Linked List

'24 MAY 22 WEDNESDAY
21st Week 143-223

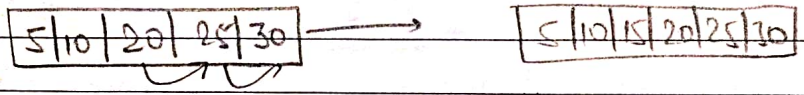
Dynamic sized array ⟶ Python → List
C++ → vector
Array list ← Java

Whenever they reach there limit, they double the size in their typical implementation and they copy the previous element to the new allocated double spaced.

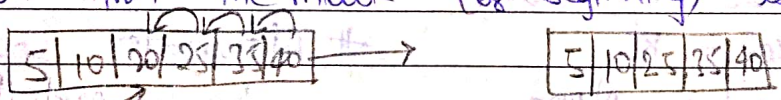
Problems with arrays :-

(1) Either size is fixed and pre-allocated (in both fixed and variable size arrays) OR the worst case insertion at the end is O(n)

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



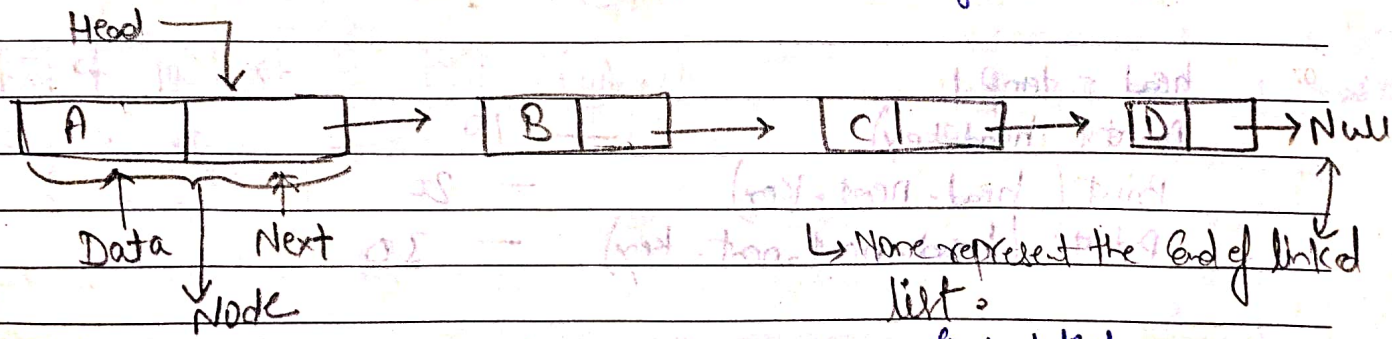
(3) Deletion from the middle (or beginning) is costly.



(4) Implementation of data structures like queue and deque is complex with arrays

(5) Contiguous memory is required.

→ Linked list is a linear data structure. Unlike arrays, linked list elements are not stored at a contiguous location; the elements are linked using pointers. They include a series of connected nodes. Here, each node stores the data and the address of the next node.



↳ None represent the end of linked list.

Arrays	Linked list
→ Cache friendliness	→ Not cache friendly.
→ Random access	→ Sequential access. [No contiguous memory]
	→ Dynamic nature.
	→ Efficient insertion & deletion.

MAY 23 '24 THURSDAY

21st Week 144-222

S	M	T	W	T	F	S
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

MAY 2024

	Insertion			deletion			Search			Access		
	begin	b/w	end	begin	b/w	End	begin	b/w	End	begin	b/w	End
Array	O(n)	O(n)	O(1)	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)	O(1)
Linked List	O(1)	O(n)	O(n)	O(1)	O(n)	O(n)	O(1)	O(n)	O(n)	O(1)	O(n)	O(n)

#2 Implementation of SLL in Py.

data = key

```
class Node:
    def __init__(self, k):
        self.key = k
        self.next = None

temp1 = Node(10)
temp2 = Node(20)
temp3 = Node(30)

temp1.Next = temp2
temp2.next = temp3

head = temp1
print(head.key)
print(head.next.key)
print(head.next.next.key)
```

```
# Alternate shorter way
head = Node(10)
head.next = Node(20)
head.next.next = Node(30)
```

Step (I) { 10 | NONE ; 20 | NONE ; 30 | NONE

Step (II) { 10 | → 20 | NONE

Step (III) { 10 → 20 → 30 | NONE

— 10 — 20 — 30

head

Vote should be weighed, not counted

Application of linked list

1. worst case insertion at the end and begin are O(1).
2. worst case deletion from the beginning is O(1).
3. Insertion and deletion in the middle are O(1) if we have reference to the previous Node.
4. Round Robin Implementation.
5. moging two sorted linked list is faster than array.
6. Implementation of simple memory manager where we need to link free blocks.
7. Easier implementation of Queue and Deque data structure.
8. Implementation of graph.
9. Dynamic memory allocation.
10. Music player — Previous & next songs.
11. Previous and Next Page of browser.
12. Image viewer — Previous & Next image.

} Real world Application

Traversing a linked list.

i/p — [10] → [20] → [30] → [40] → NONE

o/p — 10 20 30 40

```
Class Node:
    def __init__(self, key):
        self.key = key
        self.next = NONE

def printlist(head):
    curr = head
    while curr != NONE:
        print(curr.key)
        curr = currenext
```

TC = O(n)
SC = O(1)

```
head = Node(10)
head.next = Node(20)
head.next.next = Node(15)
head.next.next.next = Node(30)
Printlist(head)
```

(OR)

```
l = []
curr = head
while curr != NONE:
    l.append(curr.next)
return l.
```

MAY
25
SATURDAY
'24
21st Week
146-220

S	M	T	W	T	F	S
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

MAY 2024

Count No. of Nodes

M-1

```
l = []
curr = head_node
while curr != NONE:
    l.append(curr.next)
    curr = curr.next
return len(l)
```

M-2 else use count (cnt+=1)

