Linked List (Background & Introduction)

C++: `int arr[100];` } stack

`int arr[n];` // where n is an integer

variable size

`int *arr = new int[n];`

`vector<int> v;`

dynamically size Heap

problems with Arrays:-

① either size is fixed and pre-allocated (in both fixed and variable sized arrays). OR the worst case insertion at the end is $\Theta(n)$.

② Insertion in the middle (or beginning) is costly.

③ deletion from the middle (or beginning) is costly.

④ Implementation of data structures like queue, deque is complex with arrays.

⇒① How to implement Round Robin scheduling?

P0	P1	P2	P3	P4	P5
10	5	3	15	10	8

Token time = 5

P1	P2	P3	P4	P5	P0
5	3	15	10	8	5

P2	P3	P4	P5	P0
3	15	10	8	5

P3	P4	P5	P0
15	10	8	5

Implementation is difficult with arrays: deleting an element & adding at end of ready queue

(2) Given a sequence of items, whenever we see an item x in the sequence we need to replace it with 5 instances of another item y.

ip: d e a x q x x p y

op: d e a y y y y y q x y y y y y p y

(3) We have multiple sorted sequences and we merge them frequently.

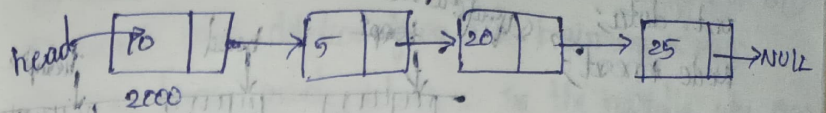
Linked list doesn't require extra space for merging

Sequences[] = { [5, 10, 15, 20], [1, 12, 18], [3, 30, 40], [100, 200] }

merge(0,1) ⇒ Sequences[] = { [1,5,10,12,15,18,20], [3,30,40], [100,200] }

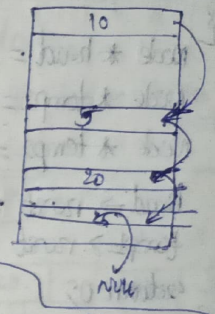
merge(1,2) ⇒ Sequences[] = { [1,5,10,12,15,18,20], [3,30,40,100,200] }

Linked List (Background & Introduction)

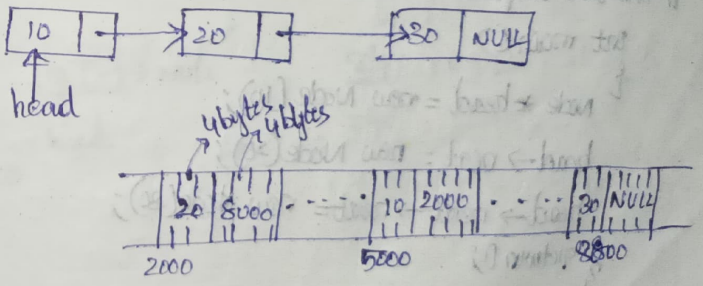


⇒ The idea is to drop the contiguous memory requirements so that insertions, deletions, can efficiently happens at the middle also.

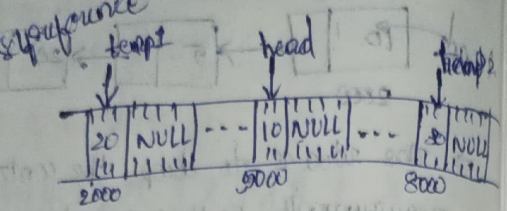
⇒ And no need to pre-allocate the space [no extra space]



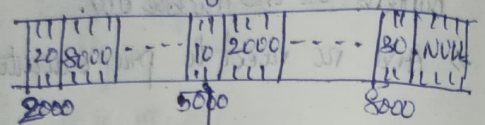
Linked List implementation in C++



```
struct Node {
    int data;
    Node *next;
    Node(int x)
    {
        data = x;
        next = NULL;
    }
};
int main()
{
    Node * head = new Node(10);
    Node * temp1 = new Node(20);
    Node * temp2 = new Node(30);
    head -> next = temp1;
    temp1 -> next = temp2;
    return 0;
}
```



After linking



head

// Shorter implementation

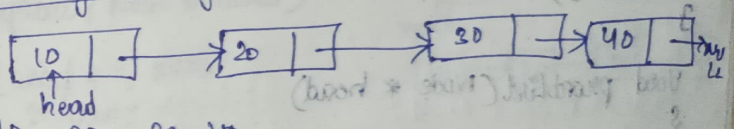
```
int main()
{
    Node * head = new Node(10);
    head -> next = new Node(20);
    head -> next -> next = new Node(30);
    return 0;
}
```

Applications of linked list

1. worst case insertion at the end and begin on $\theta(1)$
2. worst case deletion from the beginning is $\theta(1)$.
3. Insertions and deletions in the middle are $\theta(1)$ if we have reference to the previous node.
4. Round Robin Implementation.
5. merging two sorted linked lists is faster than arrays
6. Implementation of simple memory manager where we need to link free blocks.
7. easier implementation of queue & dequie data structures

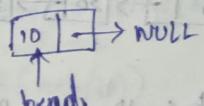
Traversing a single linked list in C++:

I/P:



O/P: 10 20 30 40

I/P:

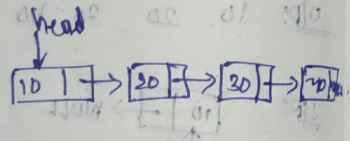


O/P: 10

I/P: NULL

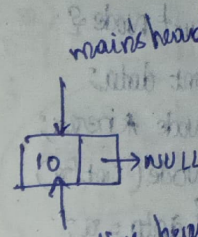
O/P:

```
struct Node {
    int data;
    Node * next;
    Node(int x) {
        data = x;
        next = NULL;
    }
};
int main()
{
    Node * head = new Node(10);
    head -> next = new Node(20);
    head -> next -> next = new Node(30);
    head -> next -> next -> next = new Node(40);
    printList(head);
return 0;
}
Void printList(Node * head)
{
    node * curr = head
    while (curr != NULL)
    {
        cout << (curr -> data);
        curr = curr -> next;
    }
}
```



Op: 10 20 30 40

```
int main()
{
    Node * head = new Node(10);
    printList(head);
    printList(head);
}
void printList (Node *head) {
{
    while (head != NULL)
    {
        cout << (head -> data) << " ";
        head = head -> next;
    }
}
```



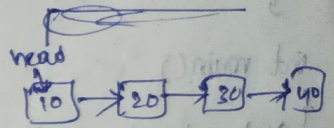
this head is limited to this function only.

T.C = O(n)
A.S = O(1)

Op: 10 10

Recursive Display of Linked List:

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



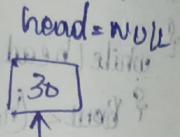

```
rprint (ref(10))
    -> print(10)
    -> rprint (ref(20))
        -> print(20)
        -> rprint (ref(30))
            -> print(30)
            -> rprint (ref(40))
                -> print(40)
                -> rprint(NULL).
```

T.C = O(n) — no of nodes
A.S = O(n) — n+1 fn calls in call stack

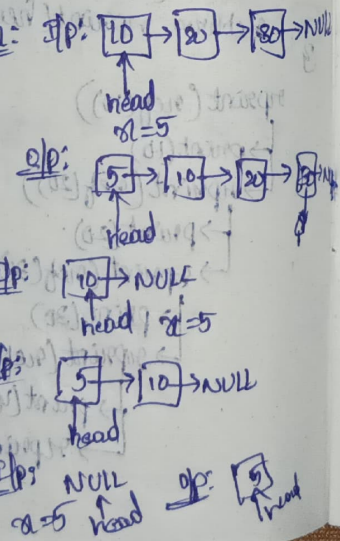
Insert at Begin of Singly linked list :-

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

T.C = O(1)

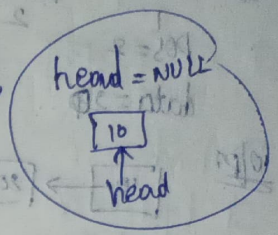


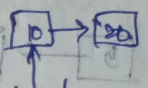

```
Node *insertBegin (Node *head, int x)
{
    Node *temp = new Node(x);
    temp -> next = head;
    return temp;
}

int main()
{
    Node *head = NULL;
    head = insertBegin(head, 30);
    head = insertBegin(head, 20);
    head = insertBegin(head, 10);
    return 0;
}
```

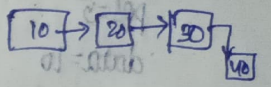


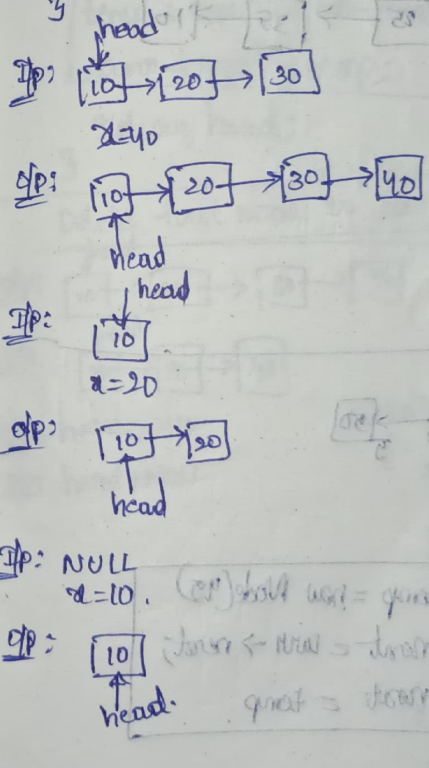
Insert at end of Singly linked list :-

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









Insert at given position in singly linked list :-

I/p: 10 → 30 → 50 → 70 (1, 2, 3, 4)

pos = 2

data = 20

O/p: 10 → 20 → 30 → 50 → 70

I/p: 5 → 15 → 25 → 35 (1, 2, 3, 4)

pos = 5

data = 10

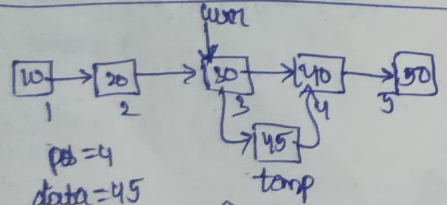
O/p: 5 → 15 → 25 → 35 → 10

I/p: 10 → 20 (1, 2)

pos = 4

data = 25

O/p: 10 → 20



pos = 4

data = 45

traverse curr to (pos-2) times

```
Node *temp = new Node(45)
temp->next = curr->next;
curr->next = temp
```

