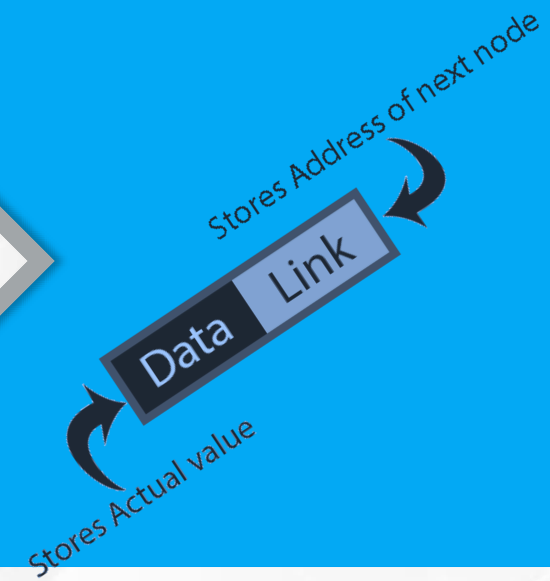
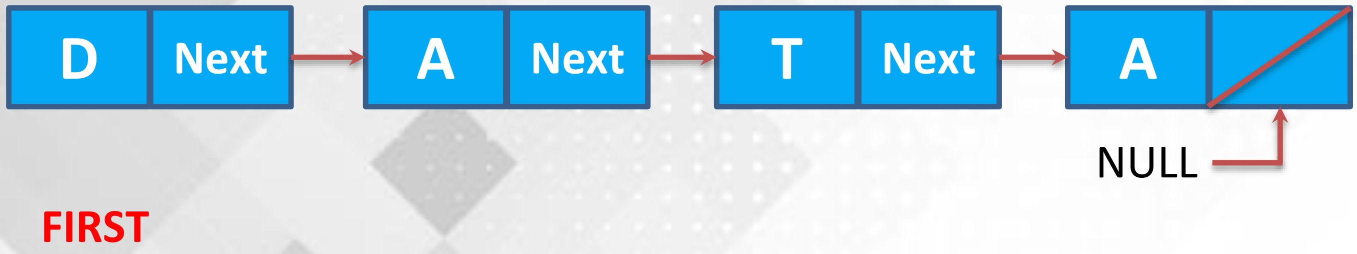
Data Structure

Linked List

Linear Data Structure



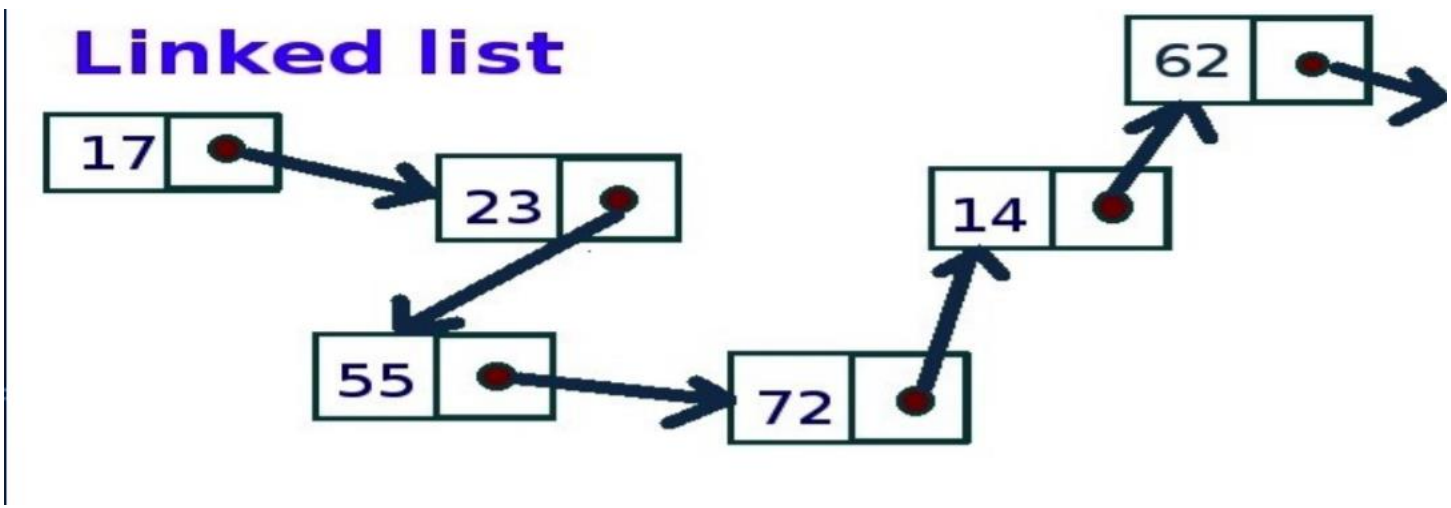


Learning Outcomes

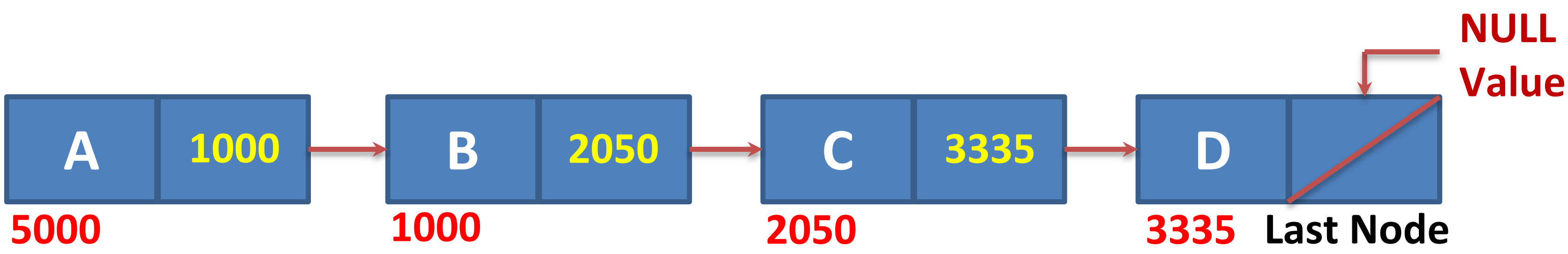
- Overview of Linked list
- Types of Linked List
- Basic operations on singly linked list :
  - Insertion of a new node in
  - the beginning of the list, at
  - the end of the list, after a
  - given node, before a given
  - node, Deleting the first and
  - last node from a linked list,
  - Count the number of nodes in linked list.
- Overview of circular linked list
- Difference between circular linked list and singly linked list
- Overview of doubly linked list
- Applications of linked list

Linked List

- It is an ordered set which consist of variable number of elements.
- Here elements are logically adjacent to each other but physically not.
- It is a collection of nodes and each node consist of two parts.
- 1) Value of a node 2) Address of next node.



Linked Storage Representation



A linked List

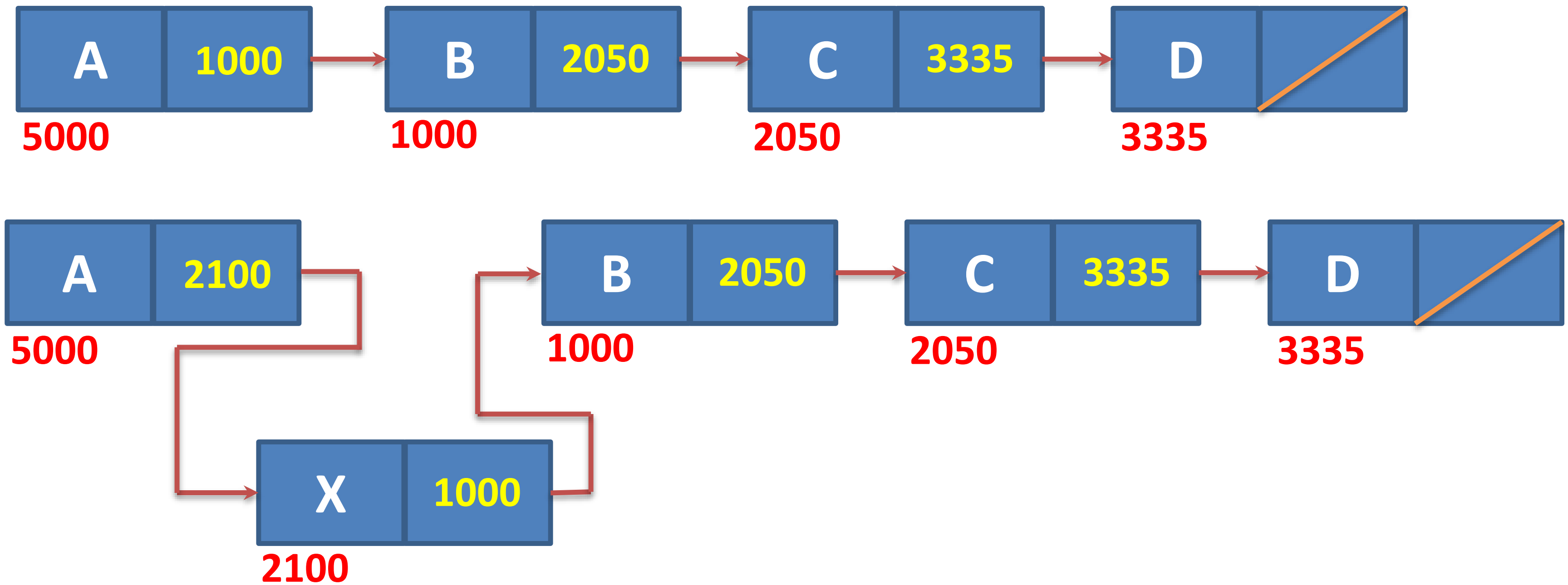
- The linked allocation method of storage can result in both efficient use of computer storage and computer time.
  - A linked list is a **non-sequential collection** of data items.
  - Each **node** is **divided** into **two parts**, the **first part** represents the **information** of the element and the **second part** contains the **address of the next mode**.
  - The **last node** of the list does not have successor node, so **null value** is stored as the address.
  - It is possible for a list to have no nodes at all, such a list is called empty list.

