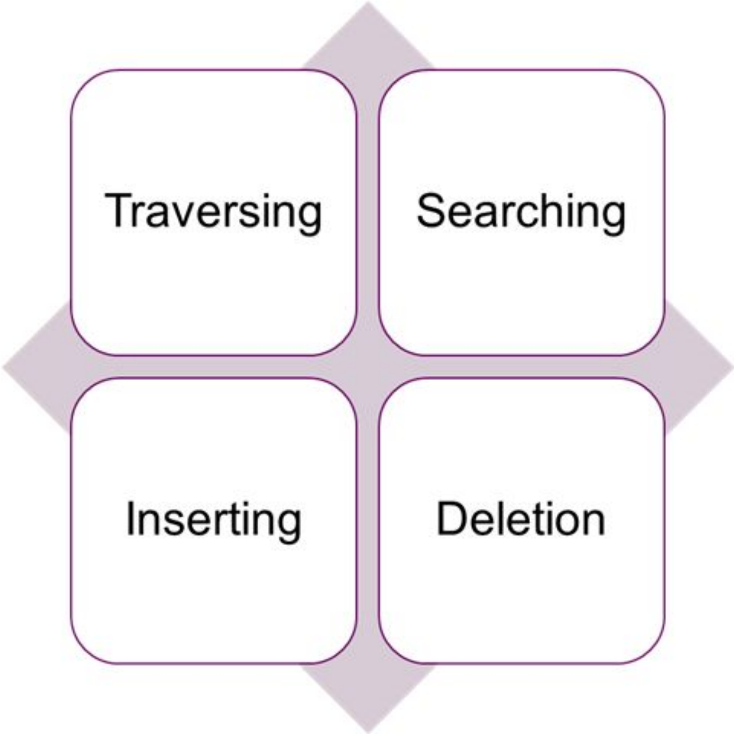
Lists & Linked Lists

Lists & Linked Lists

- List – A sequence of elements in certain linear order
- Example: Shopping list



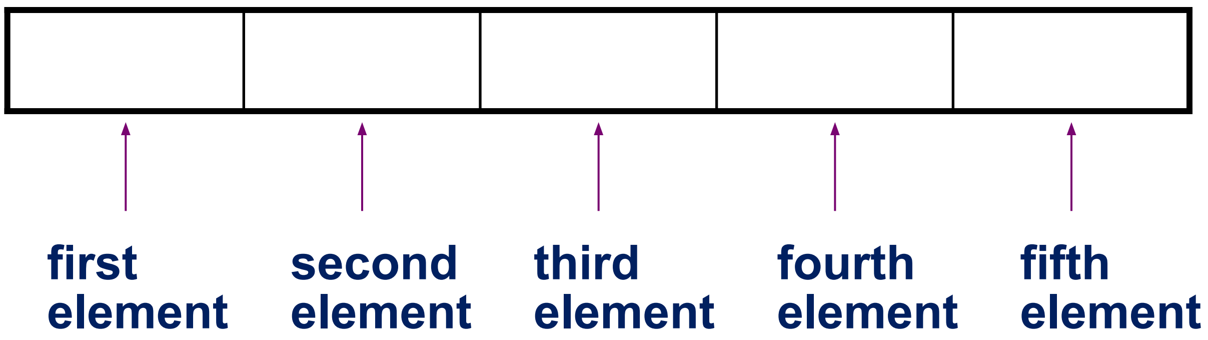
Arrays

Arrays (Storage in Memory)

- In the definition:

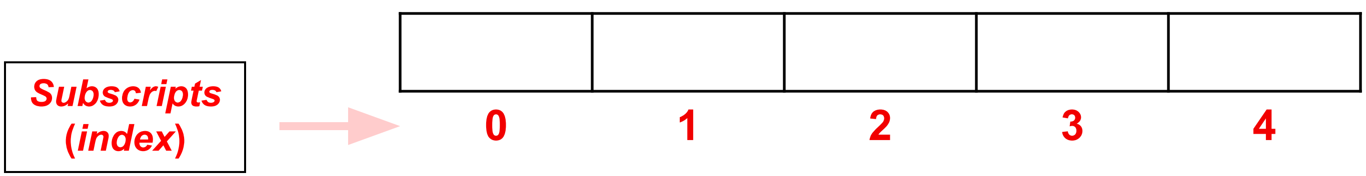
```
tests = new int[SIZE];    // SIZE is 5
```

allocates the following memory



Arrays (Accessing Array Elements)

- Each array element has a *subscript (index)*, used to access the element.
- Subscripts *(index)* start at 0



Arrays: The Pros & Cons

- **Pros:**
  - Fast element access
- **Cons:**
  - While many applications require resizing,
  - static arrays are impossible to resize
  - Required size is not always immediately available

Linked Lists

Introduction

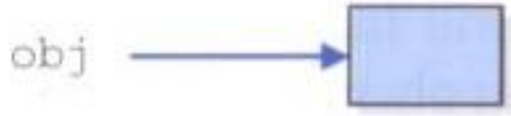
- Storing data items in arrays has at least two limitations
  - The array size is fixed once it is created: Changing the size of the array requires creating a new array and then copying all data from the old array to the new array
  - The data items in the array are next to each other in memory: Inserting an item inside the array requires shifting other items
- A linked structure is introduced to overcome limitations of arrays and allow easy insertion and deletion

Introduction (cont.)