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

Delete first node in singly linked list :

I/p: 10 → 20 → 30 → 40 (head)

O/p: 20 → 30 → 40 (head)

I/p: 10 (head)

O/p: head = NULL

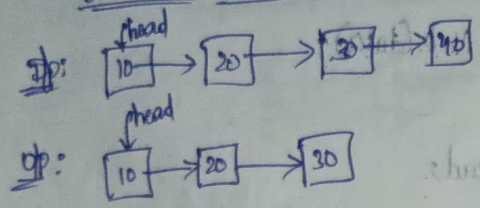
I/p: head = NULL

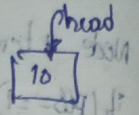
O/p: head = NULL

```
Node *delHead(Node *Head)
{
  if(head==NULL)
    return NULL;
  else
  {
    Node *temp = head->next;
    delete head;
  }
  return temp;
}
```

Memory deallocation purpose

Delete Last Node in Singly Linked List :-



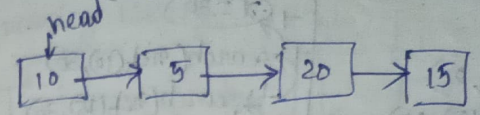


I/p: head = NULL
O/p: head = NULL

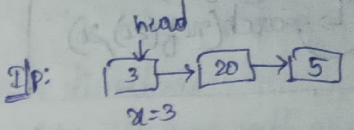
⇒
```
Node * delTail (Node * head)
{
  if (head == NULL) return NULL;
  if (head -> next == NULL)
  {
    delete head;
    return NULL;
  }
  Node * curr = head;
  while (curr -> next -> next != NULL)
      curr = curr -> next;
  delete (curr -> next);
  curr -> next = NULL;
  return head;
}
```

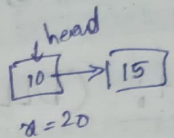
T.C = O(n)

Search in Linked List (Iterative and Recursive)



O/p: 3





O/p: 1

O/p: -1

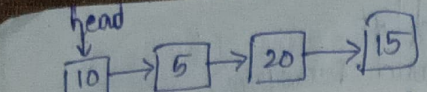
⇒
```
int search_LL (Node * head, int x)
{
  int pos = 1;
  Node * curr = head;
  while (curr != NULL)
  {
    if (curr -> data == x)
      return pos;
    else
    {
      pos++;
      curr = curr -> next;
    }
  }
  return -1;
}
```

T.C = O(n)
A.S = O(1)

Recursive:

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

T.C = O(n)
A.S = O(n)



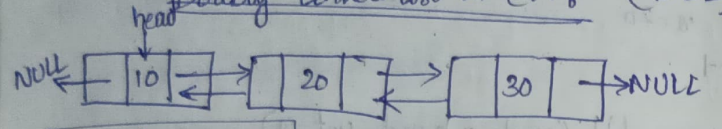
x = 20

search (ref(10), 20)
↳ search (ref(5), 20)
↳ search (ref(20), 20)

x = 30

search (ref(10), 30)
↳ search (ref(5), 30)
↳ search (ref(20), 30)
↳ search (ref(15), 30)

Doubly Linked list in c++:- (DLL)

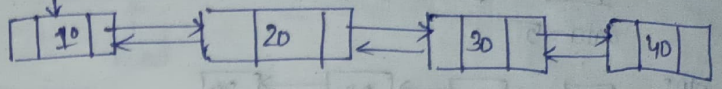



```
struct Node {
    int data;
    Node *prev;
    Node *next;
    Node (int d)
    {
        data = d;
        prev = NULL;
        next = NULL;
    }
};
```

```
int main ()
{
    Node *head = new Node(10);
    Node *temp1 = new Node(20);
    Node *temp2 = new Node(30);
    head -> next = temp1;
    temp1 -> prev = head;
    temp1 -> next = temp2;
    temp2 -> prev = temp1;
}
```

Singly vs Doubly linked lists :-

eg: Browsing History.

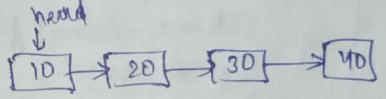


Advantages:

1) can be traversed in both directions.
2) A given delete a node in O(1) time with given reference/pointer to it.
3) Insert / delete before a given node
4) Insert / delete from both ends in O(1) time by maintaining tail.

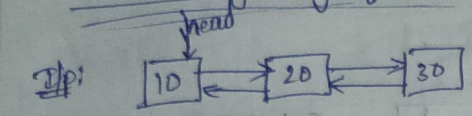
Disadvantages:

1) extra space for prev.
2) code becomes complex

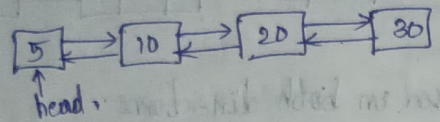


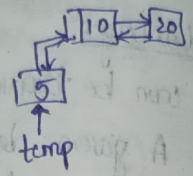
singly linked list Ex.

Insert at Beginning of DLL :-

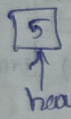
I/p: 

data = 5

O/p: 



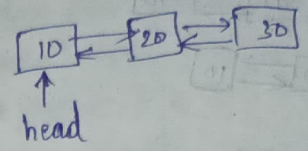
I/p: head = NULL (OR null)
data = 5

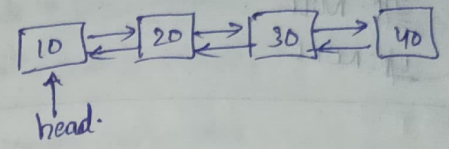
O/p: 


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

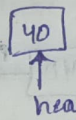
```
Node * insertBegin (Node *head,
                    int data)
{
  Node * temp = new Node(data);
  temp -> next = head;
  if(head != NULL)
    head->prev = temp;
  return temp;
}
```

Insert at End of DLL :-

I/p: 

O/p: 

I/p: if head = NULL (OR null)
data = 40

O/p: 


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

Reverse a Doubly linked list:-

I/p: 10 ⇄ 20 ⇄ 30 ⇄ 40

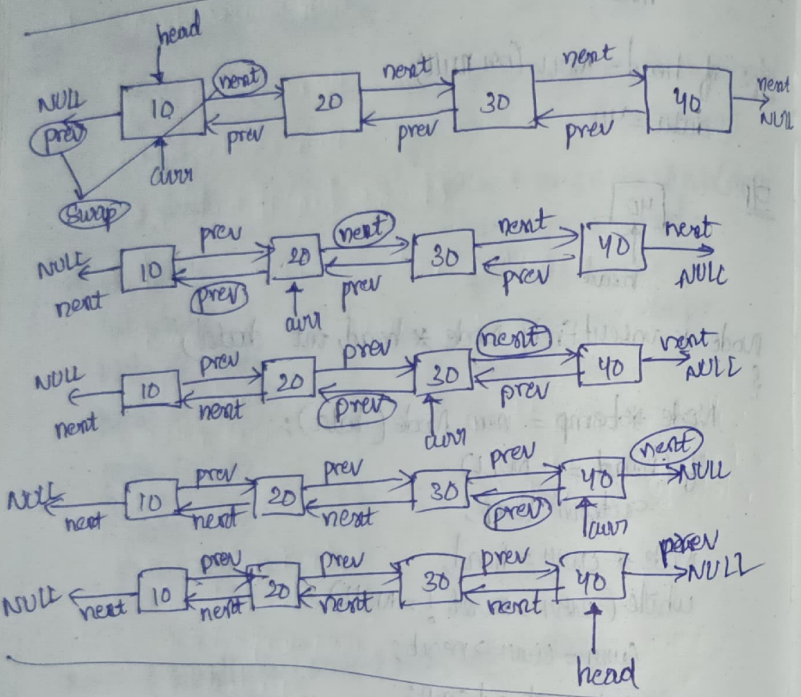
O/p: 40 ⇄ 30 ⇄ 20 ⇄ 10

I/p: 10

O/p: 10

I/p: NULL

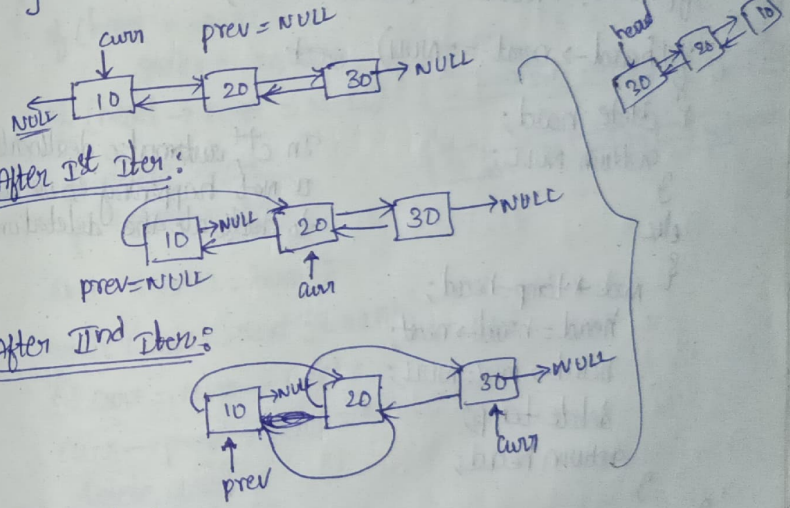
O/p: NULL




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

T.C = O(n)
A.S = O(1)

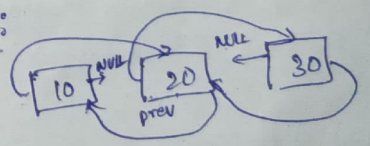
Swapping



After Ist Iter:

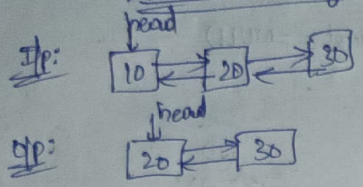
After IInd Iter:

After IIIrd Iter:



curr = NULL

Delete head of DLL:



