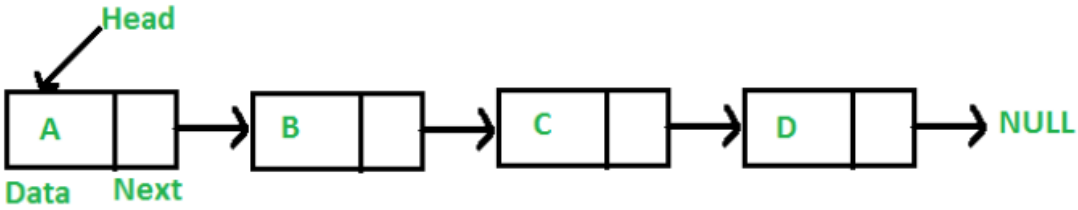
Linked List

Each element is called Node and that stores two things one is the data and one the refernce to next node



Head stores the reference to the very first node, after knowing that we can travel through the complete linkedList

Every **node** will have a data and a next by default we are keeping the next none.

In [1]:
```
## Node will contain two things data and the next refernce
class Node:
    def __init__(self,data):
        self.data = data
        self.next = None
```

In [2]:
```
a = Node(12)
b = Node(13)

a.next = b
print(a.data)
print(b.data)
print(a.next.data)
print(a)
print(a.next)
print(b)
```

```
12
13
13
<__main__.Node object at 0x000001DFF1D84280>
<__main__.Node object at 0x000001DFF1D84220>
<__main__.Node object at 0x000001DFF1D84220>
```

In [3]:
```
# if we try this
print(b.next.data)
## b.next is None and None has no data
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_2700/125603010.py in <module>
      1 # if we try this
----> 2 print(b.next.data)
      3 ## b.next is None and None has no data

AttributeError: 'NoneType' object has no attribute 'data'
```

Making a Linked List with User Input

Approach 1 : Complexity O(n^2)

In this approach we are traversing in the complete Linked List each time

In [4]:
```
class Node:
    def __init__(self,data):
        self.data = data
        self.next = None

def take_input():
    inputList = list(map(int,input().split()))
    head = None
    for curr_data in inputList:
        if curr_data == -1:
            break
        newNode = Node(curr_data)

        if head is None: # we keep the head constant, we don't change the head
            head = newNode
        else:
            curr = head
            while curr.next is not None: # using curr to reach to the end of l
                curr = curr.next
            curr.next = newNode
    return head

head = take_input()
```

```
1 2 3 4 5
```

Print Linked List

In [5]:
```
## Print Function

def printLL(head):
    while head is not None:
        print(str(head.data) + "-->" , end = "")
        head = head.next
    print(None)
    return

head = take_input()
printLL(head)
```

```
1 2 3 4 5
1-->2-->3-->4-->5-->None
```

Taking Input of Linked List (Optimised)

Approach 2 : Complexity O(n)

The only thing time taking the traversing in the Linked List again and again. We can avoid this by maintaing the tail of the linked List

In [6]:

```
def take_input():
    inputList = list(map(int, input().split()))
    head = None
    for curr_data in inputList:
        if curr_data == -1:
            break
        newNode = Node(curr_data)
        if head is None:
            head = newNode # assign head one time only
            last = head  # keep the reference of tail, which is head at this p
        else:
            last.next = newNode # connect new node to the tail.
            last = newNode      # update the tail
    return head

head = take_input()
printLL(head)
```

```
1 2 3 4 5
1-->2-->3-->4-->5-->None
```

Print the length of LinkedList (Iterative)

Complexity O(n)

In [7]:

```
def lengthLL(head):
    l = 0  # maintain a counter
    while head is not None:
        l += 1
        head = head.next
    return l


print(lengthLL(head))
```

```
5
```

Print the ith element of the Linked List

complexity O(n)

In [8]:

```
def printithLL(head, i):
    t = 0
    while head is not None and t < i:
        t += 1
        head = head.next
    print(head.data)
    return



"""
def printithLL(head,i):
    t = 0
    while head is not None:
        if t == i:
            print(head.data)
            break
        else:
            t += 1
            head = head.next
    return
"""
printithLL(head, 3)
```

```
4
```

Insert data at ith position of LinkedList (Iteratively)

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

head at 1.

Insert data = 10 at i = 2.

We need to point 2 to 10 and 10 to 3.