- A linked structure is introduced to overcome limitations of arrays and allow easy insertion and deletion
  - A collection of nodes storing data items and links to other nodes
  - If each node has a data field and a reference field to another node called next or successor, the sequence of nodes is referred to as a singly linked list
  - Nodes can be located anywhere in the memory
  - The first node is called head and the last node is called tail

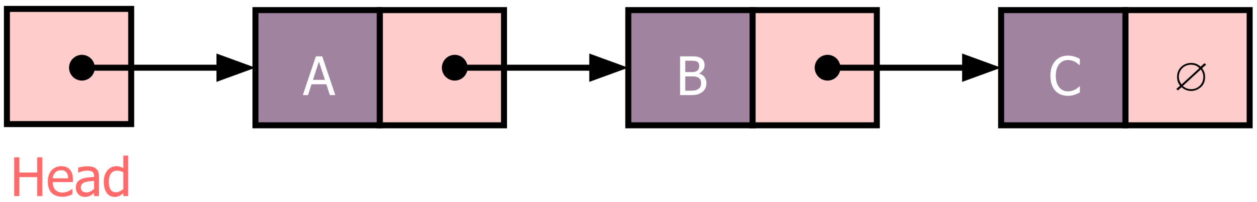
Linked Structures

- An alternative to array-based implementations are *linked structures*
- A linked structure uses object references to create links between objects
- Recall that an object reference variable holds the address of an object



Linked Lists

- D*ynamic* data structures that grow and shrink one element at a time, normally without some of the inefficiencies of arrays.
- consists of a collection of connected, dynamically allocated nodes.
- A series of connected *nodes*

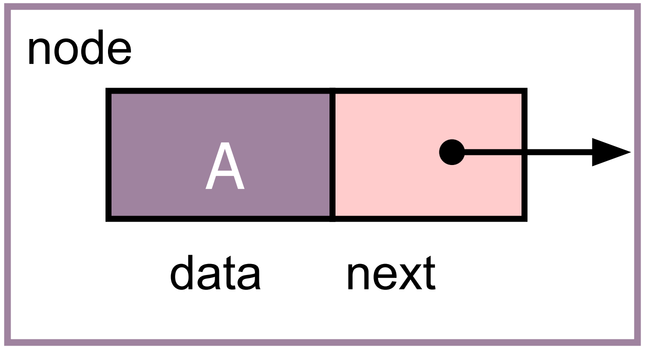


- Each node consists of the data element and a link to the next node in the list.
- The last node in the list has a null next link.

Linked Lists (cont…)

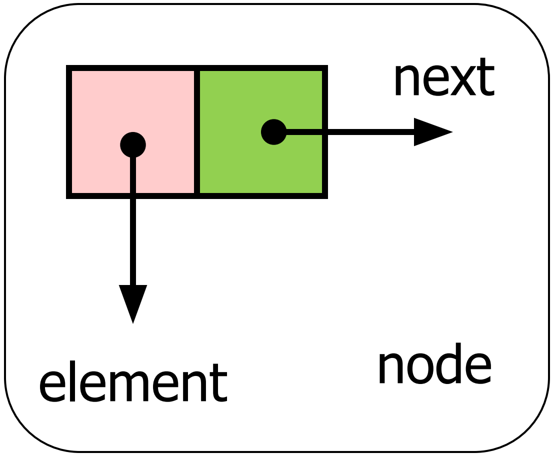
- Each node contains at least
  1. A piece of data (any type)
  2. Pointer to the next node in the list
- *head* : pointer to the first node
  - Sometimes called front, first
- The last node points to `NULL`

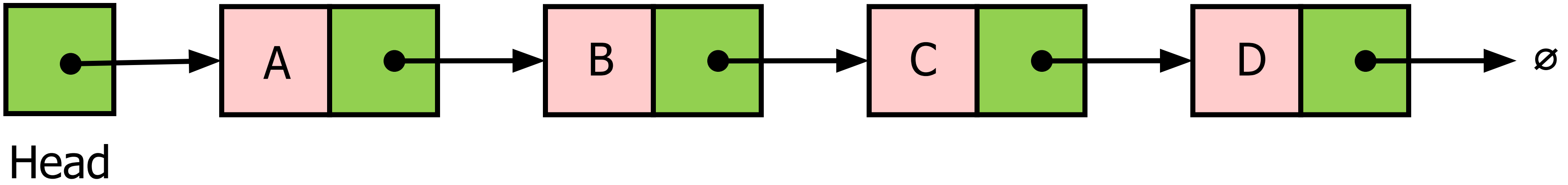
```
class Node:
    def __init__(self, data=None):
      self.data = data
      self.next = None
```

