CHAPTER

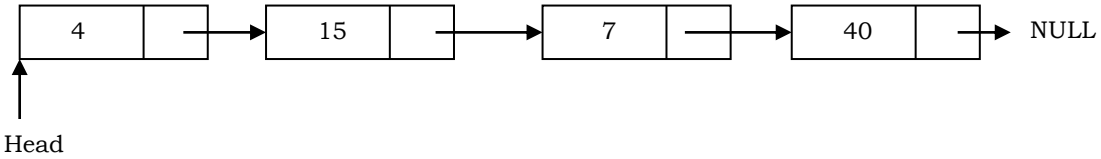
# 3

# Linked Lists

## 3.1 What is a Linked List?

A linked list is a data structure used for storing collections of data. A linked list has the following properties.

- Successive elements are connected by pointers
- The last element points to NULL
- Can grow or shrink in size during execution of a program
- Can be made just as long as required (until systems memory exhausts)
- Does not waste memory space (but takes some extra memory for pointers)




## 3.2 Linked Lists ADT

The following operations make linked lists an ADT:

**Main Linked Lists Operations**

- Insert: inserts an element into the list
- Delete: removes and returns the specified position element from the list

**Auxiliary Linked Lists Operations**

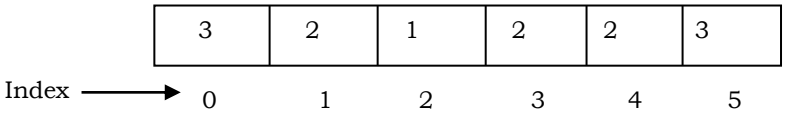
- Delete List: removes all elements of the list (dispose of the list)
- Count: returns the number of elements in the list
- Find $n^{th}$ node from the end of the list

## 3.3 Why Linked Lists?

There are many other data structures that do the same thing as linked lists. Before discussing linked lists it is important to understand the difference between linked lists and arrays. Both linked lists and arrays are used to store collections of data, and since both are used for the same purpose, we need to differentiate their usage. That means in which cases *arrays* are suitable and in which cases *linked lists* are suitable.

## 3.4 Arrays Overview

One memory block is allocated for the entire array to hold the elements of the array. The array elements can be accessed in constant time by using the index of the particular element as the subscript.




## Why Constant Time for Accessing Array Elements?

To access an array element, the address of an element is computed as an offset from the base address of the array and one multiplication is needed to compute what is supposed to be added to the base address to get the memory address of the element. First the size of an element of that data type is calculated and then it is multiplied with the index of the element to get the value to be added to the base address.

This process takes one multiplication and one addition. Since these two operations take constant time, we can say the array access can be performed in constant time.

## Advantages of Arrays

- Simple and easy to use
- Faster access to the elements (constant access)

## Disadvantages of Arrays

- **Fixed size:** The size of the array is static (specify the array size before using it).
- **One block allocation**: To allocate the array itself at the beginning, sometimes it may not be possible to get the memory for the complete array (if the array size is big).
- **Complex position-based insertion**: To insert an element at a given position, we may need to shift the existing elements. This will create a position for us to insert the new element at the desired position. If the position at which we want to add an element is at the beginning, then the shifting operation is more expensive.

## Dynamic Arrays

Dynamic array (also called *growable array*, *resizable array*, *dynamic table*, or *array list*) is a random access, variable-size list data structure that allows elements to be added or removed.

One simple way of implementing dynamic arrays is to initially start with some fixed size array. As soon as that array becomes full, create the new array double the size of the original array. Similarly, reduce the array size to half if the elements in the array are less than half.

**Note:** We will see the implementation for *dynamic arrays* in the *Stacks*, *Queues* and *Hashing* chapters.

## Advantages of Linked Lists

Linked lists have both advantages and disadvantages. The advantage of linked lists is that they can be *expanded* in constant time. To create an array, we must allocate memory for a certain number of elements. To add more elements to the array, we must create a new array and copy the old array into the new array. This can take a lot of time.

We can prevent this by allocating lots of space initially but then we might allocate more than we need and waste memory. With a linked list, we can start with space for just one allocated element and *add* on new elements easily without the need to do any copying and reallocating.

## Issues with Linked Lists (Disadvantages)

There are a number of issues with linked lists. The main disadvantage of linked lists is *access time* to individual elements. Array is random-access, which means it takes O(1) to access any element in the array. Linked lists take O($n$) for access to an element in the list in the worst case. Another advantage of arrays in access time is *spacial locality* in memory. Arrays are defined as contiguous blocks of memory, and so any array element will be physically near its neighbors. This greatly benefits from modern CPU caching methods.

Although the dynamic allocation of storage is a great advantage, the *overhead* with storing and retrieving data can make a big difference. Sometimes linked lists are *hard to manipulate*. If the last item is deleted, the last but one must then have its pointer changed to hold a NULL reference. This requires that the list is traversed to find the last but one link, and its pointer set to a NULL reference.

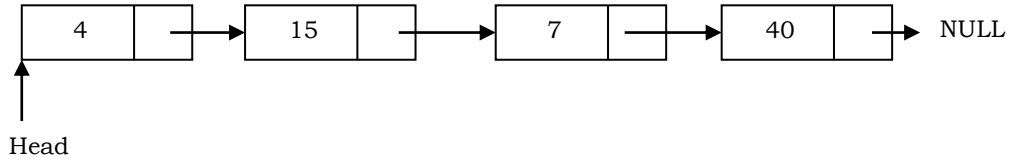
Finally, linked lists waste memory in terms of extra reference points.

# 3.5 Comparison of Linked Lists with Arrays and Dynamic Arrays

| Parameter | Linked list | Array | Dynamic array |
|---|---|---|---|
| Indexing | O(*n*) | O(1) | O(1) |
| Insertion/deletion at beginning | O(1) | O(*n*), if array is not full (for shifting the elements) | O(*n*) |
| Insertion at ending | O(*n*) | O(1), if array is not full | O(1), if array is not full<br>O(*n*), if array is full |
| Deletion at ending | O(*n*) | O(1) | O(*n*) |
| Insertion in middle | O(*n*) | O(*n*), if array is not full (for shifting the elements) | O(*n*) |
| Deletion in middle | O(*n*) | O(*n*), if array is not full (for shifting the elements) | O(*n*) |
| Wasted space | O(*n*) | 0 | O(*n*) |

# 3.6 Singly Linked Lists

Generally "linked list" means a singly linked list. This list consists of a number of nodes in which each node has a *next* pointer to the following element. The link of the last node in the list is NULL, which indicates the end of the list.




Following is a type declaration for a linked list of integers:

```
#Node of a Singly Linked List
class Node:
    #constructor
    def __init__(self):
        self.data = None
        self.next = None
     #method for setting the data field of the node
    def setData(self,data):
        self.data = data
    #method for getting the data field of the node
    def getData(self):
        return self.data
    #method for setting the next field of the node
    def setNext(self,next):
        self.next = next
    #method for getting the next field of the node
    def getNext(self):
        return self.next
    #returns true if the node points to another node
    def hasNext(self):
            return self.next != None
```

## Basic Operations on a List

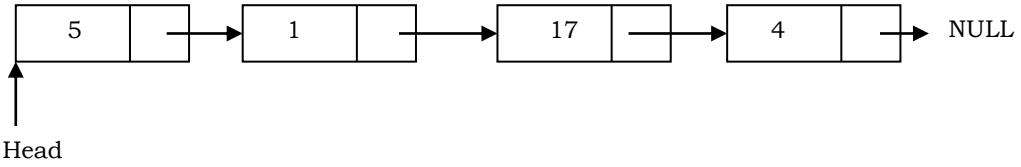
- Traversing the list
- Inserting an item in the list
- Deleting an item from the list

## Traversing the Linked List

Let us assume that the *head* points to the first node of the list. To traverse the list we do the following.

- Follow the pointers.

- Display the contents of the nodes (or count) as they are traversed.
- Stop when the next pointer points to NULL.




The ListLength() function takes a linked list as input and counts the number of nodes in the list. The function given below can be used for printing the list data with extra print function.

```
def listLength(self):
    current = self.head
    count = 0

    while current != None:
        count = count + 1
        current = current.getNext()

    return count
```

Time Complexity: O($n$), for scanning the list of size $n$.
Space Complexity: O(1), for creating a temporary variable.

## Singly Linked List Insertion

Insertion into a singly-linked list has three cases:

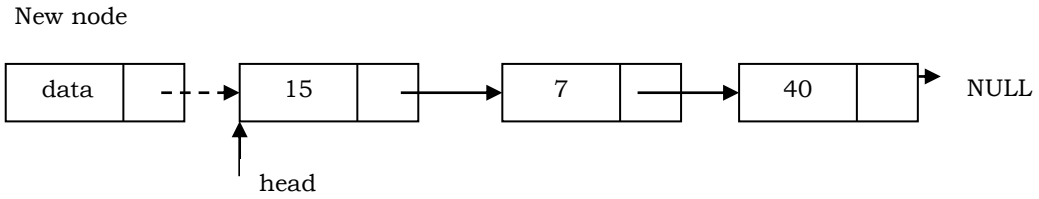
- Inserting a new node before the head (at the beginning)
- Inserting a new node after the tail (at the end of the list)
- Inserting a new node at the middle of the list (random location)

**Note:** To insert an element in the linked list at some position $p$, assume that after inserting the element the position of this new node is $p$.

