULL)
        curr = curr->next;
    curr->next = temp;
    return head;
}
```

* Delete First Node of Singly Linked List:

I/P → 10 → 20 → 30

O/P = 20 → 30

I/P → null

O/P → null.

I/P → 10

O/P → Null

```
Node *DeleteBegin(Node *head){
    if(head == NULL)
        return NULL
    Node *temp = head;
    temp = temp->next;
    delete(head);                     // deleting memory space deallocating
    return temp;
}
```

① Delete last Node of linked list

```
Node *delTail(Node *head){
    if(head == NULL){
        return NULL;
    }
    if(head->next == NULL){
        delete(head);
        return NULL;
    }
    Node *tmp = head;
    while(tmp->next != NULL){
        temp = tmp->next
    }
    Node *curr = tmp->next.
    delete(curr);
    temp->next = NULL;
    return head;
}
```

```
while(
    tmp->next->next
        != NULL)
    tmp = tmp->next
```

Ensuring atleast two nodes are present

Θ(n)

* Insert at given position at singly linked list

I/P → 10 → 20 → 30 → 40

pos = 2

data = 205

O/P → 10 → 205 → 20 → 30 → 40

I/P → 10 → 20 pos = 1

O/P → 10 → 20

- We have to run a loop (pos-2) times as we want the previous node to link
- We also want that curr != NULL

```
Node *insertPos(Node *head, int pos, int data){
    Node *temp = new Node(data);
    if(pos == 1){
        temp->next = head;
        return temp;
    }
    for(int i = 0; i < pos-2 && curr != NULL; i++)
        curr = curr->next;
    if(curr == NULL){
        delete(temp);
        return head;
    }
    temp->next = curr->next;
    curr->next = temp;
    return head;
}
```

Search in a Linked List

I/P → 10 → 5 → 20 → 15 x = 20

O/P → 3

O(1) → Aux space

```
int search(Node *head, int x){
    int pos = 1;
    Node *curr = head;
    while(curr != NULL){
        if(curr->data == x)
            return pos;
        else{
            pos++;
            curr = curr->next;
        }
    }
    return -1;
}
```

Recursive solution

O(n)

```
int search(Node *head, int x){
    if(head == NULL)
        return -1;
    if(head->data == x)
        return 1;
    int res = search(head->next, x);
    if(res == -1) return -1;
    return res+1
}
```

Taking Input as linked list

- we assume that when user enters -1, we have to terminate our linked list or the user doesn't want to continue.

* If we allocate node statically.

For ex:-

```
while (data != -1) {
    Node n(data);
}.
```

Now the scope of n is ~~struct~~ within the while loop only, everytime the loop runs the previous allocated node is deleted.

To prevent this we use dynamic allocation, the allocated node will not be deallocated unless the programmer itself deletes that.

```
Node *newNode = new Node(data);
```

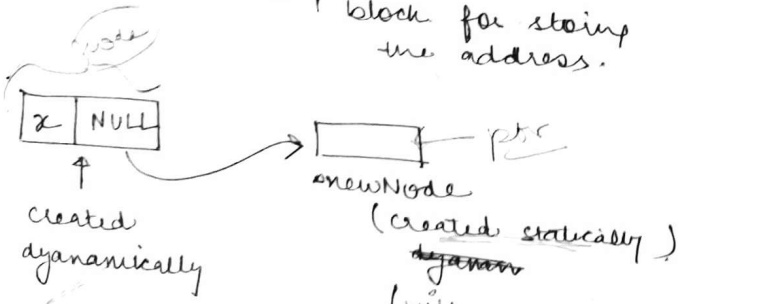
* we have to store the address of head node.

In this line

```
Node *newNode = new Node(data)
```

2 variables are made → 1 block of memory storing data

→ 1 block for storing the address.

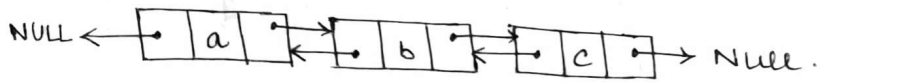



```
Node * takeinput() {
    int data;
    cin >> data;
    Node * head = NULL;
    Node * prev = NULL;
    while (data != -1) {
        Node * created = new Node(data);
        if (Head == NULL) {
            head = created;
        }
        else {
            prev -> next = created;
        }
        prev -> created;
        cin >> data;
    }
    return head;
}
```

O(n).

DOUBLY LINKED LIST

- Has two pointers → pointing to next node
  → pointing to previous node




```
struct Node {
    int data;
    Node * prev;
    Node * next;
    Node (int d) {
        data = d;
        prev = NULL; next = NULL;
    }
}
```

Structure for doubly linked list

Implementation to create a doubly linked list

```
int main(){
    Node *head = new Node(10);
    Node *n1 = new Node(20);
    Node *n2 = new Node(30);

    head->prev = NULL;
    head->next = n1;
    n1->prev = head;
    n2->prev = n1;
    n1->next = n2
    n2->next = NULL;
}
```

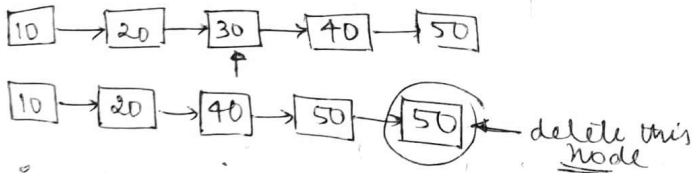
- Not needed as already declared in constructor

Singly vs Doubly linked lists

Advantages

- Can be traversed in both directions
- A given node can be deleted in O(1) time

(In singly linked list, this is not possible, the only solution is:



But this is not possible when the given node is tail of linked list.

① Insert / delete before the given node.
② Insert / delete at the end in O(1) time.

Disadvantages

① Extra space for prev
② code becomes complex.

Insert data at the beginning of linked list

I/P → 10 → 20 x = 30
O/P → 30 → 10 → 20

I/P → NULL, 30
O/P → 30