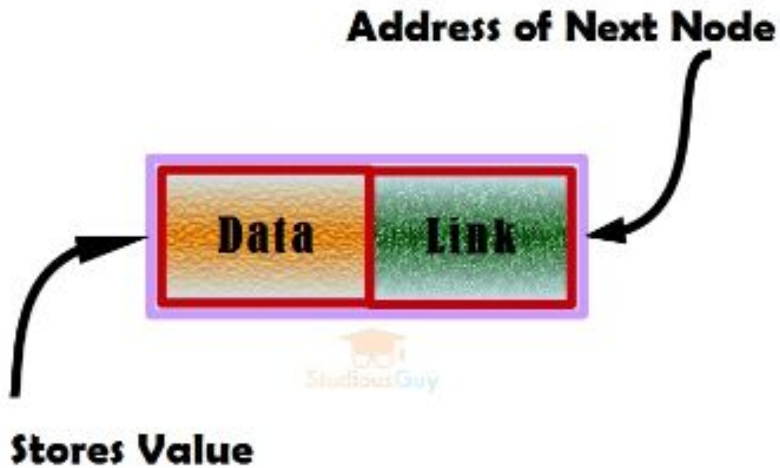


Singly Linked Lists

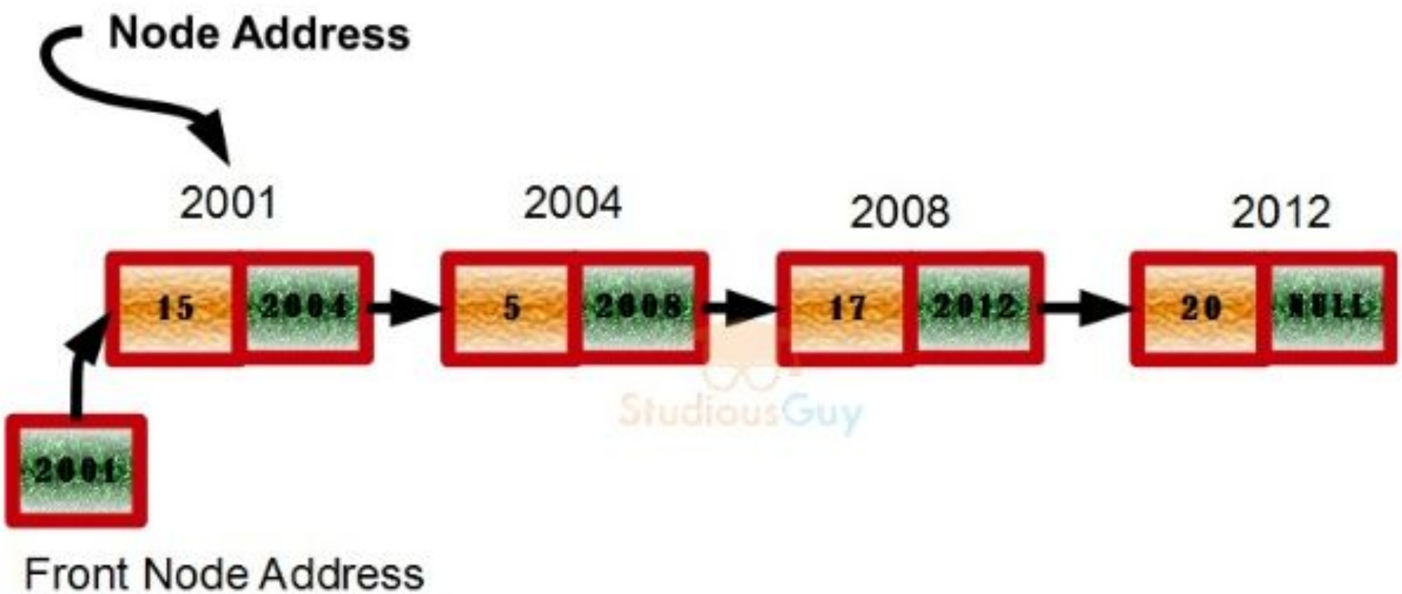
- a concrete data structure consisting of a sequence of nodes
- Each node stores
  - element
  - link to the next node