Linked List Representation:

```
class Node:
    def __init__(self,info):
        self.info=info
        self.link=None
```

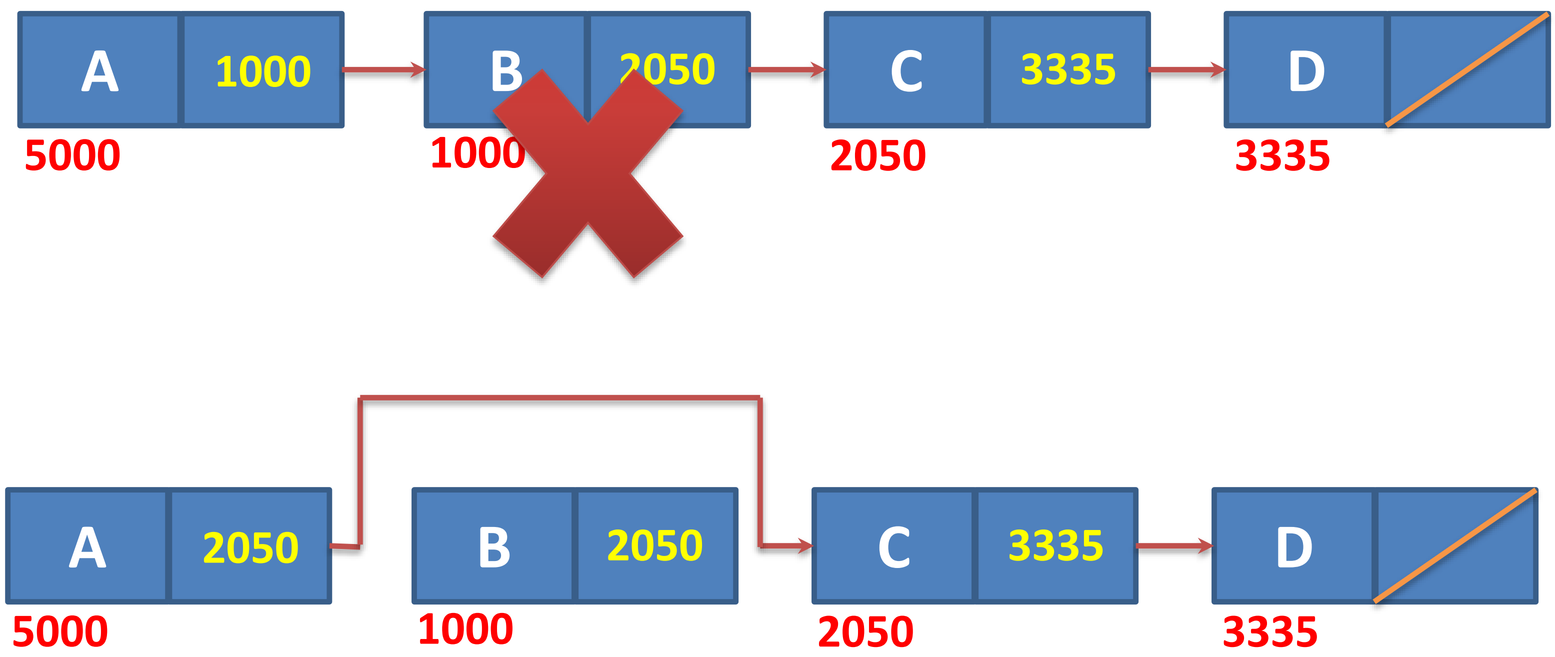
Pros & Cons of Linked Allocation

- **Insertion Operation**
  - we have an n elements in list and it is required to insert a new element between the first and second element, what to do with sequential allocation & linked allocation?
  - Insertion operation is more efficient in Linked allocation.



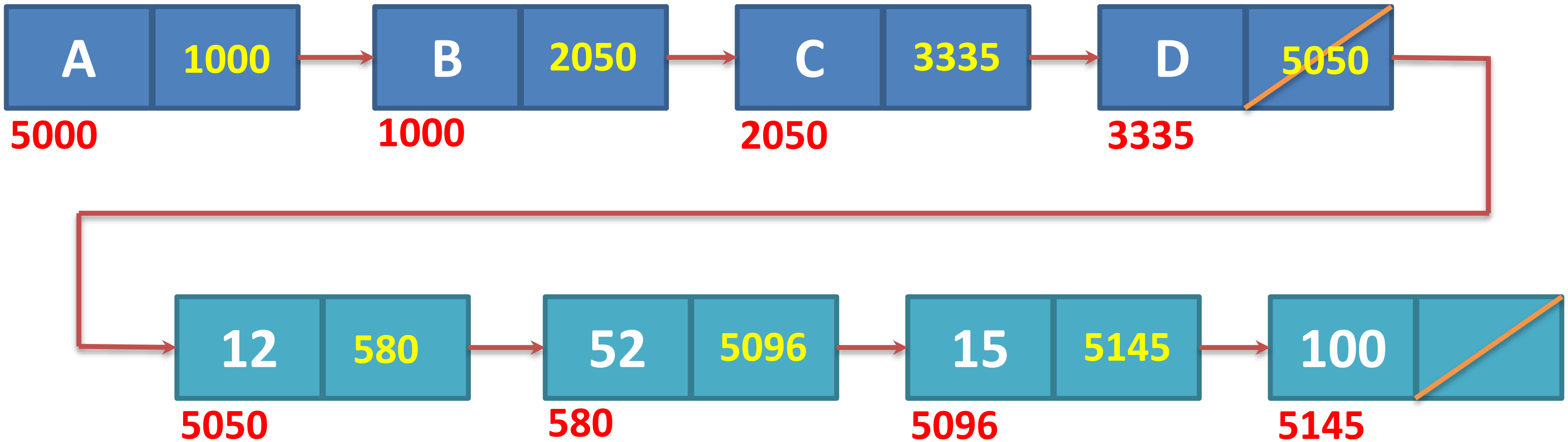
Pros & Cons of Linked Allocation

- **Deletion Operation**
  - Deletion operation is more efficient in Linked Allocation



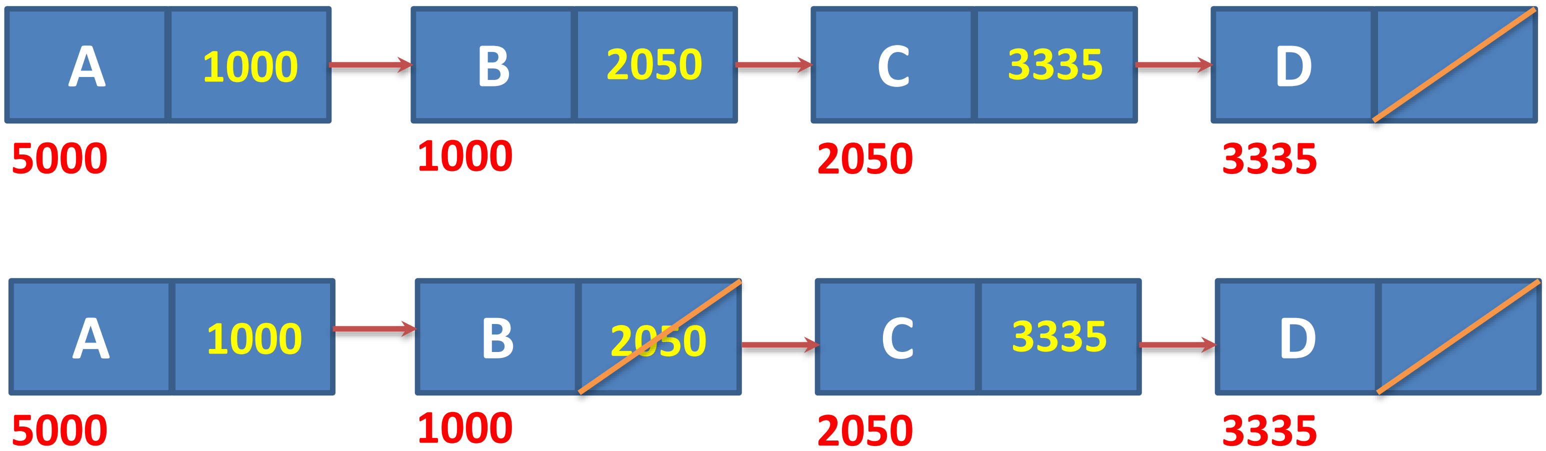
Pros & Cons of Linked Allocation

- **Search Operation**
  - **If particular node in the list is required**, it is necessary to follow links from the first node onwards until the desired node is found, in this situation **it is more time consuming** to go through linked list than a sequential list.
  - Search operation is more time consuming in Linked Allocation.
- **Join Operation**
  - Join operation is more efficient in Linked Allocation.



