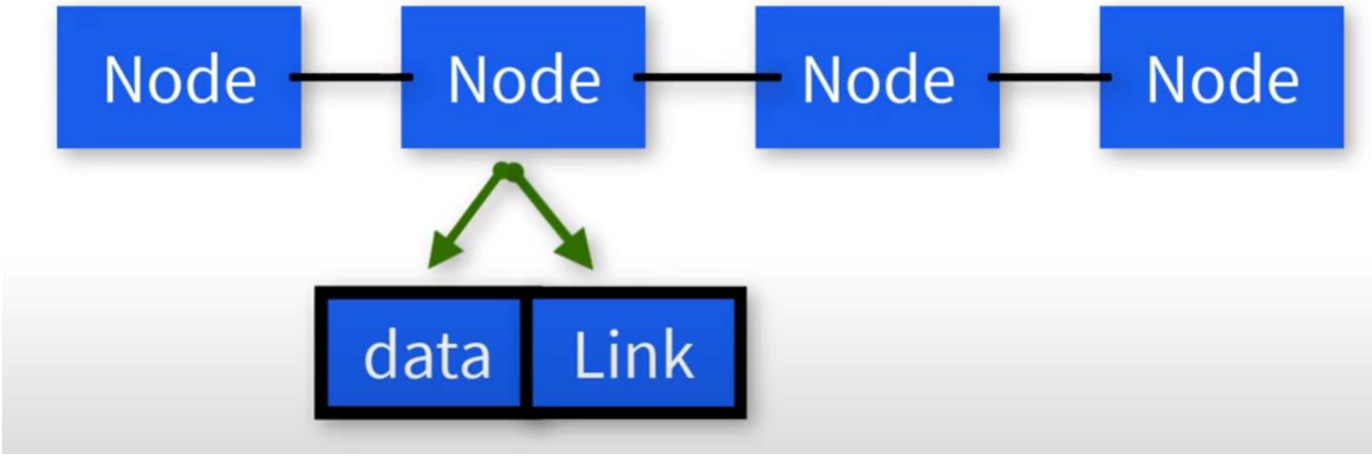
Data Structure using Python

Linked List

Introduction

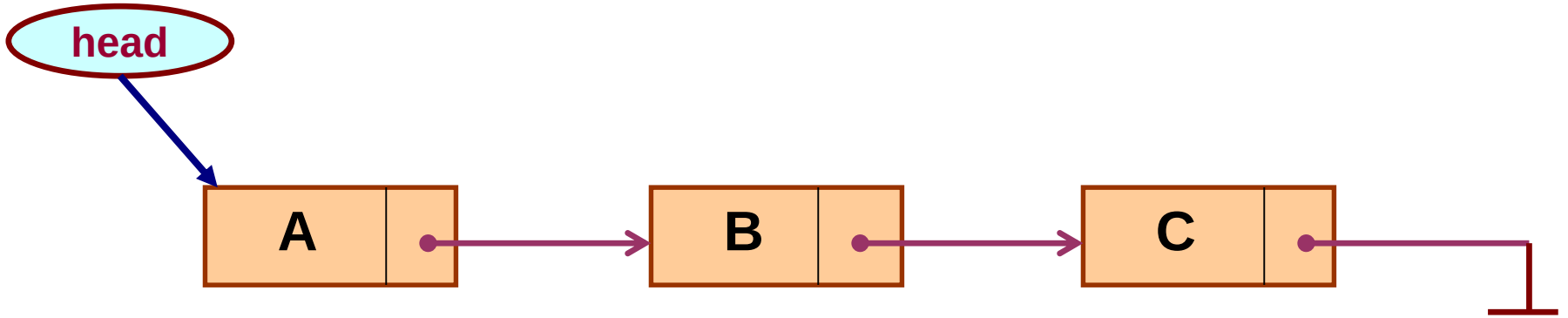
- A linked list is a data structure which can change during execution.
  - Successive elements are connected by a Link.
  - Last element points to `NULL`.
  - It can grow or shrink in size during execution of a program.
  - It can be made just as long as required.
  - It does not waste memory space.



Contd..

- Keeping track of a linked list:
  - Must know the first element of the list (called *start*, *head*, etc.).

- Linked lists provide flexibility in allowing the items to be rearranged efficiently.
  - Insert an element.
  - Delete an element.

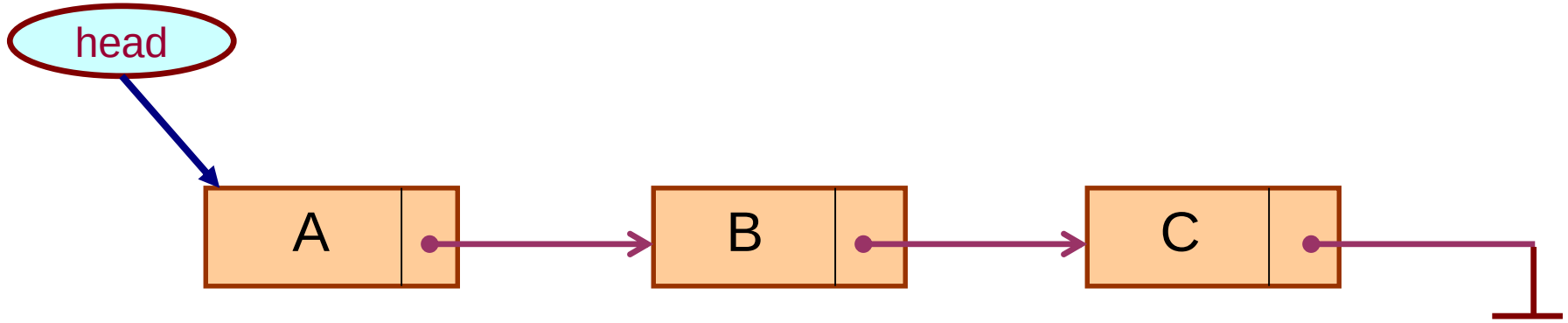


Array versus Linked Lists

- Arrays are suitable for:
  - Inserting/deleting an element at the end.
  - Randomly accessing any element.
  - Searching the list for a particular value.
- Linked lists are suitable for:
  - Inserting an element.
  - Deleting an element.
  - Applications where sequential access is required.
  - In situations where the number of elements cannot be predicted beforehand.

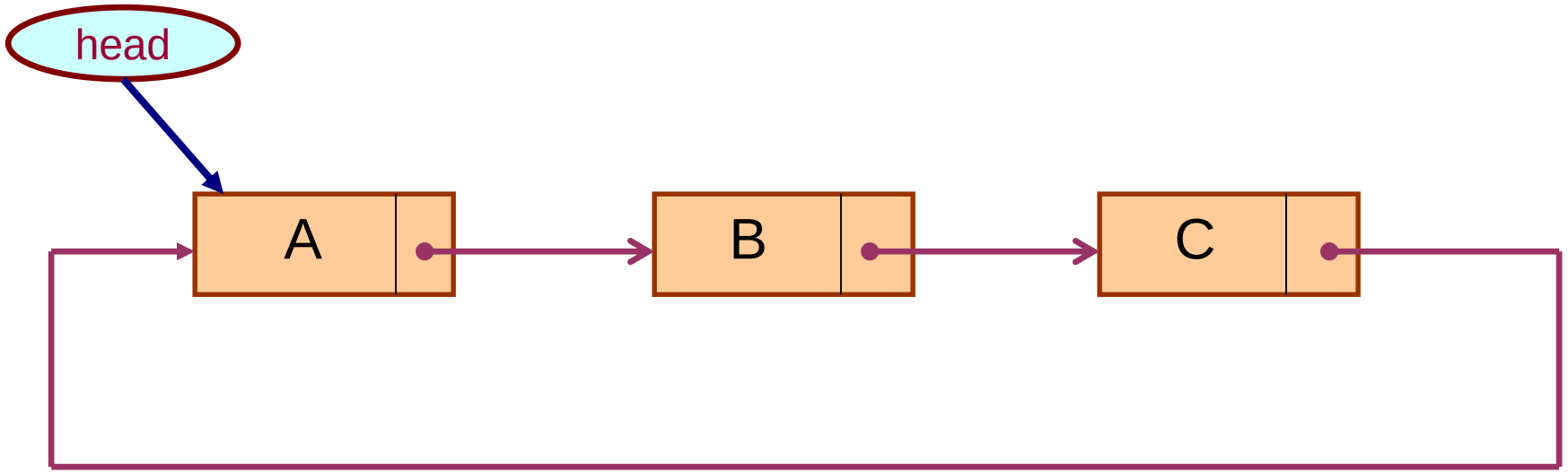
Types of Lists

- Depending on the way in which the links are used to maintain adjacency, several different types of linked lists are possible.
  - Linear singly-linked list (or simply linear list)
    - One we have discussed so far.



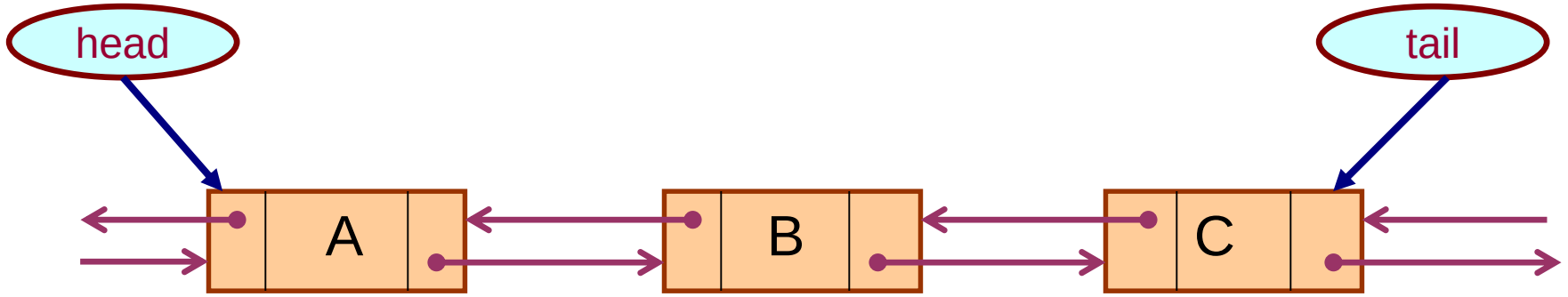
– Circular linked list

- The Link from the last element in the list points back to the first element.



– Doubly linked list

- Link exist between adjacent nodes in both directions.
- The list can be traversed either forward or backward.
- Usually two links are maintained to keep track of the list, *head* and *tail*.

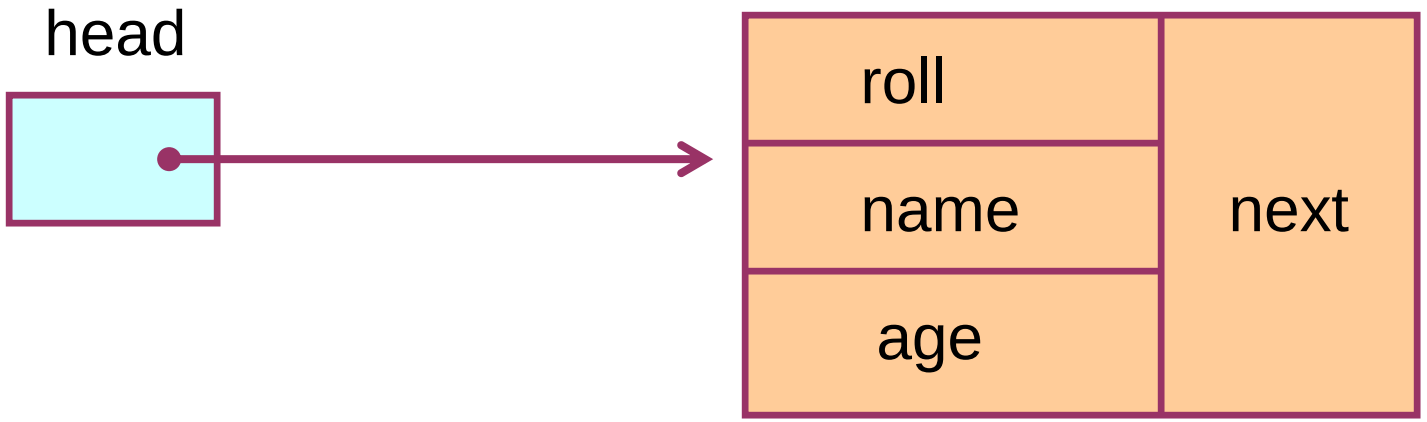


Basic Operations on a List

- Creating a list
- Traversing the list
- Inserting an item in the list
- Deleting an item from the list

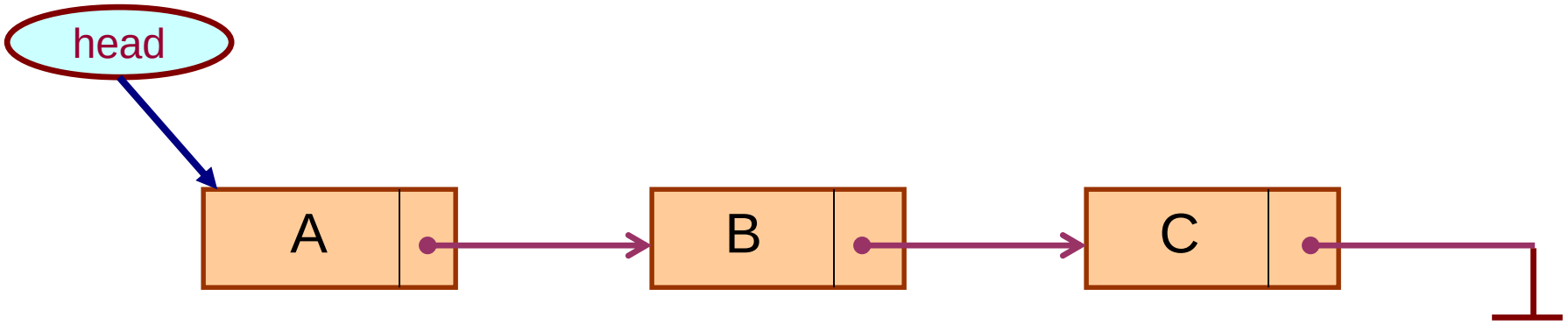
How to begin?