Sum of Nodes

M-1

```
def Sum(head):
    curr = head
    l = 0
    while curr != NONE:
        l = l + (curr.data)
        curr = curr.next
    return l
```

Search in linked list and return Position :-

i/p - 10 → 15 → 20 → 15 → NONE, n = 20
(1) (2) (3) (4)

o/p - (3)

i/p - 10 → 15 → NONE, n = 20

o/p - (-1)

```
class Node:
    def __init__(self, k):
        self.key = key
        self.Next = NONE
```

M-1

```
def search(head, n):
    curr = head
    pos = 1
    while curr != NONE:
        if curr.key == n:
            return pos
        pos = pos + 1
        curr = curr.next
    return -1
```

TC - O(n)

M-2

```
def search(head, n):
    l = []
    curr = head
    while curr != None:
        l.append(curr.data)
        curr = curr.next
    if n in l:
        return l
    else:
        return -1
```

Insert at the beginning of linked list:

i/p — [10] → [20] → [30] → NONE

o/p — [5] → [10] → [20] → [30] → NONE

```
def insert(head, key):
    temp = Node(key)
    temp.next = head
    return temp.
```

→ adding element in temprory created Node and mapping that temprory Node to head.

TC = O(1)

Insert at the end of linked list.

i/p — [10] → [20] → [30] → NONE   Key = 35

o/p — [10] → [20] → [30] → [35] → NONE.

```
def insert(head, key):
    if head == NONE:
        return Node(key)
    curr = head
    while curr.next != NONE:
        curr = curr.next
    curr.next = Node(key)
    return head.
```

} → for Empty L L.

→ expecting last Node to be Null.

else { ... }

last Node आ जाये तो come out of loop & do just curr.next = NewNode(data)

TC = O(n)

Insert at given positions of linked list

i/p → [10] → [20] → [40] → [50] → [70] → Null
(positions 1, 2, 3, 4)

Pos = 2

data = 20

o/p → [10] → [20] → [30] → [40] → [50] → [70]

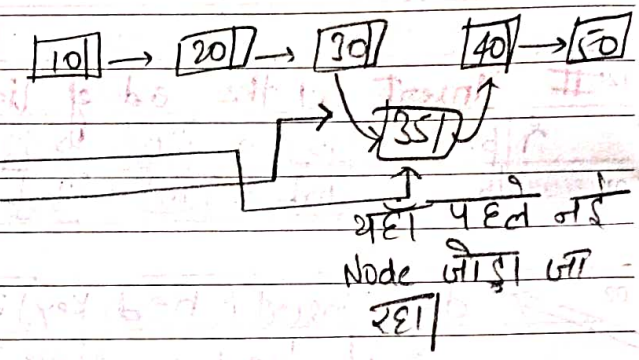
MAY
27
MONDAY
'24
22nd Week
148-218

S	M	T	W	T	F	S
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

MAY 2024

(k-2) times run the loop

```
def insert(head, data, pos):
    temp = Node(data)
    if pos == 1:                    # pos == 0
        temp.next = head
        return temp
    curr = head
    for i in range(pos-2):          # (1, pos)
        curr = curr.next
        if curr == None:
            return head.
    temp.next = curr.next
    curr.next = temp
    return head.
```



Delete First Node of linked list

i/p — 10 → 20 → 30 → 40
o/p — 20 → 30 → 40

i/p — 10
o/p — None

```
def del(head):
    if head == None:
        return None
    else:
        return head.next
```

temp = head
head = head → Next

Nth Node from the end of linked list using two pointers :—

As nth node from the end equals to (length - n+1)th node from the start, so the idea is to maintain two pointers starting from the head of the LL and move one pointer to the nth node from the start and then move both the pointer together until the pointer at the Nth position reach last node. Now the pointer which was moved later points at the nth node from the end of LL.

Delete last Node of linked list

i/p → 10 → 20 → 30 → 40 → NONE

o/p → 10 → 20 → 30 → NONE

We need to change 2nd last Node to None.

TC → O(n)

```
def del(head):
    if head == NONE:
        return NONE
    if head.next == NONE:
        return NONE
    curr = head
    while curr.next.next != NONE:
        curr = curr.next
    curr.next = NONE
    return head.
```

Handle Corner Case.

find the 2nd last node.

unlink the last node

Delete a node with pointer given to it.

Given a pointer to a node to be deleted in a singly linked list, how do you delete it

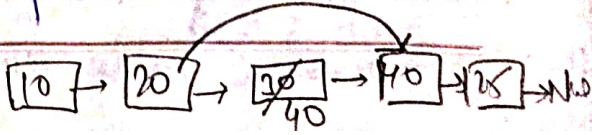
The simple solution is to traverse the linked list until you find the Node you want to delete. But this solutions requires a pointer to the head node which contradicts the problem statement.

The last solutions is to copy the data from the next node to be deleted and delete the Next node. something like this.

It is important to note that this approach will only work if it is guaranteed that the given pointer doesn't point to the last node.

```
def del(ptr):
    temp = ptr.next
    ptr.data = temp.data
    ptr.next = temp.next.
```

node.data = node.next.data
node.next = node.next.next



MAY
29
'24
22nd Week
150-216
WEDNESDAY

S	M	T	W	T	F	S
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

MAY 2024

Sorted inserted linked list in python.

i/p- 10 → 20 → 30 → 40 n = 25

o/p- 10 → 20 → 25 → 30 → 40

```
def Sort(head, n):
    temp = Node(n)
    if head == None:
        return temp
    elif n < head.key:
        temp.next = head
        return temp
    else:
        curr = head
        while curr.next != None and curr.next.key < n:
            curr = curr.next
        temp.next = curr.next
        curr.next = temp
        return head.
```

TC = O(n)

Middle of linked list.

eg i/p, 1 → 2 → 3 → 4 → 5 , o/p — 3

eg i/p, 1 → 2 → 3 → 4 → 5 → 6 , o/p — 4

M1 Traverse the whole linked list and count the No. of Nodes. Now traverse the list again till count/2 and return the Node at count/2.