```
Node *insertBegin(Node *head, int x){
    Node *temp = new Node(x);
    if(head == NULL)
        return temp;
    temp->next = head;
    head->prev = temp;
    return temp;
}
```

- temp becomes the new head

Insert at the End of linked list

I/P → 10 → 20 x = 30
O/P → 10 → 20 → 30

```
Node *insertEnd(Node *head, int x){
    Node *temp = new Node(x);
    if(head == NULL)
        return temp;
    Node *curr = head;
    while(curr->next != NULL){
        curr->next;
    curr->next = temp;
    temp->prev = curr;
    return head;
}
```

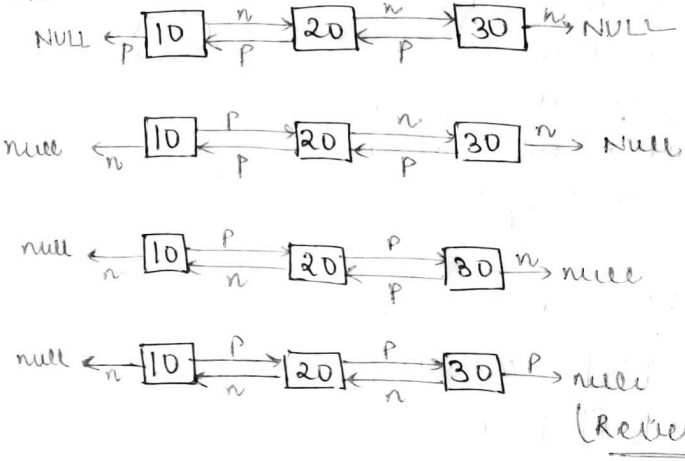
Reverse a Doubly Linked List

I/P → 10 ⇄ 20 ⇄ 30

O/P → 30 ⇄ 20 ⇄ 10

- we need to swap the prev and next pointer of every node.

Ex:-




```
Node * reverseLL (Node *head){
    if (node == NULL || node -> next == NULL)
        return head;
    while (1)
    Node * curr = head, *temp = NULL;
    while (curr != NULL){
        temp = curr -> prev;
        curr -> next = curr -> prev;
        curr -> prev = temp;
        curr = curr -> prev;
    }
    return curr -> temp -> prev;
}
```

& swapping

Delete head node of Doubly linked list

I/P → 10 ⇄ 20 ⇄ 30

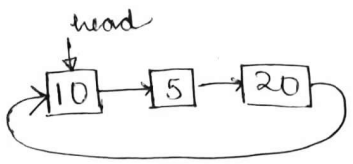
O/P → 20 ⇄ 30

```
Node * deleteHead (Node *head) {
    if (head == NULL)
        return NULL;
    if (head -> next == NULL) {
        delete (head);
        return NULL;
    }
    Node *curr = head;
    curr = curr -> next;
    curr -> prev = NULL;
    head -> next = NULL;  * not required
    delete (head);
    return curr;
}
```

Delete Tail of Doubly Linked list

```
Node * deleteTail (Node *head){
    if (head == NULL)
        return NULL;
    if (head -> next == NULL) {
        delete (head);
        return NULL;
    }
    Node *curr = head;
    while (curr -> next -> next != NULL){
        curr = curr -> next;
    }
    delete (curr -> next);
    curr -> next = NULL;
    return head; }
```

Circular linked list



tail's next is linked back to head

single node linked list → 

head → next = head

advantages

- we can traverse the whole list from any node
- implementation of algorithm like round robin
- we can insert at beginning / end by maintaining one tail pointer.

Traversal of circular LL

method-1

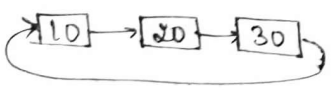
```
void print (head){
    if( head == NULL)
        return ;
    Node *p = head;
    do {
      cout << p-> data;
      P = p-> next;
    } while ( p != head);
}
```

method-2

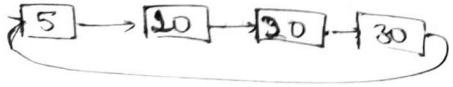
using for loop (slightly complex)

Not Required.

Insert in the beginning

I/P → 

x = 5

O/P → 

I/P → NULL x = 10 O/P 

1st Method

```
Node * insertBegin (Node * head, int x) {
    Node *temp = new Node (x);
    if( head == NULL) {
        temp-> next = temp ;
    }
    else{
        Node * curr = head ;
        while ( curr-> next != head)
            curr = curr-> next ;
        curr-> next = temp ;
        temp-> next = head;
    }
    return temp ;
}
```

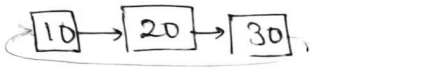
} For updating the tail link

2nd Method O(1) time

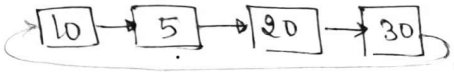
we can maintain a tail pointer and then every operation is done in O(1) time

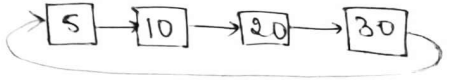
OR.

we can insert a node after head and then swap the values of head & head → next

x = 5 , 

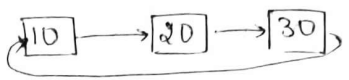
↓

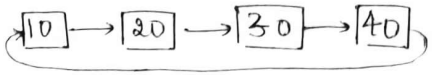




swapping the values only

Insert at the end of Circular linked list

I/P → 

O/P → 

Method-1

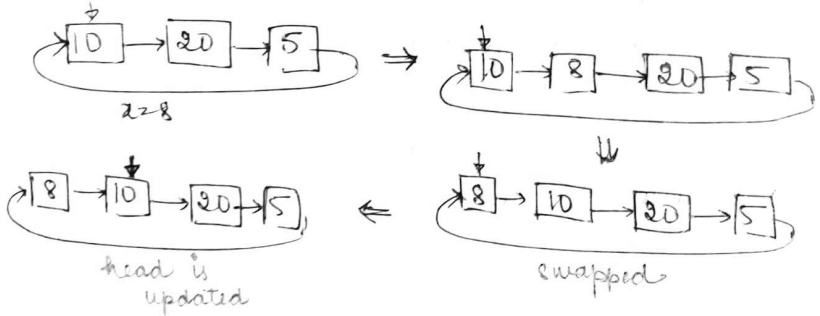
just traverse to the tail of linked list and add the new node.

Method-2

Maintain a tail pointer

Method-3

- insert a node after head
- swap the data of new node and head
- shift the head pointer to the new node.