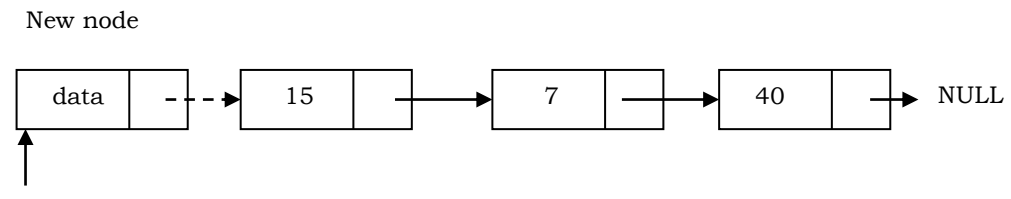
## Inserting a Node in Singly Linked List at the Beginning

In this case, a new node is inserted before the current head node. *Only one next pointer* needs to be modified (new node's next pointer) and it can be done in two steps:

- Update the next pointer of new node, to point to the current head.




- Update head pointer to point to the new node.



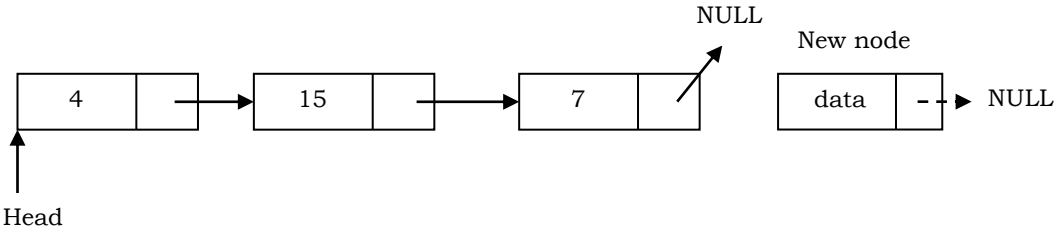

```
#method for inserting a new node at the beginning of the Linked List (at the head)
def insertAtBeginning(self,data):
    newNode = Node()
    newNode.setData(data)

    if self.length == 0:
```

```
            self.head = newNode
        else:
            newNode.setNext(self.head)
            self.head = newNode
        self.length += 1
```

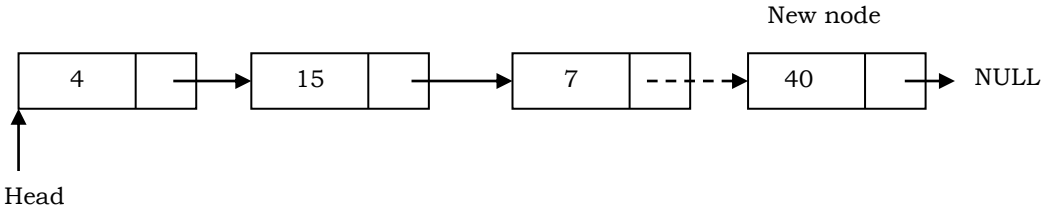
## Inserting a Node in Singly Linked List at the Ending

In this case, we need to modify *two next pointers* (last nodes next pointer and new nodes next pointer).

- New nodes next pointer points to NULL.




- Last nodes next pointer points to the new node.

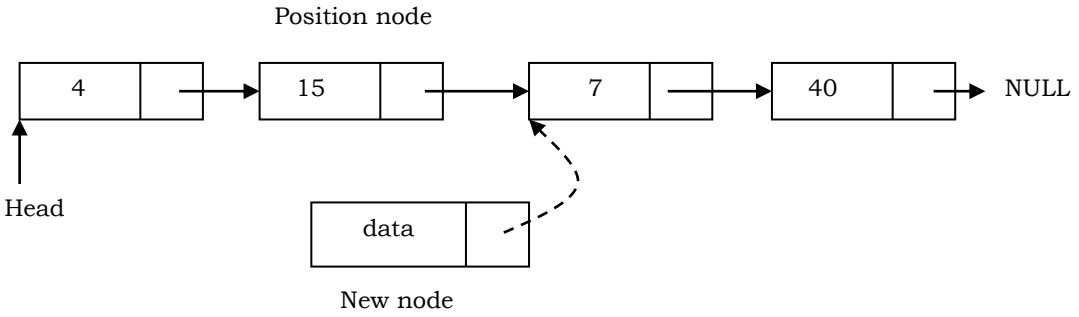



```
#method for inserting a new node at the end of a Linked List
def insertAtEnd(self,data):
    newNode = Node()
    newNode.setData(data)
    current = self.head
    while current.getNext() != None:
        current = current.getNext()
    current.setNext(newNode)
    self.length += 1
```

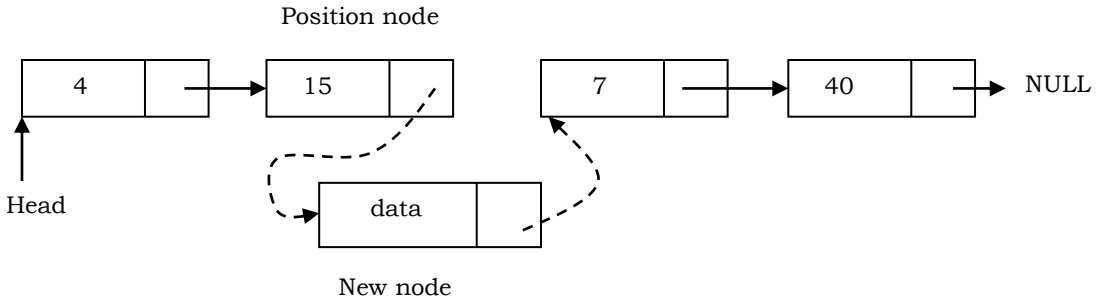
## Inserting a Node in Singly Linked List at the Middle

Let us assume that we are given a position where we want to insert the new node. In this case also, we need to modify two next pointers.

- If we want to add an element at position 3 then we stop at position 2. That means we traverse 2 nodes and insert the new node. For simplicity let us assume that the second node is called *position* node. The new node points to the next node of the position where we want to add this node.




- Position nodes next pointer now points to the new node.




Let us write the code for all three cases. We must update the first element pointer in the calling function, not just in the called function. For this reason we need to send a double pointer. The following code inserts a node in the singly linked list.

```
#Method for inserting a new node at any position in a Linked List
def insertAtPos(self,pos,data):
 if pos > self.length or pos < 0:
    return None
 else:
    if pos == 0:
           self.insertAtBeg(data)
    else:
           if pos == self.length:
              self.insertAtEnd(data)
           else:
              newNode = Node()
              newNode.setData(data)
              count = 0
              current = self.head
              while count < pos-1:
                     count += 1
                     current = current.getNext()
              newNode.setNext(current.getNext())
              current.setNext(newNode)
              self.length += 1
```

**Note:** We can implement the three variations of the *insert* operation separately.

Time Complexity: O(*n*), since, in the worst case, we may need to insert the node at the end of the list.
Space Complexity: O(1), for creating one temporary variable.

## Singly Linked List Deletion

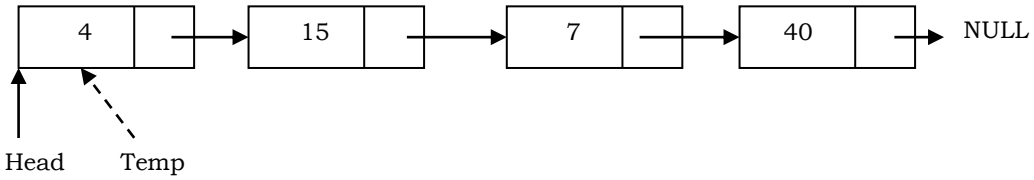
Similar to insertion, here we also have three cases.

- Deleting the first node
- Deleting the last node
- Deleting an intermediate node.

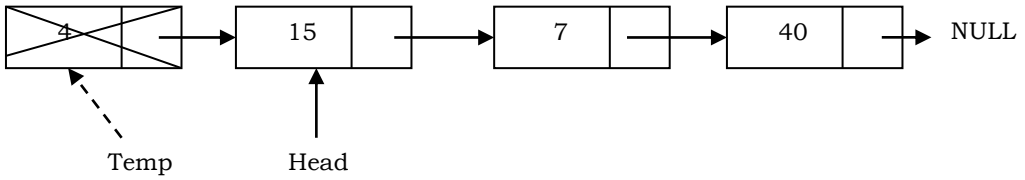
## Deleting the First Node in Singly Linked List

First node (current head node) is removed from the list. It can be done in two steps:

- Create a temporary node which will point to the same node as that of head.




- Now, move the head nodes pointer to the next node and dispose of the temporary node.