M2 Traverse linked list using two pointers. Move one pointers by one place and the other pointer by two place. when the fast pointer reaches the end, the slow pointer will reaches the middle of linked list.

Rebit

JUN 2024

S	M	T	W	T	F	S
30						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29

M-1 →

```
def pm(ptr):
    if head == None:      # for no node
        return
    count = 0
    curr = head
    while curr:
        curr = curr.next
        count += 1
    curr = head
    for i in range(count//2):
        curr = curr.next
    print(curr.data)
```

TC = Θ(n)
→ required two traversals

M-2

```
def pm(ptr):
①  if head == None:
②      return
③  slow = head
④  fast = head
⑤  while fast != None and fast.next != None:
⑥      slow = slow.next
⑦      fast = fast.next.next
⑧  print(slow.data)
```

TC = Θ(n)

→ It required one traversal.

Nth Node from end of linked list

i/p = 1, 2, 3, 4, 5   N = 3
o/p = 3 → (length − n + 1)

move at same pace

i/p = 35 → 15 → 4 → 20 , N = 4 (second pointer ↓ 35, first pointer ↓ 20)
o/p = 35

make both pointer at same pace

M-1 using length of linked list

```
def nth_end(head, N):
    len = 0
    curr = head
    while curr:            # finding length
        curr = curr.next
        len += 1
    if len < n:
        return
    curr = head
    for i in range(1, len - n + 1):
        curr = curr.next
    print(curr.data)
```

TC = Θ(n)

M-2 two pointer approach.

first pointer जितना n दिया है वहाँ रखे। second start था then दोनों को same pace पे चलाए।

```
def nth_end(head, N):
    if head == None:
        return
    first = head
    for i in range(n):
        if first == None:
            return
        first = first.next
    second = head
    while first != None:
        second = second.next
        first = first.next
    print(second.data)
```

MAY 31 '24 FRIDAY

22nd Week 152-214

S	M	T	W	T	F	S
			1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

MAY 2024

Remove the duplicate from sorted List :-

i/p = 10 → 20 → 30 → 40 → 20 → 30 → 30

o/p = 10 → 20 → 30 → 40

→ Traverse a list from the head node. while traversing, Compare each node with its next Node. If the data of the next node is the same as the current node then delete next node. Before we delete a node, we need to store the next Pointer of node.

```
def dul(head):
    curr = head
    while curr != None and curr.Next != None:
        if curr.data == curr.next.data:
            curr.next = curr.next.next
        else:
            curr = curr.next
    return head
```

Reverse a linked list :-

i/p - 10 → 20 → 30 → 40 → None

o/p - 40 → 30 → 20 → 10 → None

M-1 Naive

```
def rev(head):
    stack = []
    curr = head
    while curr is not None:
        stack.append(curr.key)
        curr = curr.next
    curr = head
    while curr is not None:
        curr.key = stack.pop()
        curr = curr.next
    return head
```

{TC = O(n), SC = O(n)}

Initially stack = []

head → 10 → 20 → 30 → None

After 1st loop stack = [10, 20, 30]

After 2nd loop stack = []

30 → 20 → 10

empty stack

putting item to stack

making Linked list

You catch more flies with honey than with vinegar

M-2

```
def rev(head):
    curr = head
    Prev = None
    While curr is not None:
        next = curr.next
        curr.next = prev
        Prev = curr
        curr = next
    return prev.
```

[Instead of changing the data we are changing the link.]

TC = O(n)
SC = O(1)

Recursive Approach.

M-1

```
def rev(head):
    if head == None:
        return head
    if head.next == None
        return head
    rest_head = rev(head.next)
    rest_tail = head.Next
    rest_tail.next = head
    head.next = None
    return rest_head.
```

Base case

TC = O(n)
SC = O(n)

M-2

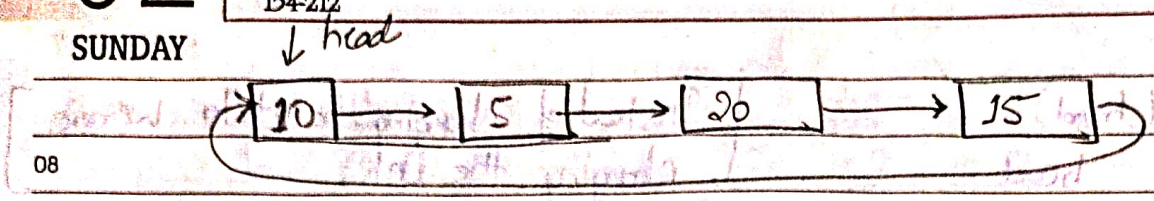
```
def rev(curr, Prev = None):
    if curr == None:
        return Prev
    next = curr.next
    curr.next = prev
    return rev(next, curr)
```

First reversing a linked list then calling a recursive function.

TC = O(n)
SC = O(n)

JUNE
02
SUNDAY
'24
23rd Week
154-212

Circular linked list



→ Circular linked list is a type of linked list that is circular in nature. In a circular linked list, every node has successor. In this data structure, every node points to the next node and the last node of the linked list points to the first node.

```
Class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

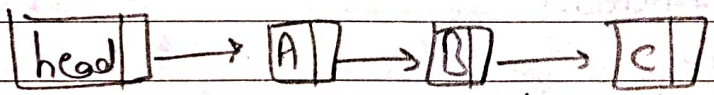
```
head = Node(10)
head.next = Node(15)
head.next.next = Node(20)
head.next.next.next = Node(15)
head.next.next.next.next = head.
```

Advantages

(a) We can traverse the whole list from any node.
(b) Implementation of algorithms like round robin.
(c) We want/can insert at the beginning and end by just maintaining one tail references/ pointers

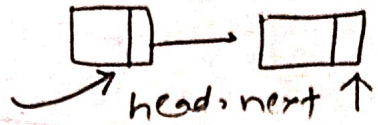
Disadvantage

(a) Implementation of operation becomes complex.



head.next = next node after the head node.
head point to first node
head.next point to A
A.next point to B
B.next point to C
C.next point to Null

(→) arrow represent the next pointer showing the direction to the next node.



Traversal — Tc - O(n)