Pros & Cons of Linked Allocation

- Split Operation
  - Split operation is more efficient in Linked Allocation



Operations & Type of Linked List

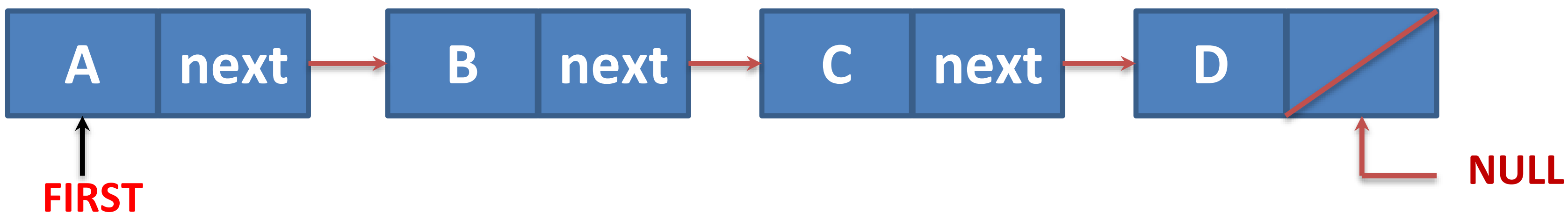
Operations on Linked List

- Insert
  - Insert at first position
  - Insert at last position
  - Insert into ordered list
- Delete
- Traverse list (Print list)
- Copy linked list

Types of Linked List

- Singly Linked List
- Doubly Linked List
- Circular Singly Linked List
- Circular Doubly Linked List

Singly Linked List



- It is basic type of linked list.
- It is a collection of variable number of nodes.
- Each **node** is **divided** into **two parts**, the **first part** represents the **information** of the element and the **second part** contains the **address of the next mode**.
- Last node's pointer is null.
- First node address is available with pointer variable **FIRST**.
- **Limitation** of singly linked list is **we can traverse only in one direction**, forward direction.

Doubly Linked Linear List

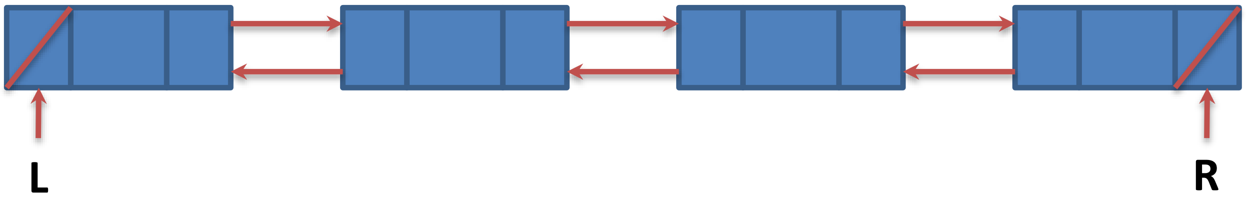
- In certain Applications, it is very desirable that a list be traversed in either forward or reverse direction.
- Each node consist of three parts.
- First part contains address of previous node (**Predecessor)**.
- Second part contains value of a node,
- Third part contains address of next node (**Successor)**.
- The links are used to denote **Predecessor** and **Successor** of node.
- The link denoting its **predecessor** is called **Left Link.**
- The link denoting its **successor** is called **Right Link.**
- A list containing this type of node is called **doubly linked list** or **two way chain**.

LPTR	INFO	RPTR

Doubly Linked Linear List

- Typical node of doubly linked linear list contains INFO, LPTR RPTR Fields
- **LPTR** is pointer variable pointing to Predecessor of a node
- **RPTR** is pointer variable pointing to Successor of a node
- Left most node of doubly linked linear list is called **L**, **LPTR** of node L **is always NULL**
- Right most node of doubly linked linear list is called **R**, **RPTR** of node **R is always NULL**
- **Limitation:** It occupies more memory than singly linked list.

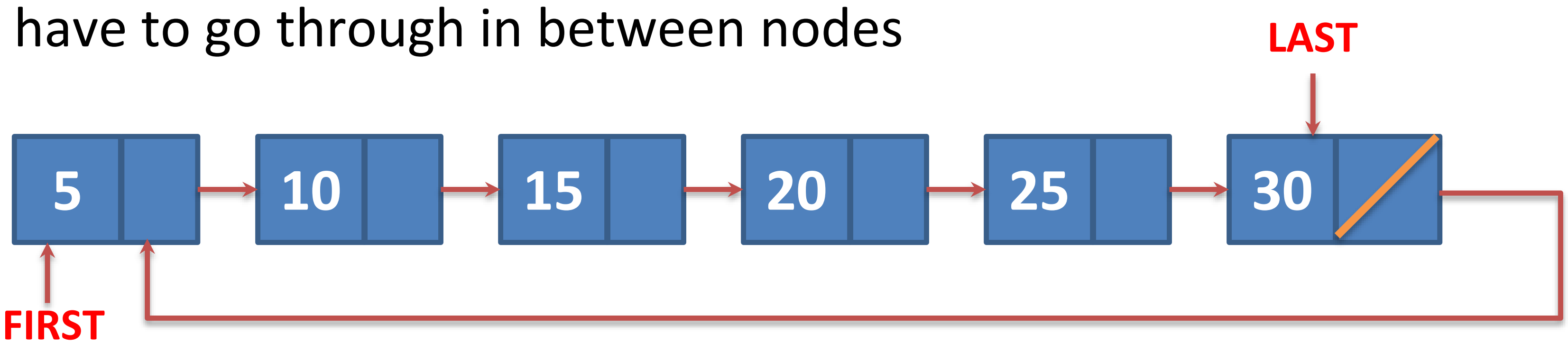
Typical node of Doubly Linked List



Doubly linked linear list

Circular Singly Linked Linear List

- If we **replace NULL** pointer of the **last node** of Singly Linked Linear List with the **address of its first node**, that list becomes circularly linked linear list or **Circular Singly linked List**.
- **FIRST** is the address of first node of Circular List
- **LAST** is the address of the last node of Circular List
- **Advantages of Circular List**
  - In circular list, every node is accessible from given node
  - It saves time when we have to go to the first node from the last node. It can be done in single step because there is no need to traverse the in between nodes. But in double linked list, we will have to go through in between nodes

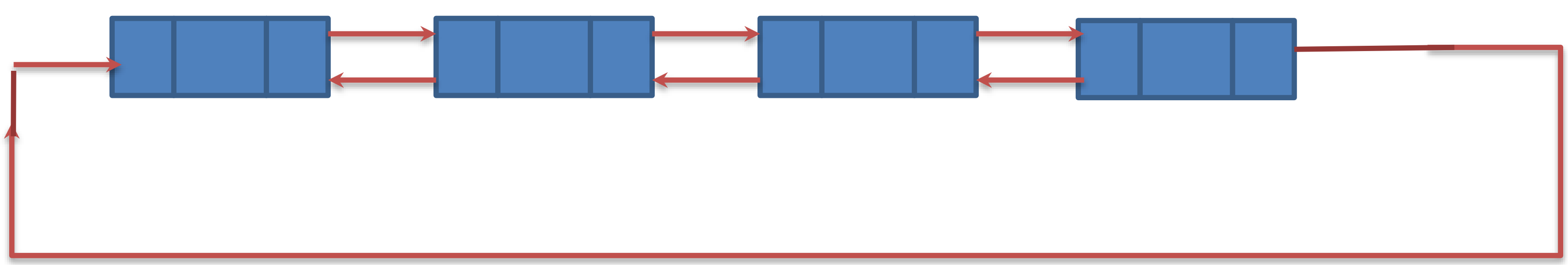


Circular Singly Linked Linear List

- **Disadvantages of Circular List**
  - It is not easy to reverse the linked list
  - If proper care is not taken, then the problem of infinite loop can occur
  - If we at a node and go back to the previous node, then we can not do it in single step. Instead we have to complete the entire circle by going through the in between nodes and then we will reach the required node

Circular Doubly Linked Linear List

- Circular doubly linked list is a more complex type of data structure in which a node contain pointers to its previous node as well as the next node.
- Circular doubly linked list doesn't contain NULL in any of the node.
- The last node of the list contains the address of the first node of the list.
- The first node of the list also contain address of the last node in its previous pointer.