- To start with, we have to create a node (the first node), and make `head` point to it.



Contd.

- If there are n number of nodes in the initial linked list:
  - Allocate n records, one by one.
  - Read in the fields of the records.
  - Modify the links of the records so that the chain is formed.



Creating the Linked List (Simple linear / Singly LL)

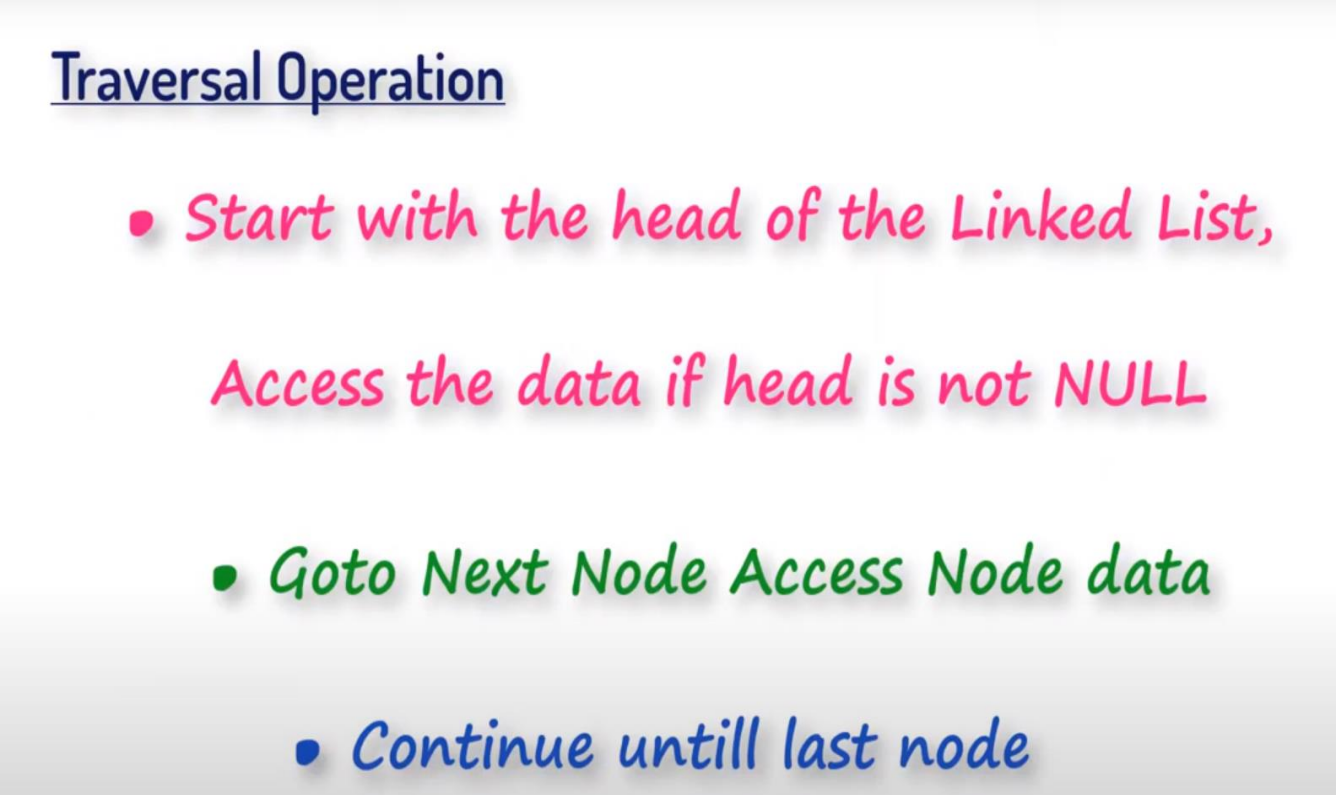
```
class Node:
    def __init__(self,data):
        self.data = data
        self.ref = None


class LinkedList:
    def __init__(self):
        self.head = None
```

Traversing of the Linked List

What is to be done?

- Once the linked list has been constructed and *head* points to the first node of the list:-

- First check if the list is empty or not:
  - Check if the head is None or not:
    - if so then LL is empty and stop the process.
  - If The list is not empty then do :
    - Follow the Link with every node starting from *head*.
    - Display the contents of the nodes as they are traversed.
    - Stop when the *next Link* points to NULL.



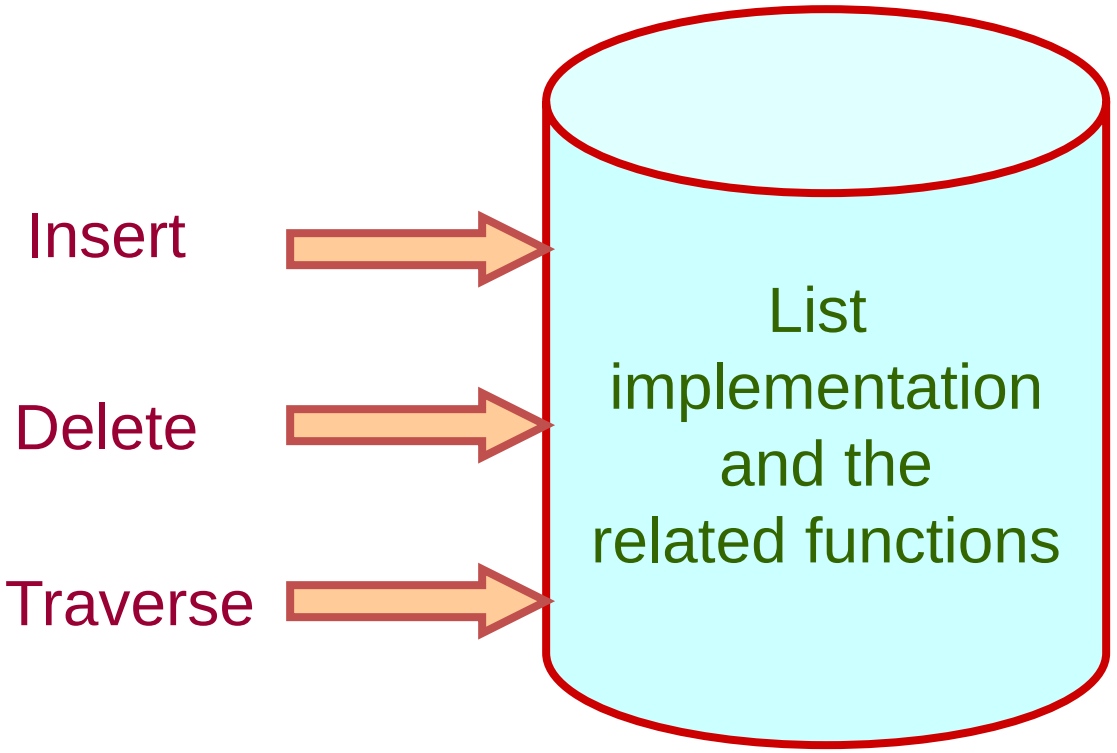
Traversing of LL using Python

```
class Node:
    def __init__(self,data):
        self.data = data
        self.ref = None

class LinkedList:
    def __init__(self):
        self.head = None

    def print_LL(self):
        if self.head is None:
            print("Linked list is empty!")
        else:
            n = self.head
            while n is not None:
                print(n.data)
                n = n.ref

LL1= LinkedList()
LL1.print_LL()
```

List is an Abstract Data Type

- What is an abstract data type?
  - It is a data type defined by the user.
  - Typically more complex than simple data types like *int*, *float*, etc.
- Why abstract?
  - Because details of the implementation are hidden.
  - When you do some operation on the list, say insert an element, you just call a function.
  - Details of how the list is implemented or how the insert function is written is no longer required.

Conceptual Idea



Insertion and Deletion within the Linked List

In essence ...

- For insertion:
  - A record is created holding the new item.
  - The next key of the new record is set to link it to the item which is to follow it in the list.
  - The next key of the item which is to precede it must be modified to point to the new item.
- For deletion:
  - The next key of the item immediately preceding the one to be deleted is altered, and made to point to the item following the deleted item.

Inserting / Adding an Element

- Inserting an element within Linked List:
  - Insert the element at beginning of the LL.
  - Insert the element at the last of the LL.
  - Insert the element at a given particular position (between the nodes).

Inserting an element at the beginning of the List (Singly Linked List)

```
class Node:
    def __init__(self,data):
        self.data = data
        self.ref = None

class LinkedList:
    def __init__(self):
        self.head = None

    def print_LL(self):
        if self.head is None:
            print("Linked list is empty!")
        else:
            n = self.head
            while n is not None:
                print(n.data)
                n = n.ref
    def add_begin(self,data):
        new_node = Node(data)
        new_node.ref = self.head
        self.head = new_node

LL1= LinkedList()
LL1.add_begin(10)
LL1.add_begin(20)
LL1.print_LL()
```

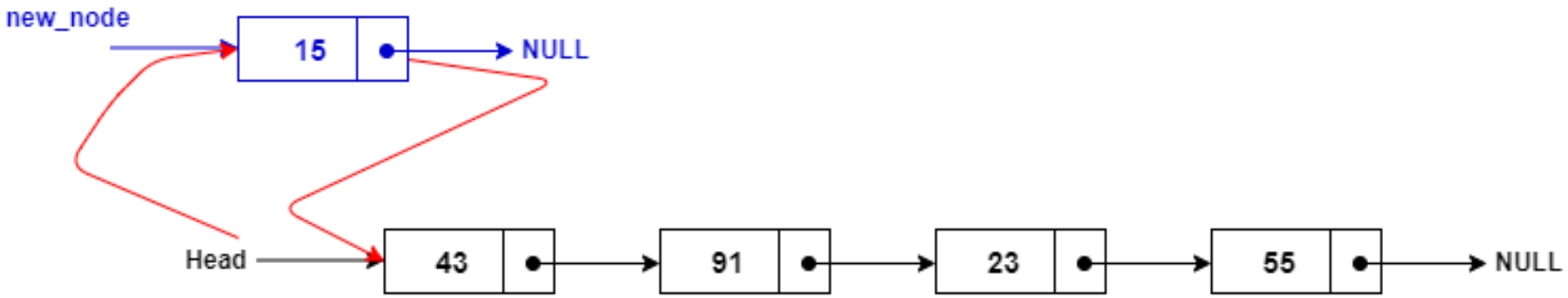
Inserting an element in the last position.