```
temp -> next = head -> next
head -> next = temp;
int t = temp -> data;
temp -> data = head -> data;
head -> data = t
return temp;
```
← New head

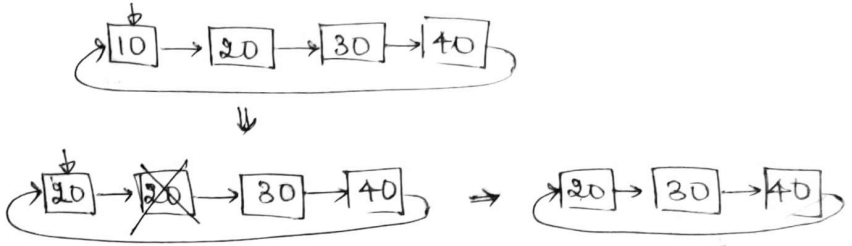
Delete the head of CLL

Method-1

Loop through the ttt linked list and move to the tail and delete the head node

method-2

Copy the content of 'head → next' to 'head' node & and then delete the head node

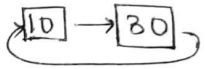


Delete k^{th} Node in linked list

I/P → 

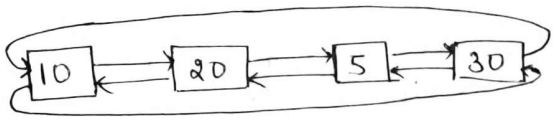
k = 2.

no of node ≥ k

O/P → 


```
Node * deleteKth (Node * head, int k) {
    if (head == NULL) return head;
    if (k == 1) return deleteHead (head);
    Node * curr = head;
    for (int i=0 ; i< k-2 ; i++)
        curr = curr -> next ;
    Node * temp = curr -> next ;
    curr -> next = curr -> next -> next ;
    delete temp ;
    return head ;
}
```

Circular Linked List



having only one node → 

Advantages

- All the advantages of circular & doubly ll.
- we can get tail pointer in O(1) operation.

Insert in CDLL (Head)

```
Node * temp = new Node(x);
if (head == NULL){
        temp -> next = temp;
        temp -> prev = temp;
        return temp;
}
    temp -> prev = head -> prev;
    temp -> next = head
    head -> prev -> next = temp
    head -> prev = temp
    return temp;
```

- Insert at the end has exactly the same code. We just need to return head only, which means that we don't need to change head.

  return temp ✗

  return head ✓.

Sorted Insert in the Linked list

I/P → 10 → 20 → 30 → 40 $x = 35$

O/P → 10 → 20 → 30 → 35 → 40

```
Node * sorted_Insert (Node * head, int x) {
    Node * temp = new Node(x);
    if (head == NULL)
        return temp ;
    if (x < head -> data) {
        temp -> next = head;
        return temp;
    }
    Node * curr = head;
    while (curr -> next != NULL &&
           curr -> next -> data < x) {
        curr = curr -> next;
    }
    temp -> next = curr -> next;
    curr -> next = temp ;
    return head;
}
```

insert before head.

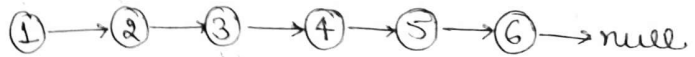
updating the position

insert the temp

MID Element of a linked list

There are two possible questions

① even length



mid element → 3, 4

② odd length

mid → 3

Method-1

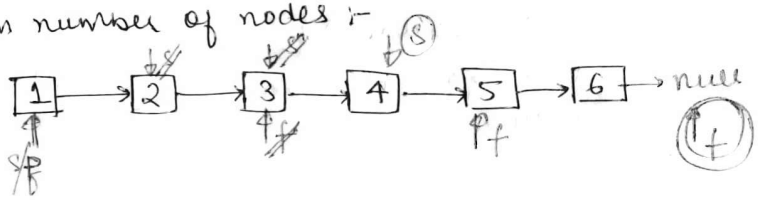
Calculate ~~Rank~~ length of linked list and just do len/2

- If we want the first mark in even length linked list, we can calculate pos = $\left(\frac{len+1}{2}\right)$.

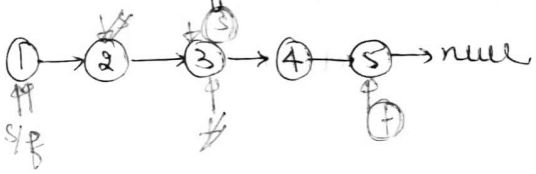
Method-2

- Declare 2 pointers → slow (moves 1 pos ahead)
  → fast (moves 2 pos ahead)

For even number of nodes :-



For odd number of nodes



condition → fast != NULL && fast->next != NULL.

```
void printMiddle (Node *head){
    if (head == NULL) return;
    Node * slow = head, * fast = head;
    while (fast != NULL && fast->next != NULL){
        slow = slow -> next;
        fast = fast -> next -> next;
    }
    cout << slow -> data;
}
```

Nth Node from End of linked list

I/P → 10 → 20 → 30 → 40 → 50 n = 2

O/P → 40

I/P → 10 → 20 n = 3

O/P → —

Method-1

If we calculate length, then the nth node from end will be the ~~n~~ (len−n+1)th node from beginning.

calculating length →

```
for (Node *curr = head; curr != NULL;)
{
    len++;
    curr = curr -> next;
}
```

Now checking

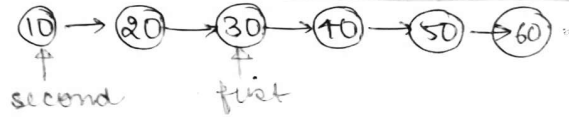
```
if (len < n){
    return;
}
```

printing the node