```
def pc(head):
    if head == None:
        return
    print(head.key, end = " ")
    curr = head.next
    while curr != head:
        print(curr.key, end = " ")
        curr = curr.next
```

→ explicitely print first (head node)
begin from 2nd node, keep on traversing and stop when you come to the head

TC = O(n)

Insert at the Beginning of circular LL.

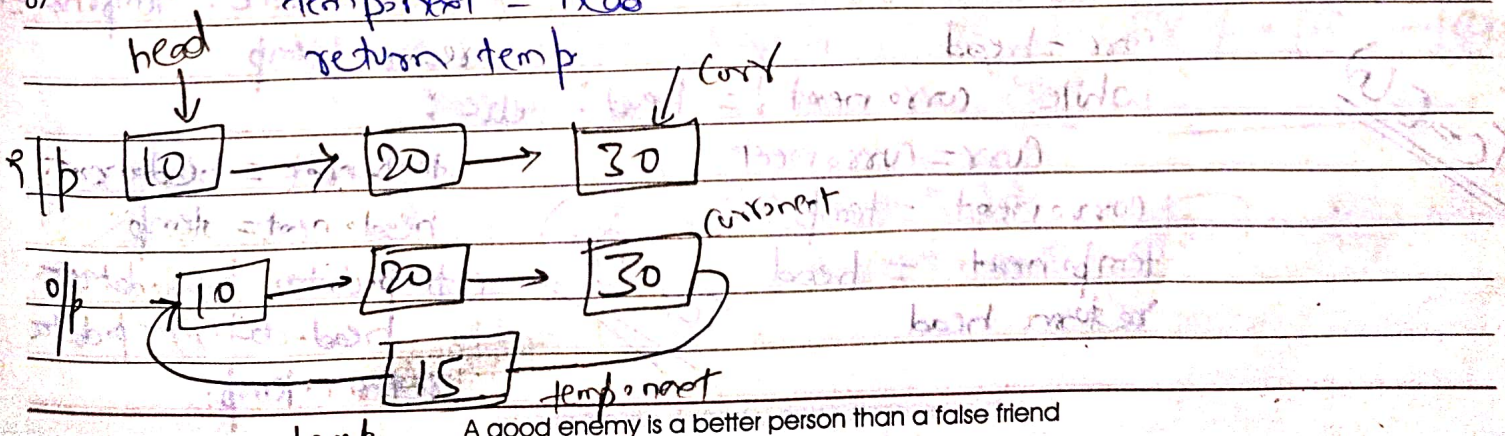
M-1

i/p — 10 → 20 → 30 (circular) n = 15

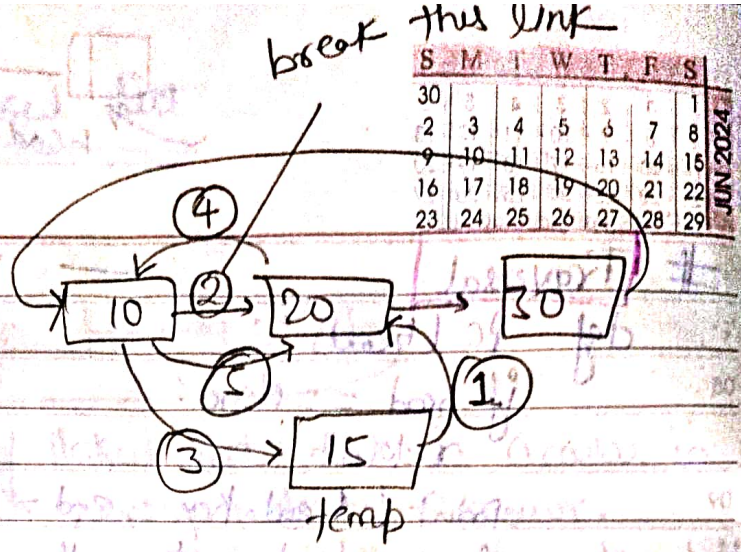
o/p — 15 → 10 → 20 → 30 (circular)

```
def insert(head, n):
    temp = Node(n)
    if head == None:                 # for empty list
        temp.next = temp
        return temp
    curr = head                      # 2nd name
    while curr.next != head:
        curr = curr.next
    curr.next = temp
    temp.next = head
    return temp
```

TC - O(n)



JUNE
04
'24
23rd Week
156-210
TUESDAY



M-2

```
def Insert(head, n):
    temp = Node(n)
    if head == None:          # for empty list
        temp.next = temp
        return temp
    else:
        temp.next = head.next
        head.next = temp
        head.key, temp.key = temp.key, head.key
        return head
```

→ TC = O(1) ✓

TC = O(1)

→ first insert at 2nd position and swap the data/key.

Insert at the end of a circular linked list

i/p = 10 → 20 → 30 (circular), n = 15, o/p = 10 → 20 → 10 → 15 (circular)

i/p = head = None, n = 10, o/p = 10 (circular)

M-1

```
def insertEnd(head, n):
    temp = Node(n)
    if head == None:
        temp.next = temp
        return temp
    else:
        curr = head
        while curr.next != head:
            curr = curr.next
        curr.next = temp
        temp.next = head
        return head
```

TC = O(n)

M-2

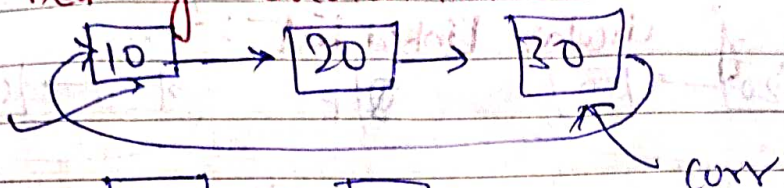
You insert the new node after the head and after that simple swap the data of head and new node.