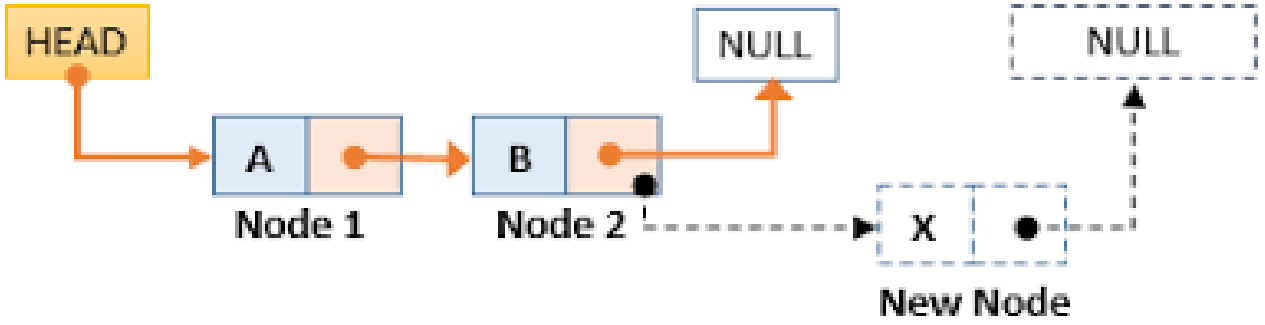
```
  def add_end(self,data):
      new_node = Node(data)
      if self.head is None:
          self.head = new_node
      else:
          n = self.head
          while n.ref is not None:
              n = n.ref
          n.ref = new_node

LL1= LinkedList()
LL1.add_begin(10)
LL1.add_end(100)
LL1.add_begin(20)
LL1.print_LL()
```

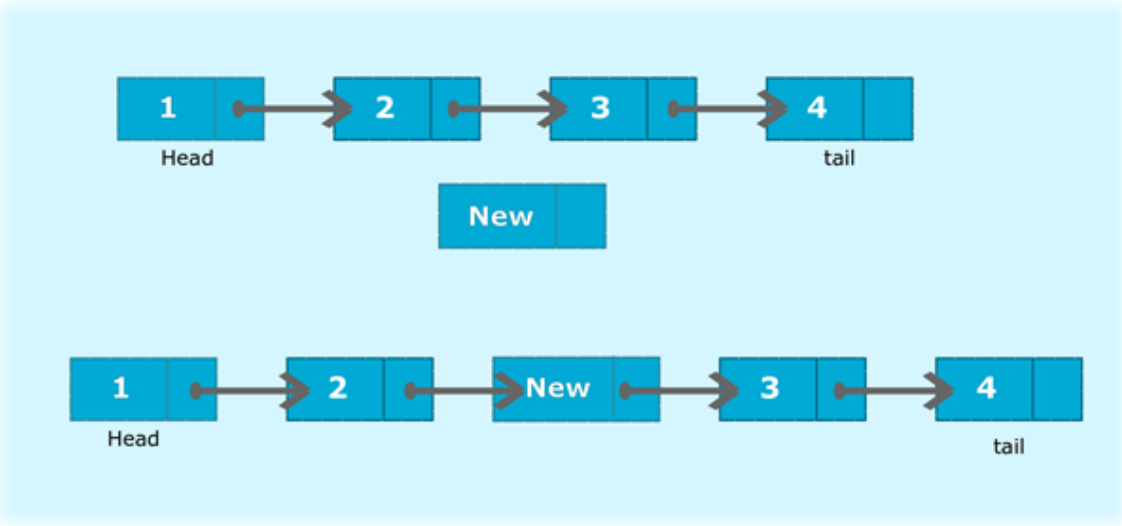
Inserting an element after and before one given position.

```
def add_after(self,data,x):
        n = self.head
        while n is not None:
            if x==n.data:
                break
            n = n.ref
        if n is None:
            print("node is not presesnt in LL")
        else:
            new_node = Node(data)
            new_node.ref = n.ref
            n.ref = new_node
LL1=LinkedList()
LL1.add_end(100)
LL1.add_after(200,100)
LL1.print_LL()
```

```
def add_before(self,data,x):
        if self.head is None:
            print("Linked List is empty!")
            return
        if self.head.data==x:
            new_node = Node(data)
            new_node.ref = self.head
            self.head = new_node
            return
        n = self.head
        while n.ref is not None:
            if n.ref.data==x:
                break
            n = n.ref
        if n.ref is None:
            print("Node is not found!")
        else:
            new_node = Node(data)
            new_node.ref = n.ref
            n.ref = new_node
LL1=LinkedList()
LL1.add_begin(10)
LL1.add_befor(20,10)
LL1.print_LL()
```


new_node
15
NULL
Head
43
91
23
55
NULL


HEAD
A
B
NULL
NULL
Node 1
Node 2
X
New Node


1
2
3
4
Head
tail
New
1
2
New
3
4
Head
tail

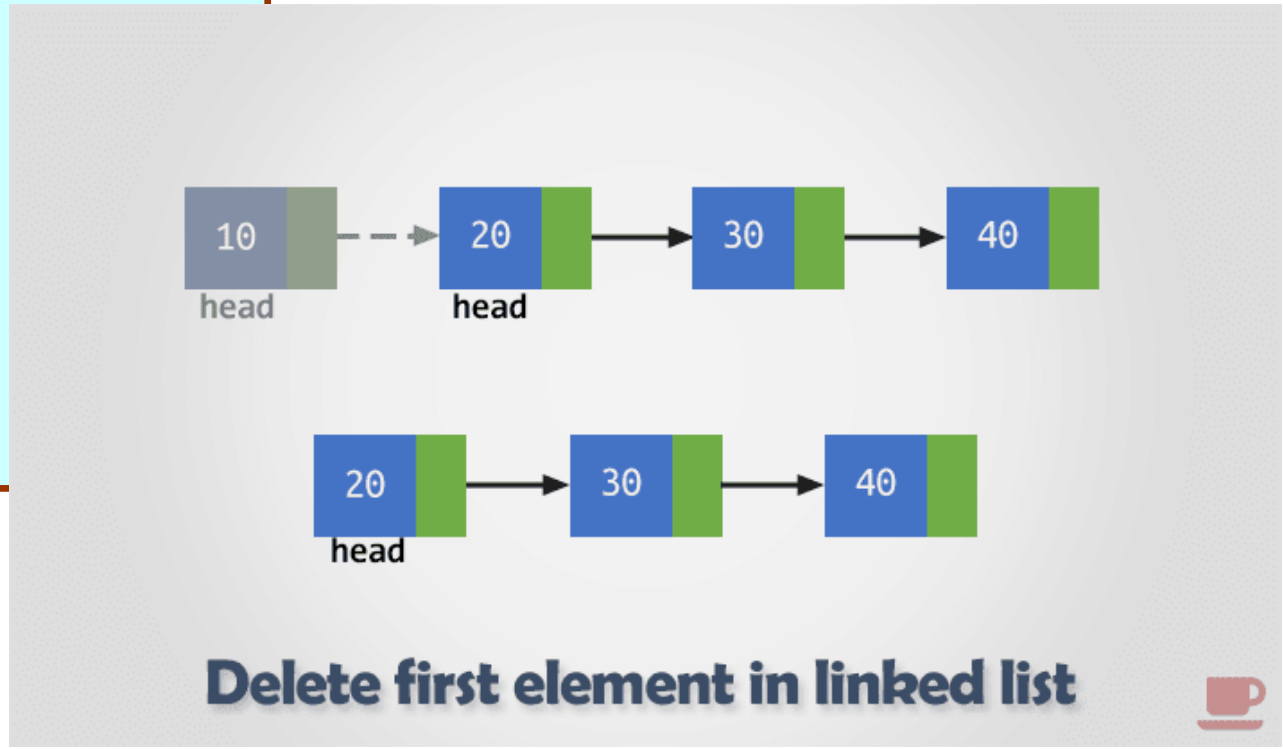
Deleting a node from the Linked List (Singly LL)

What is to be done?

- Here also we are required to delete a specified node.
  - Say, the node whose `roll` field is given.
- Here also three conditions arise:
  - Deleting the first node.
  - Deleting the last node.
  - Deleting an intermediate node.

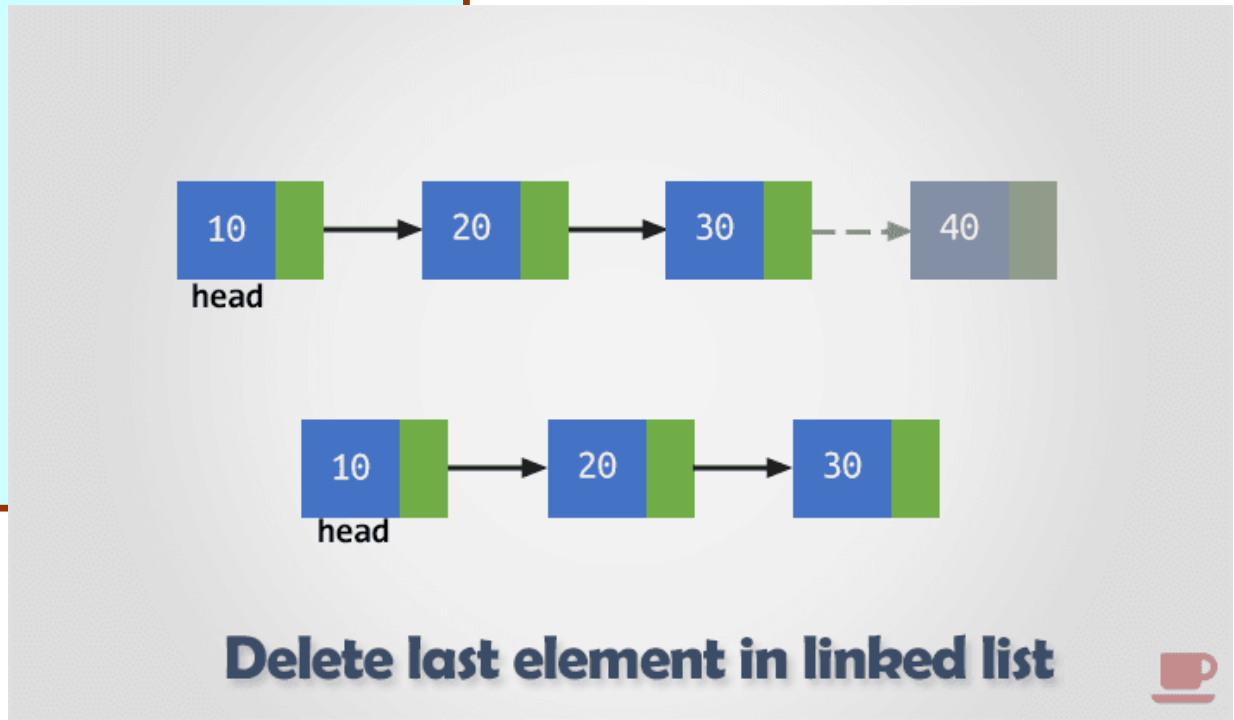
Deletion of element at the beginning of the Linked List

```
def delete_begin(self):
        if self.head is None:
            print("Linked List is empty
can't delete!")
        else:
            self.head=self.head.ref

LL1=LinkedList()
LL1.add_begin(10)
LL1.add_begin(20)
LL1.add_begin(30)
LL1.delete_begin()
LL1.print_LL()
```



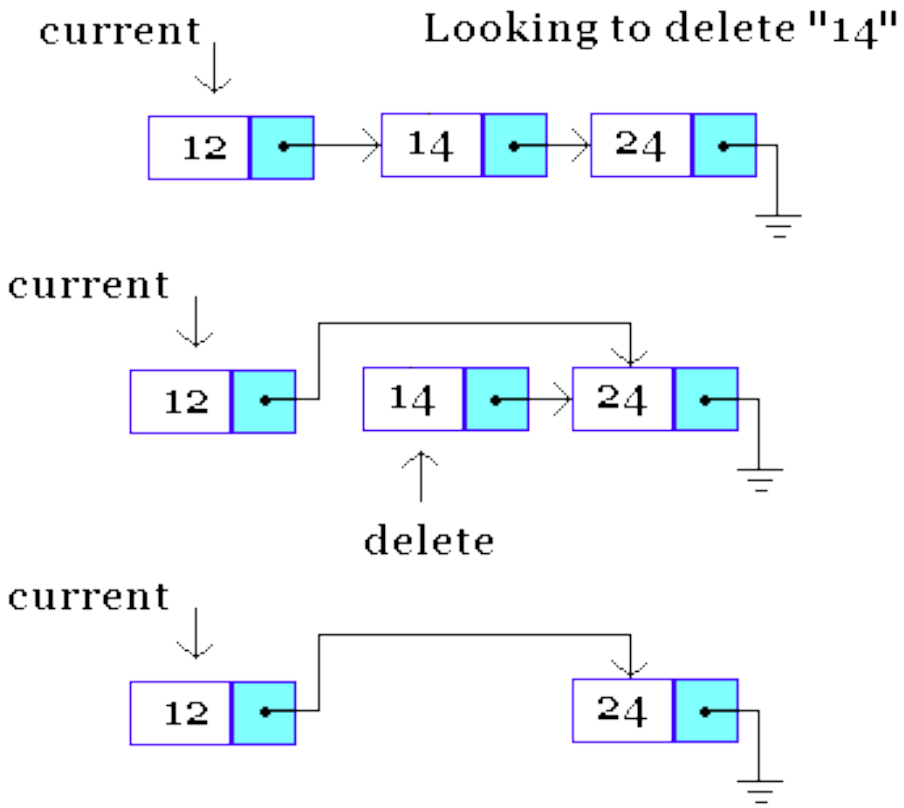
Deletion of element at the end of the Linked List

```
def delete_end(self):
        if self.head is None:
            print("Linked List is empty can't delete!")
        else:
            n=self.head
            while n.ref.ref is not None:
                  n=n.ref
            n.ref = None

LL1=LinkedList()
LL1.add_begin(10)
LL1.add_begin(20)
LL1.add_begin(30)
LL1.delete_end()
LL1.print_LL()
```