```
Node * curr = head
for (int i = 1; i < (n+len+1); i++)
    curr = curr -> next
cout << curr -> data;
```

Method-2

n = 2



- Move first pointer n positions ahead.
- Maintain a pointer named second starting from head
- We now move first & second pointer at same speed
- ~~We~~ when first reaches null, second pointer reaches at required position

```
void printNthEnd(Node * head, int n){
    if(head == NULL) return;
    Node * first = head;
    for(int i=0; i<n; i++){
        if(first == NULL) return;
        first = first -> next;
    }
    Node * second = head;
    while(first != NULL){
        second = second -> next;
        first = first -> next;
    }
    cout << second -> data;
}
```

Reverse a Linked List

I/P → 10 → 20 → 30 O/P → 30 → 20 → 10

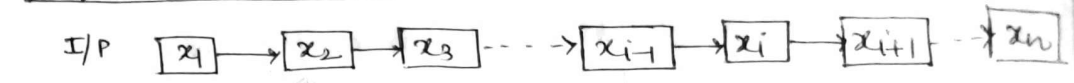
Naive Solution

O(n) time + O(n) space

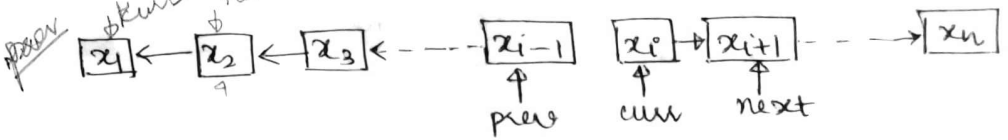
copy the contents of linked list into a vector and then create a linked list in reverse order.

```
Node * reverseList(Node * head){
    for(Node * curr = head; curr != NULL;
                curr = curr -> next){
        arr.push_back(curr -> data);
    }
    for(Node * curr = head; curr != NULL;
                curr = curr -> next){
        curr -> data = arr.back();
        arr.pop_back();
    }
    return head;
}
```

Efficient Solution



After reversing nodes from x_1 to x_{i-1}.



- keep track of prev to link curr → next
- store next so that we don't lose the remaining linked list

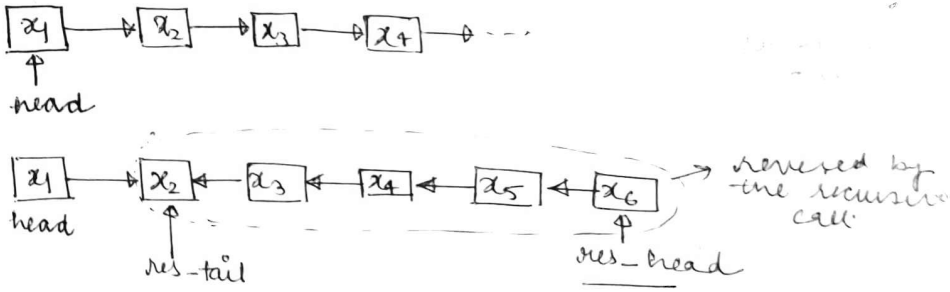
```
Node * reverse(Node * head){
    Node * curr = head;
    Node * prev = NULL;
    while(curr != NULL){
        Node * next = curr -> next;
        curr -> next = prev
        prev = curr;
        curr = next;
    }
    return prev;
}
```

C++

```
next = curr -> next
curr -> next = prev
prev = curr;
curr = next;
```

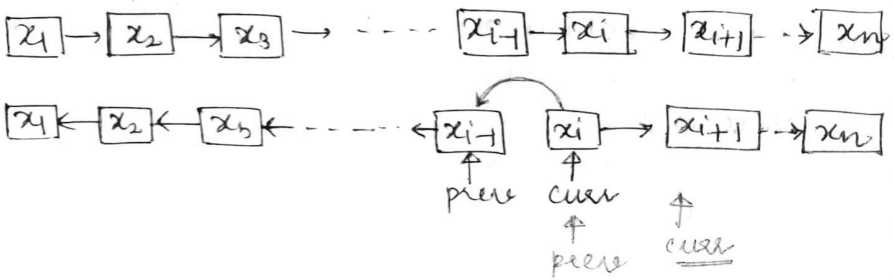
O(n)

Reverse a Linked List



```
Node * recRev (Node * head) {
    if (head == NULL || head -> next == NULL)
        return head;
    Node * new_head = recRev (head -> next);
    Node * new_tail = head -> next
    new_tail -> next = head;
    head -> next = NULL
    return new_head;
}
```

Method-2 (Recursive)




```
Node * recRev (Node * curr, Node * prev) {
    if (curr == NULL) return prev;
    Node * next = curr -> next;
    curr -> next = prev;
    return recRev (next, curr);
}
```

- we just update the link and call the next (leftover) list where we update the link and call the leftover .....

Remove Duplicates from sorted Linked list

I/P → 10 → 10 → 20 → 20 → 20 → 30 → NULL

O/P → 10 → 20 → 30 → NULL

```
void RemoveDup (Node * head) {
    Node * curr = head;
    while (curr != NULL && curr -> next != NULL)
    {
        if (curr -> data == curr -> next -> data) {
            Node * temp = curr -> next;
            curr -> next = curr -> next -> next;
            delete (temp);
        }
        else
            curr = curr -> next;
    }
}
```

Reverse Linked list in groups of size 'k'

I/P → 10 → 20 → 30 → 40 → 50 → 60 k = 3

O/P → 30 → 20 → 10 → 60 → 50 → 40

I/P → 10 → 20 → 30 → 40 → 50 k = 3

O/P → 30 → 20 → 10 → 50 → 40

I/P → 10 → 20 → 30 k = 4

O/P → 30 → 20 → 10

- If k > remaining number of nodes, then reverse the remaining

- we reverse the first k nodes ~~to~~ iteratively and r then call for reversing the next k.
- we have to connect the tail of previous reversed linked list to the new head.
- On recursive call we return the new_head to next reversed list and we store the tail of our list in the calling function.
- we then update the links.