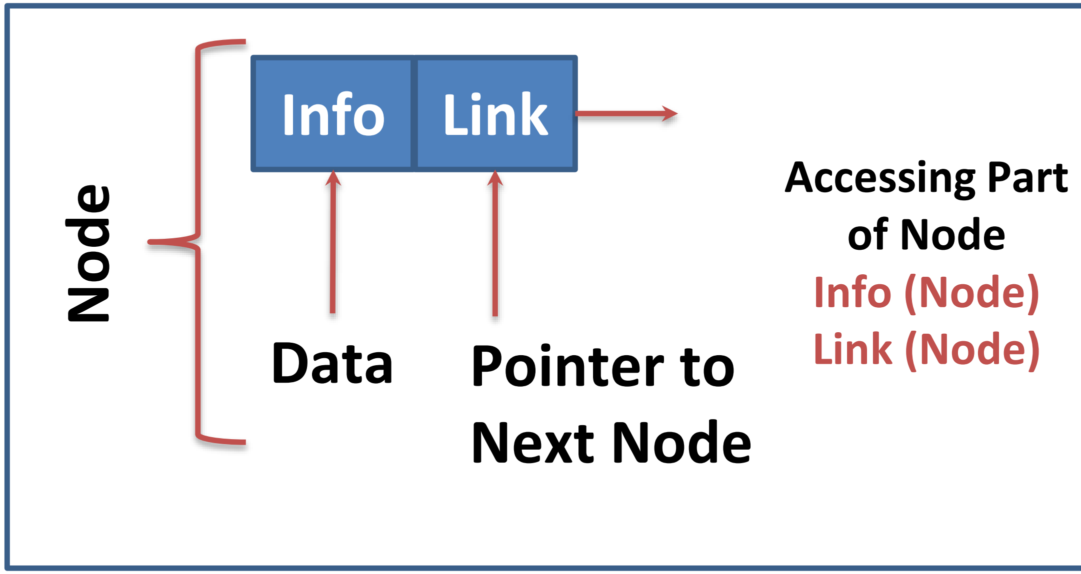


Circular Doubly Linked Linear List

- Due to the fact that a circular doubly linked list contains three parts in its structure therefore, it demands more space per node and more expensive basic operations.
- However, a circular doubly linked list provides easy manipulation of the pointers and the searching becomes twice as efficient.

Node Structure of Singly List

Typical Node



Python Structure to represent a node

```
class Node:
    def __init__(self,info):
        self.info=info
        self.link=None
```

Algorithms for singly linked list

- Insertion of a new node in the beginning of the list,
- Insertion of a new node at the end of the list,
- Insertion of a new node after a given node,
- Insertion of a new node before a given node,
- Insertion of a new node in sorted linked list,
- Deleting the first and last node from a linked list,
- Searching a node in Linked List,
- Count the number of nodes in linked list

Insert New Node at beginning of Linked List

- In order to insert a new node at the beginning of the list we have to follows the below steps:
- First we have to create a new node.
- After creating new node we have to check weather linked list is empty or not. We have two possibilities:
- (A) Linked List is empty (FIRST=None). In this case the newly created node becomes the first node of linked list.
- (B) Linked List is not empty (FIRST ≠ None). In this case we can insert new node at the beginning of linked list. Now new node becomes the first node.

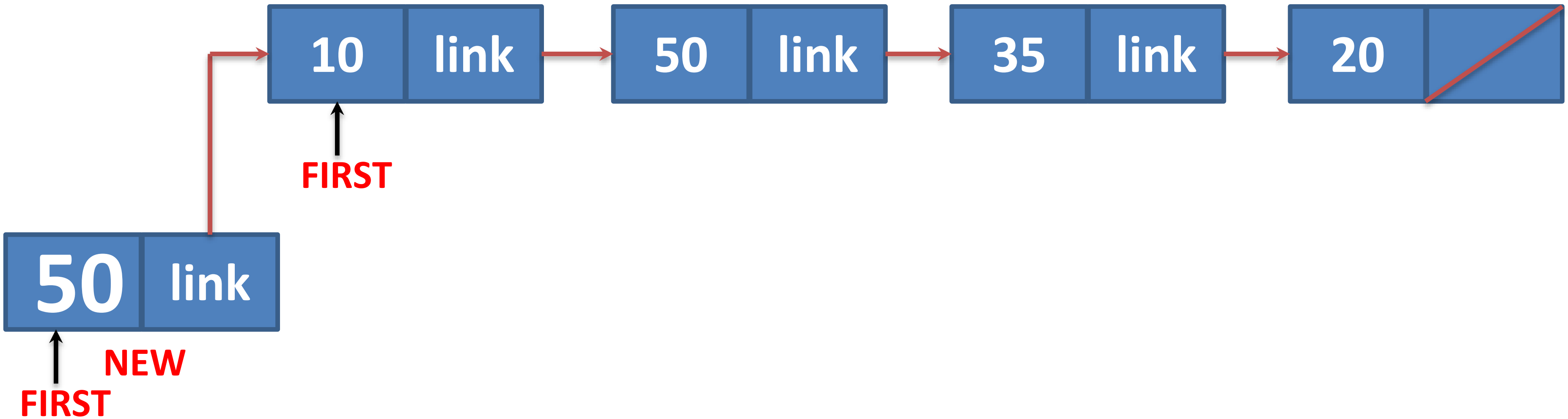
Function: INSERT(X,First)

- This function **inserts a new node at the first position** of Singly linked list.
- This function returns address of **FIRST** node.
- **X** is a new element to be inserted.
- **FIRST** is a **pointer to the first element** of a Singly linked linear list.
- Typical node contains **INFO** and **LINK** fields.
- **NEW_Node** is a temporary variable.

Insert New Node at beginning of Linked List

```
Step 1: NEW_NODE<-Node(Value)
Step 2: If self. FIRST is None then
                self. FIRST = NEW_NODE
                  Else
                NEW_NODE.LINK = self. FIRST
                  self. FIRST = NEW_NODE
Step 3: Exit
```

Example: INSERT(50,FIRST)




```
NEW_NODE.LINK = self.FIRST
 self.FIRST = NEW_NODE
```

Insert New Node at beginning of Linked List

```
class Node:
    def __init__(self,info):
        self.info=info
        self.link=None
class singlylinkedlist:
    def __init__(self):
        self.FIRST=None
        self.PREV=None
        self.SAVE=None

    def InsertBegin(self,New_Node):
        if(self.FIRST is None):
            self.FIRST=New_Node
        else:
            New_Node.link=self.FIRST
            self.FIRST=New_Node
```

Insert New Node at End of Linked List:-

- In order to insert a new node at the end of the list we have to follows the below steps:
- (1) First we have to create a new node.
- (2) After creating new node we have to check weather linked list is empty or not. We have two possibilities:
- (A) Linked List is empty (FIRST=None). In this case the newly created node becomes the first node of linked list.
- (B) Linked List is not empty (FIRST ≠ None). In this case we have to traverse from first node to last node in the list and insert new node at the end of linked list.

Function: INSEND(X, FIRST)

- This function **inserts** a new node at the **last position** of linked list.
- This function returns address of **FIRST** node.
- **X** is a new element to be inserted.
- **FIRST** is a **pointer to the first element** of a Singly linked linear list.
- Typical node contains **INFO** and **LINK** fields.
- **NEW_Node** is a temporary variable.

Function: INSEND(X, First) Cont…

```
Step 1: NEW_NODE<-Node(Value)
Step 2: If self. FIRST is None then
             self. FIRST = NEW_NODE
            Else
           self.SAVE<-self.FIRST
           Repeat while Self.SAVE.Link is not None
                self.SAVE<-= self.SAVE.LINK
           self.SAVE.link<-New_Node
Step 3: Exit
```