Deletion of any element by value from the Linked List

```
def delete_by_value(self,x):
        if self.head is None:
                print("Linked List is Empty")
                return
        if x==self.head.data:
                self.head=self.head.ref
                return
        n= self.head
        while n.ref is not None:
                if x==n.ref.data:
                        break
                n=n.ref
        if n.ref is None:
                print("Node is not present")
        else:
                n.ref=n.ref.ref
LL1=LinkedList()
LL1.add_begin(10)
LL1.add_begin(20)
LL1.add_begin(30)
LL1.delete_by_value(10)
LL1.print_LL()
```

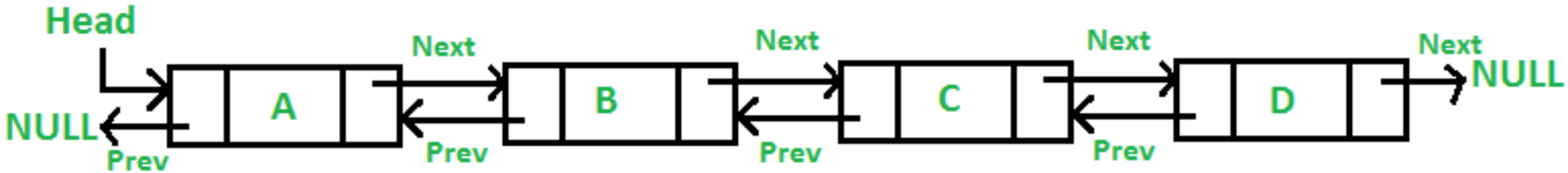


Introduction to Doubly Linked List

Implementation with Python

Introductions

- A **D**oubly **L**inked **L**ist (DLL) contains an extra node as compared with Singly LL.
- Typically three nodes:
- Called *previous link*, together with *next link* and *data* which are there in singly linked list.



Singly LL vs. Doubly LL

- **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.

- In singly linked list, to delete a node, pointer to the previous node is needed.
- To get this previous node, sometimes the list is traversed.
- In DLL, we can get the previous node using previous pointer.

Disadvantages over singly linked list

1) Every node of DLL Require extra space for an previous pointer.

2) All operations require an extra pointer previous to be maintained.

Creating the Doubly Linked List (Using Python)

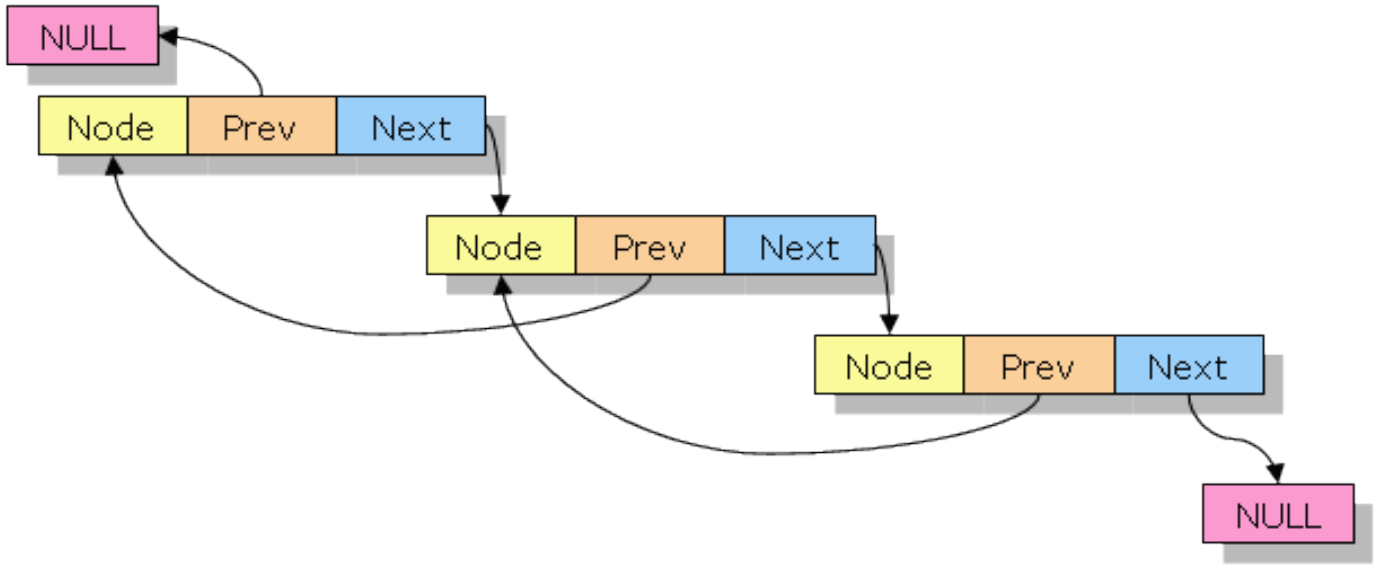
```
class Node:
    def __init__(self,data):
        self.data = data
        self.nref = None
        self.pref = None


class doublyLL:
    def __init__(self):
        self.head = None
```

Traversing operations on the Doubly Linked List

What is to be done?

- **Forward traversal of DLL**
  - Start from the first node, print its data.
  - Move to the second node print its data.
  - Continue step 2 until last node data is printed.

- **Backward traversal of DLL**
  - Start from the last node, print its data.
  - Move to the previous node, print its data.
  - Continue step 2 until the first node data is printed.



Traversing of DLL using Python (Forward & Backward)

```
class Node:
    def __init__(self,data):
        self.data = data
        self.nref = None
        self.pref = None


class doublyLL:
    def __init__(self):
        self.head = None

    def print_LL(self):
        if self.head is None:
            print("Linked List is empty!")
        else:
            n = self.head
            while n is not None:
                print(n.data)
                n = n.nref


dl1= doublyLL()
dl1.print_LL()
```

```
class Node:
    def __init__(self,data):
        self.data = data
        self.nref = None
        self.pref = None


class doublyLL:
    def __init__(self):
        self.head = None

    def print_LL_reverse(self):
        print()
        if self.head is None:
            print("Linked List is empty!")
        else:
            n = self.head
            while n.nref is not None:
                n = n.nref
            while n is not None:
                print(n.data)
                n = n.pref
dl1= doublyLL()
dl1.print_LL_reverse()
```

Insertion operations within the Doubly Linked List

In essence …

- For insertion:
  - A record is created holding the new item.
  - The next key of the new record is set to link it to the item which is to follow it in the list.
  - The previous key of the item must be modified to point to the new item.

Inserting / Adding an Element

- **Inserting an element within Doubly Linked List:**
  - Insert the element when the DLL is empty. (after creating the list)
  - Insert the element at beginning of the DLL.
  - Insert the element at the last of the DLL.
  - Insert the element at a given particular position (between the nodes).

What is to be done?

- **First check the DLL is empty or not?**
  - If the DLL is empty
  - # then we can't insert an element at beginning, end or anywhere in the DLL. #
  - Then put the first data into DLL (after creation).
- **To insert in beginning**
  - Find the *Head* position and make the new data as *Head*.
- **To insert in end**
  - Find the existing last node.
  - Add the new node there
  - From that new node make the *next* pointer to *NULL*
- **To insert after or before one given position**
  - Check the given data / position is available or not.
  - If available then add accordingly.

- **For all these operations**
  - Update the previous and next link of the added new node.

Insertion at the beginning & end of the DLL using Python

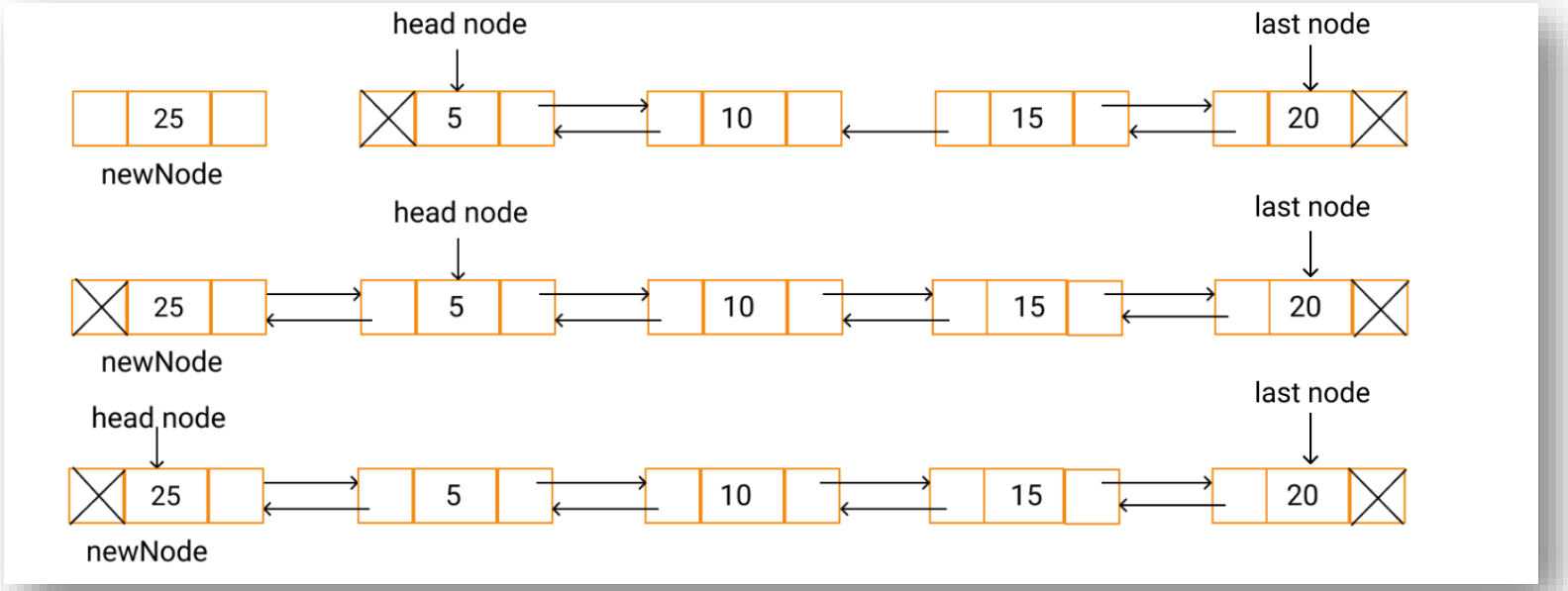
First insert one data within an empty DLL (after creation) :--