I/p: 

o/p: head = NULL (or) null

T.C = O(1)

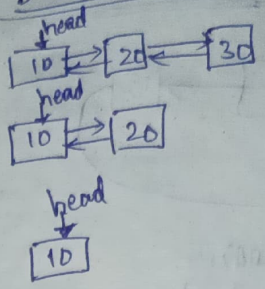
I/p: head = NULL (or) null

o/p: head = NULL (or) null

```
Node *delHeadDll(node *head)
{
  if(head==NULL) return NULL;
  if(head->next==NULL)
  { delete head;
    return NULL;
  }
  else
  { Node *temp=head;
    head=head->next;
    head->prev=NULL;
    delete temp;
    return head;
  }
}
```

In C++ automatic deallocation is not happening so we have to deallocate the deleted ones

Delete Last Node of DLL:



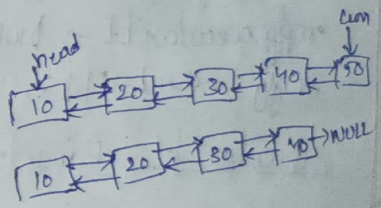
o/p: head = NULL or null

T.C = O(n)
A.S = O(1)

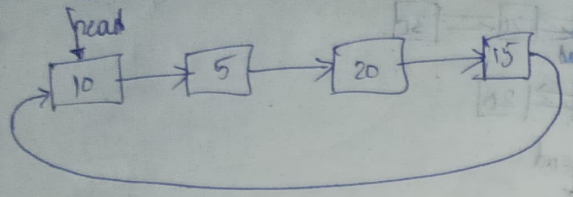
I/p: head = NULL or null

o/p: head = NULL or null

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



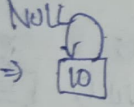
Circular linked list in C++:



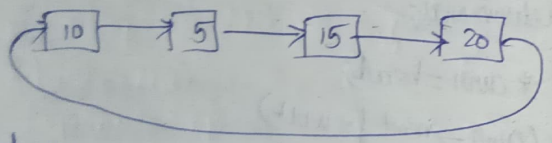

```
struct Node{
    int data;
    Node * next;
    Node(int d)
    {
        data = d;
        next = NULL;
    }
};
```

```
int main()
{
    Node *head = new Node(10);
    head->next = new Node(5);
    head->next->next = new Node(20);
    head->next->next->next = new Node(15);
    head->next->next->next->next=head;
    return 0;
}
```

empty circular L.L ⇒ NULL

single circular L.L ⇒ 

Circular L.L Advantages & disadvantages:-

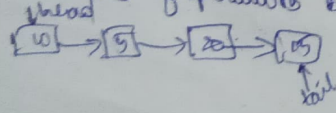


Advantages:

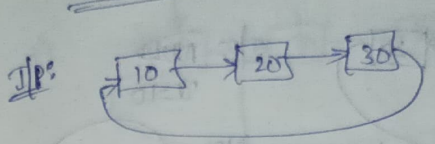
1) We can traverse the whole list from any Node.
2) Implementation of algorithms like round robin
3) We can insert at the beginning and end by just keeping maintaining one tail reference/pointer.

disadvantages:

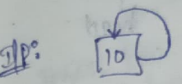
1) Implementations of operations become complex.



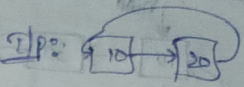
Circular L.L Traversal



clp: 10 20 30



clp: 10

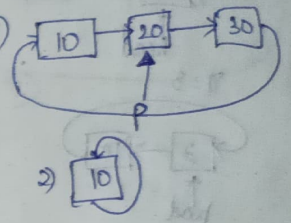


clp: 10 20

clp: NULL

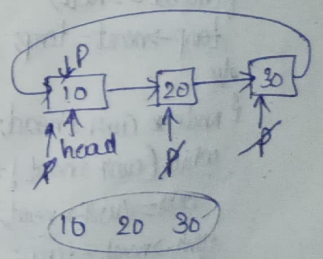
method-1 (for loop):-

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

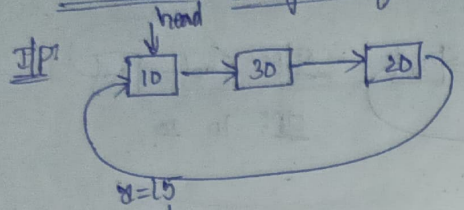


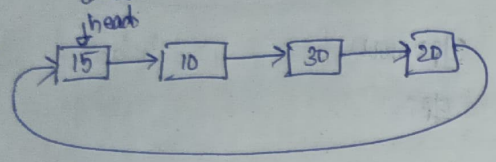
method-2 (do while loop):-

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



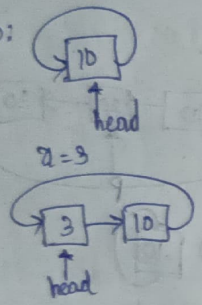
Insert at Begin of C.L.L

I/P: 

O/P: 

I/P: NULL
x=10

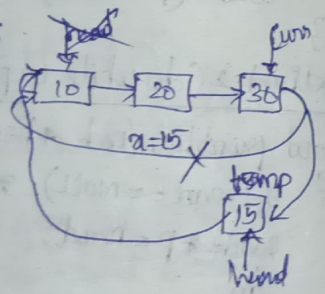
O/P: 

I/P: 

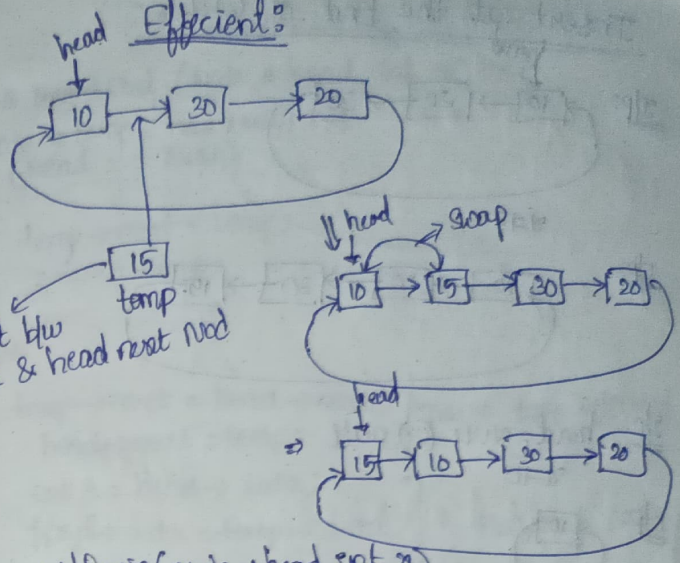
Naive:

T.C = O(n)

```
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;
}
```



Efficient:

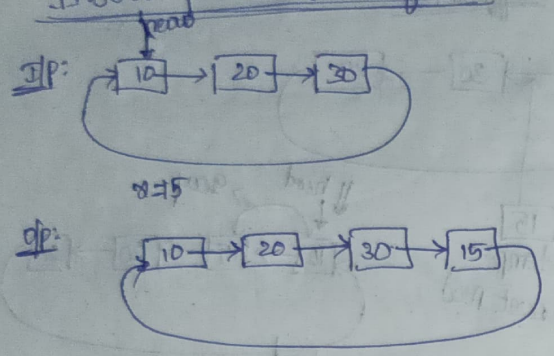
1) 

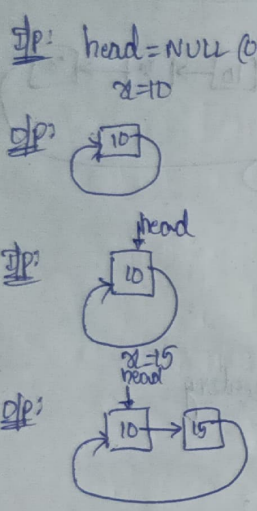

```
Node * insertBegin (node *head, int x)
{ Node *temp = new Node (x);
  if (head==NULL)
  { temp->next = temp;
    return temp;
  else
  { temp->next = head->next;
    head->next = temp;
    int t = head->data;
    head->data = temp->data;
    temp->data = t;
  }
  return head;
}
```

Inserting

another efficient solution:
maintain tail pointer & pass it as an argument.

Insert at the End of C.L.L:-

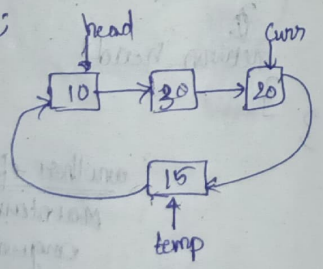




Naive:

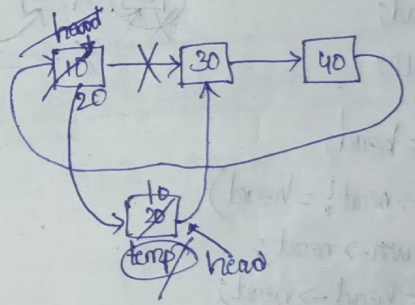
TC = O(n)

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

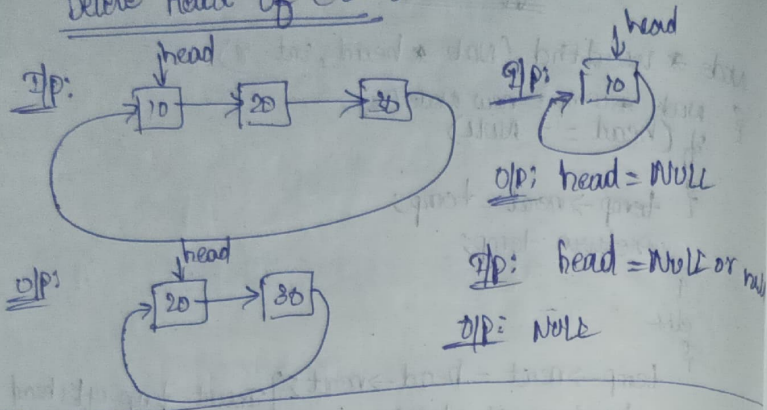


efficient:

```
Node * insertEnd (Node *head, int x)
{ Node *temp = new Node(x);
  if (head == NULL)
  { temp->next = temp;
    return temp;
  }
  else
  { temp->next = head->next;
    head->next = temp;          ] Insert temp after head
    int t = head->data;
    head->data = temp->data;    ] swapping
    temp->data = t;
    return temp; } // temp is now new head.
  }
```