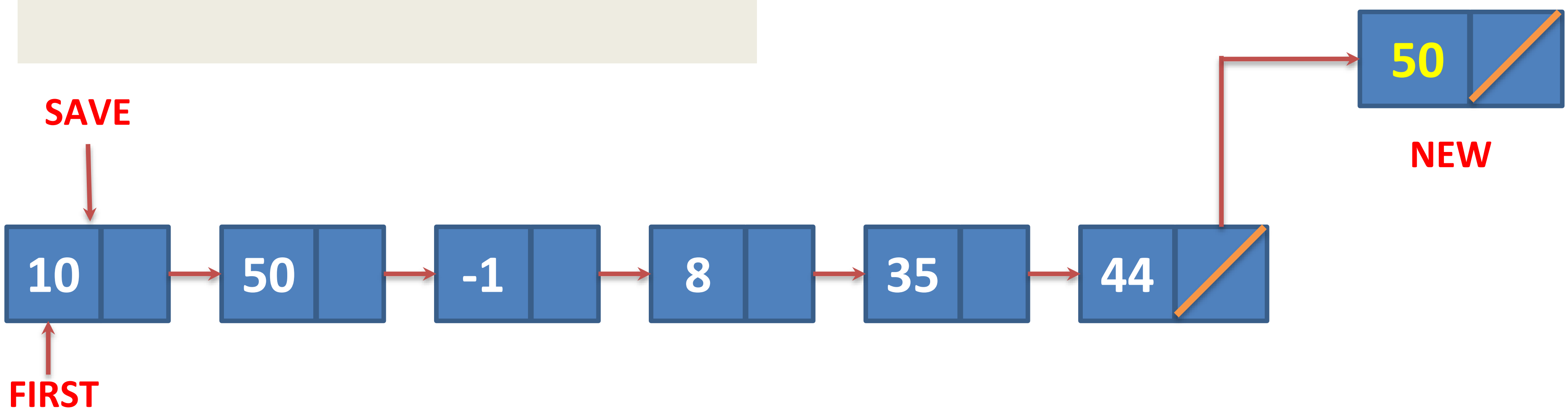
Function: INSEND(50, FIRST)

```
1. [Is the list empty?]
     If     FIRST = None
     Then   Return (NEW)

2. [Initialize search for
    a last node]
     SAVE ← FIRST
3. [Search for end of list]
 Repeat while LINK (SAVE) ≠ NULL
     SAVE ← LINK (SAVE)
```

```
4. [Set link field of last node
    to NEW]
     LINK (SAVE) ← NEW

5. [Return first node pointer]
     Return (FIRST)
```



Function: INSEND(X, First) Cont…

```
def InsertEnd(self,New_Node):
    if(self.FIRST is None):
        self.FIRST=New_Node
    else:
        self.SAVE=self.FIRST
        while(self.SAVE.link is not None):
            self.SAVE=self.SAVE.link
        self.SAVE.link=New_Node
```

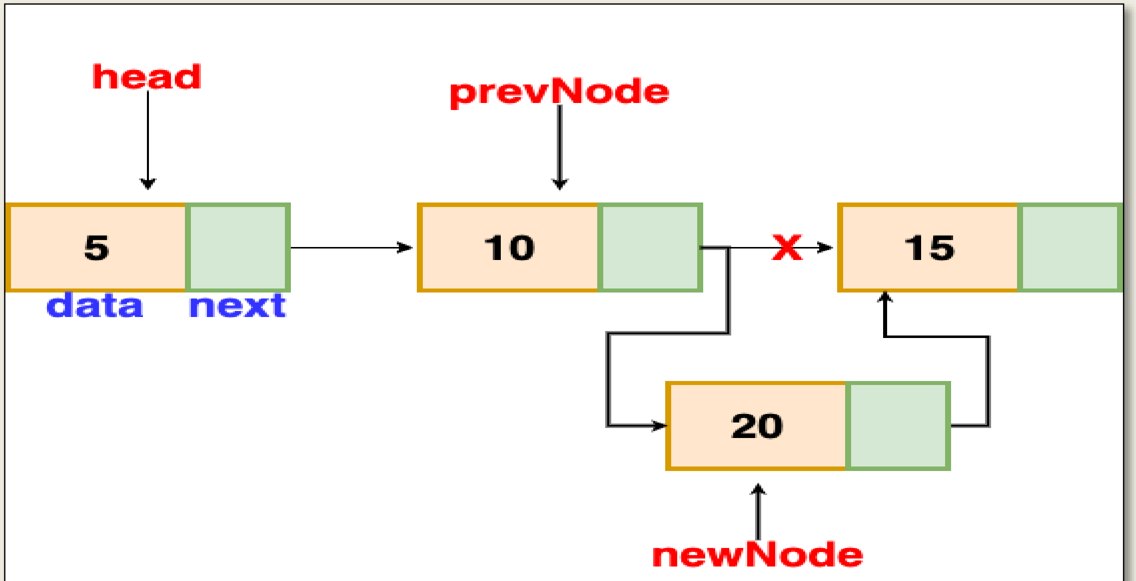
Insert New Node After Given Node in Linked List:-

- In order to insert a new node after given node in the linked list we have to follows the below steps:
- First we have to create a new node.
- After creating new node we have to check weather linked list is empty or not. We have two possibilities:
- (A) Linked List is empty (FIRST=None). Hence list is empty, specified node is not found in the linked list. In this case we cannot insert new node after given node.
- (B) Linked List is not empty (FIRST ≠ None). In this case we have to traverse from first node to last node in the list until given node is found. If node is found in the list then we can insert new node after that node otherwise we cannot insert new node after given node.

Function: INSAFG(X, First) Cont…

```
Step 1: NEW_NODE<-Node(Value)
         f<-0
         self.PREV<-none
         self.SAVE<-None
Step 2: If self.FIRST is None then
               print("Linked list is empty")
               return
Step 3: self.SAVE<-self.FIRST
Step 4: Repeat while self.SAVE.info is not None
                  if self.SAVE.info=X then
                      f=1
                      break
                  else
                      self.PREV<- self.SAVE
                      self.SAVE <-self.SAVE.LINK
Step 5: If f=1 then
           NEW_NODE.LINK<-self.SAVE.LINK
           self.SAVE.LINK<-NEW_NODE
        Else
          print("Specified Node Not Found")
Step 6: Exit
```

Function: INSAFG(X, First) Cont…



Insertion after a given node

```
SAVE = FIRST
 Repeat while X ≠ SAVE->INFO and SAVE->LINK ≠ NULL
 PRED = SAVE
 SAVE = SAVE->LINK
 If X = SAVE->INFO then
 NEW_NODE->LINK= SAVE->LINK
 SAVE->LINK=NEW_NODE
```

Function: INSAFG(X, First) Cont…

```
def InsertAfter(self,New_Node,X):
    f=0
    self.PREV=None
    self.SAVE=None
    if(self.FIRST is None):
        print("Linked list is empty")
    else:
        self.SAVE=self.FIRST
        while(self.SAVE is not None):
            if(self.SAVE.info==X):
                f=1
                break
            else:
                self.PREV=self.SAVE
                self.SAVE=self.SAVE.link
        if(f==1):
            New_Node.link=self.SAVE.link
            self.SAVE.link=New_Node
        else:
            print("Node not found")
```

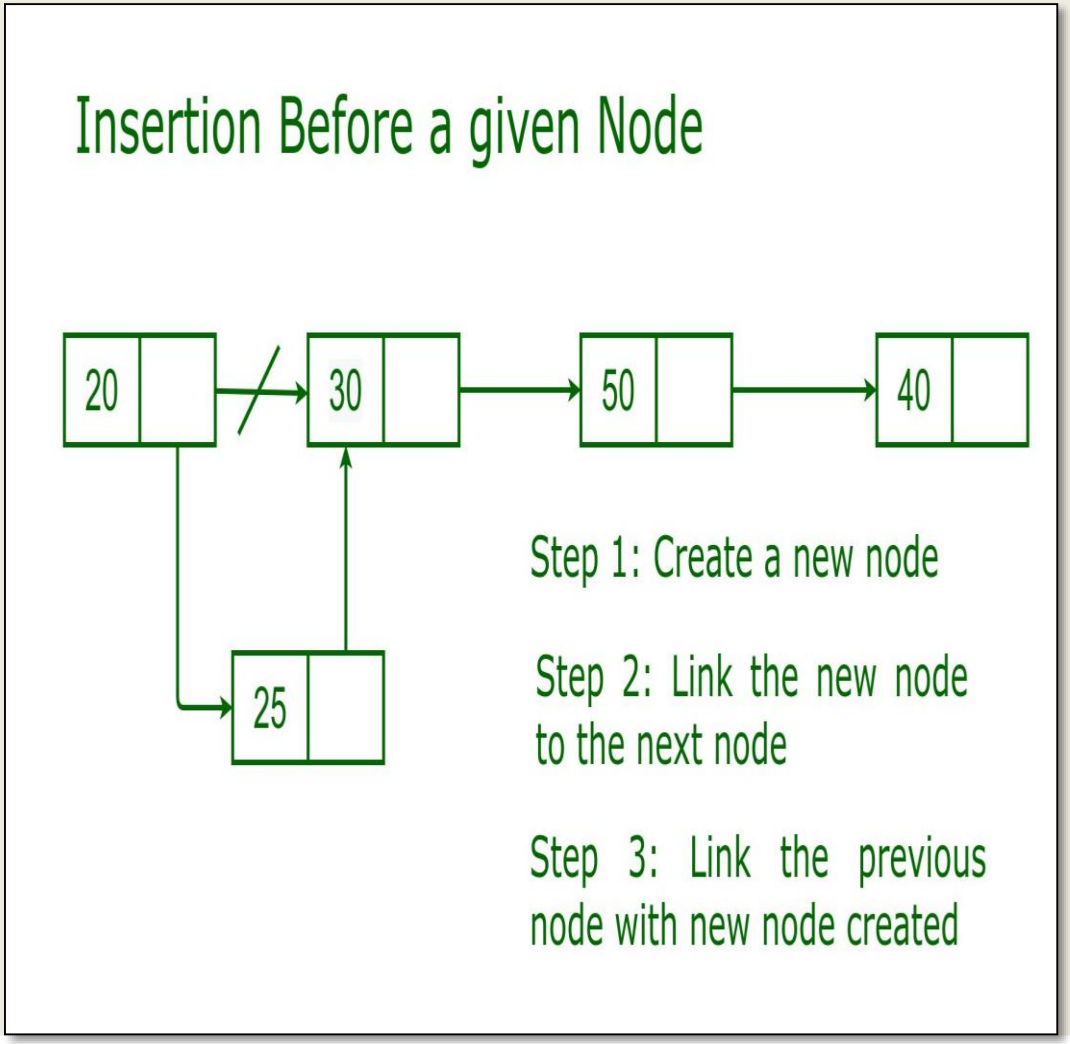
Insert New Node Before Given Node in Linked List:-

- In order to insert a new node before a given node in the linked list we have to follows the below steps:
- (1) First we have to create a new node.
- (2) After creating new node we have to check weather linked list is empty or not. We have two possibilities:
- (A) Linked List is empty (FIRST=None). Hence list is empty, specified node is not found in the linked list. In this case we can not insert new node before given node.
- (B) Linked List is not empty (FIRST ≠ None). In this case we have to traverse from first node to last node in the list until given node is found. If node is found in the linked list then we can insert new node before that node otherwise we cannot insert new node before given node.