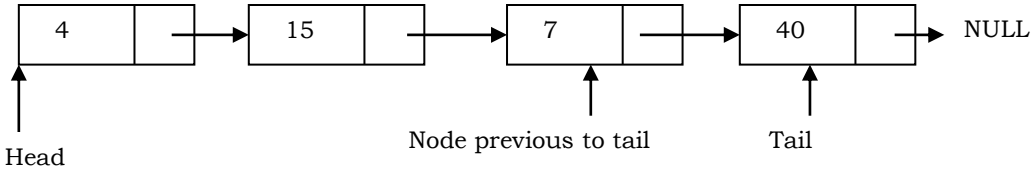



```
#method to delete the first node of the linked list
def deleteFromBeginning(self):
    if self.length == 0:
        print "The list is empty"
    else:
        self.head = self.head.getNext()
        self.length -= 1
```

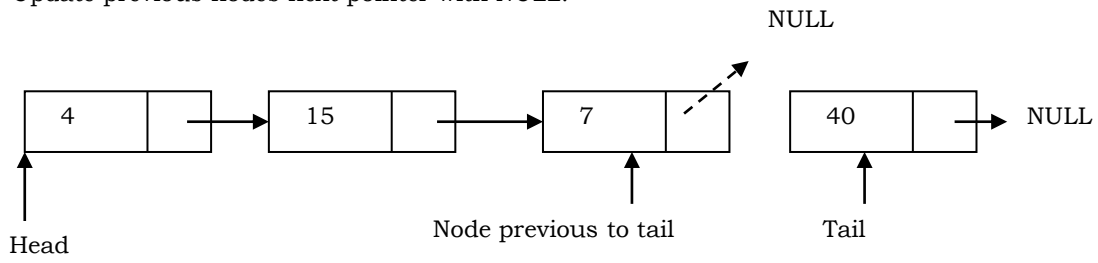
## Deleting the Last Node in Singly Linked List

In this case, the last node is removed from the list. This operation is a bit trickier than removing the first node, because the algorithm should find a node, which is previous to the tail. It can be done in three steps:

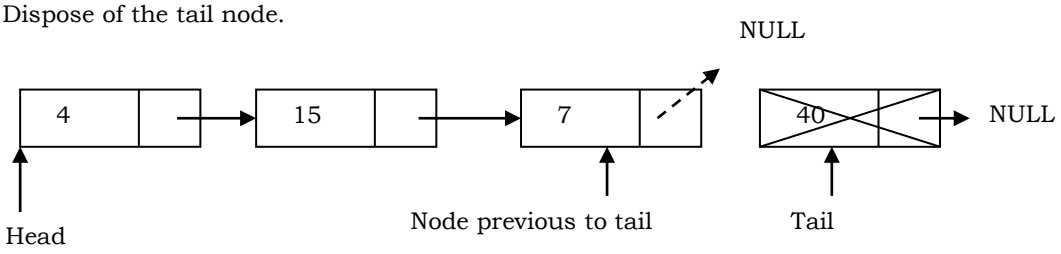
- Traverse the list and while traversing maintain the previous node address also. By the time we reach the end of the list, we will have two pointers, one pointing to the *tail* node and the other pointing to the node *before* the tail node.




- Update previous nodes next pointer with NULL.




- Dispose of the tail node.

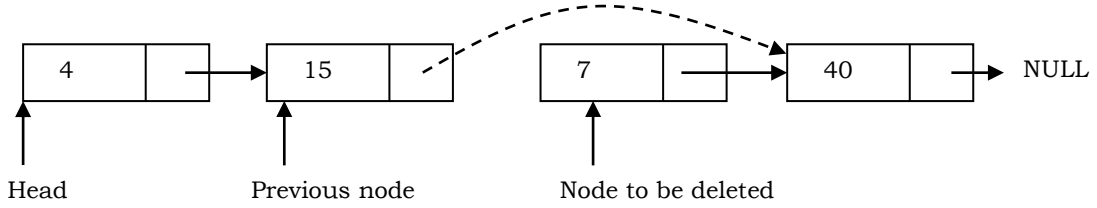



```
#Method to delete the last node of the linked list
def deleteLastNodeFromSinglyLinkedList(self):
    if self.length == 0:
        print "The list is empty"
    else:
        currentnode = self.head
        previousnode = self.head
        while currentnode.getNext() != None:
            previousnode = currentnode
            currentnode = currentnode.getNext()
        previousnode.seNext(None)
        self.length -= 1
```

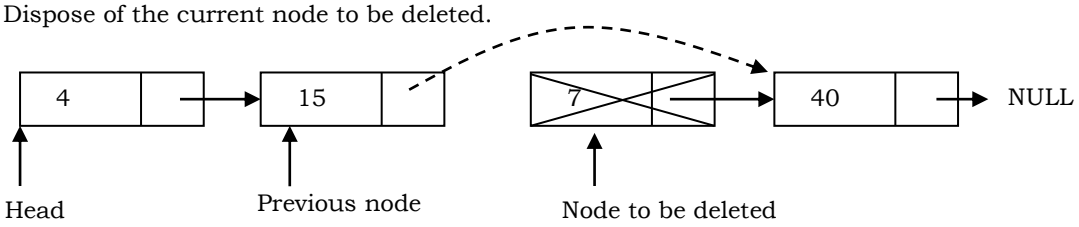
## Deleting an Intermediate Node in Singly Linked List

In this case, the node to be removed is *always located between* two nodes. Head and tail links are not updated in this case. Such a removal can be done in two steps:

- Similar to the previous case, maintain the previous node while traversing the list. Once we find the node to be deleted, change the previous node's next pointer to the next pointer of the node to be deleted.




- Dispose of the current node to be deleted.




```
#Delete with node from linked list
def deleteFromLinkedListWithNode(self, node):
    if self.length == 0:
       raise ValueError("List is empty")
    else:
       current = self.head
       previous = None
       found = False
       while not found:
            if current == node:
                    found = True
            elif current is None:
                    raise ValueError("Node not in Linked List")
            else:
                    previous = current
                    current = current.getNext()
    if previous is None:
            self.head = current.getNext()
    else:
            previous.seNext(current.getNext())
    self.length -= 1
#Delete with data from linked list
def deleteValue(self,value):
       currentnode = self.head
       previousnode = self.head
       while currentnode.next != None or currentnode.value != value:
          if currentnode.value == value:
             previousnode.next = currentnode.next
             self.length -= 1
             return
          else:
             previousnode = currentnode
             currentnode = currentnode.next
       print "The value provided is not present"
#Method to delete a node at a particular position
def deleteAtPosition(self,pos):
       count = 0
```

```
        currentnode = self.head
        previousnode = self.head
        if pos > self.length or pos < 0:
            print "The position does not exist. Please enter a valid position"
        else:
            while currentnode.next != None or count < pos:
                count = count + 1
                if count == pos:
                    previousnode.next = currentnode.next
                    self.length -= 1
                    return
                else:
                    previousnode = currentnode
                    currentnode = currentnode.next
```

Time Complexity: O(*n*). In the worst case, we may need to delete the node at the end of the list.
Space Complexity: O(1), for one temporary variable.

### Deleting Singly Linked List

Python is garbage-collected, so if you reduce the size of your list, it will reclaim memory.

```
def clear( self ) :
    self.head = None
```

Time Complexity: O(1). Space Complexity: O(1)

## 3.7 Doubly Linked Lists

The *advantage* of a doubly linked list (also called *two – way linked list*) is that given a node in the list, we can navigate in both directions. A node in a singly linked list cannot be removed unless we have the pointer to its predecessor. But in a doubly linked list, we can delete a node even if we don't have the previous node's address (since each node has a left pointer pointing to the previous node and can move backward).

The primary *disadvantages* of doubly linked lists are:

- Each node requires an extra pointer, requiring more space.
- The insertion or deletion of a node takes a bit longer (more pointer operations).

Similar to a singly linked list, let us implement the operations of a doubly linked list. If you understand the singly linked list operations, then doubly linked list operations are obvious. Following is a type declaration for a doubly linked list of integers:

```
class Node:
    # If data is not given by user,its taken as None
    def __init__(self, data=None, next=None, prev=None):
        self.data = data
        self.next = next
        self.prev = prev
    #method for setting the data field of the node
    def setData(self,data):
        self.data = data
    #method for getting the data field of the node
    def getData(self):
        return self.data
    #method for setting the next field of the node
    def setNext(self,next):
        self.next = next
    #method for getting the next field of the node
    def getNext(self):
        return self.next
    #returns true if the node points to another node
    def hasNext(self):
```

```
            return self.next != None
        #method for setting the next field of the node
        def setPrev(self,prev):
            self.prev = prev
        #method for getting the next field of the node
        def getPrev(self):
            return self.prev
        #returns true if the node points to another node
        def hasPrev(self):
            return self.prev != None
        # __str__ returns string equivalent of Object
        def __str__(self):
    return "Node[Data = %s]" % (self.data,)
```

## Doubly Linked List Insertion

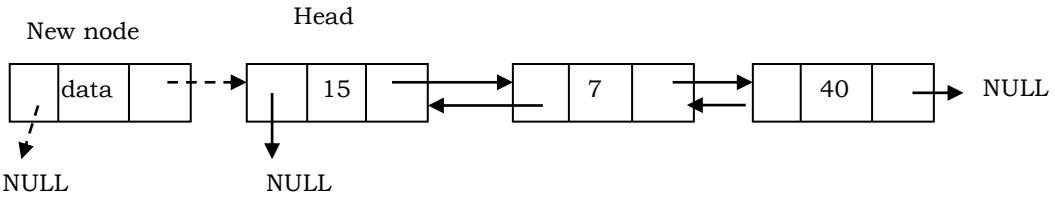
Insertion into a doubly-linked list has three cases (same as a singly linked list):

- Inserting a new node before the head.
- Inserting a new node after the tail (at the end of the list).
- Inserting a new node at the middle of the list.