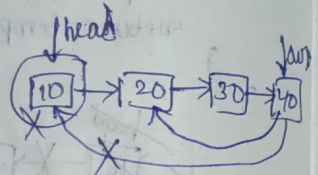
Delete Head Of C.L.L:-



Naive Solution:

T.C = O(n) (n = no of Nodes)

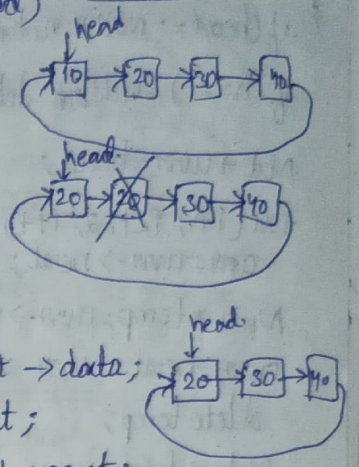
```
Node * delHead (Node * head)
{ if (head == NULL) return NULL;
  if (head -> next == head)
  { delete head;
    return NULL;
  }
  Node * curr = head;
  while (curr -> next != head)
      curr = curr -> next;
  curr -> next = head -> next;
  delete head; // bcz C++ doesn't have automatic garbage collection.
  return (curr -> next);
}
```



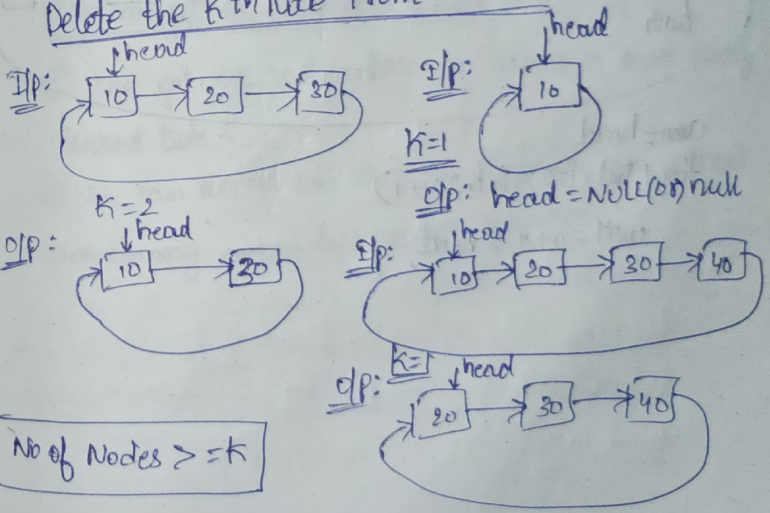
Efficient Solution:

T.C = O(1)

```
Node * delHead (Node * head)
{ if (head == NULL)
    return NULL;
  if (head -> next == NULL)
  { delete head;
    return NULL;
  }
  head -> data = head -> next -> data;
  Node * temp = head -> next;
  head -> next = head -> next -> next;
  delete temp;
  return head;
}
```

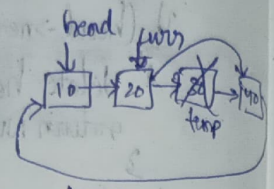


Delete the kth Node from a CLL:-

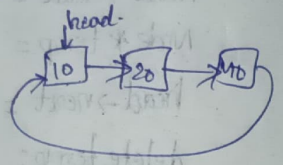


No of Nodes >= K

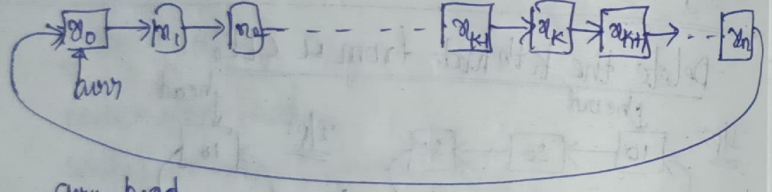
```
Node *deleteKth (node *head, int k)
{ if (head == NULL) return head;
  if (k == 1) return delHead(head);

  Node *curr = head;
  for (i=0; i<k-2; i++)
    curr = curr -> next;
  Node *temp = curr -> next;
  curr -> next = curr -> next -> next;
  delete temp;
  return head;
}
```



K=3

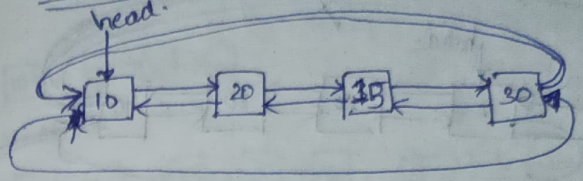


when k is not 1:




```
curr = head
for (int i=0; i<k-2; i++)
    curr = curr -> next;
```

Circular Doubly linked list:-



① previous of head is last Node.

② Next of last Node is head.

An empty circular Doubly linked list:-

head = NULL (or) null

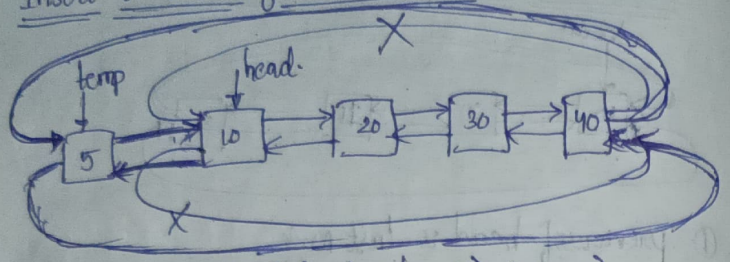
A Single Node circular Doubly linked list:

Advantages of circular DLL:-

① we get all advantages of circular and doubly linked lists.

② we can access last node in constant time without maintaining extra tail pointer reference.

Insert at Head of Circular D.L.L:




```
Node * InsertAtHead (Node *head, int x)
{ 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;
}
```

⇒ null, x=5

Insert at End of Circular D.L.L:

It is same like Insert at Head of Circular D.L.L but instead of return temp, we have to return head only.

Sorted Insert in a Linked List:

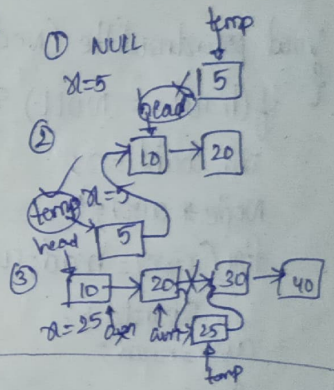
I/p: 10 → 20 → 30 → 40, x=25
O/p: 10 → 20 → 25 → 30 → 40

I/p: 10 → 25, x=5
O/p: 5 → 10 → 25

I/p: 10 → 20, x=30
O/p: 10 → 20 → 30

I/p: NULL, x=10
O/p: 10

```
Node * SortedInsert (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;
}
```



Middle of a linked list:- (if no of nodes is even take the second node)

I/p: 10 → 5 → 20 → 15 → 25

O/p: 20

I/p: 10 → 5 → 20 → 15 → 25 → 30

O/p: 15

I/p: NULL

O/p:

I/p: 10 → 20

O/p: 20

Naive solution:

```
Void printmiddle (Node *head)
{
  if (head == NULL) return;
  int count = 0;
  Node * curr;
  for (curr = head; curr != NULL; curr = curr->next)
      count++;
  curr = head;
  for (int i=0; i<(count/2); i++)
      curr = curr->next;
  cout << curr -> data;
}
```

tc-1: 10 → 20 → 30 → 40 → 50

count = 5

we need to move curr 2 times

tc-2: 10 → 20 → 30 → 40 → 50 → 60

count = 6

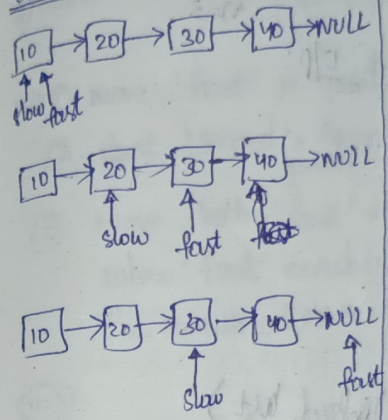
we need to move curr 3 times

Efficient solution:

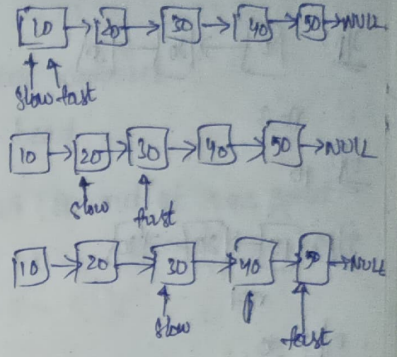
Idea: Using slow and fast pointers

when fast == NULL => return slow -> data

Even Nodes



Odd Nodes




```
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);
}
```

find n-th Node from end of Linked list:

I/p: 10 → 20 → 30 → 40 → 50

n=2

O/p: 40

I/p: 10 → 20 → 30

n=3

O/p: 10

I/p: 10 → 20 → 30

n=1

O/p: 30

I/p: 10 → 20

n=3

O/p:

method-1: (using length of linked list)

```
void printNthFromEnd(Node *head, int n)
{
    if (head == NULL)
    int len=0;
    for (Node *curr=head; curr!=NULL; curr=curr->next)
        len++;
    if (len<n)
        return;
    for (int i=1; i<len-n+1; i++)
        curr = curr->next;
    cout << (curr->data) << " ";
}
```

10 → 20 → 30 → 40 → 50 → 60

len=5, n=2

40

method-2: (using Two pointers)

n=2

10 → 20 → 30 → 40 → 50 → 60 → 70

second, first

1) move 'first' pointer to n positions a head.

2) start 'second' pointer from head

⇒ ① move 'first' n positions ahead.

② start 'second' pointer from head

③ move both 'first' and 'second' at same speed. when 'first' reaches NULL, 'second' reaches the required node.