```
def insertEnd(head, n):
    temp = Node(n)
    if head == None: temp.next = temp
        return temp
    else:
        temp.next = head.next
        head.next = temp
        temp.data, head.data = head.data, temp.data
        return temp.
```

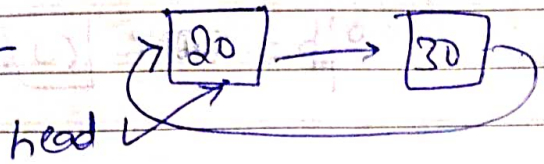
TC = O(1)

Delete head of circular linked list

i/p –

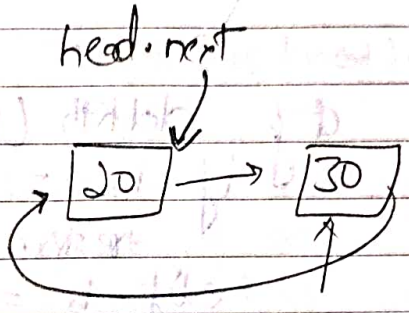


o/p –



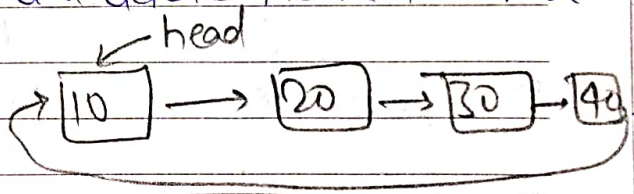
M-1

```
def delhead(head):
    if head == None:
        return None
    elif head.next == head:
        return None
    curr = head
    while curr.next != head:
        curr = curr.next
    else curr.next = head.next
    return curr.next
```

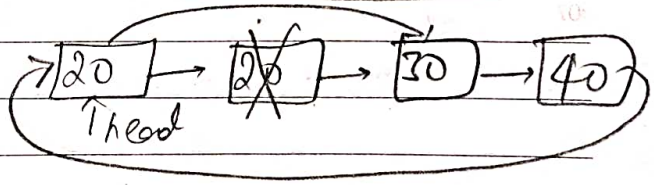


TC = O(n)

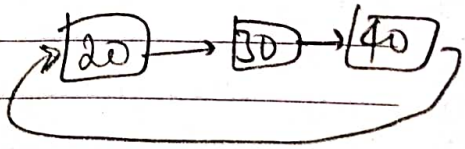
M-2 We copy the next data to the head and we wanted to delete head so we first copy the data and delete the next node.




```
def delhead(head):
    if head == None:
        return None
    elif head.next == head:
        return None
    else:
        head.data = head.next.data
        head.next = head.next.next
        return head
```



TC = O(1)



JUNE
06
'24
23rd Week
158-208
THURSDAY

S	M	T	W	T	F	S
30						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29

JUN 2024

Delete Kth Node of Circular Linked List :-

i/p - 10 → 20 → 30 (circular) K=2

o/p - 10 → 30 (circular)

i/p = 10 → 20 (circular) K=1

o/p = 20

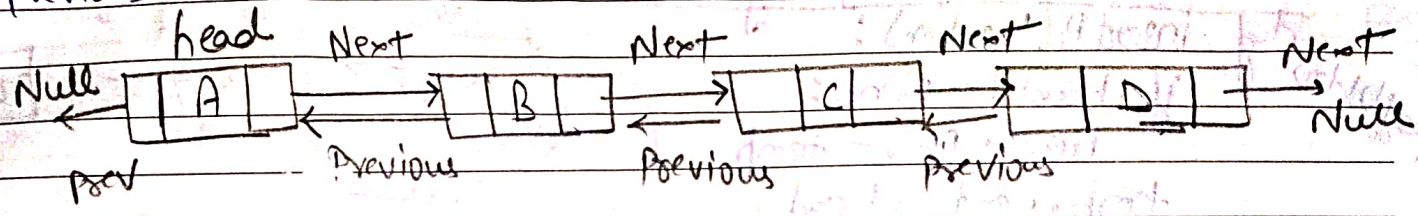
```
def delKth(head, K):
    if head == None:
        return head
    elif K == -1:
        return delFirst(head)
    else:
        curr = head
        for i in range(K-2):
            curr = curr.next
        curr.next = curr.next.next
        return head.
```

TC = O(K)

Absence makes the heart grow fonder

Doubly linked list

→ In doubly linked list every node has reference to both previous and next node.



M-1
```
def printdll(head):
    curr = head
    while curr != head :
        print(curr.data, end = " ")
        curr = curr.next
```

M-2
```
def printdll(head):
    l = []
    curr = head
    while curr != head:
        print(curr.data)
        l.append(curr.data)
    return l
```

Advantage of double linked list over singly linked list :-

(a) In SLL, the traversal can be done using the next node link only. Thus traversal is possible in one direction only.

(b) In DLL, the traversal can be done using the previous node link or the next node link. Thus traversal is possible in both direction.

(c) Complexity of deletion with a given node in singly linked list is O(n). because the previous node needs to be known, and traversal takes O(n) whereas in DLL, complexity of deletion with a given node is O(1) because the previous node can be accessed easily.

(d) We can use a DLL to execute heaps and stacks, binary tree.

(e) In case of better implementation, while searching, we prefer to use DLL

Disadvantage of DLL

(*) Extra spaces of previous Node.

(*) Code becomes more complex.

JUNE
08
'24
23rd Week
160-206
SATURDAY

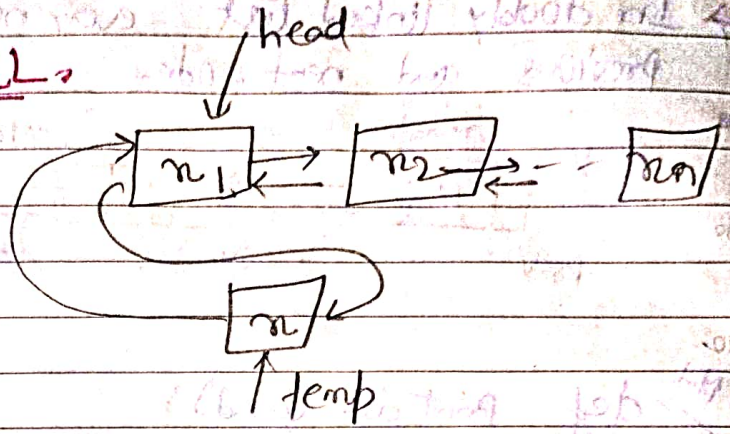
S	M	T	W	T	F	S
30						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29

JUN 2024

Insert at the Beginning of DLL.