1 --> 2 --> 10 --> 3 --> 4 --> 5

we need to keep two pointers prev and curr

In [19]:

```
def insertatI(head, i, data):
    if i < 0 or i > lengthLL(head):
        return head

    count = 0
    prev = None
    curr = head
    while count < i:
        prev = curr
        curr = curr.next

        count += 1
    newNode = Node(data)
    if prev is not None:
        prev.next = newNode
    else:
        head = newNode
    newNode.next = curr

    return head

head  = insertatI(head, 2, 10)
printLL(head)
```

```
1-->2-->10-->10-->10-->3-->4-->None
```

Delete element at position i in Linked List (Iteratively)

we just need to do .next to .next.next

In [24]:

```
def deleteatI(head, i):
    if i < 0 or i > lengthLL(head)-1:
        return head
    count = 0 # starting at 0
    prev = None
    curr = head
    while count < i: # reach one before the desired position
        prev = curr
        curr = curr.next

        count += 1
    if prev is not None:
        prev.next = curr.next
    else:
        head = head.next
    return head

head = deleteatI(head, 2)
printLL(head)
```

```
1-->2-->3-->4-->None
```

Length of Linked List (Recursively)

In [25]:
```python
# find the lenght recursively

def lenghtLLR(head):
    c = 0
    if head == None:
        return 0
    return 1+lenghtLLR(head.next)

lenghtLLR(head)
```

Out[25]: 4

Insert data into LinkedList at ith Position (Recursively)

In [29]:
```python
def InsertR(head, i, data):
    if i < 0:
        return head

    if i == 0:
        newNode = Node(data)
        newNode.next = head
        return newNode

    if head == None:
        return None

    smallhead = InsertR(head.next, i-1, data)
    head.next = smallhead
    return head

head = InsertR(head, 2,13)
printLL(head)
```

```
1-->2-->13-->13-->3-->3-->4-->None
```

Delete Node of position i Recursively in LinkedList

In [34]:
```python
def deleteR(head, i):
    if i == 0:
        return head.next
    if head == None:
        return None
    head.next = deleteR(head.next, i-1)
    return head

head = deleteR(head,3)
printLL(head)
```

```
1-->2-->13-->3-->4-->None
```

Reverse a Linked List Recursively

Complexity is O(n^2)

We divide the linked list in two parts first the head itself and 2nd the rest of LL, head is already sorted (as it single element is always sorted) and we suppose that rest of LL is sorted by Recursion. We just need to connect the head to the smallhead(head of 2nd part of sorted LL)

In [36]:
```
def reverseLL(head):
    ## Base Case
    if head == None or head.next == None:
        return head

    smallHead = reverseLL(head.next)
    curr = smallHead
    while curr.next is not None:
        curr = curr.next
    curr.next = head
    head.next = None

    return smallHead

head = reverseLL(head)
printLL(head)
```

```
4-->3-->13-->2-->1-->None
```

Reverse the Linked List in O(n) complexity

In [37]:
```
def reverseLL2(head):
    if head == None or head.next == None:
        return head, head
    smallhead, smalltail = reverseLL2(head.next) # we store the smallhead and
    smalltail.next = head
    head.next = None
    return smallhead, head

head, tail = reverseLL2(head)
printLL(head)
```

```
1-->2-->13-->3-->4-->None
```

Reverse Linked List without storing the tail of the Linked List

We dont need to store the small tail bcoz head.next = tail, i.e we already have the reference to tail

e.g. 1 => 2 => 3 => 4 => 5

```
two parts are 1 => None  and 2 => 3 => 4 => 5

Before moving to 2 we 1 has reference of 2 and 2 is going to be tail in th
e reversed LL.
```

In [38]:
```
def reverseLL3(head):
    if head == None or head.next == None:
        return head
    smallhead = reverseLL3(head.next)
    tail = head.next
    tail.next = head
    head.next = None
    return smallhead

head = reverseLL3(head)
printLL(head)
```

```
4-->3-->13-->2-->1-->None
```

Reverse Iterative

we need to point **current** to **previous** and maintain two things **prev and curr**

In [39]:
```
def reverseR1(head):
    prev = None
    curr = head
    while curr is not None:
        next = curr.next
        curr.next = prev
        prev = curr
        curr = next
    head = prev
    return head

head = reverseR1(head)
printLL(head)
```

```
1-->2-->13-->3-->4-->None
```

Finding Mid Point of a Linked List (Iteratively)

In [40]:
```
def findMid(head):
    l = lengthLL(head)
    target = l // 2
    while target -1:
        head = head.next
        target -=1
    if l % 2 == 0:
        print(head.data)
    else:
        print(head.next.data)

mid = findMid(head)
print(mid)
```

```
13
None
```

Find Mid Point of Linked List (in One Pass)

we initialize two pointers **slow** and **fast** at the head of the Linked List. **slow** will take one step while **fast** will take 2 steps. when fast will reach at the end of the LL slow will be at the mid of Linked List.