(n=3)

10 → 20 → 30 → 40 → 50 → 60 → 70 → NULL

second, first

10 → 20 → 30 → 40 → 50 → 60 → 70 → NULL

second, first

10 → 20 → 30 → 40 → 50 → 60 → 70 → NULL

second, first

10 → 20 → 30 → 40 → 50 → 60 → 70 → NULL

second, first

10 → 20 → 30 → 40 → 50 → 60 → 70 → NULL

second, first

```
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 → 40 → head

O/p: 40 → 30 → 20 → 10 → head

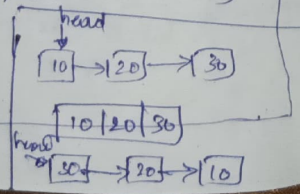
I/p: 10 → head O/p: 10 → head

I/p: NULL O/p: NULL

Naive solution:

Two Traversals
AS = O(n)

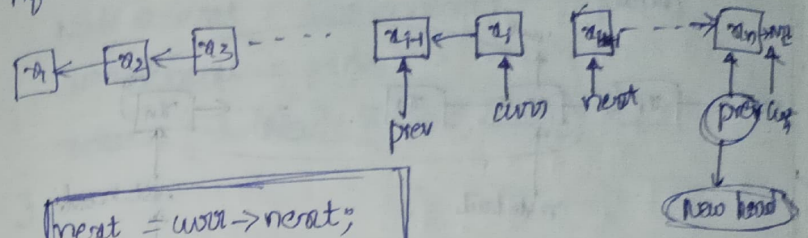
```
Node *revList(Node *head)
{ vector<int> arr;
  for (Node *curr = head; curr != NULL; curr = curr->next)
  { arr.push_back(curr->data); }
  for (node *curr = head; curr != NULL; curr = curr->next)
  {
    curr->data = arr.back();
    arr.pop_back();
  }
  return head;
}
```



Efficient Solution

I/p: $x_1 \to x_2 \to x_3 \cdots \to x_{i-1} \to x_i \to x_{i+1} \to \cdots \to x_n$

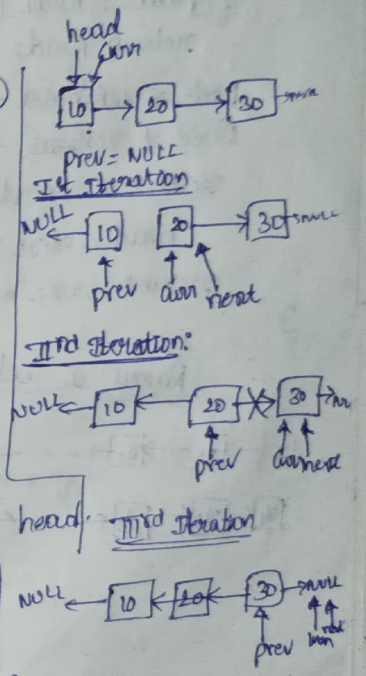
After reversing Nodes from x_1 to x_{i-1}



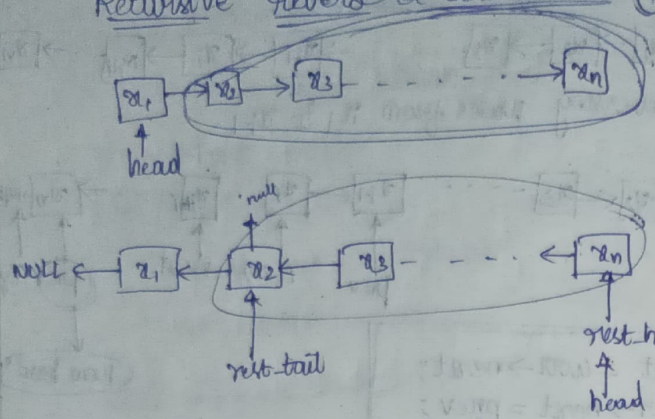

```
next = curr->next;
curr->next = prev;
prev = curr;
curr = next;
```

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

T.C = O(n)
A.S = O(1)

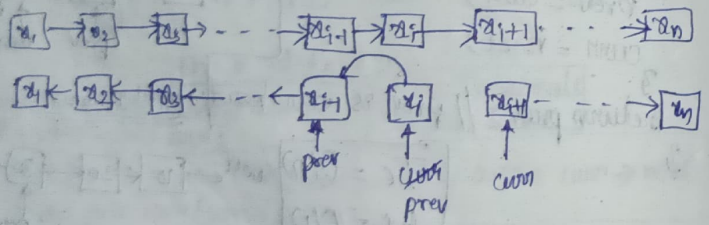


Recursive reverse a Linked list : (method-1)

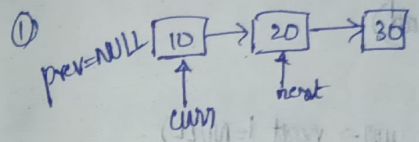


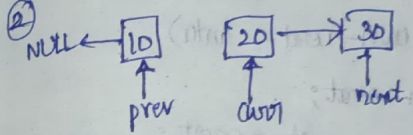

```
Node * recRevLL(Node * head)
{
  if(head == NULL || head->next == NULL)
    return head;
  Node * rest_head = recRevLL(head->next);
  Node * rest_tail = head->next;
  rest_tail->next = head;
  head->next = NULL;
  return rest_head;
}
```

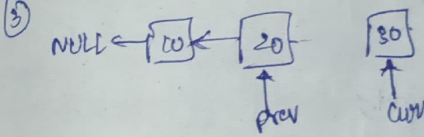
Reverse a linked list [Recursive-2]

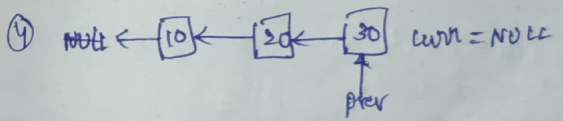



```
Node * recRevLL(Node * curr, Node * prev)
{
  if(curr == NULL) return prev;
  Node * next = curr->next;
  curr->next = prev;
  return recRevLL(next, curr);
}
```









Remove Duplicates from a Sorted List:-

I/p: 10 → 20 → 20 → 30 → 30 → 30

o/p: 10 → 20 → 30

I/p: 5 → 10 → 15 → 20

o/p: 5 → 10 → 15 → 20

I/p: NULL

o/p: NULL

```
Void RemoveDups (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 a linked list in groups:-

I/p: head → 10 → 20 → 30 → 40 → 50 → 60, K=3

o/p: 30 → 20 → 10 → 60 → 50 → 40 (head)

I/p: 10 → 20 → 30 → 40 → 50, K=3

o/p: 30 → 20 → 10 → 50 → 40 (head)

I/p: 10 → 20 → 30, K=4

o/p: 30 → 20 → 10 (head)

Recursive solution:

I/p: head → 10 → 20 → 30 → 40 → 50 → 60 → 70

o/p: 30 → 20 → 10 → 60 → 50 → 40 → 70 (head)

After Reverse of first k elements:-

10 → 20 → 30 (head, prev) 40 → 50 → 60 → 70 (curr, next)

60 → 50 → 40 → 70 (last head)

(kth node) prev is going to be new head

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

If k ≤ no of nodes in L.L then kth node is the head of our L.L

Recursive Solution:

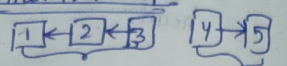
T.C = O(n)
A.S = O(n/k)

```
Node *reverseK(Node *head, int k)
{
  Node *curr = head, *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 *next_head = reverseK(next, k);
    head->next = next_head;
  }
  return prev; // prev is new head
}
```

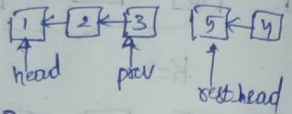
I/P: 1 → 2 → 3 → 4 → 5, K=3

O/P: 3 → 2 → 1 → 5 → 4

After first Loop:



After recursive call:



Iterative Solution

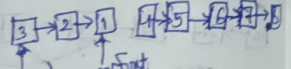
T.C = O(n)
A.S = O(1)

```
Node *reverseK(Node *head, int k)
{
  Node *curr = head, *prevFirst = NULL;
  bool isFirstPass = true;
  while (curr != NULL)
  {
    Node *first = curr, *prev = NULL;
    int 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;
}
```

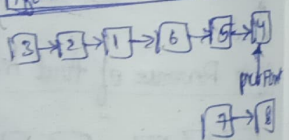
K=3

1 → 2 → 3 → 4 → 5 → 6 → 7 → 8

After I st Iteration:

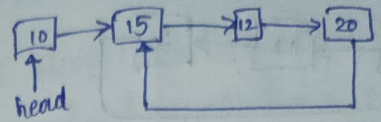


After II nd Iteration:



Detect Loop in a Linked List:

I/P:

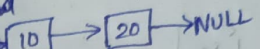


O/P: yes

I/P: head = NULL

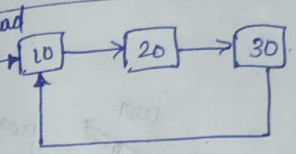
O/P: No

I/P:



O/P: No

I/P:



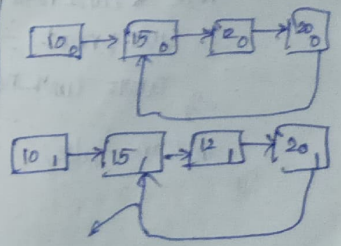
O/P: yes

Method-1 Naive: T.C = O(n²)

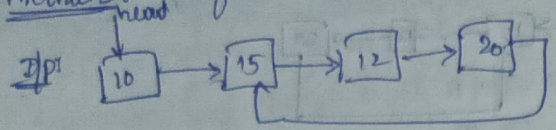
for every node we have to traverse all the nodes before it and going to check whether address of any of the nodes is going to be same as the next of curr node

Method-2: If modification to linked list structure are allowed => T.C = O(n)

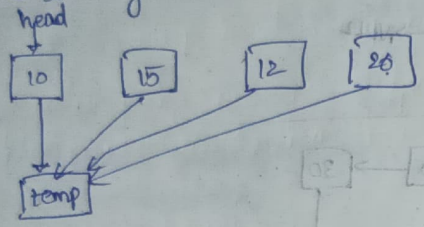
```
struct Node {
  int data;
  Node *next;
  bool visited;
  Node(int d)
  { data = d;
    next = NULL;
  } visited = false;
};
```

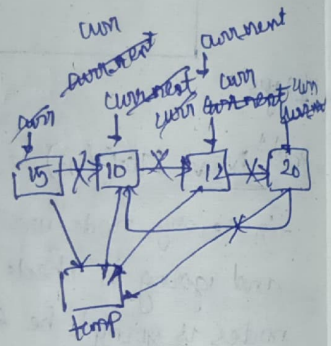


method-3: Modification to linked list pointer/references



Idea: Traverse the L-L, for every node to check if its next is pointing to dummy node. on that case return true, otherwise move to the next node & modify next nodes next as dummy node.