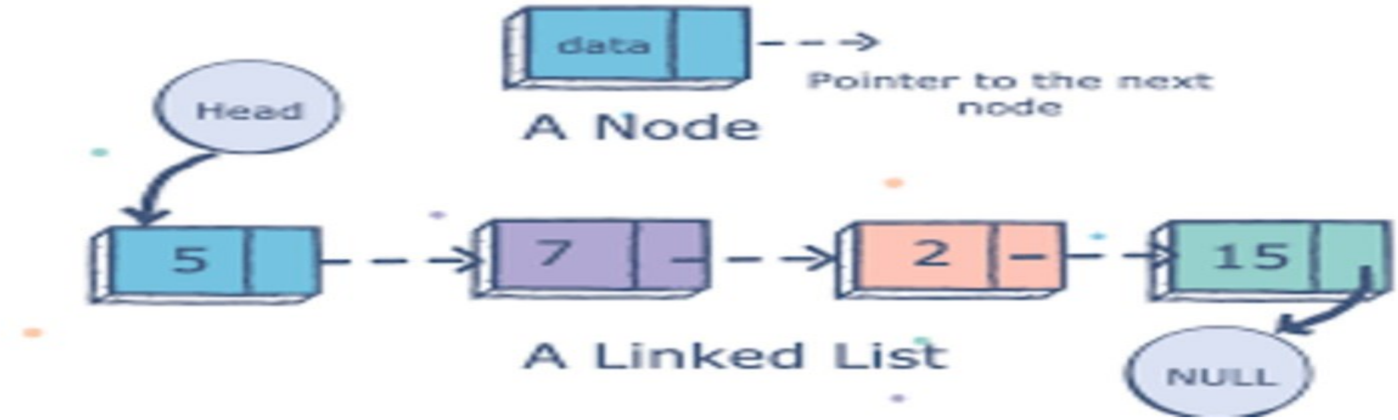






Let's see the example of a graphical depiction of a singly linked list:





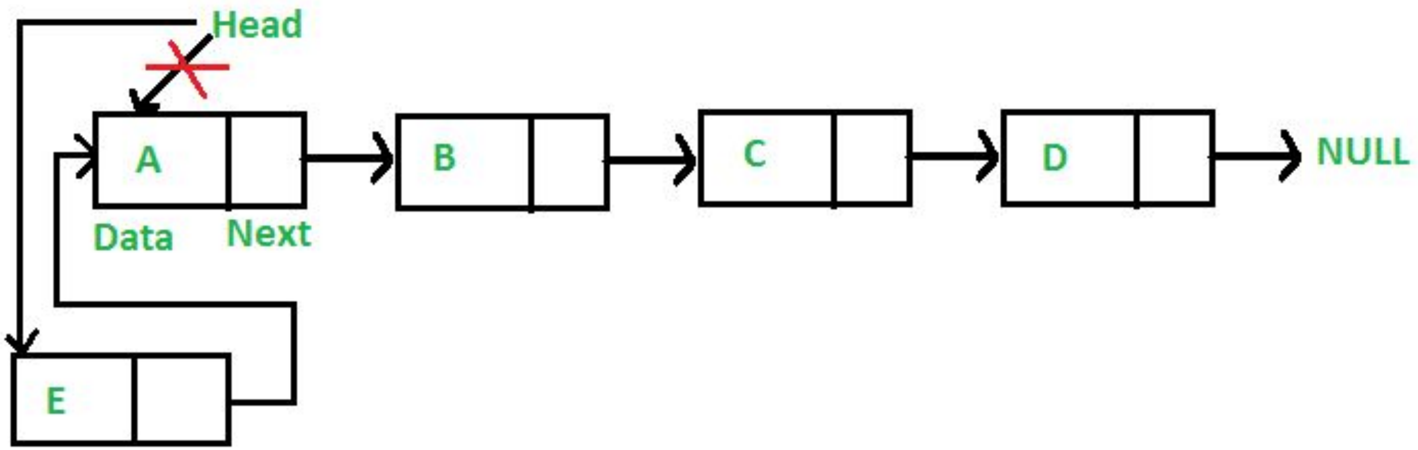
The first node is called the **head**; it points to the first node of the list and helps us access every other element in the list. The last node, points to *NULL* which represents end of the list .

Singly Linked Lists - Insertion

A node can be added in three ways

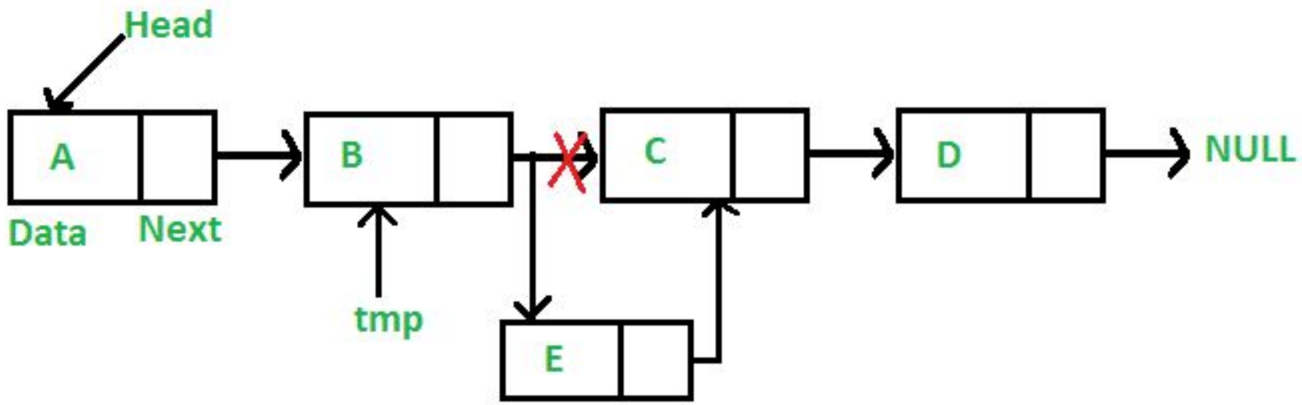
1) At the front of the linked list
2) After a given node.
3) At the end of the linked list.