```
def insertB(head,n):
    temp=Node(n)
    if head != None:
        head.prev = temp
    temp.next = head
    temp.prev = NULL
    return temp
```

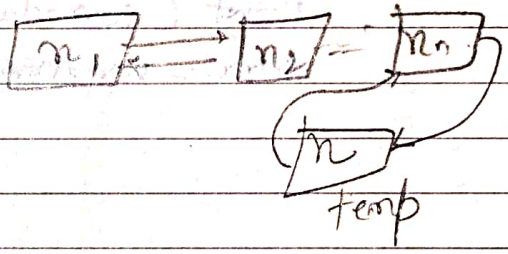
TC = O(1)



Insert at the End of DLL.

```
def insertE(head,n)
    temp = Node(n)
    if head == None:
        return temp
    else:
        curr = head
        while curr is not None:
            curr = curr.next
        curr.next = temp
        temp.prev = curr
        return head.
```

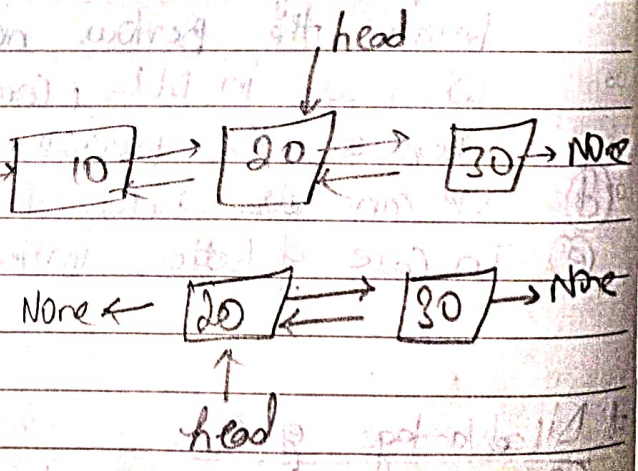
TC = O(n)



Delete head of DLL.

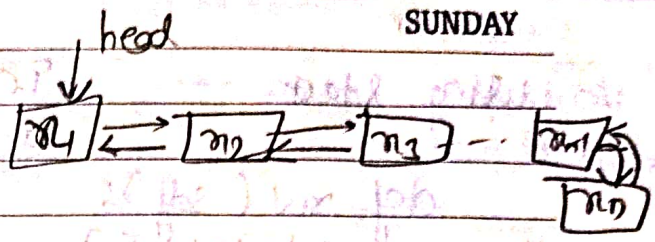
```
def delH(head):
    if head == None:
        return None
    if head.next == None:
        return None
    else:
        head = head.next
        head.prev = None
        return head
```

TC = O(1)



Won's the person who cut the leaves today, would cut the fruits tomorrow?

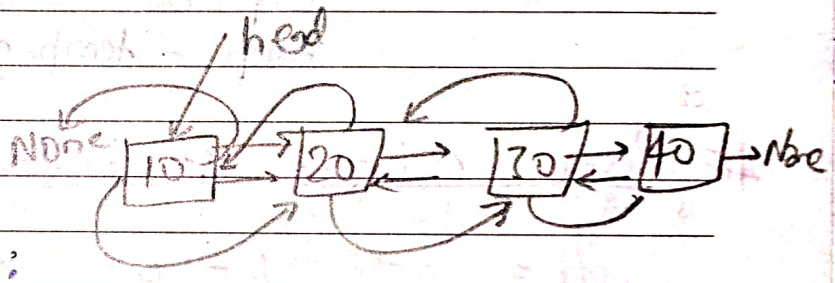
Delete last Node of DLL:




```
def dll(head):
    if head == None:
        return None
    if head.next == None:
        return None
    curr = head
    while curr.next.next != None:
        curr = curr.next
    curr.next = None
    return head.
```

TC = O(n)

Reverse a DLL




```
def Rdll(head):
    if head == None:
        return None
    if head.next == None:
        return head
    curr = head
    Prev = None
    while curr != None:
        Prev = curr
        curr.next, curr.prev = curr.prev, curr.next
        curr = curr.prev
    return prev
```

TC = O(n)

TC = O(n)
SC = O(1)

JUNE
10 '24
MONDAY
24th Week
162-204

S	M	T	W	T	F	S
30						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29

JUN 2024

Using Stack — TC = O(n)
SC = O(n)

```
def rev(self):
    stack = []
    temp = self.head
    while temp is not None:
        stack.append(temp.data)
        temp = temp.next
```

TC = O(n)
SC = O(n)

```
    temp = self.head
    while temp is not None:
        temp.data = stack.pop()
        temp = temp.next
```

Reverse a Linked list in group of size K.

i/p = 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → Null , K = 3
o/p = 3 → 2 → 1 → 6 → 5 → 4 → 8 → 7 → Null

M-1

```
def revrec(head, K):
    curr = head
    prev, next = None, None
    count = 0
    while curr != None and count < K:
        next = curr.next
        curr.next = prev
        prev = curr
        curr = next
        count += 1
    if next != None:
        rem_head(revrec(curr, K))
        head.next = rem_head
    return prev
```

TC = O(n)