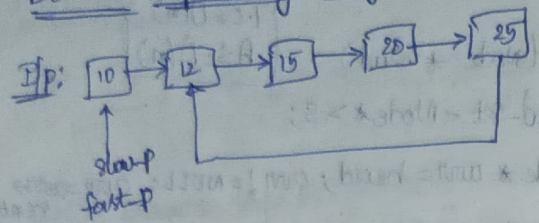

```
⇒ bool isLoop(Node *head)
{
    Node *temp = new Node;
    Node *curr = head;
    while (curr != NULL)
    { if (curr->next == NULL)
            return false;
      if (curr->next == temp)
            return true;
      Node *curr_next = curr->next;
        curr->next = temp;
        curr = curr_next;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

method-4 (storing)

T.C = O(n)
A.S = O(n)

```
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 using Floyd's cycle Detection:



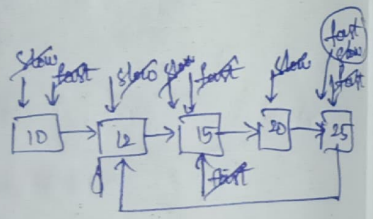
O/p: yes

Algorithm:

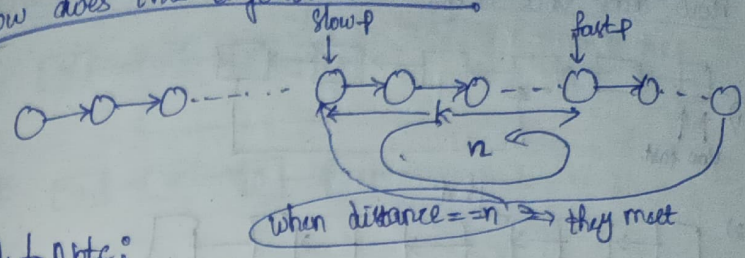
① Initialize slow_p = head, fast_p = head;

② move slow_p by one and fast_p by two.
If these pointers meet, then there is a loop.

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



How does this algorithm work?



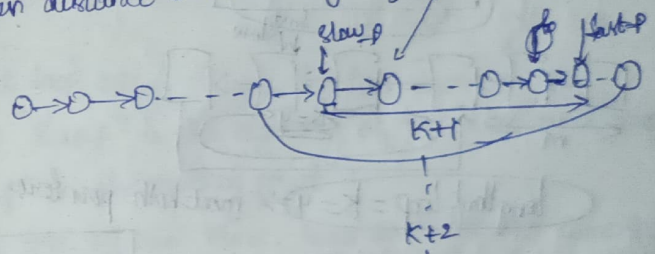
points to Note:

→ fast_p will enter into the loop before (or at the same time as slow_p)

→ Let fast_p be k distance ahead of slow_p when slow_p enters the loop where k ≥ 0

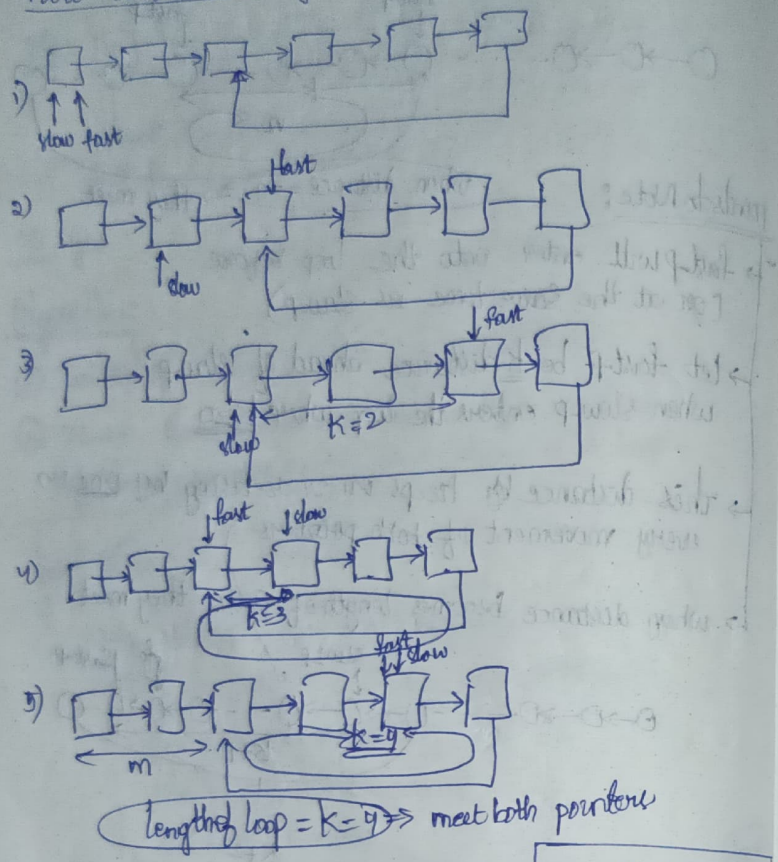
→ this distance keeps on increasing by one in every movement of both pointers.

→ when distance becomes length of cycle they meet.



when only we increment the value by one, we reach n, if we increment the value by two, we never reach n, so always move fast by two. & increment distance by 2.

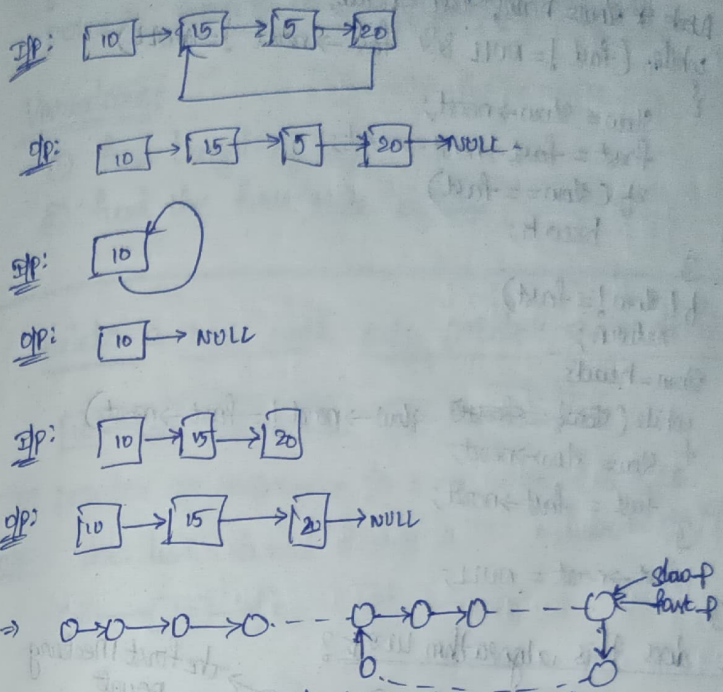
How does this algorithm (Floyds) works?



length of loop = K = 4 ⇒ meet both pointers

T.C = O(n) + O(m)
= O(m+n)

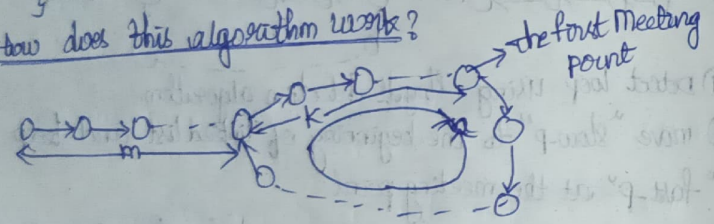
Detect and Remove loop in a linked list:



① Detect loop using floyds detection algorithm.
② move "slow-p" to the beginning of linked list and keep "fast-p" at the meeting point.
③ Now one by one move slow and fast (at same speed) the point where they meet now is the first node of the loop.

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

How does this algorithm work?



Before the 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 a multiple of n

x → No of iterations made by slow before the first meeting point

y → " " " fast " "

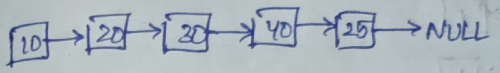
If (m+k) is a multiple of n, then second meeting point is going to be the first node of loop.

Variations:

① find length of loop.

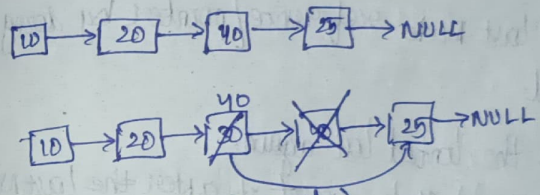
② find the first Node of loop.

Delete a node with only pointer given to it:-



I/P: pointer or reference to node with value 30.

O/P: the list should change to → Never be the last node




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

Segregate Even and Odd Nodes

I/p: 17 → 15 → 8 → 12 → 5

O/p: 8 → 12 → 17 → 15 → 5

I/p: 8 → 12 → 10

O/p: 8 → 12 → 10

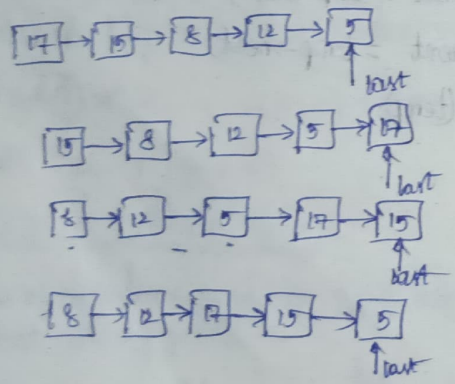
I/p: 3 → 5 → 1

O/p: 3 → 5 → 1

Naive solution (Two Traversals required):

1) find the last node reference/pointer by doing a traversal.

2) Traverse the linked list again. for every odd node, insert it after the last node. & make the newly inserted node as the new last node.