Add a node at the front:



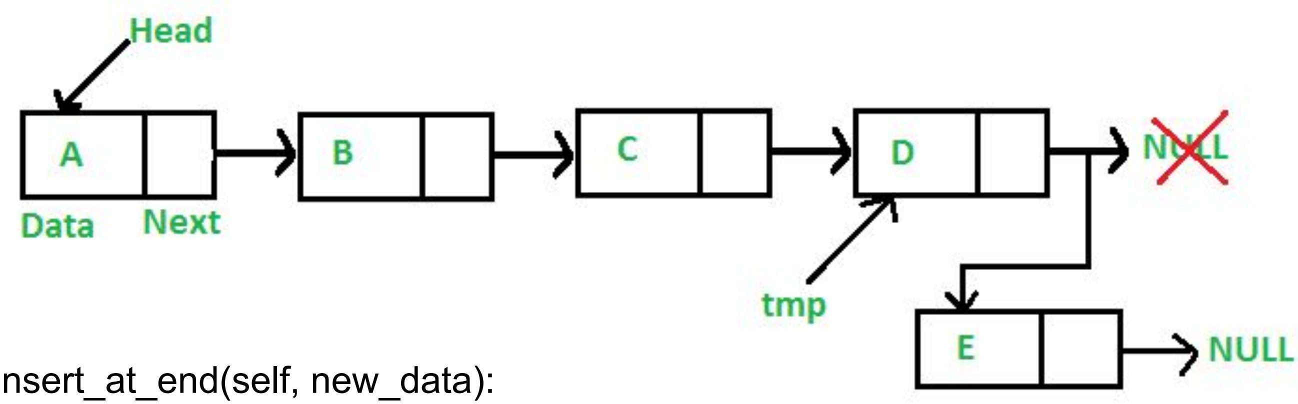

```
def insert_at_front(self, new_data):

    # 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
```

Add a node after a given node:




```
def insertAfter(self, prev_element, new_data):

    # 1. traverse till the prev_element
    temp = self.head
    while temp.data != prev_element :
        temp = temp.next

    #  2. Create new node &  3. Put in the data
    new_node = Node(new_data)

    # 4. Make next of new Node as next of temp
    new_node.next = temp.next

    # 5. make next of temp as new_node
    temp.next = new_node
```

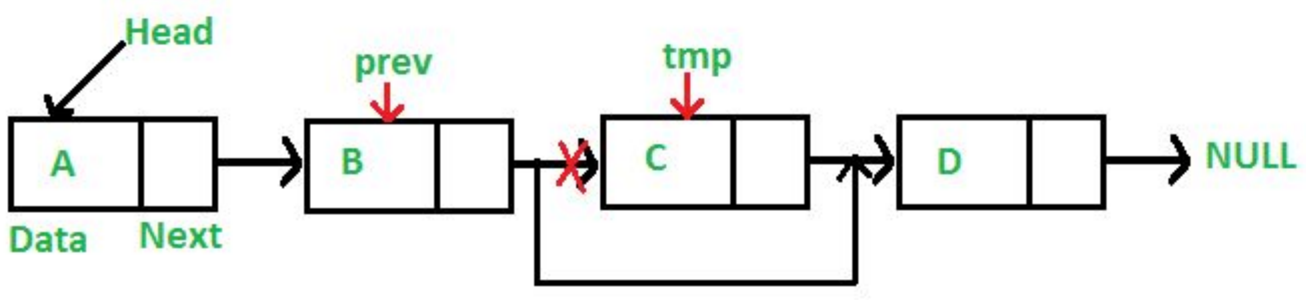
Add a node at the end:




```
def insert_at_end(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
```

