Linked List

Like arrays, 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.



**Why Linked List**

1. The size of the arrays is fixed:

**2)** Inserting a new element in an array of elements is expensive because the room has to be created for the new elements and to create room existing elements have to be shifted.

**Advantages over arrays:**

**1)** Dynamic size

**2)** Ease of insertion/deletion

**Drawbacks:**

**1)** Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists efficiently with its default implementation.

**2)** Extra memory space for a pointer is required with each element of the list.

**3)** Not cache friendly. Since array elements are contiguous locations, there is locality of reference which is not there in case of linked lists.

**Representation:**

# Node class

class Node:

    # Function to initialize the node object

    def \_\_init\_\_(self, data):

        self.data = data  # Assign data

        self.next = None  # Initialize

                          # next as null

# Linked List class

class LinkedList:

    # Function to initialize the Linked

    # List object

    def \_\_init\_\_(self):

        self.head = None

**First Simple Linked List**

# A simple Python program to introduce a linked list

# Node class

class Node:

    # Function to initialise the node object

    def \_\_init\_\_(self, data):

        self.data = data  # Assign data

        self.next = None  # Initialize next as null

# Linked List class contains a Node object

class LinkedList:

    # Function to initialize head

    def \_\_init\_\_(self):

        self.head = None

# Code execution starts here

if \_\_name\_\_=='\_\_main\_\_':

    # Start with the empty list

    llist = LinkedList()

    llist.head = Node(1)

    second = Node(2)

    third = Node(3)

    '''

Three nodes have been created.

    We have references to these three blocks as head,

    second and third

    llist.head          second              third

         |                 |                 |

         |                 |                 |

    +----+------+     +----+------+     +----+------+

    | 1  | None |     | 2  | None |     |  3 | None |

    +----+------+     +----+------+     +----+------+

    '''

    llist.head.next = second; # Link first node with second

    '''

    Now next of first Node refers to second.  So they

    both are linked.

    llist.head          second              third

         |                 |                 |

         |                 |                 |

    +----+------+     +----+------+     +----+------+

    | 1  |  o-------->| 2  | null |     |  3 | null |

    +----+------+     +----+------+     +----+------+

    '''

    second.next = third; # Link second node with the third node

    '''

    Now next of second Node refers to third.  So all three

    nodes are linked.

    llist.head          second              third

         |                 |                 |

         |                 |                 |

    +----+------+     +----+------+     +----+------+

    | 1  |  o-------->| 2  |  o-------->|  3 | null |

    +----+------+     +----+------+     +----+------+

    '''

Complete program to represent linkedlist

class node:

    def \_\_init\_\_(self,data):

        self.data= data

        self.next=None

class Linkedlist:

    def \_\_init\_\_(self):

        self.start=None

    def viewnode(self):

        if self.start==None:

            print("list is empty")

        else:

            temp=self.start

            while (temp!=None):

                print(temp.data,end=" ")

                temp=temp.next

    def deleltelast(self):

        if self.start==None:

            print("linked list empty")

        else:

            self.start=self.start.next

    def insertlast(self,value):

        newnode=node(value)

        if self.start==None:

            self.start=newnode

        else:

            temp=self.start

            while temp.next!=None:

                temp=temp.next

            temp.next=newnode

mylist=Linkedlist()

mylist.insertlast(10)

mylist.insertlast(20)

mylist.insertlast(30)

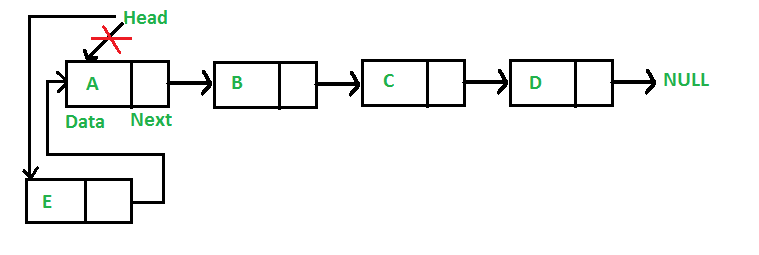
mylist.viewnode()

print("\n")

mylist.deleltelast()

mylist.viewnode()

**Add a node at the front: (A 4 steps process):**



# This function is in LinkedList class Function to insert a new node at the beginning Node class

class Node:

    # Function to initialize the node object

    def \_\_init\_\_(self, data):

        self.data = data  # Assign data

        self.next = None  # Initialize next as null

# Linked List class

class LinkedList:

    # Function to initialize the Linked List object

    def \_\_init\_\_(self):

        self.head = None

    def push(self, new\_data):

        # 1 & 2: Allocate the Node & Put in the data

        new\_node = Node(new\_data)

        # 3. Make next of new Node as head

        new\_node.next = self.head

        # 4. Move the head to point to new Node

        self.head = new\_node

    def printList(self):

        temp = self.head

        while (temp):

            print (temp.data)

            temp = temp.next

L1=LinkedList()

L1.push(1)

L1.push(2)

L1.push(3)

L1.printList()

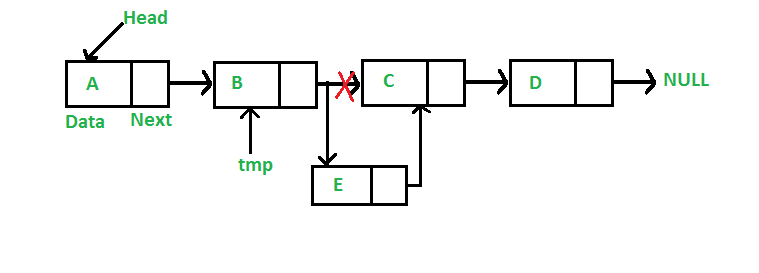
OUTPUT:

3

2

1

**Add a node after a given node: (5 steps process)**



# This function is in LinkedList class Function to insert a new node at the beginning Node class

class Node:

    # Function to initialize the node object

    def \_\_init\_\_(self, data):

        self.data = data  # Assign data

        self.next = None  # Initialize next as null

# Linked List class

class LinkedList:

    # Function to initialize the Linked List object

    def \_\_init\_\_(self):

        self.head = None

    def push(self, new\_data):

        # 1 & 2: Allocate the Node & Put in the data

        new\_node = Node(new\_data)

        # 3. Make next of new Node as head

        new\_node.next = self.head

        # 4. Move the head to point to new Node

        self.head = new\_node

    def insertAfter(self, prev\_node, new\_data):

        new\_node = Node(new\_data)

        temp=self.head

        while(temp.data!=prev\_node ):

            temp=temp.next

        prev=temp

        new\_node.next=temp.next

        prev.next =new\_node

        temp = None

    def printList(self):

        temp = self.head

        while (temp):

            print (temp.data)

            temp = temp.next

L1=LinkedList()

L1.push(1)

L1.push(2)

L1.push(3)

L1.printList()

print("\n")

L1.insertAfter(3,5)

L1.printList()

OUTPUT:

3

2

1

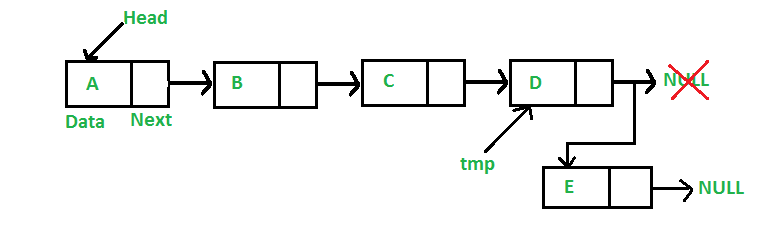
3

5

2

1

**Add a node at the end: (6 steps process)**



# This function is defined in Linked List class Appends a new node at the end.

class Node:

    # Function to initialize the node object

    def \_\_init\_\_(self, data):

        self.data = data  # Assign data

        self.next = None  # Initialize next as null

# Linked List class

class LinkedList:

    def \_\_init\_\_(self):

        self.head = None

    def append(self, new\_data):

        # 1. Create a new node

        # 2. Put in the data

        # 3. Set next as None

        new\_node = Node(new\_data)

        # 4. If the Linked List is empty, then make the new node as head

        if self.head is None:

            self.head = new\_node

            return

        # 5. Else traverse till the last node

        last = self.head

        while (last.next):

            last = last.next

        # 6. Change the next of last node

        last.next =  new\_node

# A complete working Python program to demonstrate all insertion methods of linked list Node class

class Node:

    def \_\_init\_\_(self, data):

        self.data = data  # Assign data

        self.next = None  # Initialize next as null

class LinkedList:

    def \_\_init\_\_(self):

        self.head = None

    def push(self, new\_data):

        new\_node = Node(new\_data)

        new\_node.next = self.head

        self.head = new\_node

    def insertAfter(self, prev\_node, new\_data):

        if prev\_node is None:

            print ("The given previous node must inLinkedList.")