Function: INSBFG(X, First) Cont…

```
Step 1: NEW_NODE<-Node(Value)
         f<-0, self.PREV<-none, self.SAVE<-None
Step 2: If self.FIRST is None then
              print("Linked list is empty")
              return
Step 3: self.SAVE<-self.FIRST
Step 4: Repeat while self.SAVE.info is not None
                 if self.SAVE.info=X then
                     f=1
                     break
                 else
                     self.PREV<- self.SAVE
                     self.SAVE <-self.SAVE.LINK
Step 5: If f=1 then
          if(self.PREV is None)
              NEW_NODE.LINK<-self.SAVE
              self.FIRST<-NEW_NODE
          else
              self.PREV.LINK<-NEW_NODE
              NEW_NODE.link=self.SAVE
       else
       Print("Node not found")
```

Function: INSBFG(X, First) Cont...



Function: INSBFG(X, First) Cont...

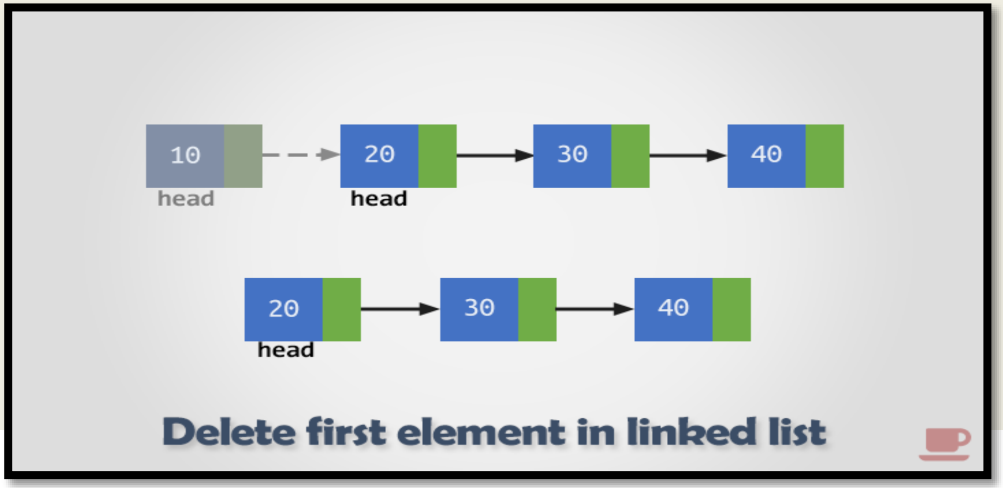
```
def InsertBefore(self,New_Node,X):
    f=0
    self.PREV=None
    self.SAVE=None
    if(self.FIRST is None):
        print("Linked list is empty")
    else:
        self.SAVE=self.FIRST
        while(self.SAVE is not None):
            if(self.SAVE.info==X):
                f=1
                break
            else:
                self.PREV=self.SAVE
                self.SAVE=self.SAVE.link
        if(f==1):
            if(self.PREV is None):
                New_Node.link=self.SAVE
                self.FIRST=New_Node
            else:
                self.PREV.link=New_Node
                New_Node.link=self.SAVE
        else:
            print("Node not found")
```

Delete First Node from Singly Linked List:-

- In order to delete first node from linked list we have to consider three possibilities:
- List is Empty (FIRST = None). In this case we can not delete node from linked list.
- There is only one node in the linked list (FIRST.LINK=None). In this case we can delete the first node and then linked list becomes empty (FIRST=None).
- There are more than one node in the linked list. In this case we can delete the first node. After deleting the first node we have to move FIRST pointer to next node so that it can points to the newly first node in the linked list.

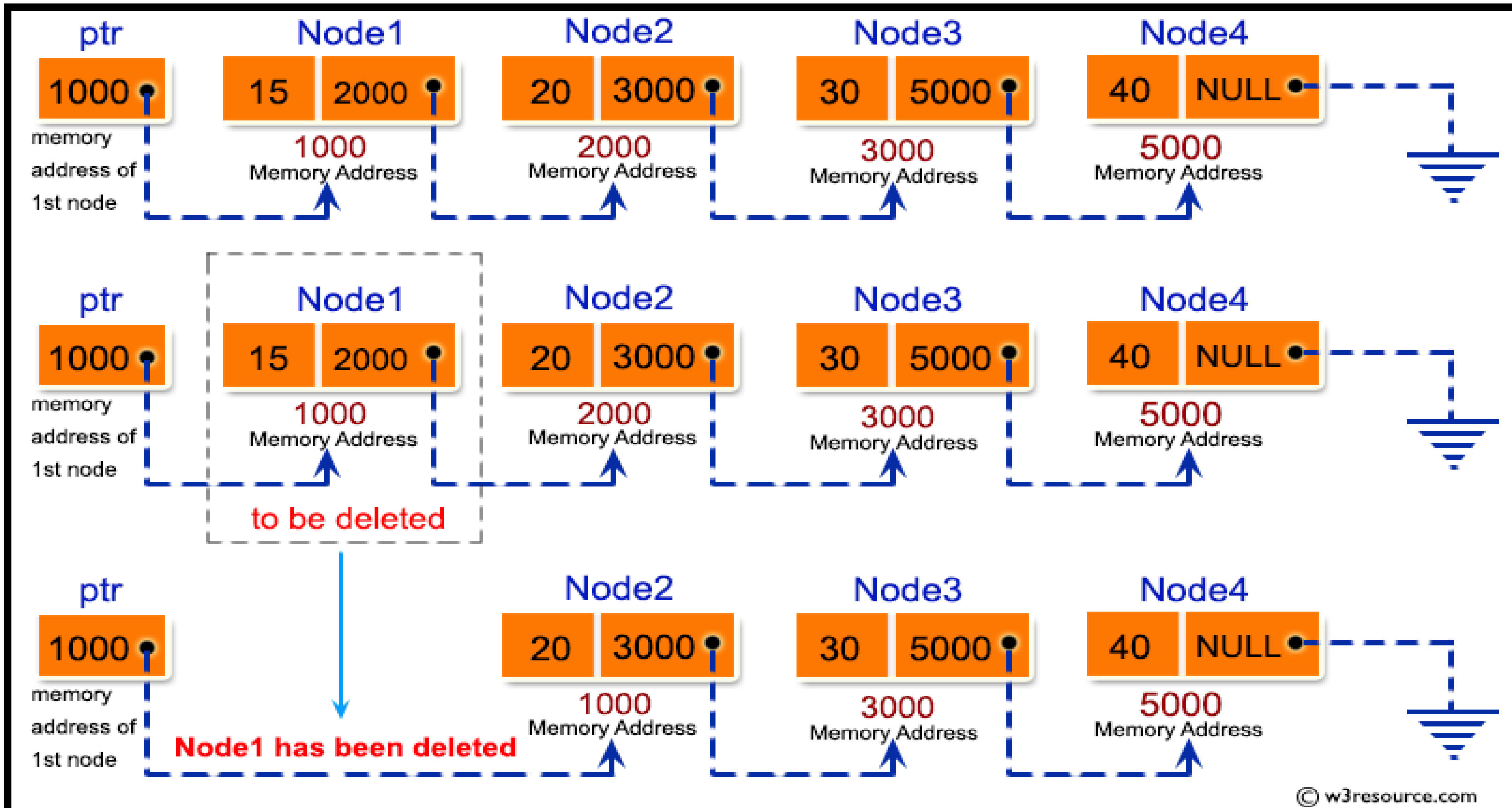
Function: DELETE(X, First)

```
Step 1: If self.FIRST is None then
           Write "Linked List is Empty"
          return
Step 2: If self.FIRST.LINK is None then
           return(self.FIRST.info)
           self.FIRST=None
           Else
           return(self.FIRST.Info)
           self.FIRST=self.FIRST.LINK
Step 3: Exit
```



Procedure: DELETE(X, FIRST)

- This algorithm **delete** a node whose address is given by variable **X**.
- **FIRST** is a **pointer to the first element** of a Singly linked linear list.
- Typical node contains **INFO** and **LINK** fields.