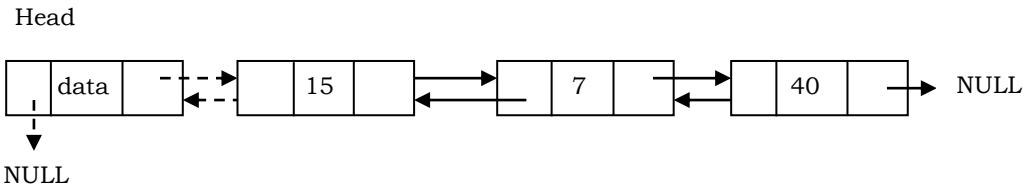
## Inserting a Node in Doubly Linked List at the Beginning

In this case, new node is inserted before the head node. Previous and next pointers need to be modified and it can be done in two steps:

- Update the right pointer of the new node to point to the current head node (dotted link in below figure) and also make left pointer of new node as NULL.




- Update head node's left pointer to point to the new node and make new node as head.

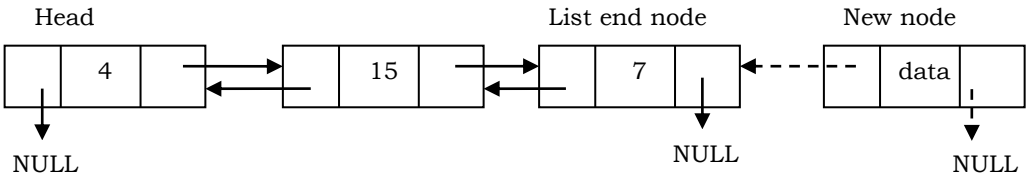



```
    def insertAtBeginning(self, data):
        newNode = Node(data, None, None)
        if (self.head == None):     # To imply that if head == None
            self.head = self.tail = newNode
        else:
            newNode.setPrev(None)
            newNode.setNext(self.head)
            self.head.setPrev(newNode)
            self.head = newNode
```

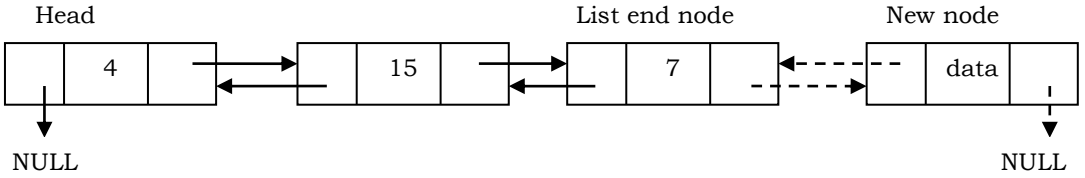
## Inserting a Node in Doubly Linked List at the Ending

In this case, traverse the list till the end and insert the new node.

- New node right pointer points to NULL and left pointer points to the end of the list.




- Update right pointer of last node to point to new node.

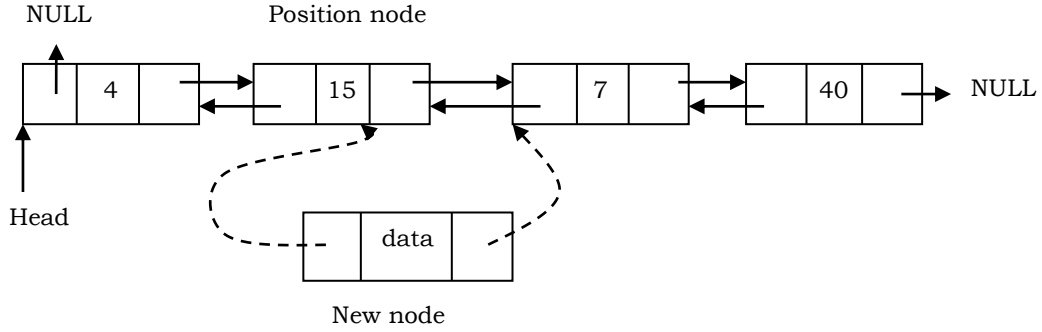



```
def insertAtEnd(self, data):
    if (self.head == None):     # To imply that if head == None
        self.head = Node(data)
        self.tail = self.head
    else:
        current = self.head
        while(current.getNext() != None):
            current = current.getNext()
        current.setNext(Node(data, None, current))
        self.tail = current.getNext()
```

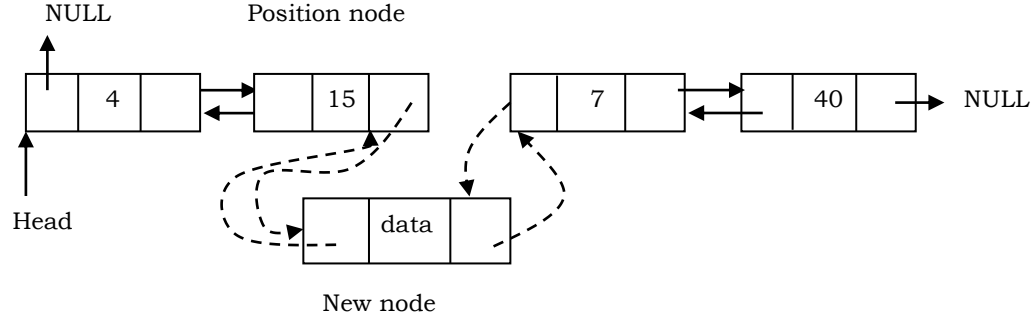
## Inserting a Node in Doubly Linked List at the Middle

As discussed in singly linked lists, traverse the list to the position node and insert the new node.

- *New node* right pointer points to the next node of the *position node* where we want to insert the new node. Also, *new node* left pointer points to the *position node*.




- Position node right pointer points to the new node and the *next node* of position node left pointer points to new node.




Now, let us write the code for all of these three cases. We must update the first element pointer in the calling function, not just in the called function. For this reason we need to send a double pointer. The following code inserts a node in the doubly linked list.

```
def getNode(self, index):
    currentNode = self.head
    if currentNode == None:
        return None
    i = 0
    while i < index and currentNode.getNext() is not None::
        currentNode = currentNode.getNext()
        if currentNode == None:
            break
        i += 1
    return currentNode
def insertAtGivenPosition(self, index, data):
    newNode = Node(data)
    if self.head == None or index == 0:
        self.insertAtBeginning(data)
    elif index > 0:
        temp = self.getNode(index)
        if temp == None or temp.getNext() == None:
            self.insert(data)
        else:
            newNode.setNext(temp.getNext())
            newNode.setPrev(temp)
            temp.getNext().setPrev(newNode)
            temp.setNext(newNode)
```

Time Complexity: O($n$). In the worst case, we may need to insert the node at the end of the list.
Space Complexity: O(1), for a temporary variable.

## Doubly Linked List Deletion

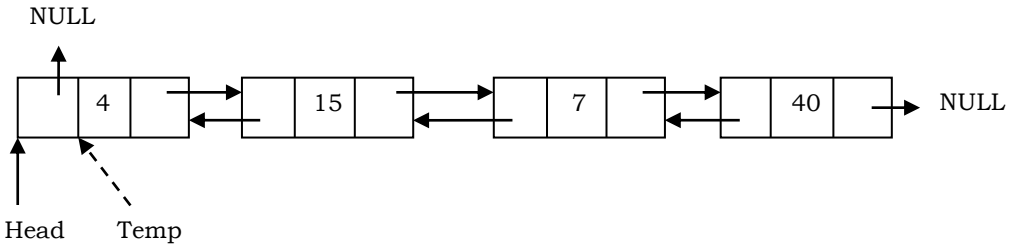
Similar to singly linked list deletion, here we have three cases:

- Deleting the first node
- Deleting the last node
- Deleting an intermediate node

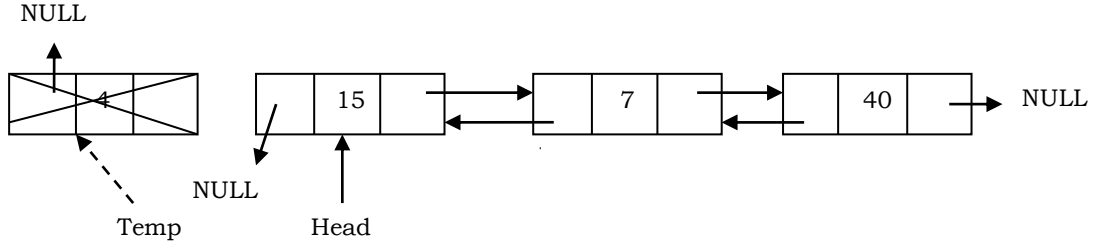
## Deleting the First Node in Doubly Linked List

In this case, the first node (current head node) is removed from the list. It can be done in two steps:

- Create a temporary node which will point to the same node as that of head.