            return

        new\_node = Node(new\_data)

        new\_node.next = prev\_node.next

        prev\_node.next = new\_node

    def append(self, new\_data):

        new\_node = Node(new\_data)

        if self.head is None:

            self.head = new\_node

            return

        last = self.head

        while (last.next):

            last = last.next

        last.next =  new\_node

    def printList(self):

        temp = self.head

        while (temp):

            print (temp.data)

            temp = temp.next

if \_\_name\_\_=='\_\_main\_\_':

    llist = LinkedList()

    llist.append(6)

    llist.push(7);

    llist.push(1);

    llist.append(4)

    llist.insertAfter(llist.head.next, 8)

    print ('Created linked list is:')

    llist.printList()

OUTPUT:

1

7

8

6

4

Delete a node from linked list:

# Python program to delete a node from linked list Node class

class Node:

    # Constructor to initialize the node object

    def \_\_init\_\_(self, data):

        self.data = data

        self.next = None

class LinkedList:

    # Function to initialize head

    def \_\_init\_\_(self):

        self.head = None

    # Function to insert a new node at the beginning

    def push(self, new\_data):

        new\_node = Node(new\_data)

        new\_node.next = self.head

        self.head = new\_node

    # Given a reference to the head of a list and a key,

    # delete the first occurence of key in linked list

    def deleteNode(self, key):

        # Store head node

        temp = self.head

        # If head node itself holds the key to be deleted

        if (temp is not None):

            if (temp.data == key):

                self.head = temp.next

                temp = None

                return

        # Search for the key to be deleted, keep track of the

        # previous node as we need to change 'prev.next'

        while(temp is not None):

            if temp.data == key:

                break

            prev = temp

            temp = temp.next

        # if key was not present in linked list

        if(temp == None):

            return

        # Unlink the node from linked list

        prev.next = temp.next

        temp = None

    # Utility function to print the linked LinkedList

    def printList(self):

        temp = self.head

        while(temp):

            print (" %d" %(temp.data))

            temp = temp.next

# Driver program

llist = LinkedList()

llist.push(7)

llist.push(1)

llist.push(3)

llist.push(2)

print ("Created Linked List: ")

llist.printList()

llist.deleteNode(1)

print ("\nLinked List after Deletion of 1:")

llist.printList()

OUTPUT:

Created Linked List:

2

3

1

7

Linked List after Deletion of 1:

2

3

7

# Search an element in a Linked List:

# Node class

class Node:

    def \_\_init\_\_(self, data):

        self.data = data # Assign data

        self.next = None # Initialize next as null

class LinkedList:

    def \_\_init\_\_(self):

        self.head = None # Initialize head as None

    def push(self, new\_data):

        new\_node = Node(new\_data)

        new\_node.next = self.head

        self.head = new\_node

    # This Function checks whether the value x present in the linked list

    def search(self, x):

        # Initialize current to head

        current = self.head

        # loop till current not equal to None

        while current != None:

            if current.data == x:

                return True # data found

            current = current.next

        return False # Data Not found

# Code execution starts here

if \_\_name\_\_ == '\_\_main\_\_':

    llist = LinkedList()

    llist.push(10);

    llist.push(30);

    llist.push(11);

    llist.push(21);

    llist.push(14);

    if llist.search(21):

        print("Yes")

    else:

        print("No")

OUTPUT: YES

# Doubly Linked List

A **D**oubly **L**inked **L**ist (DLL) contains an extra pointer, typically called previous pointer, together with next pointer and data which are there in singly linked list.