Procedure: DELETE(X, FIRST)

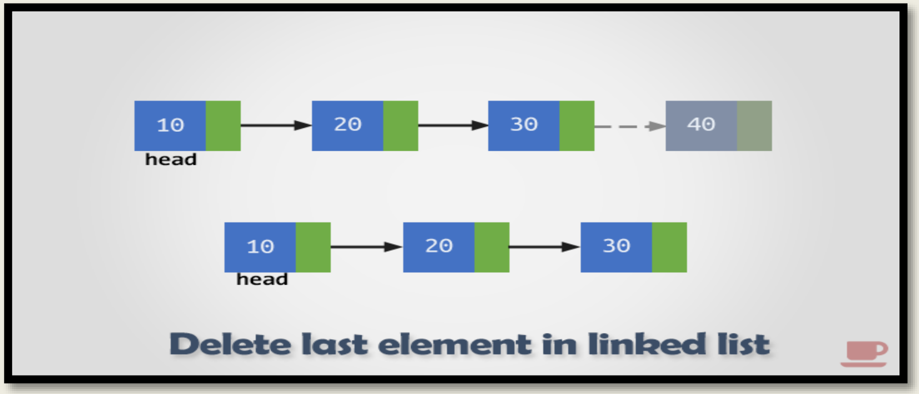
```
def DeleteFirst(self):
    if(self.FIRST is None):
        print("Linked list is empty")
    elif(self.FIRST.link is None):
        print("Deleted element is {}".format(self.FIRST.info))
        self.FIRST=None
    else:
        print("Deleted element is {}".format(self.FIRST.info))
        self.FIRST=self.FIRST.link
```

Delete Last Node from Singly Linked List:-

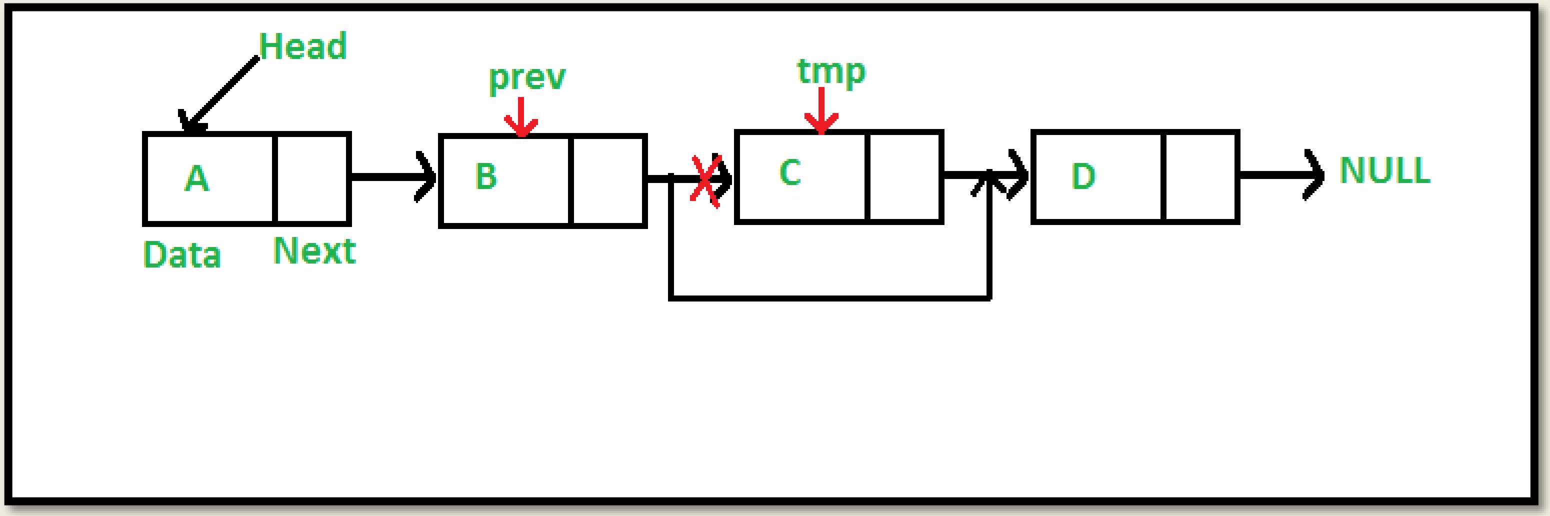
- In order to delete first node from linked list we have to consider three possibilities:
- List is Empty (FIRST = None). In this case we can not delete node from linked list.
- There is only one node in the linked list (FIRST.LINK=None). In this case we can delete the node and then linked list becomes empty (FIRST=None).
- There is more than one node in the linked list. In this case we have to traverse from first node to last node and then delete the last node. After deleting the last node we have to set NULL value in the LINK part of the previous node.

Function: DELETE(X, First)

```
Step 1: If self.FIRST is None then
            Write "Linked List is Empty"
            return
Step 2: If self.FIRST.LINK is None then
            return(self.FIRST.info)
            self.FIRST=None
        Else
            self.SAVE=self.FIRST
            Repeat while self.SAVE.LINK ≠ None
                self.PREV=self.SAVE
                self.SAVE=self.SAVE.LINK
            Return self.SAVE.INFO
            self.PREV.LINK=None
Step 3: Exit
```



Function: DELETE(X, First)




```
self.SAVE=self.FIRST
    Repeat while self.SAVE.LINK ≠ None
        self.PREV=self.SAVE
        self.SAVE=self.SAVE.LINK
    Return self.SAVE.INFO
    self.PREV.LINK=None
```

Function: DELETE(X, First)

```
def DeleteLast(self):
    if(self.FIRST is None):
        print("Linked list is empty")
    elif(self.FIRST.link is None):
        print("Deleted element is {}".format(self.FIRST.info))
        self.FIRST=None
    else:
        self.SAVE=self.FIRST
        while(self.SAVE.link is not None):
            self.PREV=self.SAVE
            self.SAVE=self.SAVE.link
        print("Deleted element is {}".format(self.SAVE.info))
        self.PREV.link=None
```

Function: SEARCH_NODES(FIRST)

- In order to search particular node in a linked list we have to traverse from first node to last node in a linked list and compare the search value against each node in a linked list. Whenever a node is found we set the flag to indicate successful search.
- **FIRST** is a **pointer to the first element** of a Singly linked linear list.
- Typical node contains **INFO** and **LINK** fields.
- **SAVE** is a Temporary pointer variable.

Search Node in Linked List:-

```
Step 1: FLAG = 0
        self.SAVE=self.FIRST
Step 2: Repeat step 3 while self.SAVE ≠ None
Step 3: If self.SAVE.INFO = X then
           FLAG = 1
           break
        Else
           self.SAVE=self.SAVE.LINK
Step 4: If FLAG = 1 then
           Write “Search is Successful”
           Else
           Write “Search is not successful”
Step 5: Exit
```

Search Node in Linked List:-

```
def SearchNode(self,X):
    f=0
    self.SAVE=self.FIRST
    while(self.SAVE is not None):
        if(self.SAVE.info==X):
            f=1
            break
        else:
            self.SAVE=self.SAVE.link
    if(f==1):
        print("Node found")
    else:
        print("Node not found")
```

Function: COUNT_NODES(FIRST)

- This function **counts** number of nodes of the linked list and returns **COUNT**.
- **FIRST** is a **pointer to the first element** of a Singly linked linear list.
- Typical node contains **INFO** and **LINK** fields.
- **SAVE** is a Temporary pointer variable.