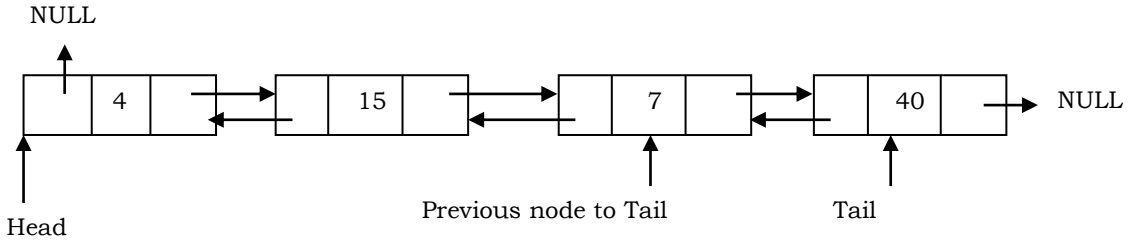

- Now, move the head nodes pointer to the next node and change the heads left pointer to NULL. Then, dispose of the temporary node.



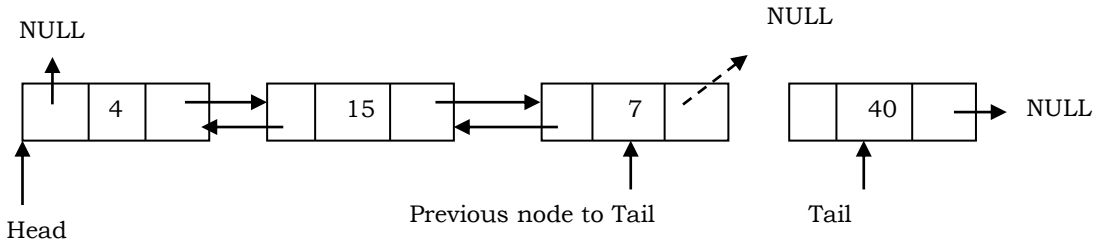
## Deleting the Last Node in Doubly Linked List

This operation is a bit trickier, than removing the first node, because the algorithm should find a node, which is previous to the tail first. This can be done in three steps:

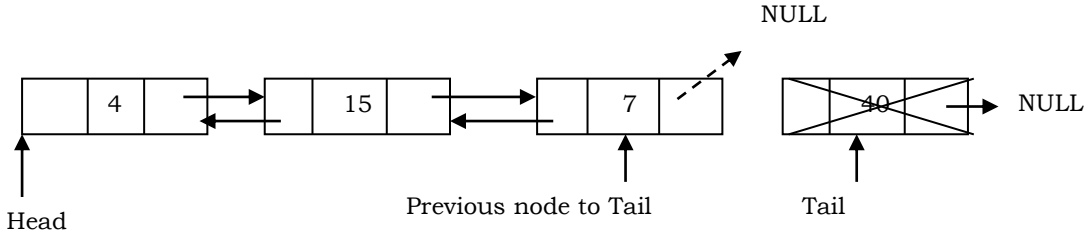
- Traverse the list and while traversing maintain the previous node address also. By the time we reach the end of the list, we will have two pointers, one pointing to the tail and the other pointing to the node before the tail.




- Update the next pointer of previous node to the tail node with NULL.



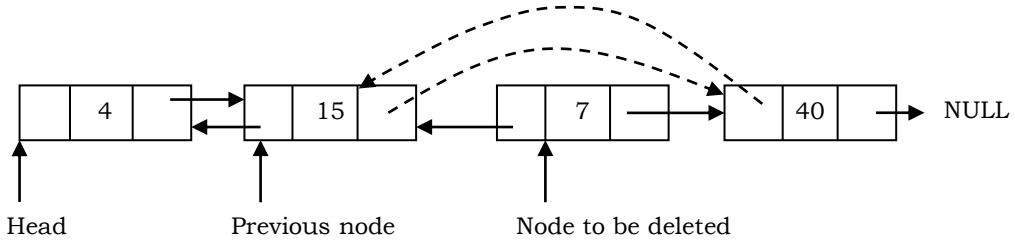

- Dispose of the tail node.



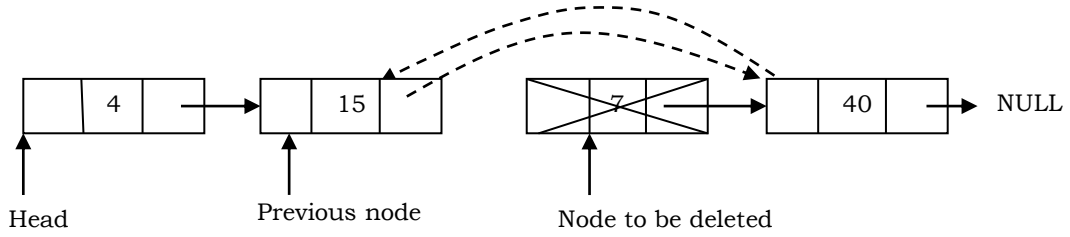

## Deleting an Intermediate Node in Doubly Linked List

In this case, the node to be removed is *always located between* two nodes, and the head and tail links are not updated. The removal can be done in two steps:

- Similar to the previous case, maintain the previous node while also traversing the list. Upon locating the node to be deleted, change the previous node's next pointer to the next node of the node to be deleted.




- Dispose of the current node to be deleted.




```
#Deleting element at given position
def getNode(self, index):
    currentNode = self.head
    if currentNode == None:
        return None
    i = 0
    while i <= index:
        currentNode = currentNode.getNext()
        if currentNode == None:
            break
        i += 1
    return currentNode

def delteAtGivenPosition(self, index):
    temp = self.getNode(index)
    if temp:
        temp.getPrev.setNext(temp.getNext())
        if temp.getNext():
            temp.getNext().setPrev(temp.getPrev())
        temp.setPrev(None)
        temp.setNext(None)
        temp.setData(None)

#Deleting with given data
def deleteWithData(self, data):
    temp = self.head
    while temp is not None:
        if temp.getData() == data:
            # if it's not the first element
            if temp.getNext() is not None:
                temp.getNext().setNext(temp.getNext())
                temp.getNext().setPrev(temp.getPrev())
            else:
                # otherwise we have no prev (it's None), head is the next one, and prev becomes None
                self.head = temp.getNext()
                temp.getNext().setPrev(None)
        temp = temp.getNext()
```

Time Complexity: O($n$), for scanning the complete list of size $n$.
Space Complexity: O(1), for creating one temporary variable.

## 3.8 Circular Linked Lists

In singly linked lists and doubly linked lists, the end of lists are indicated with NULL value. But circular linked lists do not have ends. While traversing the circular linked lists we should be careful; otherwise we will be traversing the list infinitely. In circular linked lists, each node has a successor. Note that unlike singly linked lists, there is no node with NULL pointer in a circularly linked list. In some situations, circular linked lists are useful. There is no difference in the node declaration of circular linked lists compared to singly linked lists.

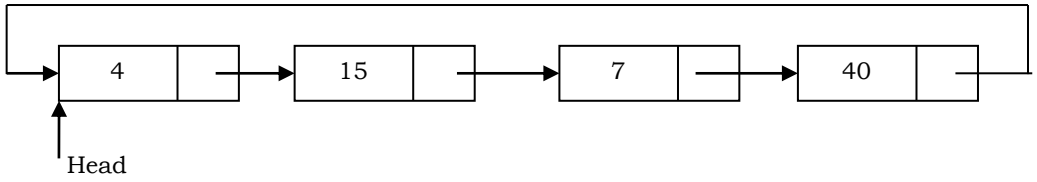
For example, when several processes are using the same computer resource (CPU) for the same amount of time, we have to assure that no process accesses the resource before all other processes do (round robin algorithm). The following is a type declaration for a circular linked list:

```
#Node of a Circular Linked List
class Node:
    #constructor
    def __init__(self):
```

```
        self.data = None
        self.next = None
    #method for setting the data field of the node
    def setData(self,data):
        self.data = data
    #method for getting the data field of the node
    def getData(self):
        return self.data
    #method for setting the next field of the node
    def setNext(self,next):
        self.next = next
    #method for getting the next field of the node
    def getNext(self):
        return self.next
    #returns true if the node points to another node
    def hasNext(self):
            return self.next != None
```

In a circular linked list, we access the elements using the *head* node (similar to *head* node in singly linked list and doubly linked lists).

## Counting Nodes in a Circular List

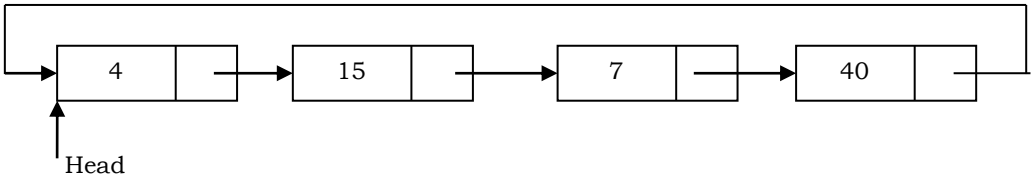



The circular list is accessible through the node marked *head*. To count the nodes, the list has to be traversed from the node marked *head*, with the help of a dummy node *current*, and stop the counting when *current* reaches the starting node *head*. If the list is empty, *head* will be NULL, and in that case set $count = 0$. Otherwise, set the current pointer to the first node, and keep on counting till the current pointer reaches the starting node.

```
    #This method would be a member of other class (say, CircularList)
    def circularListLength(self):
        currentNode = self.head
        if currentNode == None:
                return 0
        count = 1
        currentNode = currentNode.getNext()
        while currentNode !=  self.head:
                currentNode = currentNode.getNext()
                count =  count + 1
        retur  count
```

Time Complexity: O($n$), for scanning the complete list of size $n$. Space Complexity: O(1), for temporary variable.