Odd case 1 => 2 => 3 => 4 => 5 => None

stop when fast.next == None

even case 1 => 2 => 3 => 4 => 5 => 6 => None

stop when fast.next.next == None

In [41]:
```
def findMid1(head):
    slow = head
    fast = head
    while fast.next != None and fast.next.next != None:
        slow = slow.next
        fast = fast.next.next
    return slow

mid = findMid1(head)
print(mid.data)
```

```
13
```

Merge two Soretd Linked Lists

In [43]:
```
def mergeLL(h1, h2):
    head = None

    while h1 != None and h2 != None:
        if h1.data < h2.data:
            if head == None:
                head = h1
                last = head
            else:
                last.next = h1
                last = h1
            h1 = h1.next
        else:
            if head == None:
                head = h2
                last = head
            else:
                last.next = h2
                last = h2
            h2 = h2.next
    if h1 != None:
        last.next = h1
    if h2 != None:
        last.next = h2
    return head
```

In [44]:
```
h1 = take_input()
h2 = take_input()
head = mergeLL(h1,h2)
printLL(head)
```

```
1 3 5 7
2 4 6 8
1-->2-->3-->4-->5-->6-->7-->8-->None
```

Merge Sort on Linked List

Steps:

1. Break the LL in two halves
2. Call Recursion on two halves
3. Merge the two halves h1 and h2

In [45]:
```
def merge_sort(head):
    ## base case
    if head.next is None:
        return head
    mp = findMid1(head)
    h1 = head
    h2 = mp.next
    mp.next = None
    merge_sort(h1)
    merge_sort(h2)

    mergeLL(h1,h2)
```

In [48]:
```
head = take_input()
merge_sort(head)
printLL(head)
```

```
1 3 5 7 2 4 6 8
1-->2-->3-->4-->5-->6-->7-->8-->None
```

Find a Node in LL (Recursively)

Given a LL & an integer n, find the index of first occurance of n

In [19]:
```
def findNode(head,n):
    if head is None:
        return -1
    if head.data == n:
        return 0
    if head.next is not None:
        return 1 + findNode(head.next, n)
```

Even After Odd

In a given LL, arrange all the elements such that all the even numbers are placed after the odd numbers. Relative order should not change.

In [50]:
```
def oddEven(head):
    oddH = None
    oddT = None
    evenH = None
    evenT = None

    while head is not None:
        if head.data %2 != 0:
            if oddH is None:
                oddH = head
                oddT = oddH
            else:
                oddT.next = head
                oddT = head
        else:
            if evenH is None:
                evenH = head
                evenT = evenH
            else:
                evenT.next = head
                evenT = head
        head = head.next
    if evenT is not None:
        evenT.next = None
    if oddT is not None:
        oddT.next = None
        oddT.next = evenH
        return oddH
    else:
        return evenH
```

In [51]:
```
printLL(oddEven(head))
```

```
1-->3-->5-->7-->2-->4-->6-->8-->None
```

Delete every N nodes

Delete every n nodes after every m nodes.

In [52]:
```
def deleteN(head,m,n):
    curr = head
    while curr:
        for c1 in range(1,m):
            if curr is None:
                break
            curr = curr.next
        if curr is None:
            break
        temp = curr.next
        for c2 in range(1,n+1):
            if temp is None:
                break
            temp = temp.next
        curr.next = temp
        curr = temp
    return head
```

In [54]:

```
head = take_input()
deleteN(head, 2,3)
printLL(head)
```

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14
1-->2-->6-->7-->11-->12-->None
```

Swap nodes at poistoin "i" and "j" of a LL

In [42]:

```python
# given a linked list head and two intergers i and j. swap the nodes of the li
def swap(head, i, j):
    p1 = head
    p2 = head
    if abs(i-j) == 1:
        if i == 0 or j == 0:
            c1 = head
            c2 = head.next
            c1.next = c2.next
            c2.next = c1
            return c2
        else:
            while i-1:
                p1 = p1.next
                i -= 1
            c1 = p1.next
            c2 = c1.next
            p1.next = c2
            c1.next = c2.next
            c2.next = c1
        return head
    if i == 0 or j == 0:
        c1 = head
        print("c1")
        printLL(c1)
        p2 = head
        while i>1 or j>1:
            p2 = p2.next
            i -= 1
            j -= 1
        print("p2")
        printLL(p2)
        t = head.next
        print("t")
        printLL(t)
        c2 = p2.next
        print("c2")
        printLL(c2)
        p2.next = c1
        c1.next = c2.next
        c2.next = t
        return c2

    else:

        while i-1:
            p1 = p1.next
            i -= 1
        c1 = p1.next
        t = c1.next
        print("p1")
        printLL(p1)
        print("c1")
        printLL(c1)
        while j-1:
            p2 = p2.next
            j-=1
        c2 = p2.next
        print("p2")
        printLL(p2)
        print("c2")
        printLL(c2)

```

```
        p1.next = c2
        p2.next = c1
        c1.next = c2.next
        c2.next = t
        return head
```

In [12]:

```
head = makeLL()
# printLL(swap(head, 0, 1))
```

```
0=>1=>2=>3=>4=>5=>6=>7=>8=>9=>10=>11=>12=>13=>14=>None
```