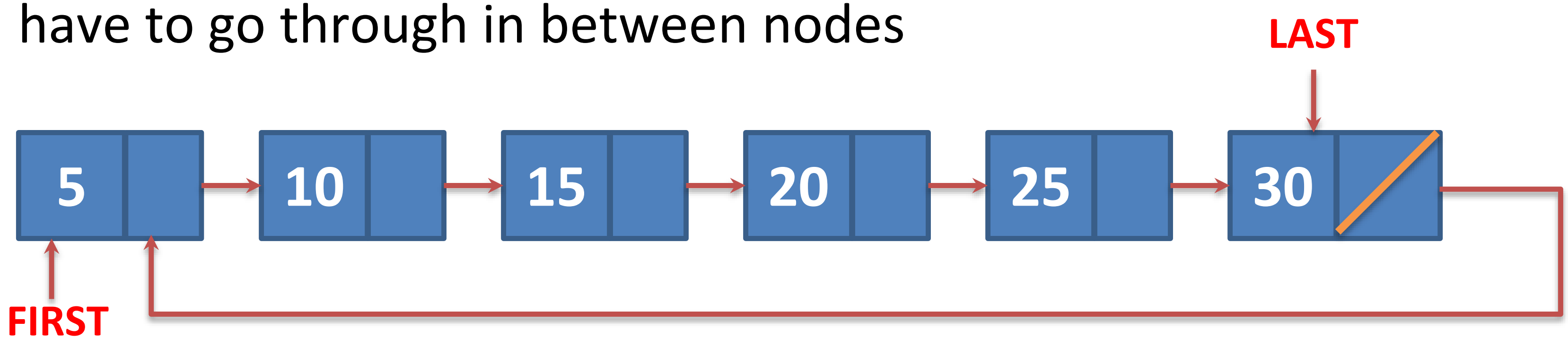
Function: COUNT_NODES(FIRST) Cont…

```
Step 1: Count = 0
        self.SAVE = self.FIRST
Step 2: Repeat step 3 while self.SAVE ≠ None
Step 3: Count= Count + 1
        self.SAVE=self.SAVE.LINK
Step 4: Return Count
```

```python
def CountNode(self):
    count=0
    self.SAVE=self.FIRST
    while(self.SAVE is not None):
        count=count+1
        self.SAVE=self.SAVE.link
    print("Number of node:",count)
```

Circularly Linked Linear List

- If we **replace NULL** pointer of the **last node** of Singly Linked Linear List with the **address of its first node**, that list becomes circularly linked linear list or **Circular List**.
- **FIRST** is the address of first node of Circular List
- **LAST** is the address of the last node of Circular List
- **Advantages of Circular List**
  - In circular list, every node is accessible from given node
  - It saves time when we have to go to the first node from the last node. It can be done in single step because there is no need to traverse the in between nodes. But in double linked list, we will have to go through in between nodes



Circularly Linked Linear List

- **Disadvantages of Circular List**
  - It is not easy to reverse the linked list
  - If proper care is not taken, then the problem of infinite loop can occur
  - If we at a node and go back to the previous node, then we can not do it in single step. Instead we have to complete the entire circle by going through the in between nodes and then we will reach the required node

Singly Linked List Vs Circular Linked List

Singly Linked List	Circular Linked List
Only one pointer is maintained in a node of singly list which contains the address of next node in sequence another will keep the address of.	Two pointers are maintained in a node of circular list, one will keep the address of first previous node and first next node in sequence.
In singly, we can not move in backward direction because each in node has next node pointer which facilitates us to move in forward direction	In circular list, we can move backward as well as forward direction as each node keeps the address of previous and next node in sequence
During deletion of a node in between the singly list, we will have to keep two nodes address one the address of the node to be deleted and the node just previous of it.	During deletion, we have to keep only one node address i.e the node to be deleted. This node will give the address of previous node automatically.

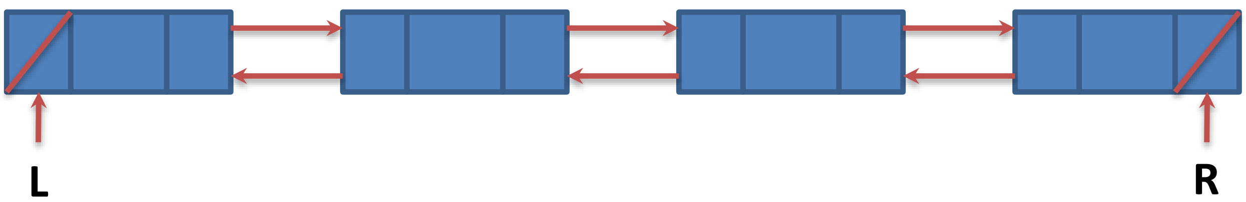
Doubly Linked Linear List

- In certain Applications, it is very desirable that a list be traversed in either forward or reverse direction.
- This property implies that each node must contain two link fields instead of usual one.
- The links are used to denote **Predecessor** and **Successor** of node.
- The link denoting its **predecessor** is called **Left Link.**
- The link denoting its **successor** is called **Right Link.**
- A list containing this type of node is called **doubly linked list** or **two way chain**.

Doubly Linked Linear List

- Typical node of doubly linked linear list contains INFO, LPTR RPTR Fields
- **LPTR** is pointer variable pointing to Predecessor of a node
- **RPTR** is pointer variable pointing to Successor of a node
- Left most node of doubly linked linear list is called **L**, **LPTR** of node L **is always NULL**
- Right most node of doubly linked linear list is called **R**, **RPTR** of node **R is always NULL**

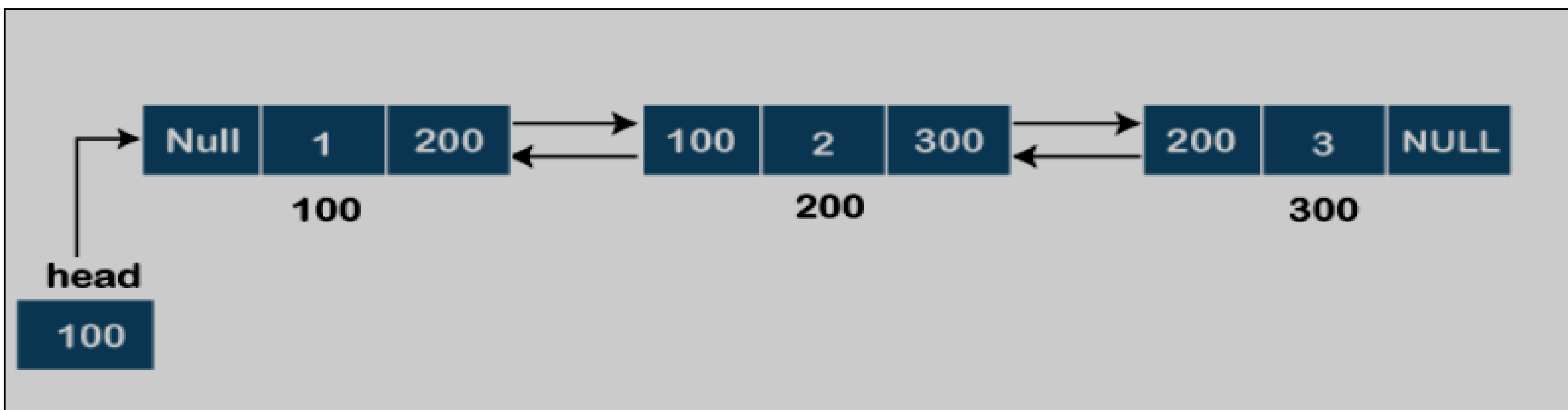
Typical node of Doubly Linked List



Doubly linked linear list

Algorithm of Doubly Linked List

- Insertion of a new node in the beginning of the list,
- Insertion of a new node at the end of the list,
- Insertion of a new node after a given node,
- Insertion of a new node before a given node,
- Insertion of a new node in sorted linked list,
- Deleting the first and last node from a linked list,
- Searching a node in Linked List,
- Count the number of nodes in linked list



SLL V/S DLL

Basis of comparison	Singly linked list	Doubly linked list
Definition	A single linked list is a list of nodes in which node has two parts, the first part is the data part, and the next part is the pointer pointing to the next node in the sequence of nodes.	A doubly linked list is also a collection of nodes in which node has three fields, the first field is the pointer containing the address of the previous node, the second is the data field, and the third is the pointer containing the address of the next node.
Access	The singly linked list can be traversed only in the forward direction.	The doubly linked list can be accessed in both directions.
List pointer	It requires only one list pointer variable, i.e., the head pointer pointing to the first node.	It requires two list pointer variables, **head** and **last**. The head pointer points to the first node, and the last pointer points to the last node of the list.
Memory space	It utilizes less memory space.	It utilizes more memory space.
Efficiency	It is less efficient as compared to a doubly-linked list.	It is more efficient.
Implementation	It can be implemented on the stack.	It can be implemented on stack, heap and binary tree.
Complexity	In a singly linked list, the time complexity for inserting and deleting an element from the list is **O(n)**.	In a doubly-linked list, the time complexity for inserting and deleting an element is **O(1)**.

Application of Linked List

- **Dynamic Memory Allocation:** As we know, we can dynamically allocate memory in a linked list, so it can be very helpful when we don't know the number of elements we are going to use.
- **Implementing advanced data structures:** We can implement data structures like stacks and queues with the help of a linked list.
- **Manipulating polynomials:** We can do polynomials manipulation with the help of a linked list by storing the constants in the nodes of the linked list.
- **Arithmetic operations on long integers:** As integers have a limit, we cannot perform arithmetic operations on long integers. But, if we use a linked list to represent the long integers, we can easily perform the operations.
- **Graph Adjacency list Representation:** Linked List helps in storing the adjacent vertices of a graph in adjacency list representation.

Applications of Linked List in real world

- In web browsers, you might have seen that we can always access the previous and next URL using the back and forward button. Access to previous and next URL searched is possible because they are linked using a linked list.
- The songs in the Music Player are linked to the next and the previous song. We can play songs either from the starting or the end of the list.
- In an Image Viewer, the next and the previous images are linked; hence they can be accessed by the previous and the next button.