## Printing the Contents of a Circular List




We assume here that the list is being accessed by its *head* node. Since all the nodes are arranged in a circular fashion, the *tail* node of the list will be the node previous to the *head* node. Let us assume we want to print the

contents of the nodes starting with the *head* node. Print its contents, move to the next node and continue printing till we reach the *head* node again.

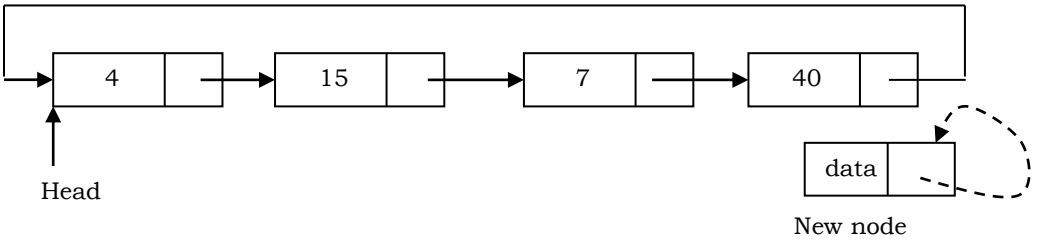
```
def printCircularList(self):
    currentNode = self.head
    if currentNode == None:   return 0
    print (currentNode.getData())
    currentNode = currentNode.getNext()
    while currentNode !=  self.head:
            currentNode = currentNode.getNext()
            print (currentNode.getData())
```

Time Complexity: O($n$), for scanning the complete list of size $n$. Space Complexity: O(1), for temporary variable.

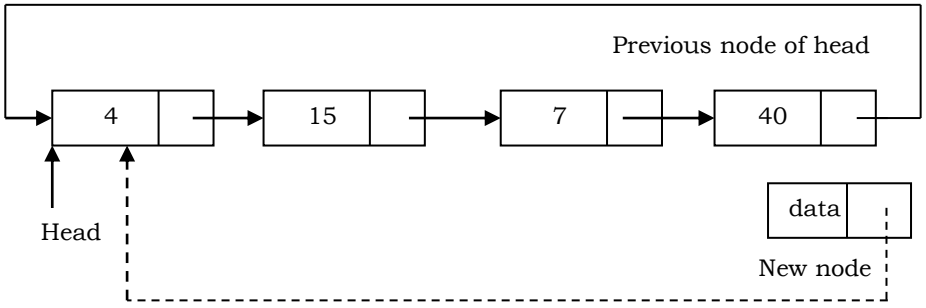
## Inserting a Node at the End of a Circular Linked List

Let us add a node containing *data*, at the end of a list (circular list) headed by *head*. The new node will be placed just after the tail node (which is the last node of the list), which means it will have to be inserted in between the tail node and the first node.

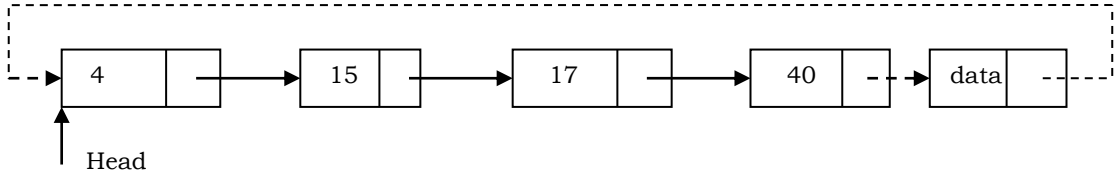
- Create a new node and initially keep its next pointer pointing to itself.




- Update the next pointer of the new node with the head node and also traverse the list to the tail. That means in a circular list we should stop at the node whose next node is head.




- Update the next pointer of the previous node to point to the new node and we get the list as shown below.




```
def insertAtEndInCLL (self, data):
    current = self.head
    newNode = Node()
    newNode.setData(data)
    while current.getNext != self.head:
            current = current.getNext()
    newNode.setNext(newNode)
    if self.head == None:
            self.head = newNode;
    else:
```

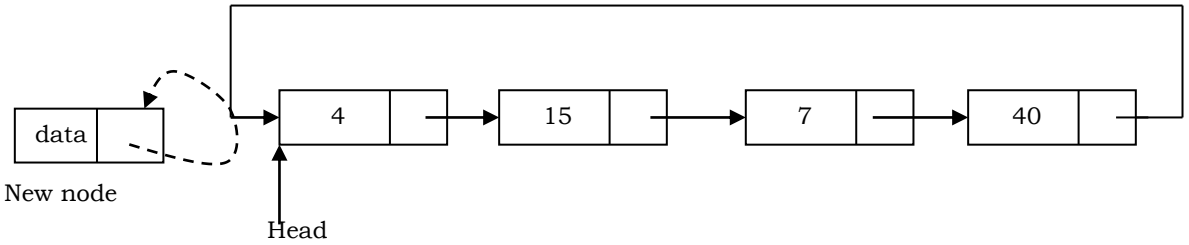
```
            newNode.setNext(self.head)
            current.setNext(newNode)
```

Time Complexity: O($n$), for scanning the complete list of size $n$. Space Complexity: O(1), for temporary variable.

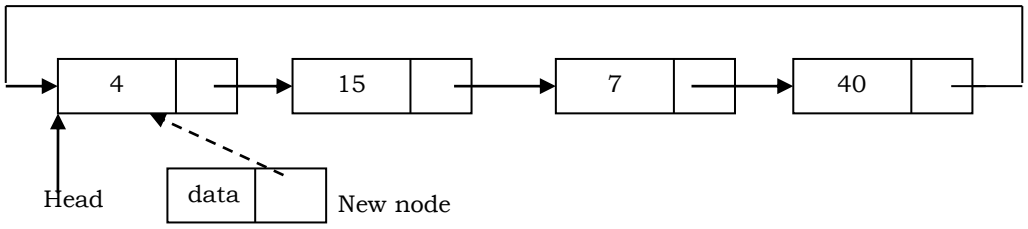
## Inserting a Node at the Front of a Circular Linked List

The only difference between inserting a node at the beginning and at the end is that, after inserting the new node, we just need to update the pointer. The steps for doing this are given below:

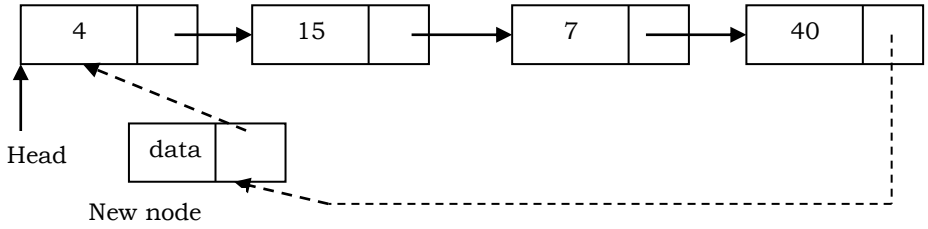
- Create a new node and initially keep its next pointer pointing to itself.



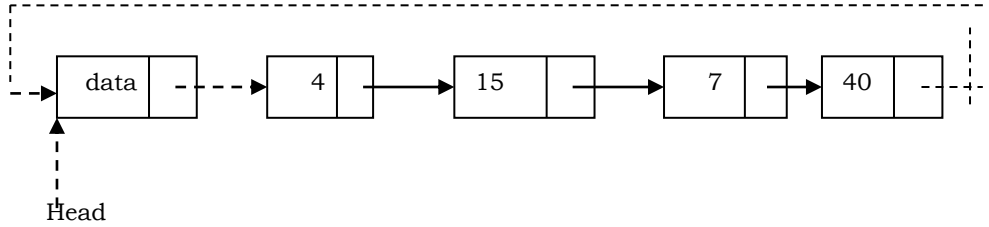

- Update the next pointer of the new node with the head node and also traverse the list until the tail. That means in a circular list we should stop at the node which is its previous node in the list.




- Update the previous head node in the list to point to the new node.




- Make the new node as the head.




```
def insertAtBeginInCLL (self, data):
    current = self.head
    newNode = Node()
    newNode.setData(data)
    while current.getNext != self.head:
            current = current.getNext()
    newNode.setNext(newNode)
    if self.head == None:
            self.head = newNode;
    else:
            newNode.setNext(self.head)
            current.setNext(newNode)
```

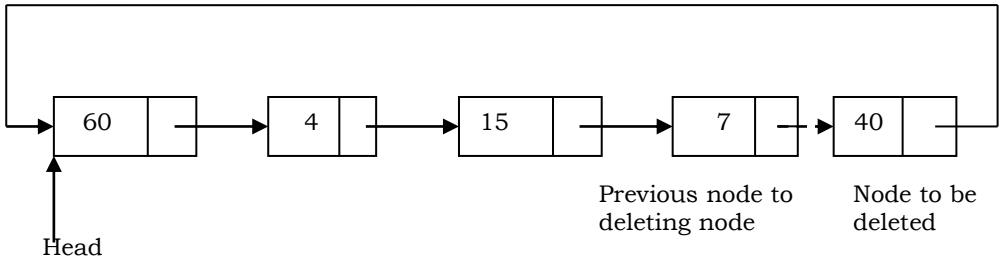
```
            self.head = newNode
```

Time Complexity: O($n$), for scanning the complete list of size $n$. Space Complexity: O(1), for temporary variable.