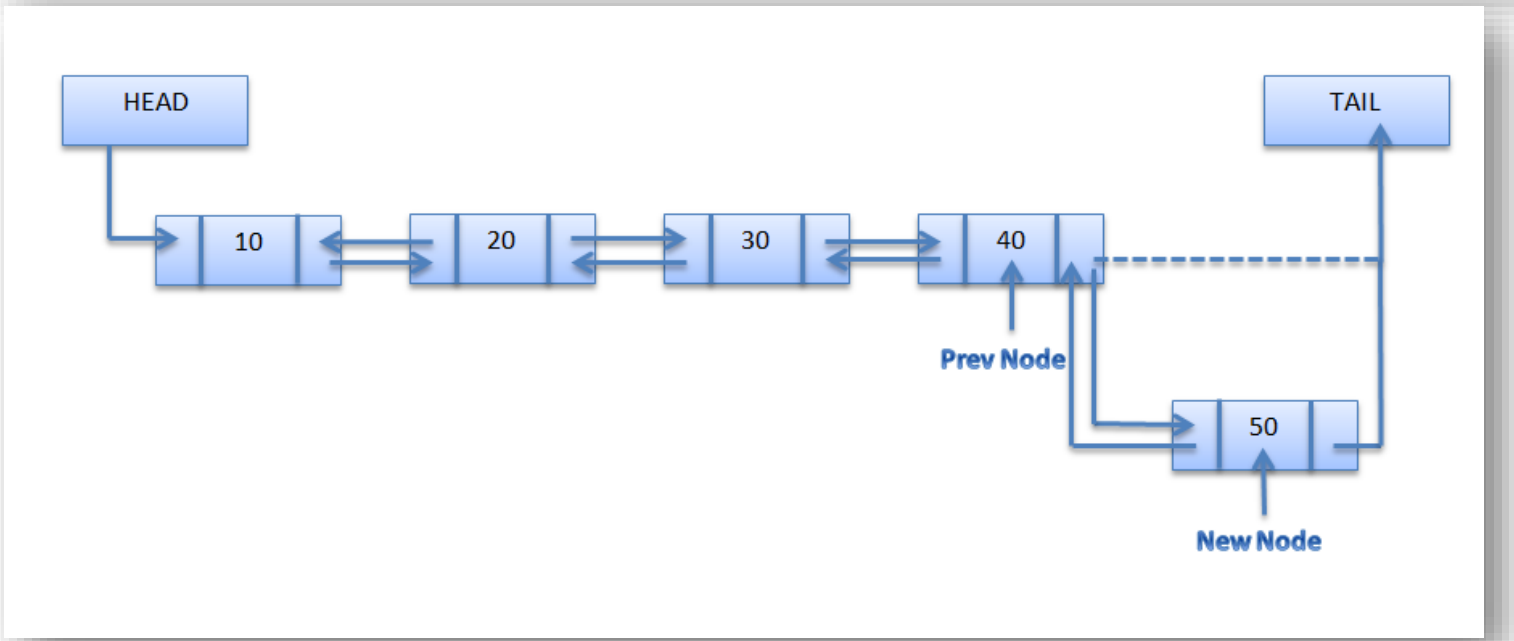
```
def insert_empty(self,data):
        if self.head is None:
            new_node = Node(data)
            self.head = new_node
        else:
            print("Linked List is not empty!")

    def add_begin(self,data):     #Insertion at beginning
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
        else:
            new_node.nref = self.head
            self.head.pref = new_node
            self.head = new_node



dl1= doublyLL()
dl1.insert_empty(10)
dl1.add_begin(20)
dl1.print_LL()
dl1.print_LL_reverse()
```

First insert one data within an empty DLL (after creation) :--

```
def insert_empty(self,data):
        if self.head is None:
            new_node = Node(data)
            self.head = new_node
        else:
            print("Linked List is not empty!")

        def add_end(self,data):          #Insertion at end
              new_node = Node(data)
              if self.head is None:
                    self.head = new_node
              else:
                    n = self.head
                    while n.nref is not None:
                          n = n.nref
                    n.nref = new_node
                    new_node.pref = n

dl1= doublyLL()
dl1.insert_empty(10)
dl1.add_begin(20)
dl1.add_end(50)
dl1.print_LL()
dl1.print_LL_reverse()
```

Examples:-


head node
last node
25
5
10
15
20
newNode
head node
last node
25
5
10
15
20
newNode
head node
last node
25
5
10
15
20
newNode

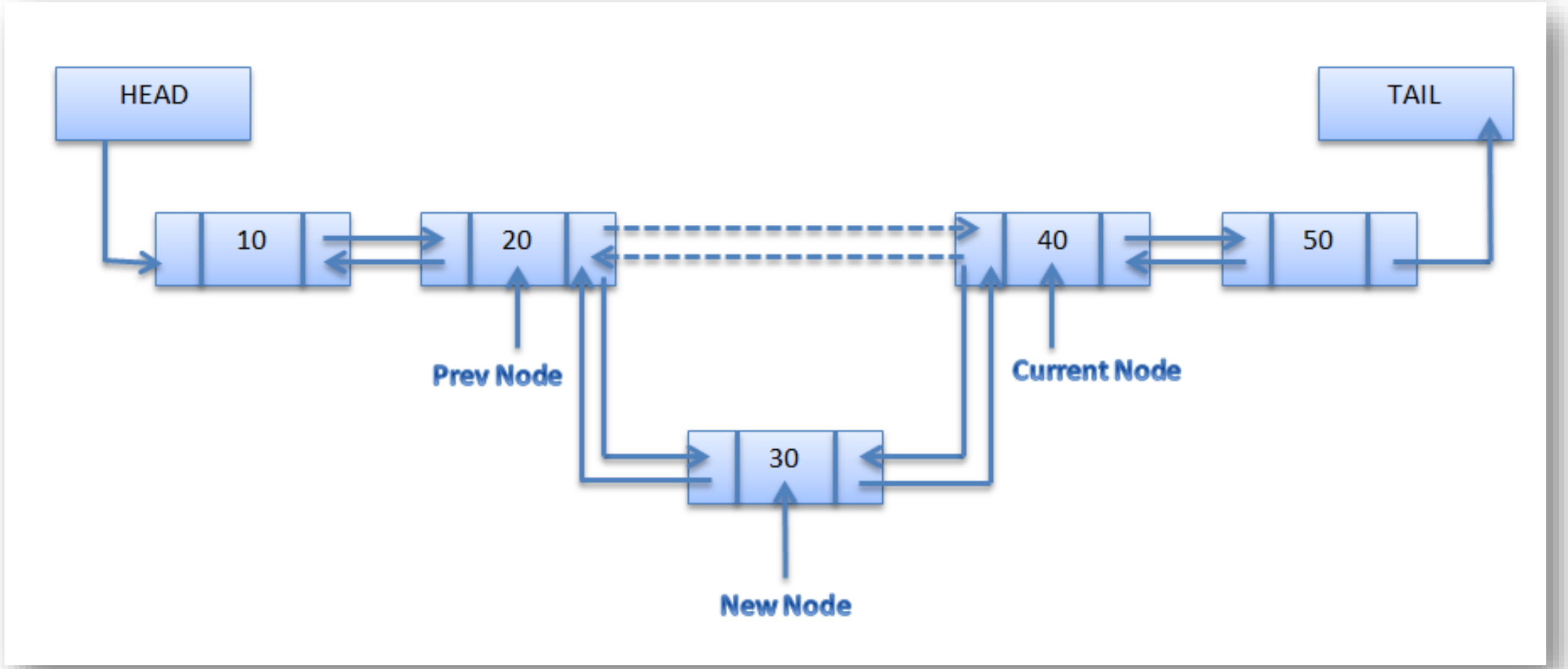

HEAD
TAIL
10
20
30
40
Prev Node
50
New Node

Insertion before and after one given node within the DLL using Python

```
  def add_after(self,data,x):
      if self.head is None:
          print("LL is empty!")
      else:
          n =self.head
          while n is not None:
              if x==n.data:
                  break
              n = n.nref
          if n is None:
              print("Given Node is not present in DLL")
          else:
              new_node = Node(data)
              new_node.nref = n.nref
              new_node.pref = n
              if n.nref is not None:
                  n.nref.pref = new_node
              n.nref = new_node

dl1 = doublyLL()
dl1.add_begin(4)
dl1.add_after(10,4)
dl1.print_LL()
dl1.print_LL_reverse()
```

```
  def add_before(self,data,x):
      if self.head is None:
          print("LL is empty!")
      else:
          n = self.head
          while n is not None:
              if x==n.data:
                  break
              n = n.nref
          if n is None:
              print("Given Node is not present in DLL")
          else:
              new_node = Node(data)
              new_node.nref = n
              new_node.pref = n.pref
              if n.pref is not None:
                  n.pref.nref = new_node
              else:
                  self.head = new_node
              n.pref = new_node

dl1 = doublyLL()
dl1.add_begin(4)
dl1.add_before(10,4)
dl1.print_LL()
dl1.print_LL_reverse()
```

Examples:-



Deletion operations within the Doubly Linked List

Deletion from the Doubly Linked List

- **Here also we are required to delete a specified node.**
  - Say, the node whose roll field is given.

- **Here also three conditions arise:**
  - Deleting the first node.
  - Deleting the last node.
  - Deleting an intermediate node.

What is to be done?

- First check the DLL is empty or not?
  - If the DLL is empty
  - # then we can't delete an element at beginning, end or anywhere in the DLL. #
  - Then put the first data into DLL (after creation).
- To insert in beginning
  - Find the *Head* position and make the *next* data as Head.
- To insert in end
  - Find the existing last node.
  - Delete that node.
  - From previous node make the *next* pointer to NULL
- To insert after or before one given position
  - Check the given data / position is available or not.
  - If available then delete accordingly.

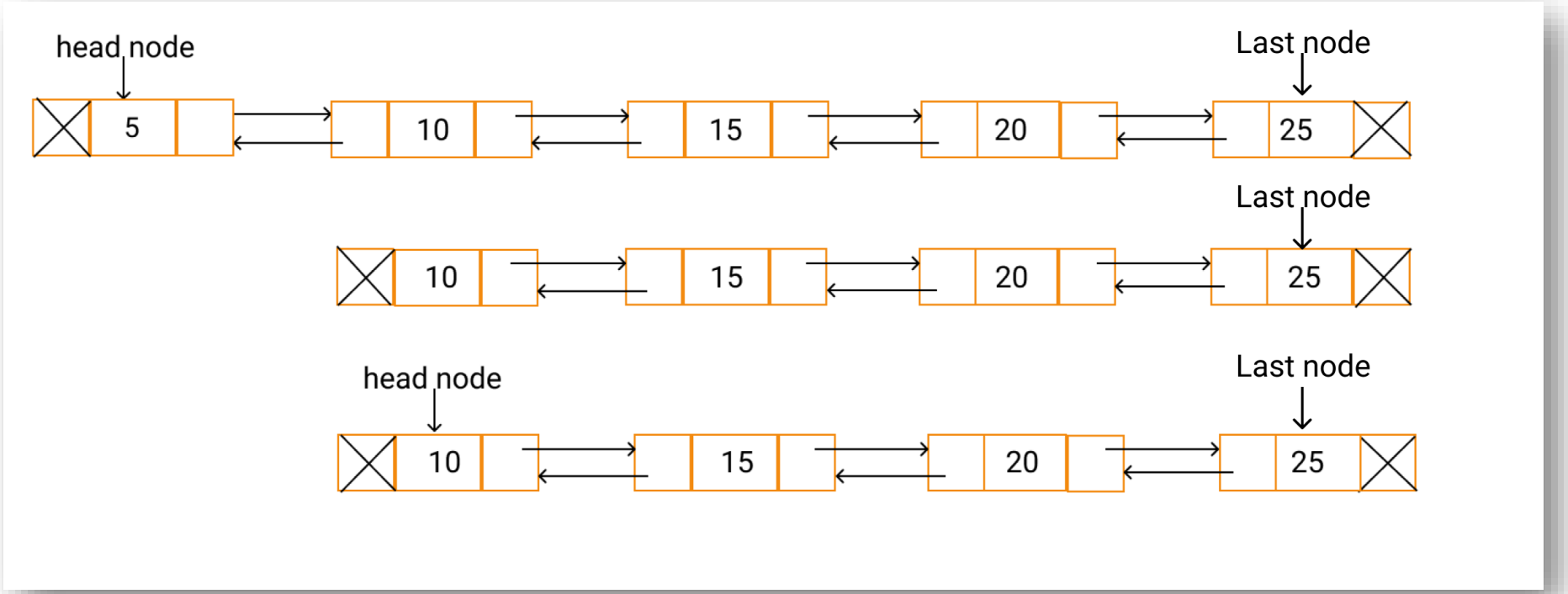
- For all these operations
  - Update the previous and next link after deleting a node.