Singly Linked List: Deletion

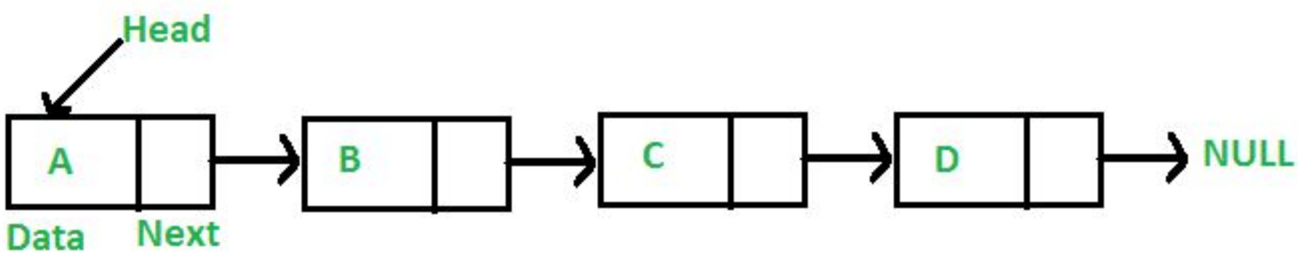
To delete a node from linked list, we need to do following steps.
1) Find previous node of the node to be deleted.
2) Change the next of previous node.
3) Free memory of the node to be deleted.



def deleteNode(self, key):

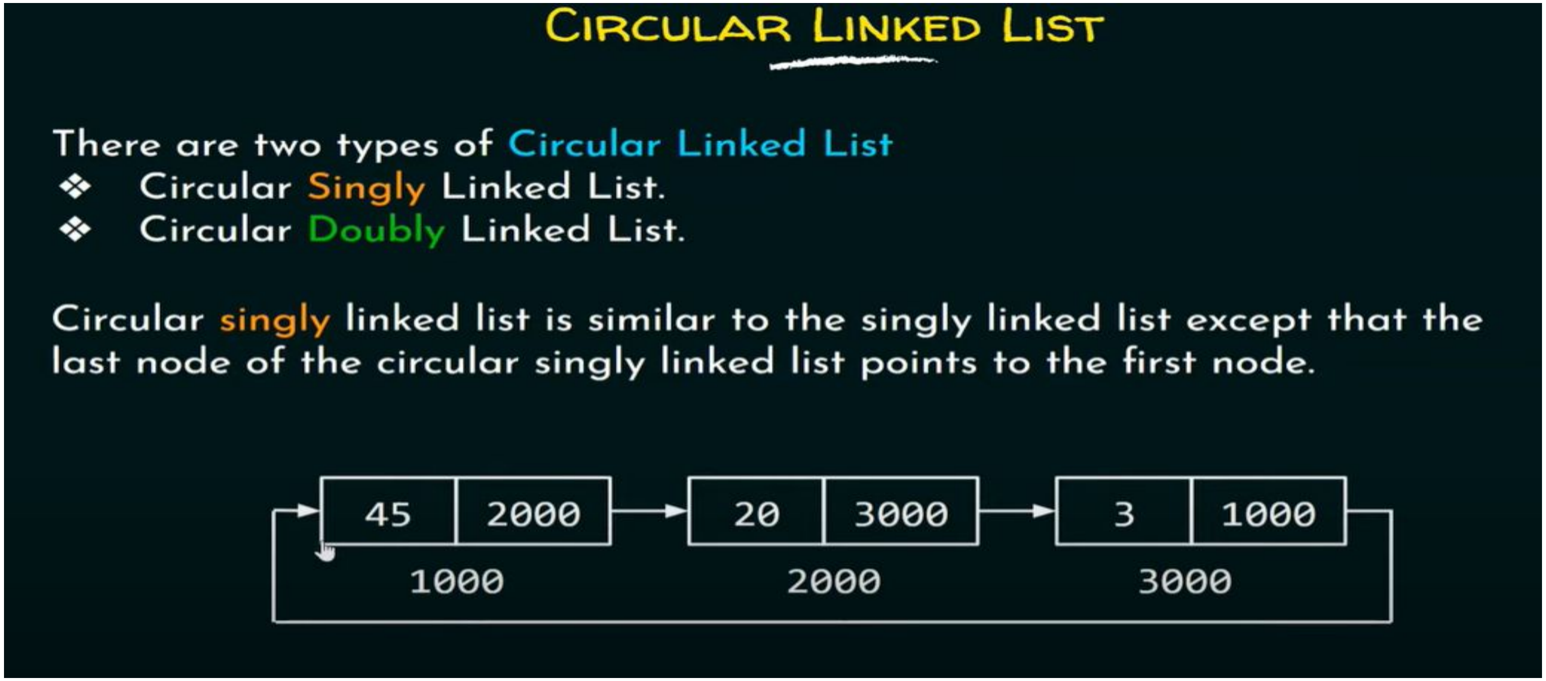
```
# If list is empty
if s.head == None:
  return
# If head node itself holds the key to be deleted
if s.head.data == key:
  s.head = s.head.next
  return
# Search the key to be deleted, keep track of the previous node also
t = s.head
while t != None and t.data != key:
    prev = t
    t = t.next
# if key was not present in linked list
if t == None:
  print("Key not found")
  return
# Unlink the node from linked list
prev.next = t.next
del t
```

Print elements of Linked List




```
# to print the linked LinkedList
def printList(self):
    temp = self.head
    # traverse till the end of the list
    while(temp is not None):
        print (temp.data)
        temp = temp.next
```

CIRCULAR LINKED LIST



Why Circular? In a singly linked list, for accessing any node of the linked list, we start traversing from the first node. If we are at any node in the middle of the list, then it is not possible to access nodes that precede the given node. This problem can be solved by slightly altering the structure of a singly linked list. In a singly linked list, the next part (pointer to next node) is NULL. If we utilize this link to point to the first node, then we can reach the preceding nodes. Refer to this for more advantages of circular linked lists.