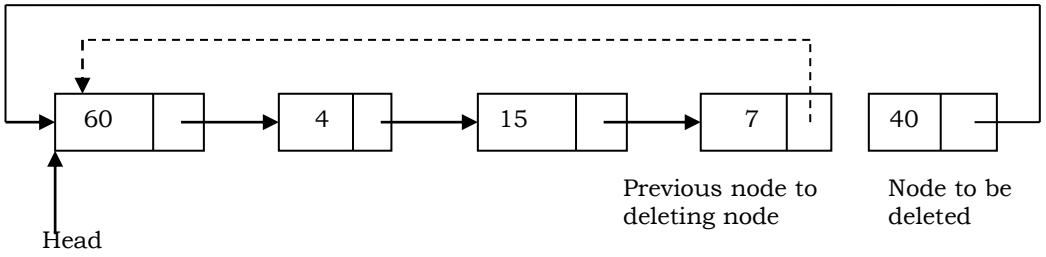
## Deleting the Last Node in a Circular List

The list has to be traversed to reach the last but one node. This has to be named as the tail node, and its next field has to point to the first node. Consider the following list. To delete the last node 40, the list has to be traversed till you reach 7. The next field of 7 has to be changed to point to 60, and this node must be renamed $pTail$.

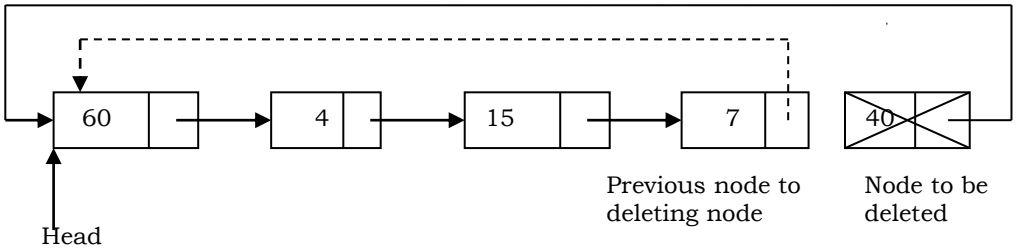
- Traverse the list and find the tail node and its previous node.




- Update the tail node's previous node pointer to point to head.




- Dispose of the tail node.



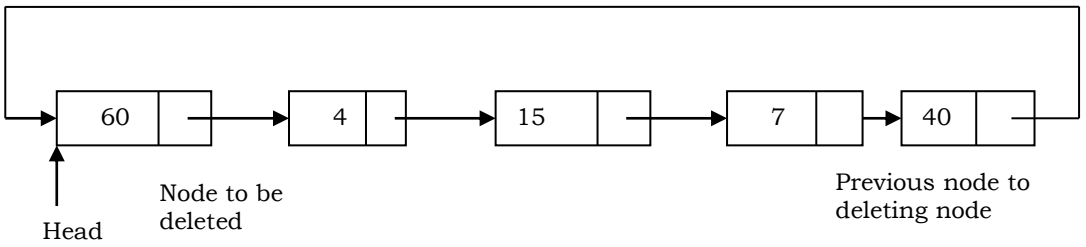

```
def deleteLastNodeFromCLL (self):
    temp = self.head
    current = self.head
    if self.head == None:
        print ("List Empty")
        return
    while current.getNext() != self.head:
        temp = current;
        current = current.getNext()
    temp.setNext(current.getNext())
    return
```

Time Complexity: O($n$), for scanning the complete list of size $n$. Space Complexity: O(1)

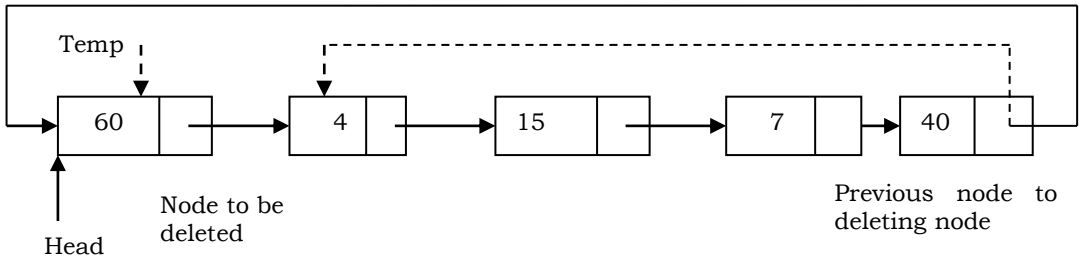
## Deleting the First Node in a Circular List

The first node can be deleted by simply replacing the next field of the tail node with the next field of the first node.

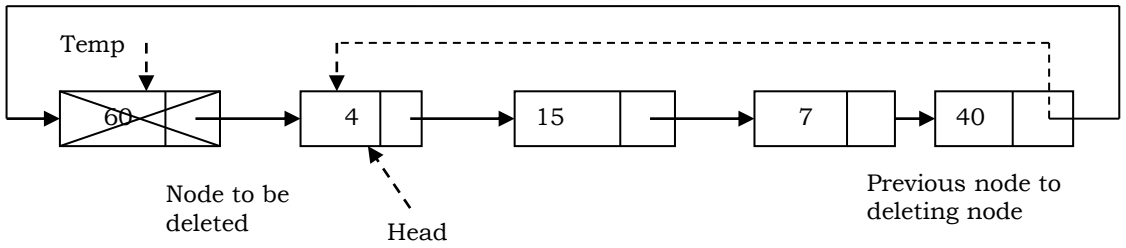
- Find the tail node of the linked list by traversing the list. Tail node is the previous node to the head node which we want to delete.




- Create a temporary node which will point to the head. Also, update the tail nodes next pointer to point to next node of head (as shown below).




- Now, move the head pointer to next node. Create a temporary node which will point to head. Also, update the tail nodes next pointer to point to next node of head (as shown below).




```
def deleteFrontNodeFromCLL (self):
  current = self.head
  if self.head == None:
     print ("List Empty")
     return
  while current.getNext() != self.head:
     current = current.getNext()
  current.setNext(self.head.getNext())
  self.head = self.head.getNext()
  return
```

Time Complexity: O($n$), for scanning the complete list of size $n$. Space Complexity: O(1)

## Applications of Circular List

Circular linked lists are used in managing the computing resources of a computer. We can use circular lists for implementing stacks and queues.

# 3.9 A Memory-efficient Doubly Linked List

In conventional implementation, we need to keep a forward pointer to the next item on the list and a backward pointer to the previous item. That means elements in doubly linked list implementations consist of data, a pointer to the next node and a pointer to the previous node in the list as shown below.

**Conventional Node Definition**

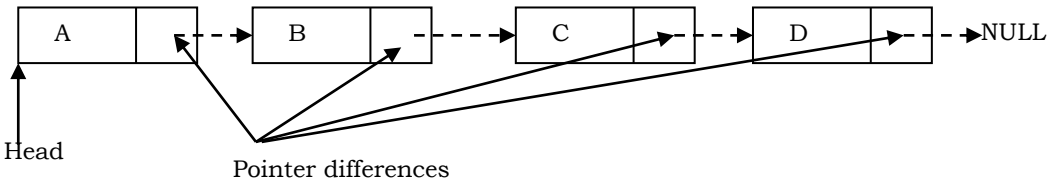
```
#Node of a Singly Linked List
class Node:
   #constructor
   def __init__(self):
      self.data = None
```

```
        self.next = None
    #method for setting the data field of the node
    def setData(self,data):
        self.data = data
    #method for getting the data field of the node
    def getData(self):
        return self.data
    #method for setting the next field of the node
    def setNext(self,next):
        self.next = next
    #method for getting the next field of the node
    def getNext(self):
        return self.next
    #returns true if the node points to another node
    def hasNext(self):
            return self.next != None
```

Recently a journal (Sinha) presented an alternative implementation of the doubly linked list ADT, with insertion, traversal and deletion operations. This implementation is based on pointer difference. Each node uses only one pointer field to traverse the list back and forth.

**New Node Definition**

```
class Node:
    #constructor
    def __init__(self):
        self.data = None
        self.ptrdiff = None
    #method for setting the data field of the node
    def setData(self,data):
        self.data = data
    #method for getting the data field of the node
    def getData(self):
        return self.data
    #method for setting the pointer difference field of the node
    def setPtrDiff(self, prev, next):
        self.ptrdiff = prev ^ next
    #method for getting the next field of the node
    def getPtrDiff(self):
        return self.ptrdiff
```




The $ptrdiff$ pointer field contains the difference between the pointer to the next node and the pointer to the previous node. The pointer difference is calculated by using exclusive-or (⊕) operation.

$$ptrdiff = pointer\ to\ previous\ node \oplus pointer\ to\ next\ node.$$

The $ptrdiff$ of the start node (head node) is the ⊕ of NULL and *next* node (next node to head). Similarly, the $ptrdiff$ of end node is the ⊕ of *previous* node (previous to end node) and NULL. As an example, consider the following linked list.