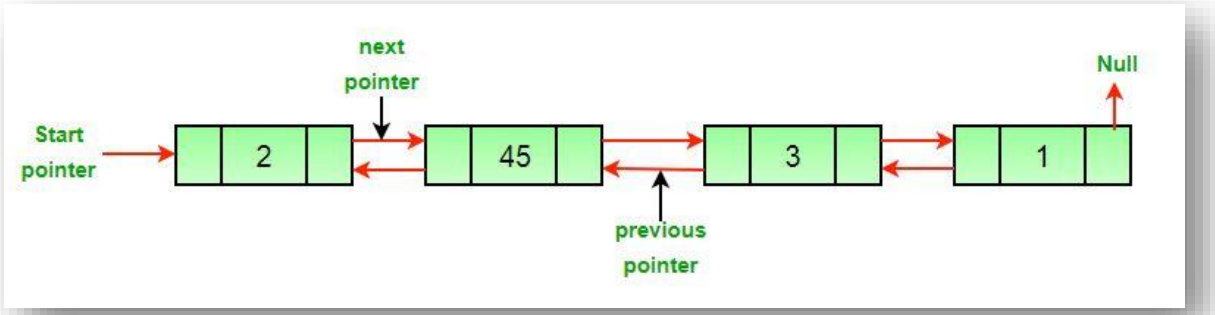
Deletion from the beginning and end of the DLL using Python

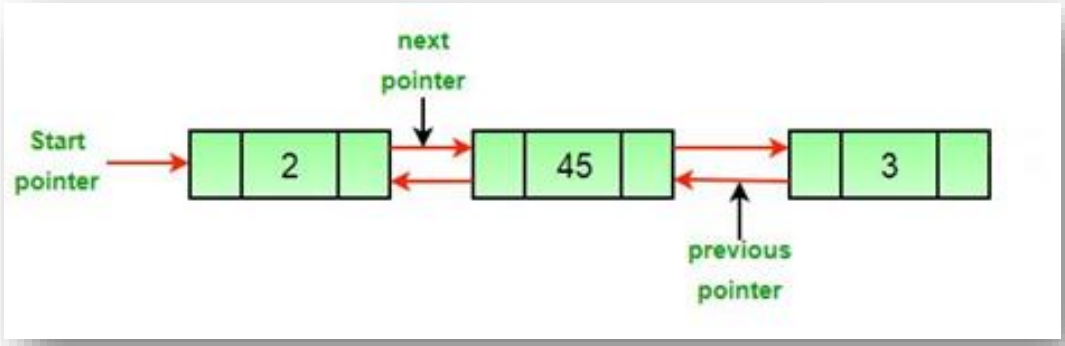
```
def delete_begin(self):
    if self.head is None:
        print("DLL is empty can't delte !")
        return
    if self.head.nref is None:
        self.head = None
        print("DLL is empty after deleting the node!")
    else:
        self.head = self.head.nref
        self.head.pref = None

dl1 = doublyLL()
dl1.add_begin(4)
dl1.add_before(10,4)
dl1.delete_begin()
dl1.print_LL()
dl1.print_LL_reverse()
```

```
def delete_end(self):
    if self.head is None:
        print("DLL is empty can't delte !")
        return
    if self.head.nref is None:
        self.head = None
        print("DLL is empty after deleting the node!")
    else:
        n = self.head
        while n.nref is not None:
            n = n.nref
        n.pref.nref = None

dl1 = doublyLL()
dl1.add_begin(4)
dl1.add_before(10,4)
dl1.delete_end()
dl1.print_LL()
dl1.print_LL_reverse()
```

Examples:-

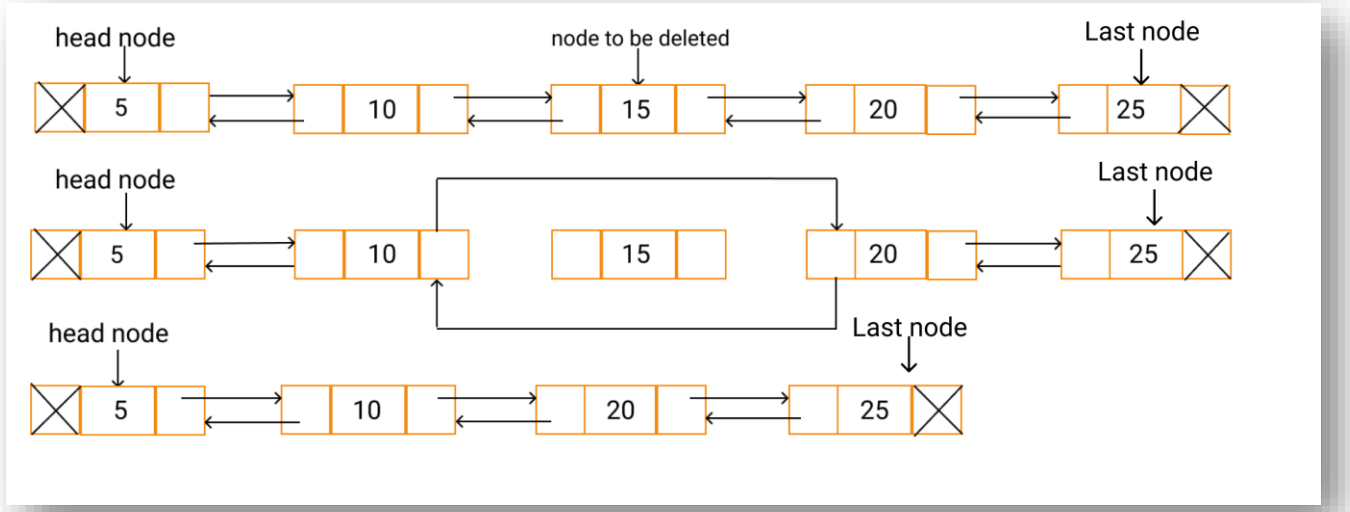

head node
Last node
5
10
15
20
25
Last node
10
15
20
25
head node
Last node
10
15
20
25


next
pointer
Null
Start
pointer
2
45
3
1
previous
pointer


next
pointer
Start
pointer
2
45
3
previous
pointer

Deletion of any element by value from the Doubly Linked List

```
def delete_by_value(self,x):
        if self.head is None:
            print("DLL is empty can't delete !")
            return
        if self.head.nref is None:
            if x==self.head.data:
                self.head = None
            else:
                print("x is not present in DLL")
            return
        if self.head.data == x:
            self.head = self.head.nref
            self.head.pref = None
            return
        n = self.head
        while n.nref is not None:
            if x==n.data:
                break
            n = n.nref
        if n.nref is not None:
            n.nref.pref = n.pref
            n.pref.nref = n.nref
        else:
            if n.data==x:
                n.pref.nref = None
            else:
                print("x is not present in dll!")
dl1 = doublyLL()
dl1.add_begin(4)
dl1.add_before(10,4)
dl1.delete_by_value(10)
dl1.print_LL()
dl1.print_LL_reverse()
```



STACK

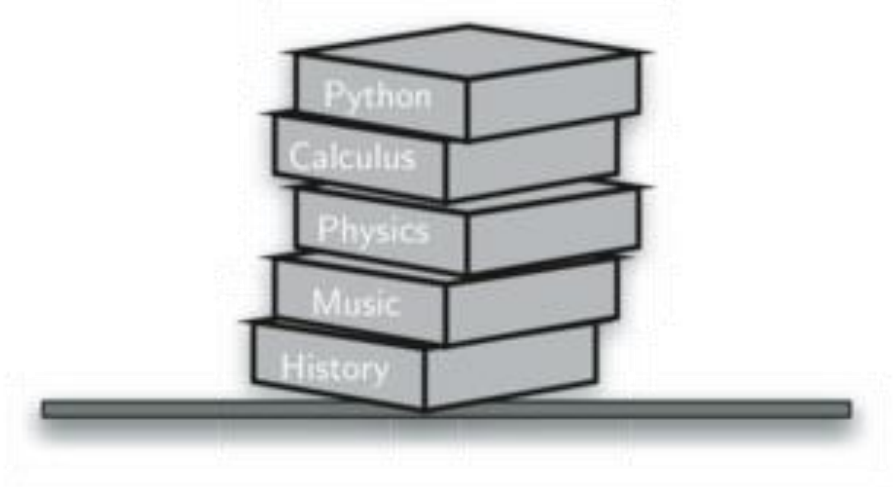
Implementation with Python

Introduction

- A *stack* is an ordered collection of items where the addition of new items and the removal of existing items always takes place at the same end.
- This end is commonly referred to as the “top”, and the opposite end is known as the “base”.
- The most recently added item is always on the top of the stack and thus will be removed first.
- In one word Stack is nothing but a sequence of objects where we can put and remove the object one after another in a sequential manner.

Introduction (contd..)

- There are many examples of stacks in everyday situations.
- Consider a stack of plates on a table, where it's only possible to add or remove plates to or from the top.
- Or imagine a stack of books on a desk. The only book whose cover is visible is the one on top. So from the top we can remove it or can add one more book on it.



Stack vs. Linked List

- One of the most useful features of stacks comes from the observation that the insertion order is the reverse of the removal order.
- Starting with a clean desk, place books on top of each other one at a time and consider what happens when you begin removing books: the order that they're removed is exactly the reverse of the order that they were placed. This ability to reverse the order of items is what makes stacks so important.
- Stack is a linear data structure which follows a particular order in which the operations are performed.
- The size of the Stack is Predefined.
- We can not add item to the stack if it's size is full and also can not remove item from stack if it is empty.

How to begin ?

- Stack can be implemented using the List.
- First create an empty stack.
- Add items to the empty stack.
- Adding items only happened after the last added element.
- We can not add an item anywhere else at any step.
- Remove the item you want from the stack; removing will be happened in a sequential manner.
- We can not remove any element at any step from the Stack.

Basic Terminology

- ***LIFO :*** Last In First Out
- The item that is added into the stack at last position will be removed from the stack first.
- ***FILO :*** First In Last Out
- The item that is added first into the stack will be removed from the stack at last position.
- Both the **LIFO** & **FILO** are conceptually same.
- Within the Stack this two mechanism are used to access the elements.

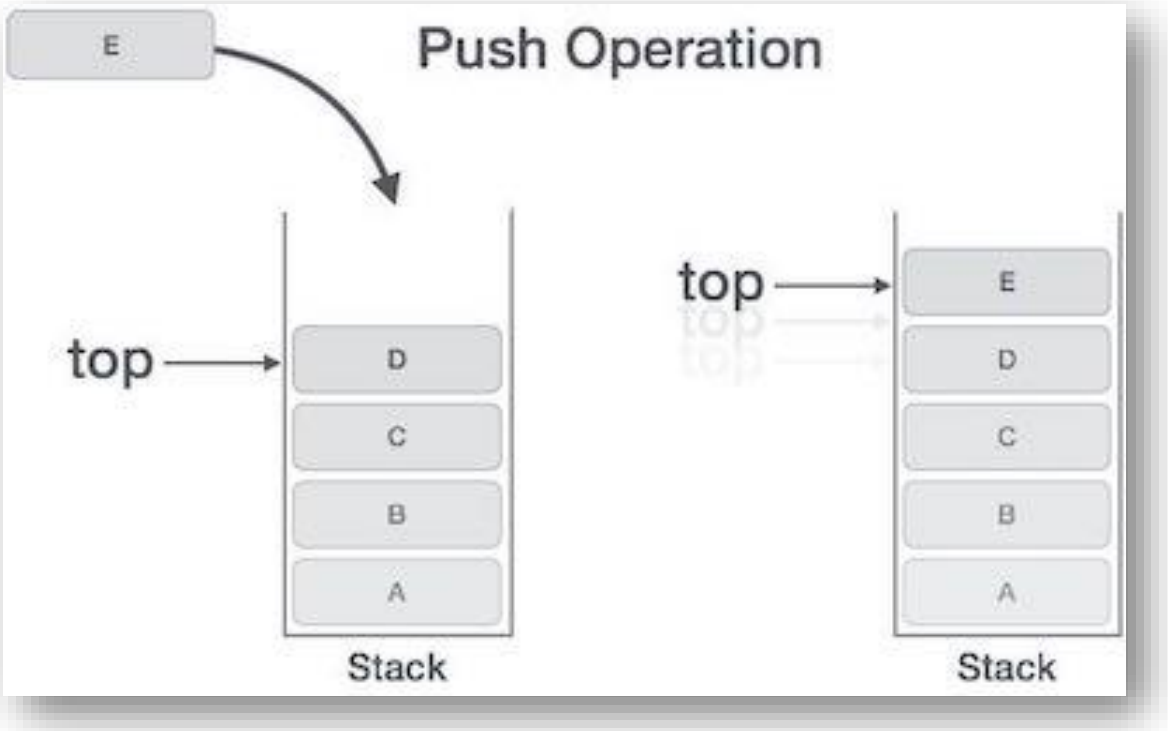
Basic Operations on Stack

- **Four main operations :**

- ***PUSH :-*** Add element in the Stack.
- ***POP :-*** Remove element from the Stack.
- ***Is Empty :-*** Check whether the Stack is empty or not?
- ***Size:-*** Retrieve the size of the Stack (Total number of elements or simply the length of the Stack)