```
def itrev(head, K):
    curr, prev, first = head, None
    firstPass = True
    while curr != None:
        first, prev = curr, None
        count = 0
        while curr != None & count < K
            next = next.next
```

Compliments are only lies in court cloths

Palindrome linked list

→ Use stack data structure — The idea is to use a stack and push all the nodes into a stack, then again iterate over the linked list to validate if the linked list is palindrome or not.

M-1 using stack

```
def pal(head):
    stack = []
    curr = head
    while (curr != None):
        stack.append(curr.data)
        curr = curr.next
    curr = head
    while (curr != None):
        if (stack.pop() != curr.data):
            return False
        curr = curr.next
    return True
```

TC = O(n)
SC = O(n)

using slice

```
return stack == stack[::-1]
```

M-2 two pointer approach

```
def pal(head):
    if (head == None): return True
    slow, fast = head, head
    while (fast.next != None and fast.next.next != None):
        slow = slow.next
        fast = fast.next.next
    rev = reverseList(slow.next)
    curr = head
    while (rev != head):
        if (rev.data != curr.data):
            return False
        rev = rev.next
        curr = curr.next
    return True
```

TC = O(n)
SC = O(1)

M-2

Just change while loop and

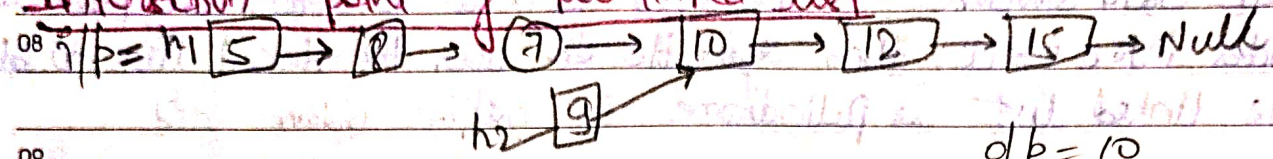
```
left, right = 0, len(value)-1
while left < right:
    if value[left] != value[right]
        return False
    left += 1
    right -= 1
return True
```

JUNE
12 '24
24th Week
164-202
WEDNESDAY

S	M	T	W	T	F	S
30						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29

JUN 2024

Intersection point of two linked list



→ We will use set, we will put all the element of the first LL in our set after that we start traversing the 2nd LL and at any node we find that it is already present in our set then we return that node that node will be our intersection point.

M1

```
def Inter(head1, head2):
    S = set()
    curr = head1
    while curr != None:
        S.add(curr)
        curr = curr.next
    curr = head2
    while curr != None:
        if curr in S:
            return curr.data
        curr = curr.next
    return -1
```

TC = $\Theta(m+n)$
SC = $\Theta(m)$

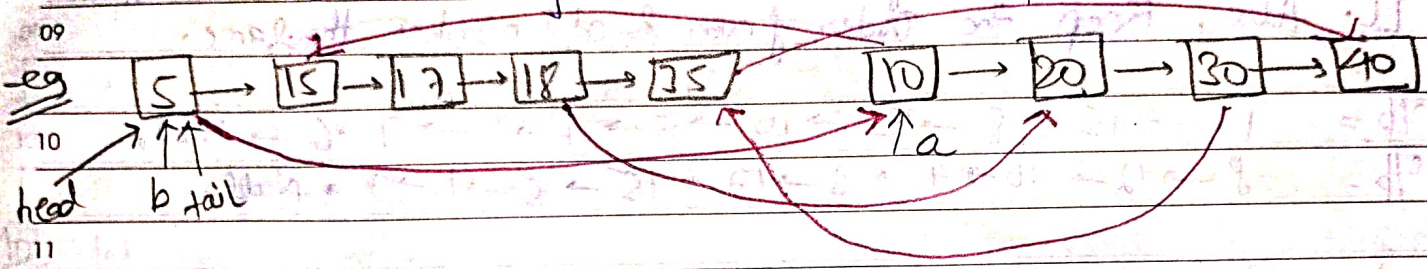
M2

```
def Inter(d, head1, head2):
    curr1 = head1  # 1st Node
    curr2 = head2  # 2nd Node
    for i in range(d):
        if curr1 == None:
            return -1
        curr1 = curr1.next
    while curr1 != None and
          curr2 != None:
        if curr1 == curr2:
            return curr1.data
        curr1 = curr1.next
        curr2 = curr2.next
```

{ TC = $\Theta(m+n)$, SC = $\Theta(1)$ }

Merge two sorted linked list

→ write a sorted merge() function that takes two lists, each of which is sorted in increasing order, and merges the two together into one list which is in increasing order.



→ Simply pick smaller of these two and mark the smaller as head & tail both and whichever you marked head & tail you move ahead in LL. then compare a & b and whichever is smaller is going to add after tail.