```
Node *reversek( Node *head, int k){
    Node *curr = head;
    Node *next = NULL, prev = NULL;
    int count = 0;
    while (curr != NULL && count < k) {
        next = curr -> next;
        curr -> next = prev;
        prev = curr;
        curr = next;
        count++;
    }
    if (next != NULL){
        Node res_head = reversek (next, k);
        head -> next = res_head;
    }
    return prev;
}
```

Reverse it in groups of 'k'

calling & update links

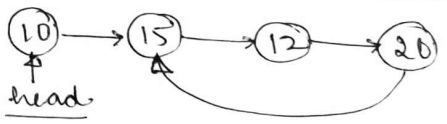
O(n)

Auxiliary space → O(n/k)

Iterative solution to Reduce complexity (Space) O(1)

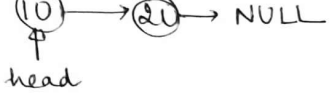
```
Node *reversek( Node *head, int k) {
    Node * curr = head;
    /* prevFirst = NULL;
    bool isFirstPass = true;
    while (curr != NULL) {
        Node *first = curr; prev = NULL;
        * count = 0;
        while( curr != NULL && count < k) {
            Node * next = curr -> next;
            curr -> next = prev;
            prev = curr;
            curr = next
            count ++;
        }
        if (isFirstPass) { head = prev; isFirstPass
                                = false; }
        else { prevFirst -> next = prev; }.
        prevFirst = first;
    }
    return head;
}
```

DETECT LOOP IN LINKED LIST

I/P → 

O/P → Yes.

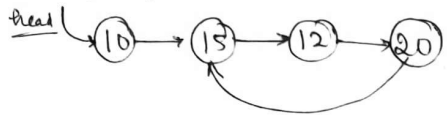
Or I/P → head = NULL O/P = NO

I/P → 

O/P = NO

Naive solution

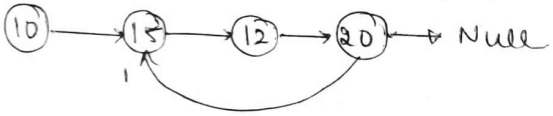
At any a node, run the loop from head to that node to check if next of curr is same as any of previous node.



→ at curr→data = 15
run loop from 10 to 15

→ at 20, run a loop from 10 to 15, 20→next = 15

Method-2 (modification in linked list structure is allowed).



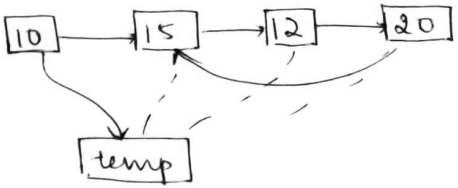
Structure of Linked List

```
struct Node{
    int data;
    Node * next;
    bool visited;
    Node (int data){
        data = data;
        next = NULL;
        visited = false;
    }
};
```

- we mark the node visited whenever we visited it
- whenever we see a node already visited, we detect a loop

(Method-3) Modification to linked list pointers

I/P



we traverse the linked list and at every node we change the next of it to temp.
whenever we reach a node whose next is already pointing to temp, we detect a loop. we stop, when curr→next = NULL

Algorithm :-

```
bool isLoop (Node * head) {
    Node * temp = new Node;
    Node * curr = head;
    while (curr != NULL) {
        if (curr → next == NULL) return false;
        if (curr → next == temp) return true;
        Node * curr_next = curr → next;
        curr → next = temp;
        curr = curr_next;
    }
    return false;
}
```

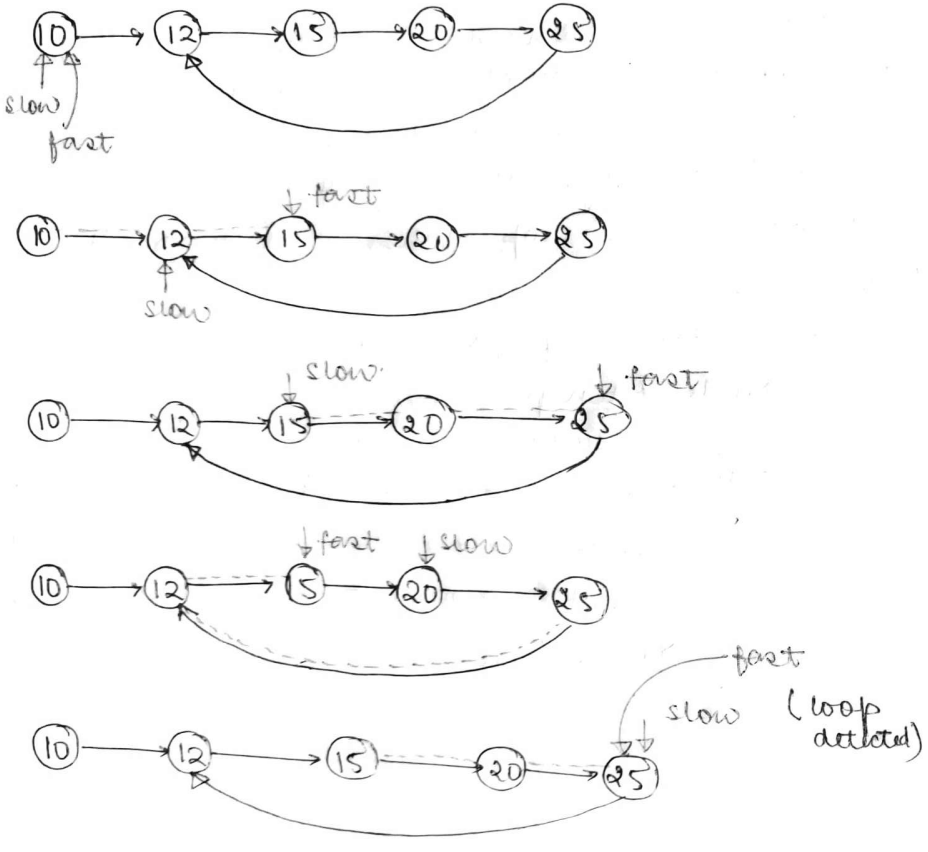
Method-4 (using Hashing)

O(n) Time
O(n) Aux space