# Node of a doubly linked list

class Node:

    def \_\_init\_\_(self, next=None, prev=None, data=None):

        self.next = next # reference to next node in DLL

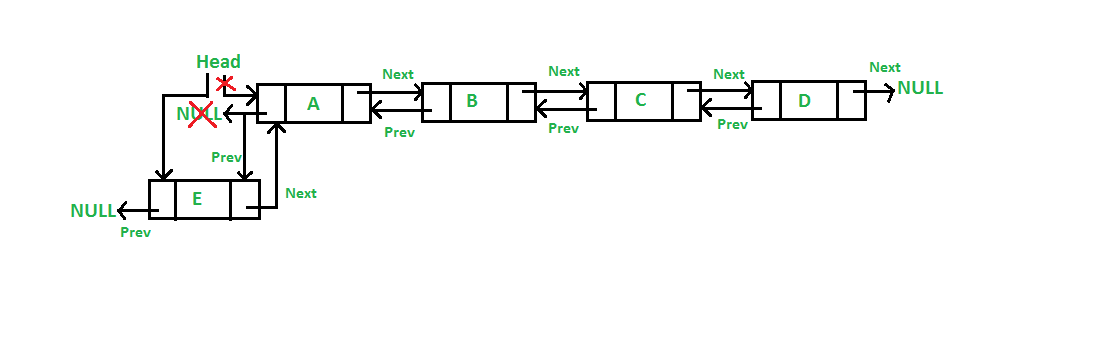
        self.prev = prev # reference to previous node in DLL

        self.data = data

**Advantages over singly linked list**  
**1)** A DLL can be traversed in both forward and backward direction.  
**2)** The delete operation in DLL is more efficient if pointer to the node to be deleted is given.  
**3)**We can quickly insert a new node before a given node.

**Disadvantages over singly linked list**  
**1)** Every node of DLL Require extra space for an previous pointer. It is possible to implement DLL with single pointer though.  
**2)** All operations require an extra pointer previous to be maintained.

**1) Add a node at the front: (A 5 steps process)**



# Node of a doubly linked list

class Node:

    def \_\_init\_\_(self, next=None, prev=None, data=None):

        self.next = next # reference to next node in DLL

        self.prev = prev # reference to previous node in DLL

        self.data = data

class DLinkedlist:

    def \_\_init\_\_(self):

        self.head=None

    def push(self, new\_data):

        # 1 & 2: Allocate the Node & Put in the data

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

        # 3. Make next of new node as head and previous as NULL

        new\_node.next = self.head

        new\_node.prev = None

        # 4. change prev of head node to new node

        if self.head is not None:

            self.head.prev = new\_node

        # 5. move the head to point to the new node

        self.head = new\_node

    def printList(self):

        print ("\nTraversal in forward direction")

        temp=self.head

        while(temp is not None):

            print (" % d" %(temp.data))

            last = temp

            temp = temp.next

        print ("\nTraversal in reverse direction")

        while(last is not None):

            print (" % d" %(last.data))

            last = last.prev

DL=DLinkedlist()

DL.push(10)

DL.push(11)

DL.push(12)

DL.printList()

OUTPUT:

Traversal in forward direction

12

11

10

Traversal in reverse direction

10

11

12

# Circular Linked List

# ***Circular linked list*** is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.

https://media.geeksforgeeks.org/wp-content/uploads/CircularLinkeList.png

**Advantages of Circular Linked Lists:**  
**1)**Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.

**2)**Useful for implementation of queue. Unlike other implementation, we don’t need to maintain two pointers for front and rear if we use circular linked list. We can maintain a pointer to the last inserted node and front can always be obtained as next of last.

**3)** Circular lists are useful in applications to repeatedly go around the list.

# Structure for a Node

class Node:

    # Constructor to create  a new node

    def \_\_init\_\_(self, data):

        self.data = data

        self.next = None

class CircularLinkedList:

    # Constructor to create a empty circular linked list

    def \_\_init\_\_(self):

        self.head = None

    # Function to insert a node at the beginning of a circular linked list

    def push(self, data):

        ptr1 = Node(data)

        temp = self.head

        ptr1.next = self.head

        # If linked list is not None then set the next of last node

        if self.head is not None:

            while(temp.next != self.head):

                temp = temp.next

            temp.next = ptr1

        else:

            ptr1.next = ptr1 # For the first node

        self.head = ptr1

    # Function to print nodes in a given circular linked list

    def printList(self):

        temp = self.head

        if self.head is not None:

            while(True):

                print ("%d" %(temp.data))

                temp = temp.next

                if (temp == self.head):

                    break

# Driver program to test above function Initialize list as empty

cllist = CircularLinkedList()

# Created linked list will be 11->2->56->12

cllist.push(12)

cllist.push(56)

cllist.push(2)

cllist.push(11)

print ("Contents of circular Linked List")

cllist.printList()

OUTPUT:

11

2

56

12