Idea for the One Traversal solution:-

we maintain 4 variables.

es → Reference/pointer to start of the even sublist

ee → " " " end " " "

os → " " " start " odd sublist

oe → " " " end " " "

All of the above are initialized as NULL/None

while Traversing

a) If current node is even, we insert it after ee and update ee. Also update es if this is the first node.

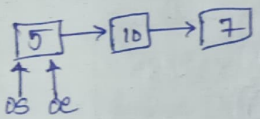
b) similar to (a) for odd nodes.

After the loop connect the two lists

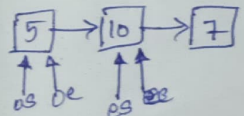
ee->next = os

⇒ 5 → 10 → 7

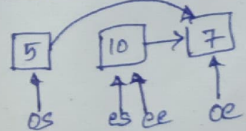
After I st traversal:



After II nd traversal:-

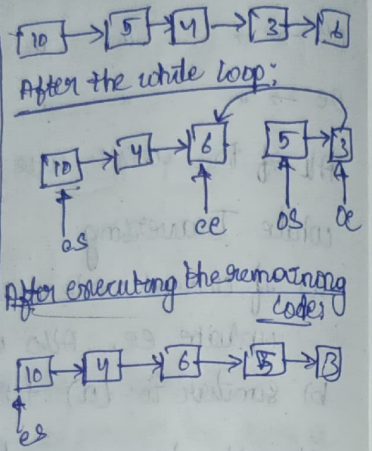


After III rd traversal:-

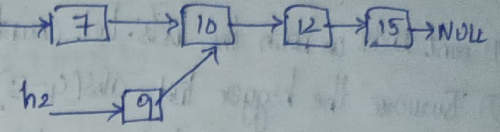


```
Node * Segregate (Node * head)
{
  Node *es = NULL, *ee = NULL , *os = NULL, *oe = NULL;
  for (Node * curr = head; curr != NULL; curr = curr->next)
  {
    x = curr->data;
    if (x % 2 == 0)
    {
      if (es == NULL)
        es = ee = curr;
      else
      {
        ee->next = curr;
        ee = ee->next;
      }
    }
    else
    {
      if (os == NULL)
        os = oe = curr;
      else
      {
        oe->next = curr;
        oe = oe->next;
      }
    }
  }
  if (os == NULL || es == NULL)
    return head;
  ee->next = os;
  oe->next = NULL;
  return es;
}
```

T.C = O(n)
A.S = O(1)



Intersection point of two linked lists



o/p: 10

```
int getIntersection (Node * h1, Node * h2)
{
}
```

hs{ add(5), add(8), add(7), add(10), add(12), add(15) }

method-1:

1) Create an empty hash set, hs.
2) Traverse the first list and put all of its nodes into the hs.
3) Traverse the second list and look for every node in hs. As soon as we find a node present in hs, we return value of it.

T.C = O(m)
A.S = O(m)

⇒ O(n)

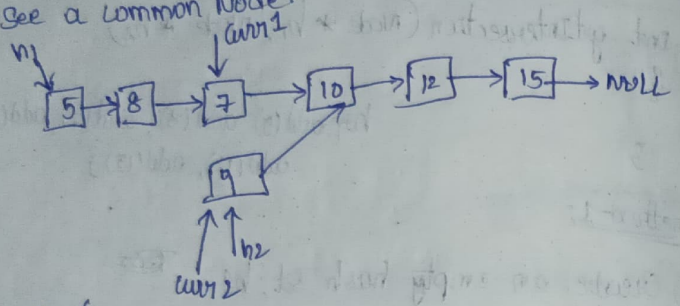
T.C = O(m+n)
A.S = O(m)

method 2:

① Count Nodes in both the lists. Let count be c_1 and c_2.

② Traverse the bigger list abs(c_1-c_2) times.

③ Traverse both the lists simultaneously until we see a common Node.



$c_1 = 6$

$c_2 = 4$

$d = abs(c_1 - c_2) = (6-4) = 2$

Move curr1 'd' time ahead in higher length list

```
while (curr1 != NULL && curr2 != NULL)
{
    if (curr1 == curr2)
        return (curr1->data);
    curr1 = curr1->next;
    curr2 = curr2->next;
}
return -1;
```

pairwise Swap Nodes:-

ip: 1 → 2 → 3 → 4

op: 2 → 1 → 4 → 3

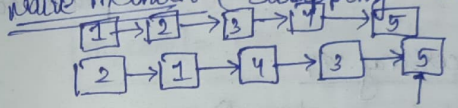
ip: 1 → 2 → 3 → 4 → 5

op: 2 → 1 → 4 → 3 → 5

ip: 1

op: 1

Naive method: (Swapping Data)



Run a loop while we have at least one node ahead.

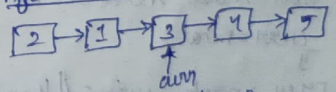
a) Swap data of current node with its next node.

b) move current two nodes ahead.

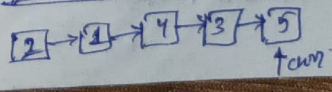
T.C = O(n)
A.S = O(1)

```
Node *pairwiseSwap(Node *head)
{
    Node *curr = head;
    while (curr != NULL && curr->next != NULL)
    {
        Swap(curr->data, curr->next->data);
        curr = curr->next->next;
    }
    return head;
}
```

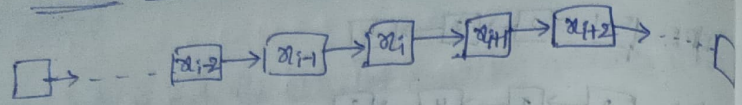
After I st Iteration:



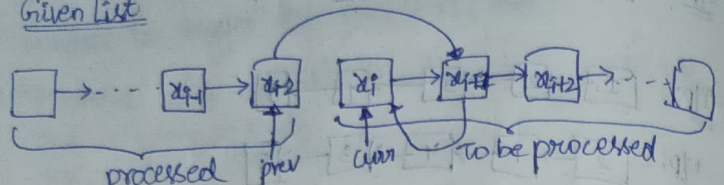
After II nd Iteration:



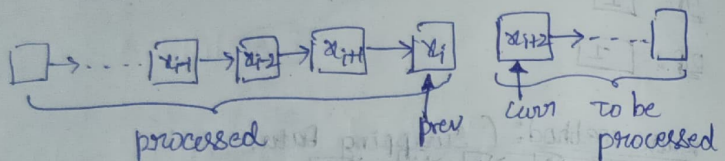
Idea for the Efficient Solution:



Given list



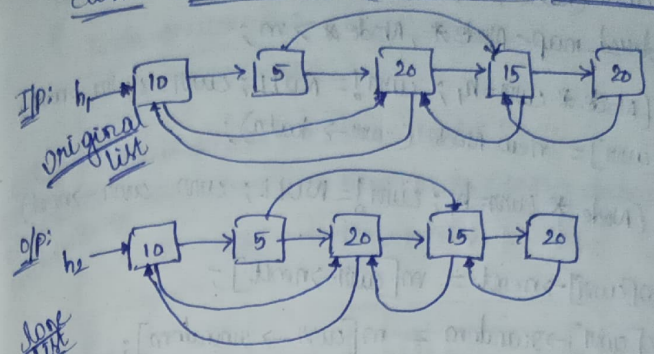
Before we reach x_i



After processing x_i

```
Node *pairwiseSwap(Node *head)
{
  if(head==NULL || head->next==NULL)
    return head;
  Node * curr = head->next->next;
  Node * prev = head;
  head = head->next;
  head->next = prev;
  while (curr!=NULL && curr->next!=NULL)
  {prev->next = curr->next;
   prev = curr;
   Node *next = curr->next->next;
   curr->next->next = curr;
   curr=next;
  }
  prev->next = curr; // Handle both even and odd cases
  return head;
}
```

Clone a linked list with Random connections

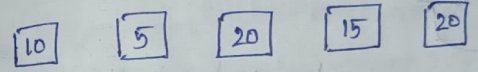


Method-1: (Use Hashing):

1) create an unordered_map (m)

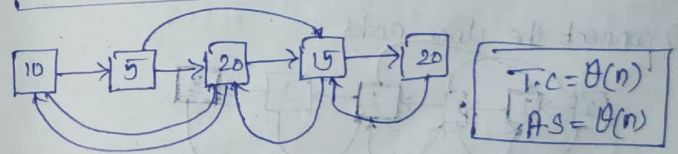
2) Do following for every node curr in the given list.

```
m[curr] = new Node(curr->data);
```



3) Traverse the given list again and do following for every node curr.

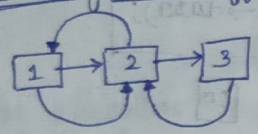
```
m[curr]->next = m[curr->next];
m[curr]->random = m[curr->random];
```



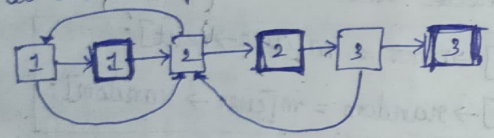
4) Return m[h1]

```
Node *clone(Node *h1)
{
    unordered_map<Node*, Node*> m;
    for (Node *curr=h1; curr!=NULL; curr=curr->next)
        m[curr] = new Node(curr->data);
    for (Node *curr=h1; curr!=NULL; curr=curr->next)
    {
        m[curr]->next = m[curr->next];
        m[curr]->random = m[curr->random];
    }
    return m[h1];
}
```

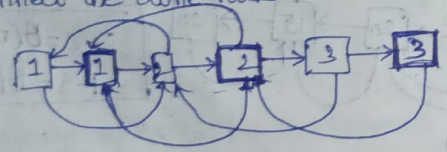
Idea for the efficient solution:-



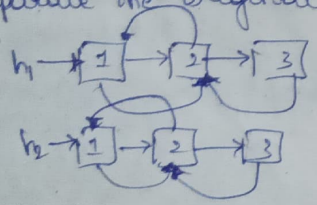
1) Create clone nodes and insert on the given list at alternate positions