```
def mSorted(a,b):
    if (a==None):
        return b
    if (b==None):
        return a
    head, tail = None, None
    while (a is not None and b is not None):
        if (a.data <= b.data):
            tail.next = a
            tail = a
            a = a.next
        else:
            tail.next = b
            tail = b
            b = b.next
    if (a==None):
        tail.next = b
    else:
        tail.next = a
    return head.
```

} out of while loop.

TC = O(m+n)
SC = O(1)

JUNE
14
FRIDAY
'24
24th Week
166-200

S	M	T	W	T	F	S
30						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29

JUN 2024

Segregate Even & Odd Nodes :-

→ Given a LL of integers, write a function to modify the LL such that all even number appear before all the odd number in the modified LL. Also, Keep the order of even & odd number the same.

i/p = 17 → 15 → 8 → 12 → 10 → 5 → 4 → 1 → 7 → 6 → Null.

o/p = 8 → 12 → 10 → 4 → 6 → 17 → 15 → 5 → 1 → 7 → Null.

Idea!

(a) Find the last node reference by doing traversal.
(b) Move all odd nodes to the end.
(i) Consider all odd nodes before the first even node and move them to end.
(ii) Change the head pointer to point to the first even node.
(iii) Consider all odd nodes after the first even node & move them to end.

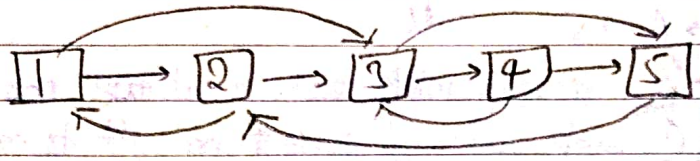
```
def segregate(head):
    es, ee, os, oe = None, None, None, None
    curr = head
    while curr != None:
        n = curr.data
        if n%2 == 0:
            if es == None:
                es = curr
                ee = es
            else:
                ee.next = curr
                ee = ee.next
        else:
            if os == None:
                os = curr
                oe = os
            else: oe.next = curr
                oe = oe.next
        curr = curr.next
    if os == None or es == None:
        return head
    ee.next = os
    os.next = None
    return es
```

O(n)
O(n)
O(1)

Clear statements is argument

Clone a linked list with Random pointer:-

→ Given a linked list of size N where each node has two links: one pointer points to the next node and the second pointer points to any node in the list. The task is to create a clone of the LL in O(n) time.



Using Dictionary

M1

```
def clone(head):
    d = {None : None}
    curr = head
    while curr != None :
        d[curr] = Node(curr.data)
        curr = curr.next
    curr = head
    while curr != None :
        d[curr].next = d[curr.next]
        d[curr].next = d[curr.random]
        curr = curr.next
    return d[head].
```

TC = $\theta(n)$
SC = $\theta(n)$

(I) Create an empty dictionary d.
(II) Traverse through the list and do the following for every node:
d[curr] = Node(curr.data)
(III) Traverse through the list again and do:
d[curr].next = d[curr.next]
d[curr].random = d[curr.random]
(IV) Return d[h]

M2

```
def clone(h):
    curr = h1
    while curr != None :
        next = curr.next
        curr.next = Node(curr.data)
        curr.next.next = next
    h2, clone, curr = h1.next, h2, h1
    while curr != None :
        curr.next = curr.next.next
        clone.next = None if (clone.next == None) else clone.next.next
        clone = clone.next
    return h2
```

Same Scope

```
curr = h1
while curr != None :
    curr.next.random = curr.random.next
    curr = curr.next.next
```

(I) create clone

JUNE
16
SUNDAY
'24
25th Week
168-198

S	M	T	W	T	F	S
30						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29

JUN 2024

Pairwise Swap nodes

08
I/p :- 1 → 2 → 3 → 4 → 5 → 6 → Null
09 O/p :- 2 → 1 → 4 → 3 → 6 → 5 → Null

M-1 Naive

10 def pairwise (head):
Run a loop while we have atleast one node ahead
(a) Swap data of current node with its next node.
(b) move current two nodes ahead.
curr = head
11 while curr != None and curr.next != None:
curr.data, curr.next.data = curr.next.data, curr.data
curr = curr.next.next
return head.

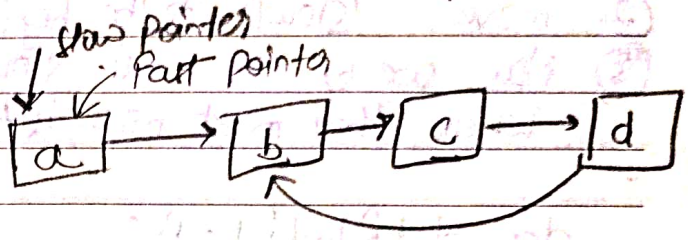
TC = O(n)

01
M-2
def pairwise (head):
02 if head == None or head.next == None:
return head
03 curr = head.next.next
Running the loop and changing the links.
prev = head
04 head = head.next
head.next = prev
05 while curr != None and curr.next != None:
prev.next = curr.next
06 prev = curr
next = curr.next.next
07 curr.next.next = curr
curr = next
prev.next = curr
return head.

Detect loop using floyd cycle detection algorithm.

→ Floyd's cycle finding algorithm (or) Hare-Tortoise algorithm is a pointer algorithm that uses only two pointers, moving through the sequence at different speed.

Idea!



- Initialize two pointer & start traversing.
- move the slow pointer by one positions.
- mov the fast pointer by two positions.
- If both pointer meet at some point then a loop exist and if the fast pointer meets the end position then no loop exists.

M-1

```
def loop (head):
    slow = head
    fast = head
    while fast != None and fast.next != None:
        slow = slow.next
        fast = fast.next.next
        if slow == fast:
            return True
    return False.
```

TC = O(n)
SC = O(1) no space

M-2 Hashing

```
def loop (self):
    S = set()
    temp = self.head
    while (temp):
        if (temp in S):
            return True
        S.add(temp)
        temp = temp.next
    return False
```

TC = O(n)
SC = O(n)

Note: — • fast pointer will enter into the loop before (or at same time as slow)
• Let fastp be k distance ahead of slowp when slowp enters this loop (K>0)
• This distance keeps on increasing by one in every moment of both pointers.
• When distance becomes length of cycle, they must meet at some node.

M-3

```
def loop (h):
    while (h != None):
        if (h.flag == 1):
            return true
        h.flag = 1
        h = h.next
    return False
```

TC: O(n) (or) O(n+n)

JUNE
18
TUESDAY
'24
25th Week
170-196

S	M	T	W	T	F	S
30						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29

JUN 2024

Detect & remove loop in LL.

(1) Detect loop using floyd's detection algorithms
(2) move 'slowp' to the beginning of linked list & keep 'fastp at meeting Po
(3) Now one by one move slow and fast (at same speed). The point where they meet now is the first node of this loop.

```
def detRloop(head):
    slow = fast = head
    while fast != None and fast.next != None:
        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 = None
```

Detect loop

Remove loop

Variation of above Question.

(1) find length of loop.
(2) find the first node of loop.