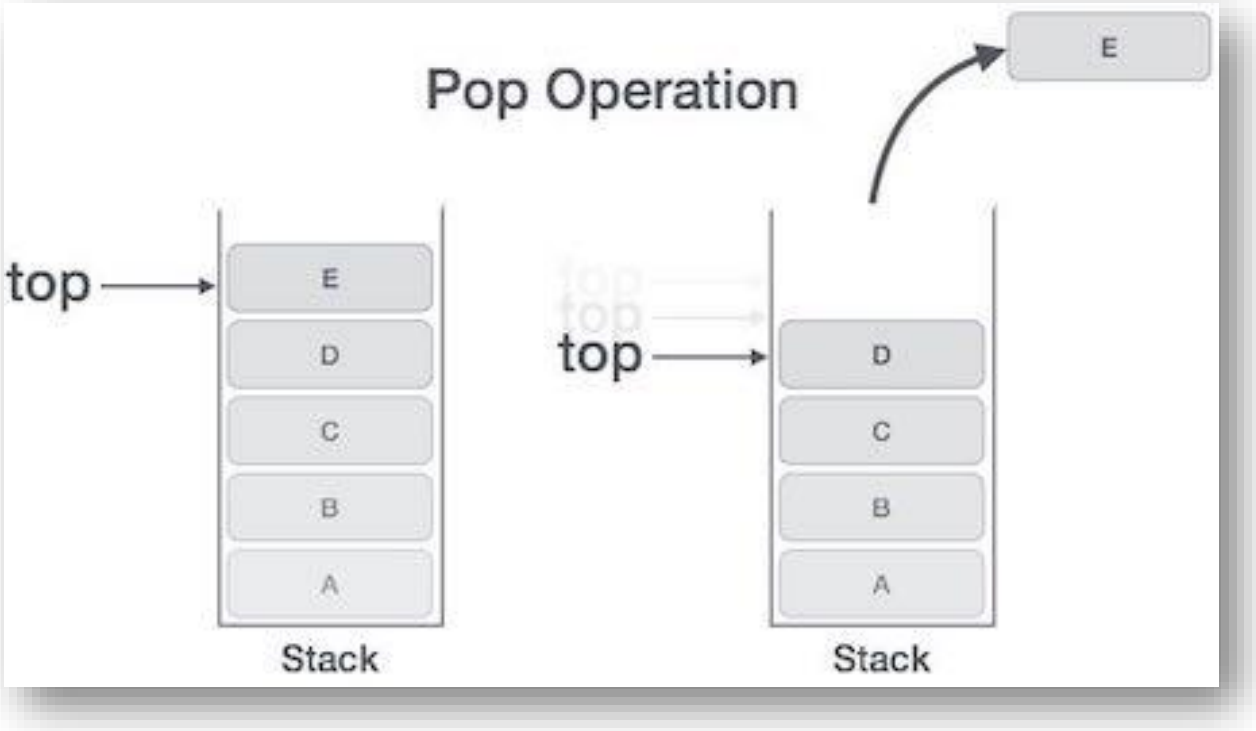
PUSH() Operation on Stack

- The process of putting a new data element onto stack is known as a Push Operation.
- Push operation involves a series of steps −
  - **Step 1** − Checks if the stack is full.
  - **Step 2** − If the stack is full, produces an error and exit.
  - **Step 3** − If the stack is not full, detect the **top element** to find next empty space.
  - **Step 4** − Add data element to the after the **top** location.
  - **Step 5** − Returns success.

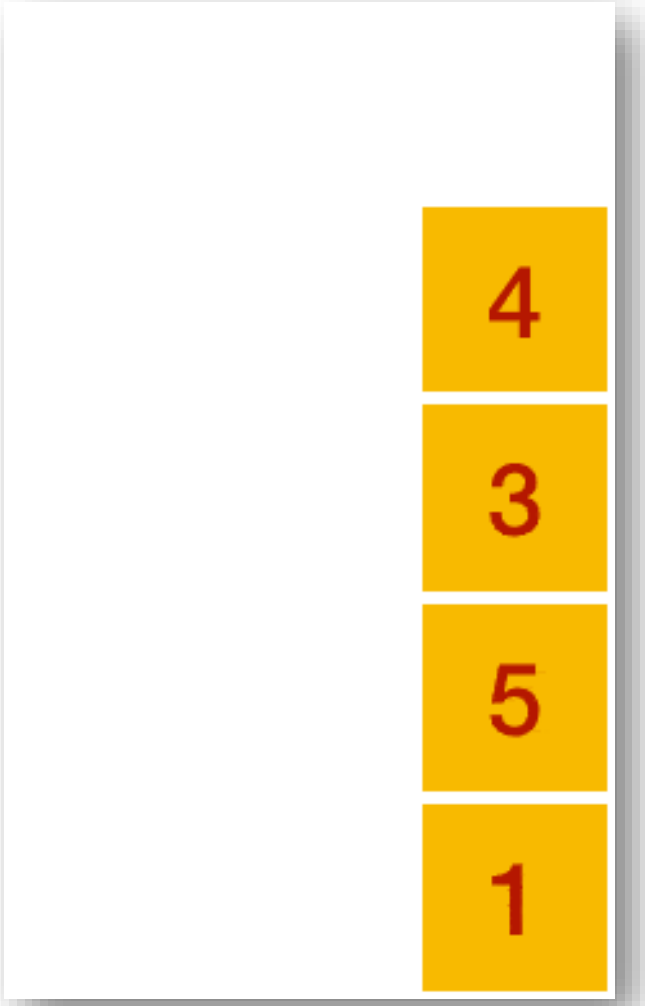


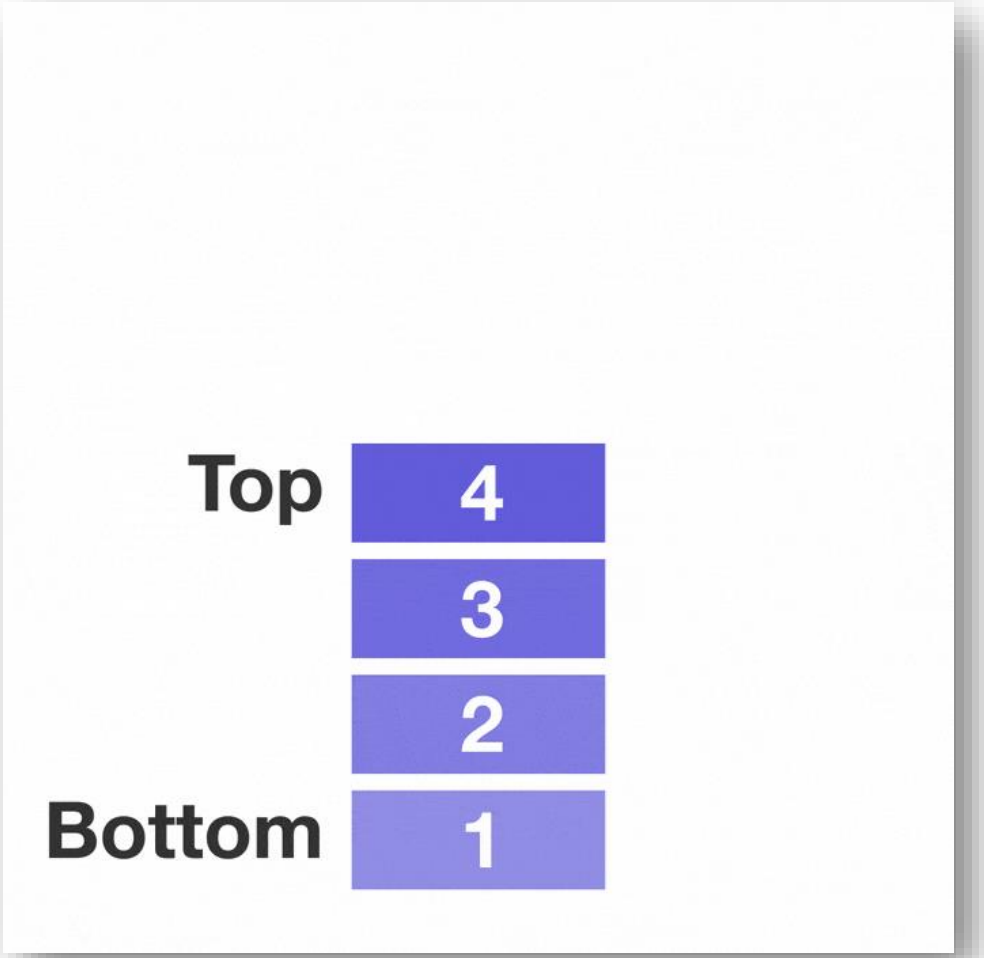
POP() operation on the stack

- Accessing the content while removing it from the stack, is known as a Pop Operation.
- A Pop operation may involve the following steps −
  - **Step 1** − Checks if the stack is empty.
  - **Step 2** − If the stack is empty, produces an error and exit.
  - **Step 3** − If the stack is not empty, find the data element at which **top** is pointing.
  - **Step 4** − Remove that element.
  - **Step 5** − Returns success.



Have a look with PUSH() & POP() operation!!





Implementing the STACK using python

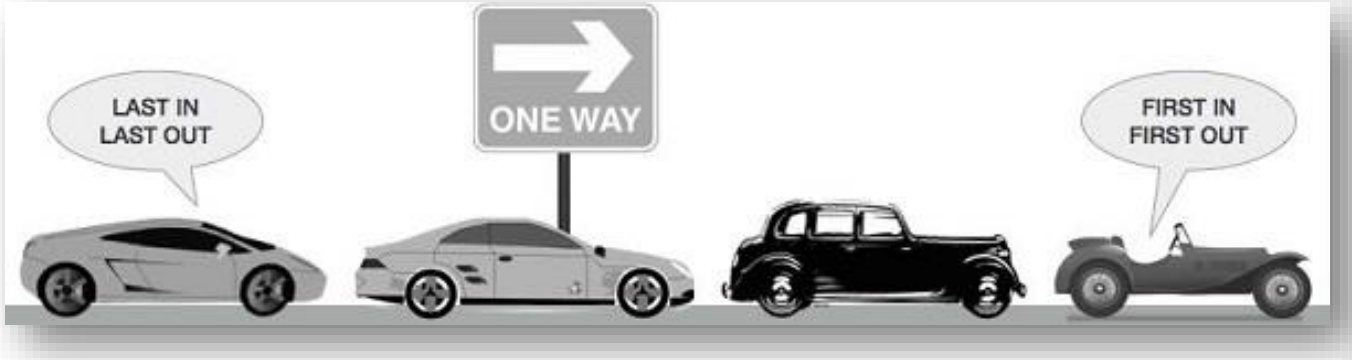
```
stack=[]
def push():
    element=input("Enter the element")
    stack.append(element)
    print("Stack is =",stack)

def pop_element():
    if not stack:
        print("Stack is empty can't pop item")
    else:
        e=stack.pop()
        print("Removed element=",e)
        print("Stack is=",stack)

while True:
    print("Select the operation 1.push 2.pop 3.quit")
    choice=int(input())
    if choice==1:
        push()
    elif choice==2:
        pop_element()
    elif choice==3:
        break
    else:
        print("Enter the correct operation!!")
```

QUEUE

Implementation with Python

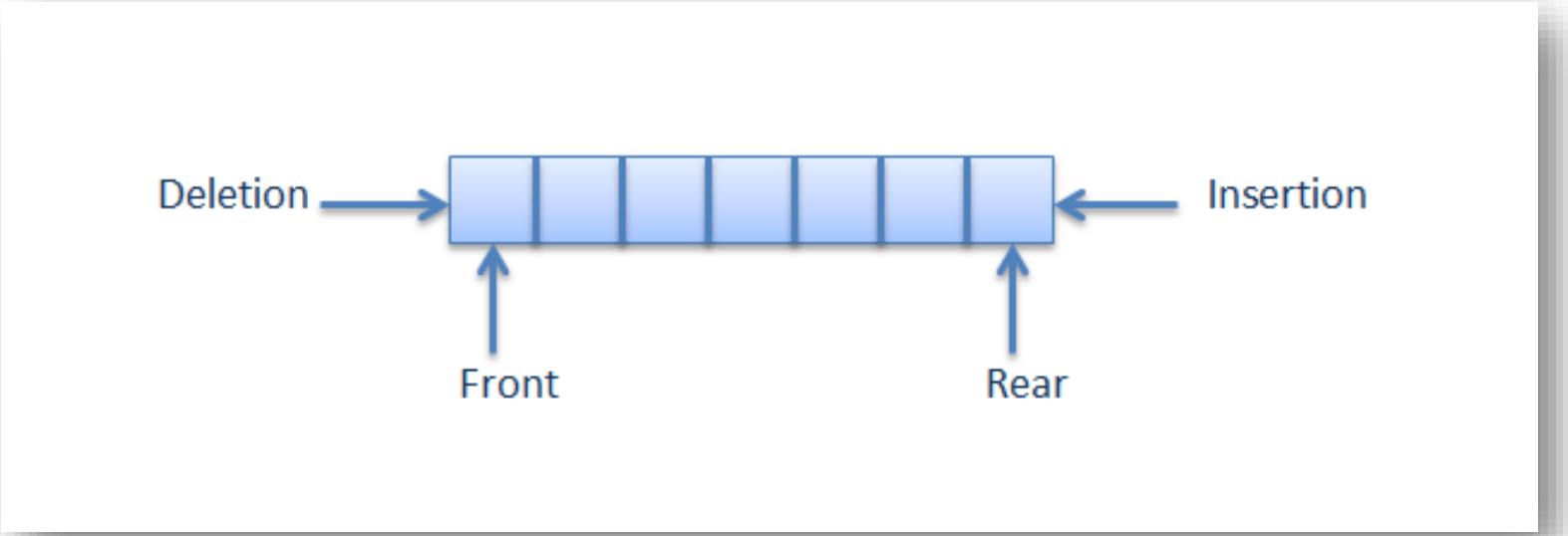
Introduction

- A Queue is a linear structure which follows a particular order in which the operations are performed.
- Queue follows a linear structure where the addition of element happens at the end of the queue and removal of element from the front or first position of queue.
- Suppose you stand in a line to buy a ticket in cinema hall. Now the first person in the queue will get the ticket first and leave the line. Also the last person in the line will get the ticket at last.
- Queue follows the same concept as real life linear operation.