```
bool isLoop (Node * head) {
    unordered_set < Node * > s;
    for (Node * curr = head; curr != NULL; curr = curr→next)
        if (s.find(curr) != s.end()) return true;
        s.insert(curr);
    return false;
}
```

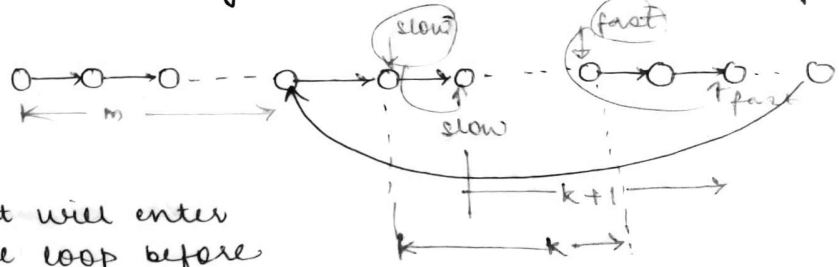
Detect Loop (Floyd's Cycle Detection) (Hare & Tortoise Algorithm)

- There is a slow and fast pointer running one & two steps at a time Respectively




```
bool isLoop (Node * head) {
    Node * slow = head, * fast = head;
    while (fast != NULL && fast -> next != NULL) {
        slow = slow -> next;
        fast = fast -> next -> next;
        if (slow == fast) return true;
    } return false; }
```

How does the algorithm work (Mathematically)



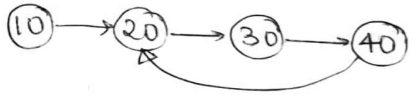
① fast will enter the loop before slow

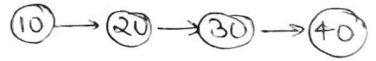
② on every iteration the distance b/w them is increased by 1.

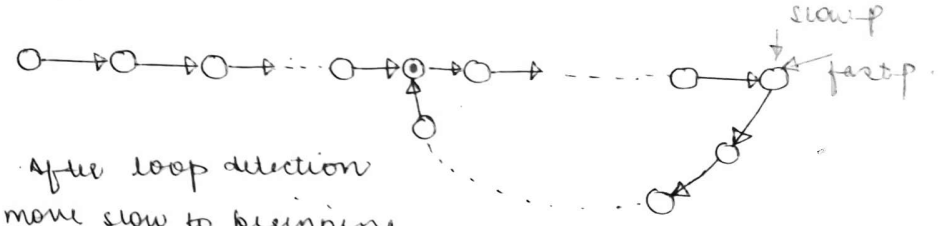
③ After some iteration their may be a case when distance becomes (n) → linked list length.
→ they are at the same node

Time complexity → O(m+n) m → n → length of loop
→ O(length) → of linked list

Detect and Remove the loop

I/P → 

⇒ 



① ~~move~~ After loop detection
→ move slow to beginning

② And move both slow and fast pointer by 1 position ahead

③ The claim is they both will meet at the starting of loop (Red node)

```
void detectLoop (Node *head){
    Node *slow = head, *fast = head;
    while (fast != NULL && fast->next != NULL){
        slow = slow->next;
        fast = fast->next->next;
        if (slow == fast)
            break;
    }
    if (slow != fast)
        return;
```

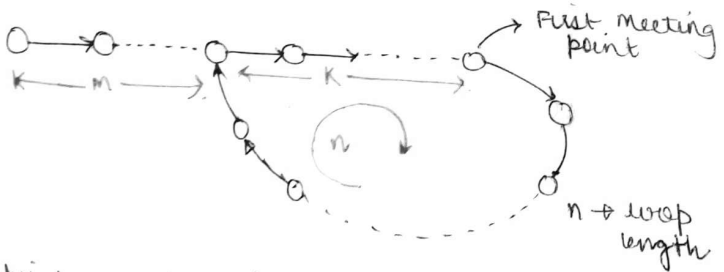
Loop Detection

```
    slow = head;
    while(slow->next != fast->next){
        slow = slow->next;
        fast = fast->next;
    }
    fast->next = NULL;
    return;
}
```

Loop Removal

- we compare the next pointers so that we stop 1 node before starting point

How this algorithm work?



Before first meeting point

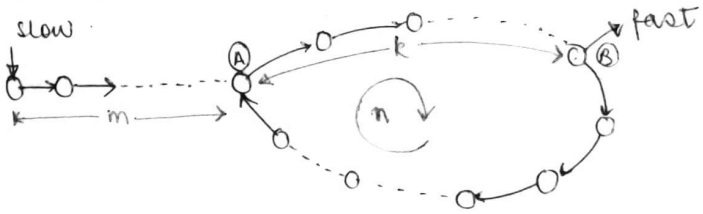
Distance travelled by slow * 2 = Distance travelled by fast

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

$$(m+k) = n(y-2x)$$

$m+k$ is multiple of n.

x → no of cycles slow made before first meeting pt



* slow will have to travel m distance to reach A.
* fast will travel 'n' or (m+k) to reach at same position i.e B.
* To reach B, fast has to travel 'k' steps less. (A−B)
  m+k−k = m steps
* i.e at m steps slow and fast both will reach A, their second meeting point

- Find length of loop
- Find first node of loop → just did it, i.e we have to calculate A

Just fix slow point and move fast one steps ahead and then increament the count till it reaches the same position.

Delete the node with only pointer given to it;-

I/P - 10 → 20 → 30 → 40 → 50

reference of node 30 is given

O/P → 10 → 20 → 40 → 50

Trick → copy the content of next node to the curr node whose pointer is given and then delete the next node

* will not work for last node