Insertion

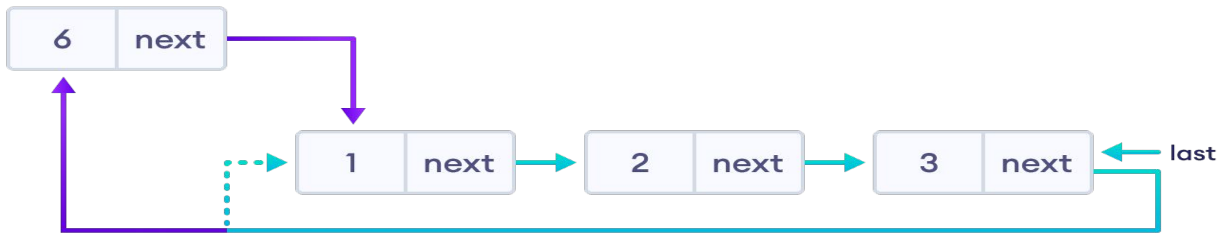
- **Insertion on a Circular Linked List**
- We can insert elements at 3 different positions of a circular linked list:

1. Insertion at the beginning
2. Insertion in-between nodes
3. Insertion at the end

1. Insertion at the Beginning

1)store the address of the current first node in the newNode (6- data) (i.e. pointing the newNode to the current first node)

2)point the last node to newNode (i.e making newNode as head)




```
# add node to the front
  def addFront(self, data):
    # check if the list is empty
    if self.last == None:
      return self.addToEmpty(data)
    # allocate memory to the new node and add data to the node
    newNode = Node(data)
    # store the address of the current first node in the newNode
    newNode.next = self.last.next
    # make newNode as last
    self.last.next = newNode
    return self.last
```

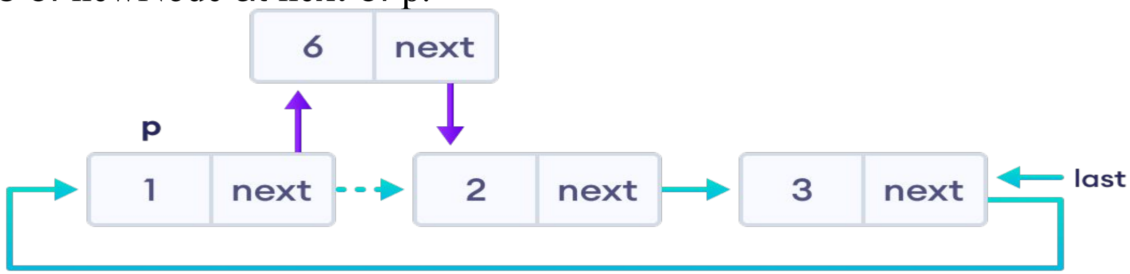
2.Insertion in between two nodes

Let's insert newNode after the first node.

1)travel to the node given (let this node be p).

2)point the next of newNode to the node next to p.

3)store the address of newNode at next of p.




```
# insert node after a specific node
 def addAfter(self, data, item):
   # check if the list is empty
   if self.last == None:
     return None
   newNode = Node(data)
   p = self.last.next
   while p:
     # if the item is found, place newNode after it
     if p.data == item:
       # make the next of the current node as the next of newNode
       newNode.next = p.next
       # put newNode to the next of p
       p.next = newNode
       if p == self.last:
         self.last = newNode
         return self.last
       else:
         return self.last
     p = p.next
     if p == self.last.next:
       print(item, "The given node is not present in the list")
       break
```

3. Insertion at the end

- store the address of the head node to next of newNode (making newNode the last node)
- point the current last node to newNode
- make newNode as the last node


```
# add node to the end
def addEnd(self, data):
    # check if the node is empty
    if self.last == None:
        return self.addToEmpty(data)
     # allocate memory to the new node and add data to the node
    newNode = Node(data)
     # store the address of the last node to next of newNode
    newNode.next = self.last.next
     # point the current last node to the newNode
    self.last.next = newNode
     # make newNode as the last node
    self.last = newNode
     return self.last
```

- **Deletion on a Circular Linked List**
- Suppose we have a linked list with elements 1, 2, and 3.