Introduction (contd..)

- Queue is nothing but simple a line or a linear structure of elements.
- The last person or element in the queue is called the back /rear / tail.
- The first element in the queue is called front / head.
- The addition of element is done at back / rear.
- Deletion or removal of element is done from front / head.



Queue vs. Stack

- The difference between stacks and queues is in removing of element.
- In a stack we remove the item the most recently added.
- In a queue, we remove the item the least recently added.
- Unlike stacks, a queue is open at both its ends.
- One end is always used to insert data (enqueue) and the other is used to remove data (dequeue).
- Like stack, queue is also an ordered list of elements of similar data types.
- Queue is a FIFO(First in First Out) structure.
- Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.
- peek( ) function is often used to return the value of first element without dequeuing it.

How to begin ?

- Queue can be implemented using the List.
- First create an empty queue.
- Add items to the empty stack.
- Adding items only happened after the last added element, like stack.
- We can not add an item anywhere else at any step.
- Remove the item you want from the stack; removing will be happened in a sequential manner.
- We can not remove any element at any step from the Stack.

Basic Terminology

- ***FIFO :*** First In First Out
- The item that is added into the queue at first position will be removed from the queue first.
- ***LILO :*** Last In Last Out
- The item that is added last into the queue will be removed from the queue at last position.
- Both the ***FIFO & LILO*** are conceptually same.
- Within the Queue this two mechanism are used to access the elements.

Basic Operations on Stack

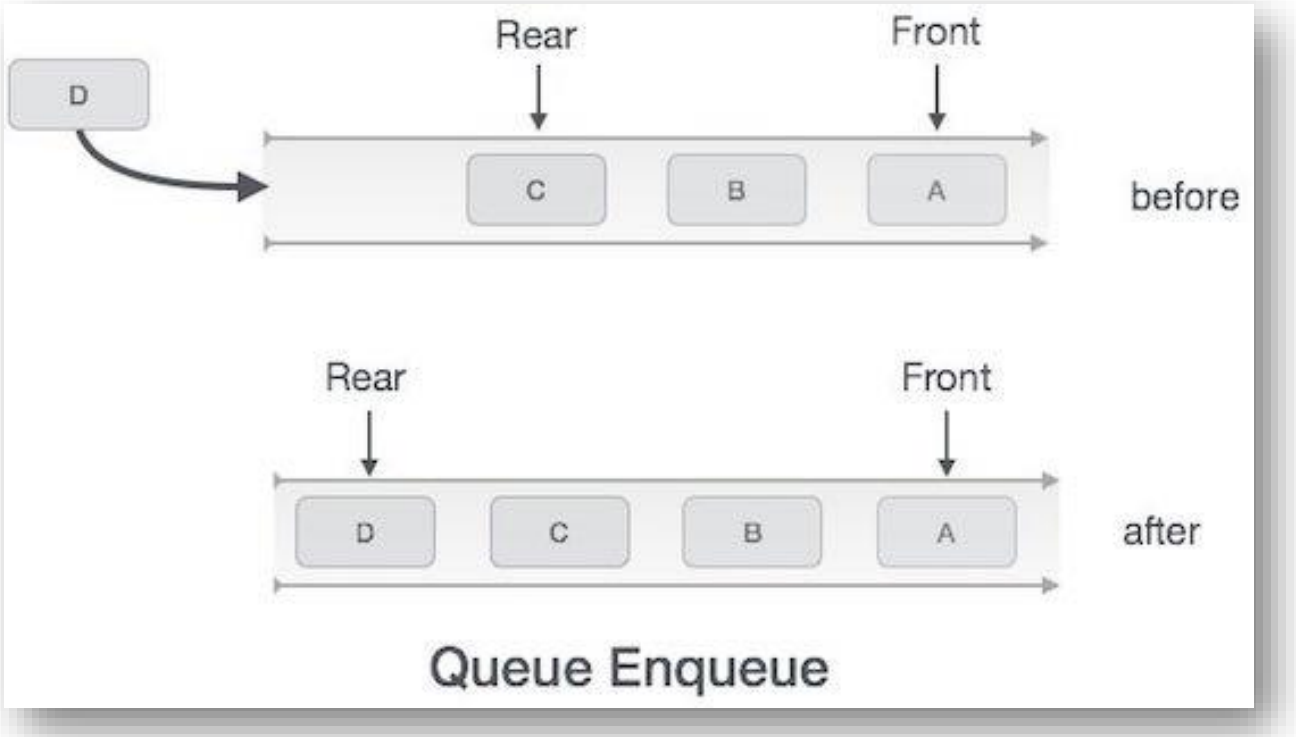
➢ Four main operations :

- ***enqueue :-*** Add element in the Queue.
- ***dequeue :-*** Remove element from the Queue.
- ***Is Empty :-*** Check whether the Queue is empty or not?
- ***Is full:-*** Check whether the queue is full or not?

You can not add item if the queue is full as well as can not remove item from queue if it is empty.

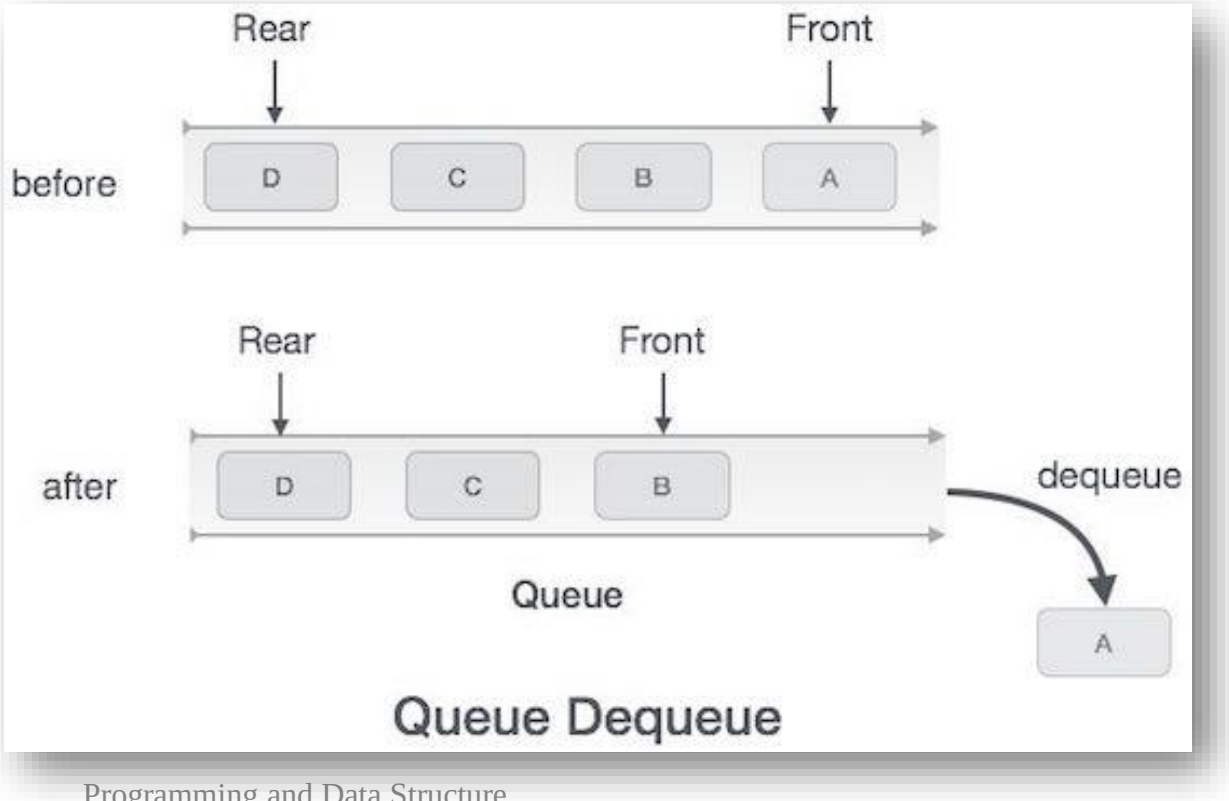
Enqueue() operation on Queue

- Queues maintain two identifier, **front** and **rear**.
- **The following steps should be taken to enqueue (insert) data into a queue −**
  - **Step 1** − Check if the queue is full.
  - **Step 2** − If the queue is full, produce overflow error and exit.
  - **Step 3** − If the queue is not full, find **rear** position to identify the next empty space.
  - **Step 4** − Add data element to that queue location, where the rear is pointing.
  - **Step 5** − return success.



Dequeue() operation on Queue

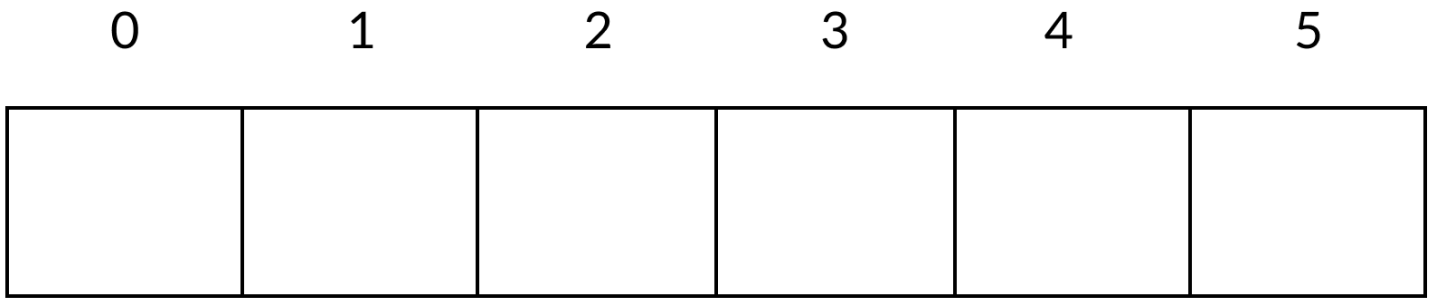
- Queues maintain two identifier, **front** and **rear**.
- **The following steps should be taken to enqueue (insert) data into a queue −**
  - **Step 1** − Check if the queue is empty.
  - **Step 2** − If the queue is empty, produce underflow error and exit.
  - **Step 3** − If the queue is not empty, identify the data where **front** is pointing.
  - **Step 4** − Remove the **front** data and make the next available data as **front.**
  - **Step 5** − Return success



Have a look with queue operations !!

Queue Operations

Front = Rear = -1



Empty Queue

faceprep

Implementing the QUEUE using python

```
queue=[]
def enqueue():
    element=input("Enter the element")
    queue.append(element)
    print("Queue =",queue)

def dequeue():
    if not queue:
        print("queue is empty can't pop item")
    else:
        e=queue.pop(0)
        print("Removed element=",e)
        print("Queue is=",queue)

while True:
    print("Select the operation 1.ENQUEUE  2.DEQUEUE  3.QUIT")
    choice=int(input())
    if choice==1:
        enqueue()
    elif choice==2:
        dequeue()
    elif choice==3:
        break
    else:
        print("Enter the correct operation!!")
```