```
void deleteNode (Node *ptr) {
    Node *temp = ptr->next
    ptr->data = temp->data;
    ptr->next = ptr->next->next;
    delete (ptr); }
```

Segregate Even and odd value in linked list.

I/P → 17 → 15 → 8 → 12 → 10 → 5 → 4

O/P → 8 → 12 → 10 → 4 → 17 → 15 → 5

I/P → 8 → 12 → 10

O/P → 8 → 12 → 10

- We maintains pointers like Evenstart (ES), Eventail (ET), Oddstart (OS), oddtail (OT)

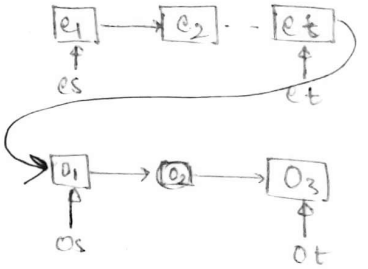
⊙ whenever we see a even nodd we append it after eventail, same goes with odd.

⊙ At the end, we just append the odd start after even tail.

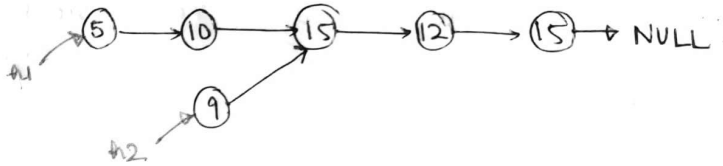
```
Node * segregate (Node *head) {
    Node *es = NULL, *et = NULL, *os = NULL, ot=NULL;
    for (Node *curr = Head, curr != NULL; curr =
                                        curr → next){

        int x = curr → data;
        if (x % 2 == 0) {
            if(es == NULL){
                es = curr;
                et = es;
            }
            else{
                et → next = curr;
                et = eE → next;
            }
        }
        else { if(ot == NULL) {
                ot = curr; os = ot;
            } else {
                ot → next = curr;
                ot = ot → next;
            }
        }
    }
```

// After for loop

```
if (ot == NULL || et == NULL)
        return head;
ø. et → next = os;
   ot → next = NULL;
   return es;
}
```



Intersection point of two linked list



O/P → 15

Method-1

* create a hash set storing pointer every node we encounter
* As soon as we found a node whose address is already present that is an intersection

O(n) space

O(n+m) time complexity

Method-2

① count the number of nodes in both the list as c_1 and c_2

② ~~Moo~~ Traverse the bigger list abs($c_1 - c_2$)-times

③ Move both the list at same speed and they will meet at intersection point

Pairwise swap nodes.

I/P → 1→2→3→4→5→6

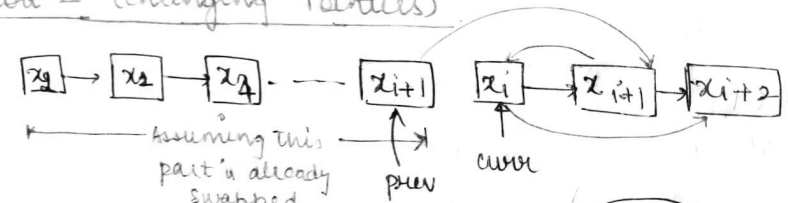
O/P → 2→1→4→3→6→5

Method-1 (Swapping data)

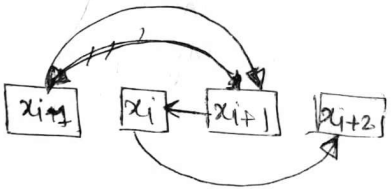
Swap the values of curr and curr→next ~~and~~ & move curr to curr→next→next.

```
void pairswap(Node *head){
    Node *curr = head;
    while(curr != NULL && curr->next != NULL){
        swap(curr->data, curr->next->data)
        curr = curr->next->next;
    }
}
```

Method-2 (Changing Pointers)

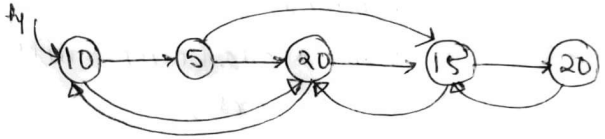


for swapping



```
Node *pairSwap_better(Node *head){
    Node *curr = head, *next, *prev = NULL,
         *temp = NULL;
    while(curr != NULL && curr->next != NULL){
        next = curr->next->next;
        temp = curr->next;
        temp->next = curr;
        curr->next = next;
        if(prev != NULL)
            prev->next = temp;
        else
            head = temp;
        prev = curr;
        curr = next;
    }
    return head;
}
```

Clone the linked list with Random pointers



Method-1

Using hash maps.

① Create a hashmap, m.

② for(curr = h1, curr != null; curr = curr->next)
    m[curr] = new Node(curr->data);

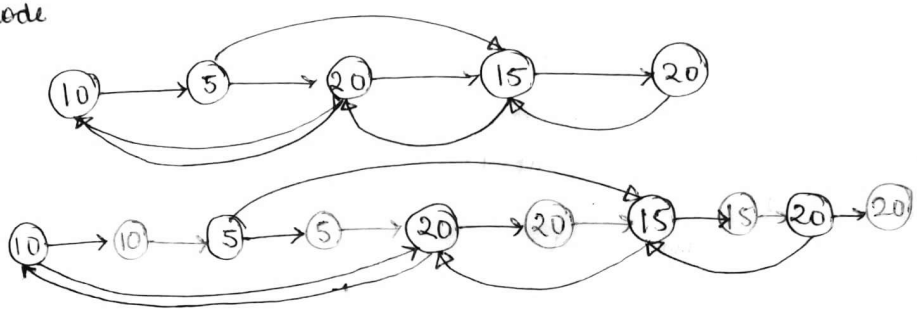
③ for(curr = h1; curr != NULL; curr = curr->next)
{
    clonecurr = m.~~find~~[curr]
    clonecurr->next = m.~~find~~[curr->next]
    clonecurr->random = m[curr->random]
}

Node *head2 = m[head]
return head2

}

Method - 2 O(1) Aux space

- We will add another node in front of each node



Creation of copy nodes

```
for (curr = hL ; curr != NULL;) {
    next = curr -> next;
    curr -> next = new Node (curr -> data)
    curr -> next -> next = next;
    curr = next;
}
```

Linking the random pointers