2) connect the clone nodes



3) Separate the original and clone nodes



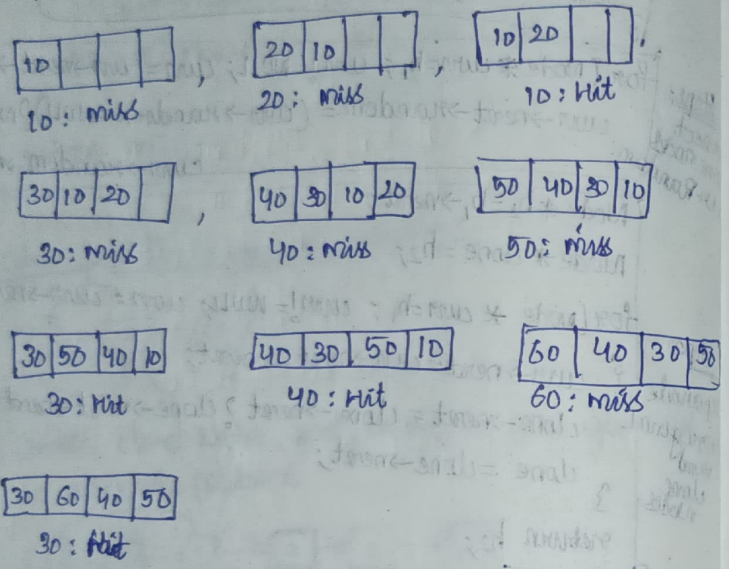

```
Node *clone(Node *h1)
{
    Node *curr = h1;
    // step 1: Insert clone nodes Alternatively
    while (curr != NULL)
    {
        Node *next = curr->next;
        curr->next = new Node(curr->data);
        curr->next->next = next;
        curr = next;
    }
    // step 2: connect clone nodes with random.
    for (Node *curr=h1; curr!=NULL; curr=curr->next->next)
        curr->next->random = (curr->random==NULL)?NULL:
                             curr->random->next;
    // step 3: separate original and clone nodes
    Node *h2 = h1->next;
    Node *clone = h2;
    for (Node *curr=h1; curr!=NULL; curr=curr->next)
    {
        curr->next = curr->next->next;
        clone->next = clone->next ? clone->next->next : NULL;
        clone = clone->next;
    }
    return h2;
}
```

LRU cache Design

cache size : 4

Reference Sequence : 10, 20, 10, 30, 40, 50, 30, 40, 60, 30

Expected cache Behaviour:



Simple Implementation : Array

T.C

Hit : O(n)

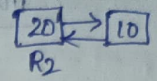
Miss : O(n)

where 'n' is capacity of cache

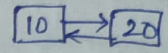
Efficient Implementation

10 : miss (10, R_1)

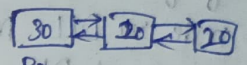
20 : miss (10, R_1) (20, R_2)



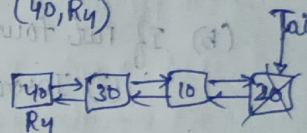
10 : Hit No change



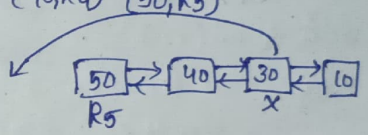
30 : miss (10, R_1) (20, R_2) (30, R_3)



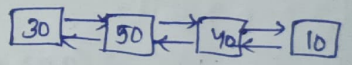
40 : miss (10, R_1) ~~(20, R_2)~~ (30, R_3) (40, R_4)



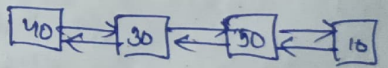
50 : miss (10, R_1) (30, R_3) (40, R_4) (50, R_5)



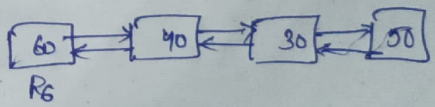
30 : Hit No change



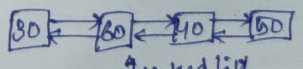
40 : Hit No change



60 : miss (30, R_3) (40, R_4) (50, R_5) (60, R_6)



30 : Hit No change



Hash Table

Linked list

Hit

T.C:-
Hit = O(1)
miss = O(1).

HT: Hash Table
DLL: Doubly LL

Refer(x)
{
Look for x in HT.

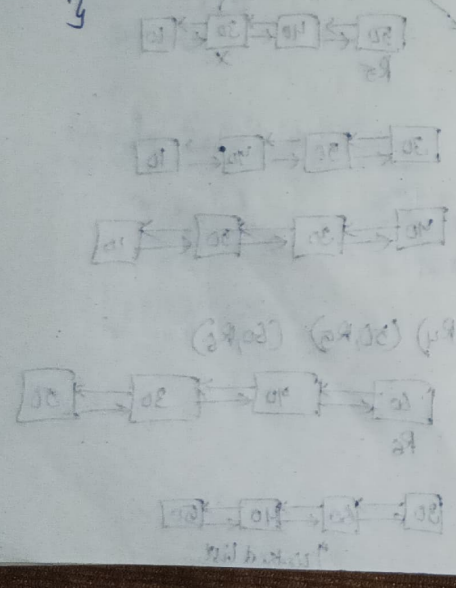
(a) If found (Hit), find the reference of the node in DLL. Move the node to the front of DLL.

(b) If not found (miss)

(i) Insert a new node at the front of DLL.

(ii) Insert an entry into the HT.

}

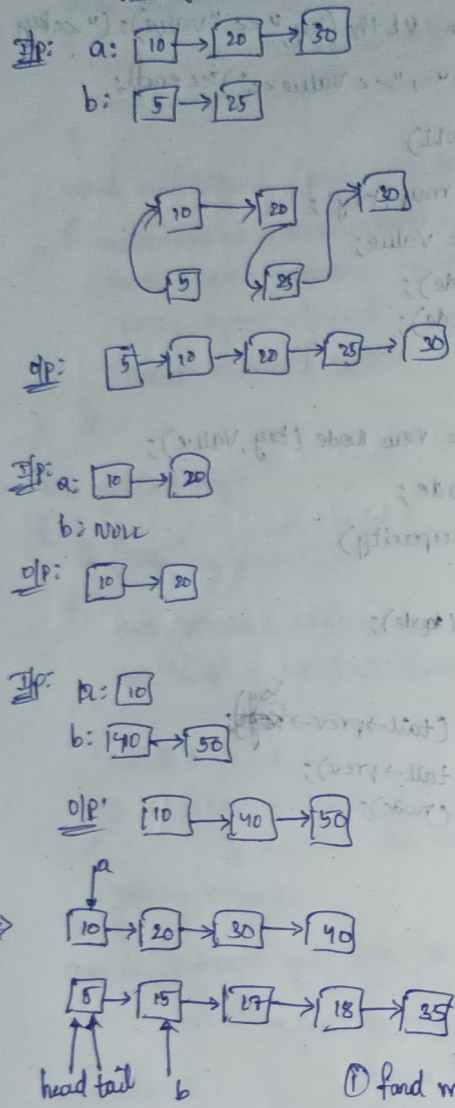


```
class Node
struct Node
{ int key;
  int value;
  Node * prev, * next;
  Node (int k, int v)
  { key = k;
    value = v;
    prev = NULL;
    next = NULL;
  }
};

struct LRUCache
{ unordered_map < int, Node *> map;
  int capacity, count;
  Node * head, * tail;
  LRUCache(int c)
  { capacity = c;
    head = new Node (0,0);
    tail = new Node (0,0);
    head -> next = tail;
    tail -> prev = head;
    head -> prev = NULL;
    tail -> next = NULL;
    count = 0;
  }
```

```
void deleteNode (Node * node)
{ node->prev->next = node->next;
  node->next->prev = node->prev;
}
void addToHead (Node *node)
{ node->next = head->next;
  node->next->prev = node
  node->prev = head;
  head->next = node;
}

int get(int key)
{
  if (map[key]!=NULL)
  {
    Node *node = map[key];
    int result = node->value;
    deleteNode (node);
    addToHead(node);
    cout<<"Got the value: "<<
        result<<" for the key: "<<key<<"\n";
    return result;
  }
  cout<<"did not get any value"<<
      " for the key: "<<key<<endl;
  return -1;
}
```

```
void set (int key, int value)
{ cout<<"going to set the (key,"<<"value): ("<<key
      <<" ,"<<value<<")"<<endl;

  if (map[key]!=NULL)
  { Node * node = map[key];
    node->value = value;
    deleteNode(node);
    addToHead (node);
  }
  else
  { Node * node = new Node (key, value);
    map[key] = node;
    if (count < capacity)
    { count++;
      addToHead(node);
    }
    else
    { map.erase (tail->prev->key);
      deleteNode(tail->prev);
      addToHead(node);
    }
  }
}
};
int main()
{ LRUCache cache(2);
  cache.set(1,10);
  cache.get (2,20);
  cout<<"value for the key: 1 is "<<cache.get(1)<<endl;
  cache.set (3,30);
  cout<<"value for the key: 2 is "<<cache.get(2)<<endl;
  cache.set (4,40);
  cout<<"value for the key: 3 is "<<cache.get(3)<<endl;
}
```

Merge two sorted Linked Lists



① find min from both LL heads & point tail & head to min head LL.

② move pointer, one link ahead in new head L.L

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

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

T.C = O(m+n)
A.S = O(1)

palindrome Linked List

I/p: R → A → D → A → R

o/p: yes.

I/p: G → F → G

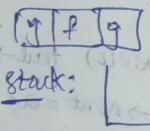
o/p: yes

I/p: Q → E → E → K

o/p: NO

Naive method:

```
bool ispalindrome(node *head)
{
  stack<char> 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;
    st.pop();
  }
  return true;
}
```

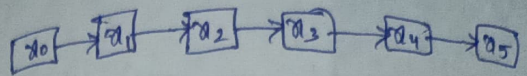


stack:

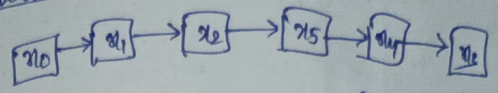
After first loop:

After second loop:

efficient approach



find the second half & reverse it.



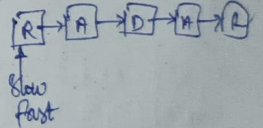
now one by one compare first half & reversed second half

x_0 with x_5

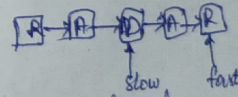
x_1 with x_4

x_2 with x_3

```
bool ispalindrome(node *head)
{ 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;
}
```



After first loop:



After reversing second