1. If the node to be deleted is the only node(only one node in list)

•free the memory occupied by the node
•store NULL in last

```
# delete a node
  def deleteNode(self, last, key):
      # If linked list is empty
     if last == None:
        return
      # If the list contains only a single node
     if (last).data == key and (last).next == last:
         last = None
```

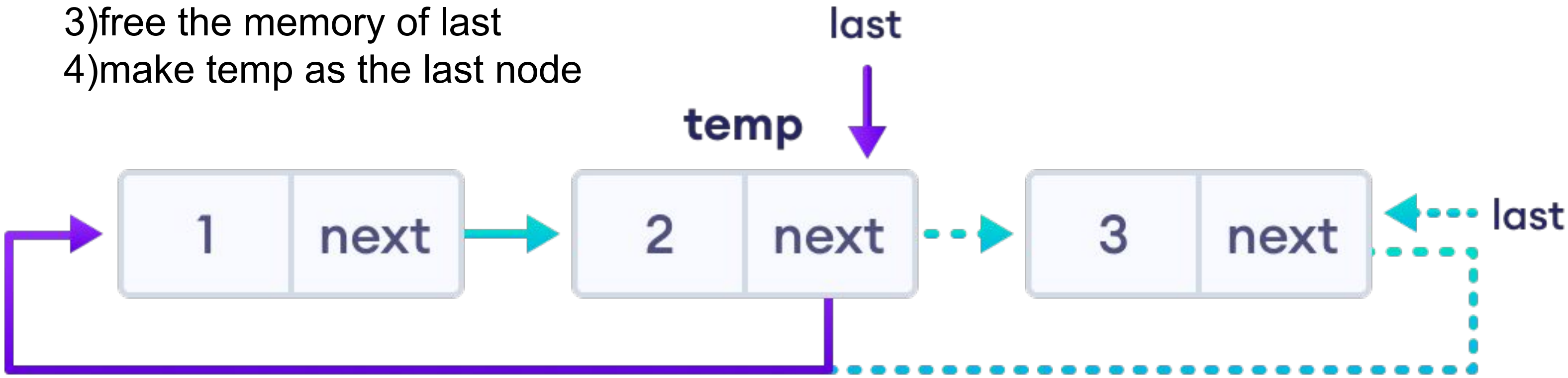
2. If last node is to be deleted

1)find the node before the last node (let it be temp)

2)store the address of the node next to the last node (1st node) in temp

3)free the memory of last

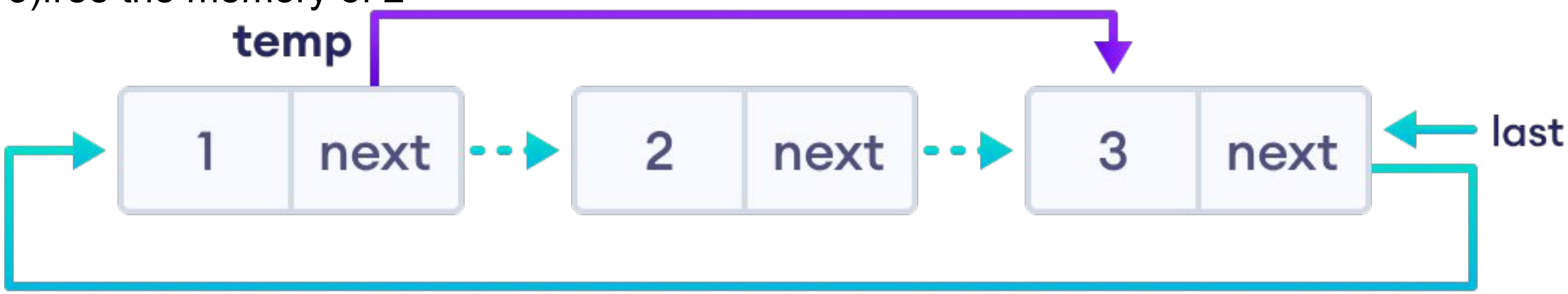
4)make temp as the last node




```
# if last node is to be deleted
    if (last).data == key:
        # find the node before the last node
        while temp.next != last:
            temp = temp.next
        # point temp node to the next of last i.e. first node
        temp.next = (last).next
        last = temp.next
```

3. If any other nodes are to be deleted(middle element)

1)travel to the node to be deleted (here we are deleting node 2)
2)let the node before node 2 be temp
3)store the address of the node next to 2 in temp
3)free the memory of 2




```
# travel to the node to be deleted
        while temp.next != last and temp.next.data != key:
            temp = temp.next
         # if node to be deleted was found
        if temp.next.data == key:
            d = temp.next
            temp.next = d.next
         return last
```

Traversal in Circular Linked List

```
def traverse(self):
    if self.last == None:
        print("The list is empty")
        return
    newNode = self.last.next
    while newNode:
        print(newNode.data, end=" ")
        newNode = newNode.next
        if newNode == self.last.next:
            break
```