```
for (curr = hL ; curr != NULL ; curr = curr -> next -> next)
{
    curr -> next -> random =
        (curr -> random != NULL) ?
            curr -> random -> next : NULL;
}
```

Step-3 extracting the given nodes from the linked list.

LRU Cache Design

Least Recently Used

Really close to CPU and less access time

Small in size so we need to have efficient utilization

- It used the concept of - temporal Locality
  - The item which is just accessed has a very high probability to be accessed again.
- we keep the k recently used items in cache and we remove the least recent items.

Ex :-

Cache - size - 4

Reference sequence - 10, 20, 10, 30, 40, 50, 30, 40, 60, 30

miss → not present

hit → present

Expected Cache Behaviour :-

| 10 | | | |

10 ; Miss

| 20 | 10 | | |

20 ; Miss

| 10 | 20 | | |

10 ; Hit (10 is recently used i.e comes in front)

| 30 | 10 | 20 | |

30 ; Miss (inserted in front)

| 40 | 30 | 10 | 20 |

40 ; Miss (inserted in front)

| 50 | 40 | 30 | 10 |

50 ; Miss (Remove the LRU item)

| 30 | 50 | 40 | 10 |

30 ; Hit

| 40 | 30 | 50 | 10 |

40 ; Hit

| 60 | 40 | 30 | 50 |

60 ; Miss

| 30 | 60 | 40 | 50 |

30 ; Hit

Simple Implementation – Array

time complexity — O(n) of both hit and miss

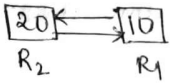
n → capacity of cache.

Effective approach

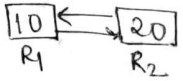
- Use of hashing for quick access and insert
- use of doubly linked list ~~to~~ to maintain order

10 : Miss (10, R_1)

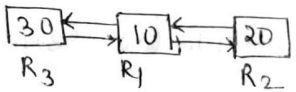
20 ; Miss (10, R_1) (20, R_2)



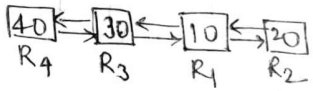
10 ; ~~Mo~~ Hit NO change



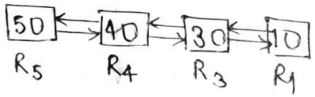
30 ; Miss (10, R_1) (20, R_2) (30, R_3)



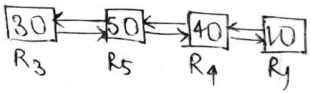
40 ; Miss (40, R_4) (10, R_1) (20, R_2) (30, R_3)



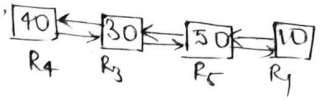
50 ; Miss (10, R_1) (30, R_3) (40, R_4) (50, R_5)



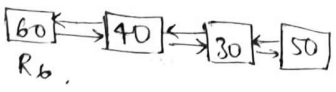
30 ; Hit NO change



40 ; Hit NO change



60 ; Miss (40, R_4), (30, R_3) (50, R_5), (60, R_6)



30 ; Hit NO change

- using the tail pointer we can remove the last node, when cache the full.
- Removal of any middle node is possible in O(1) time in doubly linked list.
- Doubly linked list works as queue, but supports additional operation of moving the middle item (Hit) in the front.

```
Refer(x) {
    Look for x in the Hash table.
    (a) If found (Hit), find the reference
        of the node. Move it to front.
    (b) If Not found (Miss).
        (i) Insert a new node at front
            of DLL
        (ii) Insert an entry into the H.T
}.
```

Merge Sorted linked list

I/P ; a : 10 → 20 → 30 b : 5 → 25

O/P ; 5 → 10 → 20 → 25 → 30

We maintain four pointers

① a → curr element in A
② b → curr element in B
③ head → head of resultant
④ tail → tail of res.

```
Node *sortedMerge (Node *h1, Node *h2) {
    if (h1 == NULL) return h2;
    if (h2 == NULL) return h1;
    Node *head = NULL, *tail = NULL;
    Node *a = h1, *b = h2;
```

```
if (a->next
        if(a->data <= b->data) {
            head = a
            a = a->next;
        } else {
            head = b
            b = b->next;
        }
        tail = head;
        while( a != NULL && b != NULL){
            if (a->data <= b->data){
                tail -> next = a ;
                a = a -> next;
            }
            else {
                tail -> next = b;
                b = b -> next;
            }
        }
        if (a == NULL) tail ->next = b;
        else tail -> next = a;
        return head;
    }
```

Auxiliary space - O(1)
Time complexity - O(m+n)

Palindrome Linked list

I/P - (10) → (5) → (5) → (10)
O/P - Yes

I/P - (G) → (F) → (F)
O/P - Yes

I/P - (G) → (E) → (E) → (K)
O/P - NO.

Naive solution

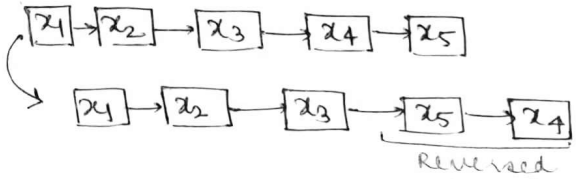
- Use a stack and push all the elements into it
- Run second loop and check if it is equal to curr → data.

```
bool isPalendrome ( Node * head){
    stack<int> st ;
    for (Node * curr = head ; curr != NULL ;
                              curr = curr -> next){

        st. push (curr -> data) ;
    }.
    for (Node * curr = head ; curr != NULL ;
                              curr = curr -> next){

        if (st.top () != curr -> data)
            return false ;
    }
    return true;
}.
```

O(n) with two traversal
O(n) space

Efficient Approach

- Find the middle point and Reverse the second part
- Now one by one compare first half and reversed second half.



compare one by one.

x_1 with x_5
x_2 with x_4
⋮ ⋮

```
bool isPalindrome(Node *root){
    if(head == NULL) return true;
    Node * slow = head, *fast = head;
    while(fast->next != NULL &&
          fast->next->next != NULL){

        slow = slow->next;
        fast = fast->next->next;
    }.
    Node *rev = reverseList(slow->next);
    Node * curr = head;
    while(rev != NULL){
        if(rev->data != curr->data)
            return false;
        rev = rev->next;
        curr = curr->next;
    }.
    return true;
}
```