In the example above,

- The next pointer of A is: NULL ⊕ B
- The next pointer of B is: A ⊕ C
- The next pointer of C is: B ⊕ D
- The next pointer of D is: C ⊕ NULL

**Why does it work?**

To find the answer to this question let us consider the properties of ⊕:

$X \oplus X = 0$
$X \oplus 0 = X$
$X \oplus Y = Y \oplus X$ (symmetric)
$(X \oplus Y) \oplus Z = X \oplus (Y \oplus Z)$ (transitive)

For the example above, let us assume that we are at C node and want to move to B. We know that C's $ptrdiff$ is defined as B $\oplus$ D. If we want to move to B, performing $\oplus$ on C's $ptrdiff$ with D would give B. This is due to the fact that

$$(B \oplus D) \oplus D = B \text{ (since, } D \oplus D=0)$$

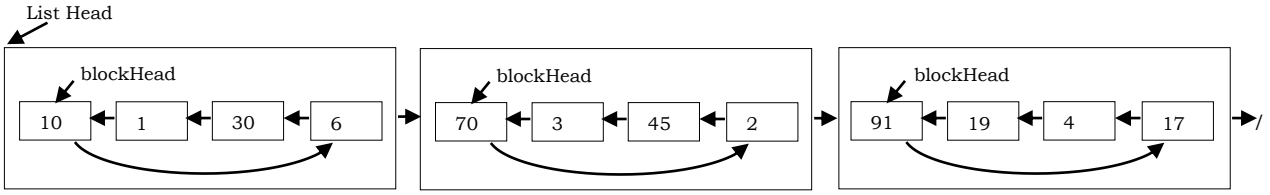
Similarly, if we want to move to D, then we have to apply $\oplus$ to C's $ptrdiff$ with B to give D.

$$(B \oplus D) \oplus B = D \text{ (since, } B \oplus B=0)$$

From the above discussion we can see that just by using a single pointer, we can move back and forth. A memory-efficient implementation of a doubly linked list is possible with minimal compromising of timing efficiency.

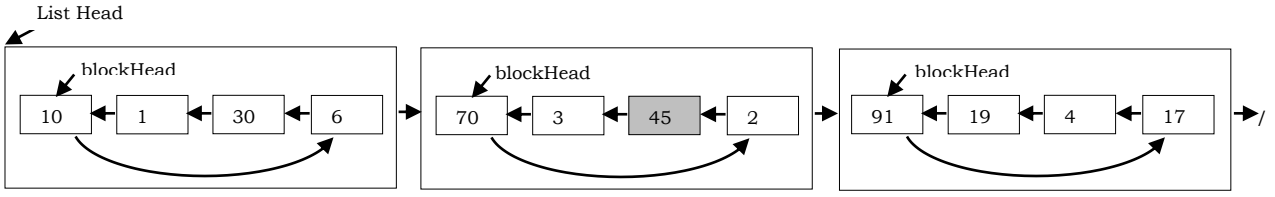
# 3.10 Unrolled Linked Lists

One of the biggest advantages of linked lists over arrays is that inserting an element at any location takes only O(1) time. However, it takes O($n$) to search for an element in a linked list. There is a simple variation of the singly linked list called *unrolled linked lists*. An unrolled linked list stores multiple elements in each node (let us call it a block for our convenience). In each block, a circular linked list is used to connect all nodes.




Assume that there will be no more than $n$ elements in the unrolled linked list at any time. To simplify this problem, all blocks, except the last one, should contain exactly $\lceil\sqrt{n}\rceil$ elements. Thus, there will be no more than $\lfloor\sqrt{n}\rfloor$ blocks at any time.

## Searching for an element in Unrolled Linked Lists

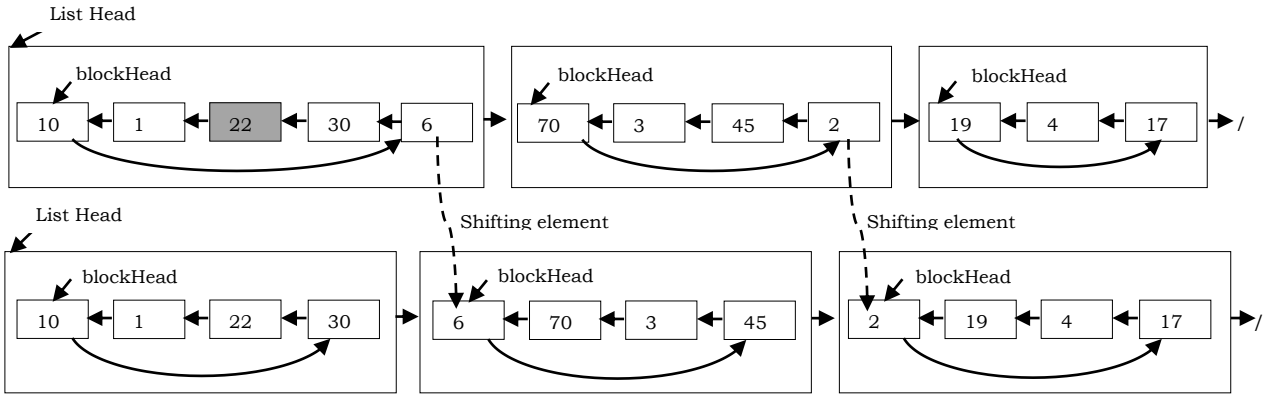



In unrolled linked lists, we can find the $k^{th}$ element in O($\sqrt{n}$):

1. Traverse the *list of blocks* to the one that contains the $k^{th}$ node, i.e., the $\left\lceil\frac{k}{\lceil\sqrt{n}\rceil}\right\rceil^{\text{th}}$ block. It takes O($\sqrt{n}$) since we may find it by going through no more than $\sqrt{n}$ blocks.
2. Find the ($k$ mod $\lceil\sqrt{n}\rceil$)$^{\text{th}}$ node in the circular linked list of this block. It also takes O($\sqrt{n}$) since there are no more than $\lceil\sqrt{n}\rceil$ nodes in a single block.

## Inserting an element in Unrolled Linked Lists

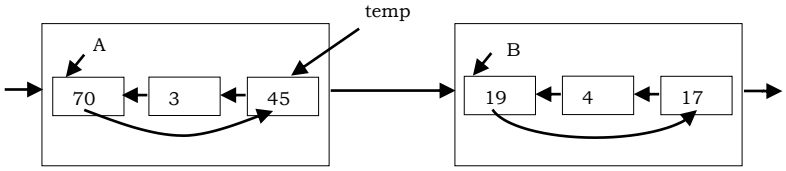
When inserting a node, we have to re-arrange the nodes in the unrolled linked list to maintain the properties previously mentioned, that each block contains $\lceil\sqrt{n}\rceil$ nodes. Suppose that we insert a node $x$ after the $i^{th}$ node, and $x$ should be placed in the $j^{th}$ block. Nodes in the $j^{th}$ block and in the blocks after the $j^{th}$ block have to be shifted toward the tail of the list so that each of them still have $\lceil\sqrt{n}\rceil$ nodes. In addition, a new block needs to be added to the tail if the last block of the list is out of space, i.e., it has more than $\lceil\sqrt{n}\rceil$ nodes.



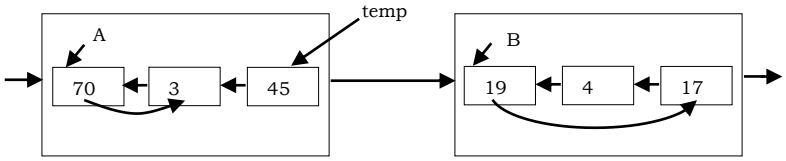

## Performing Shift Operation

Note that each *shift* operation, which includes removing a node from the tail of the circular linked list in a block and inserting a node to the head of the circular linked list in the block after, takes only O(1). The total time complexity of an insertion operation for unrolled linked lists is therefore O($\sqrt{n}$); there are at most O($\sqrt{n}$) blocks and therefore at most O($\sqrt{n}$) shift operations.

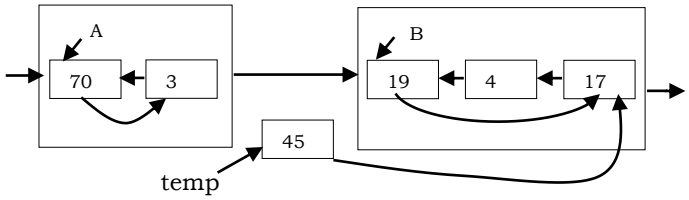
1. A temporary pointer is needed to store the tail of *A*.



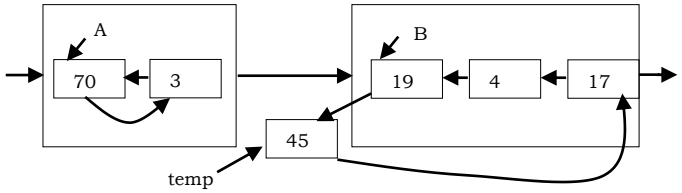

2. In block *A*, move the next pointer of the head node to point to the second-to-last node, so that the tail node of *A* can be removed.



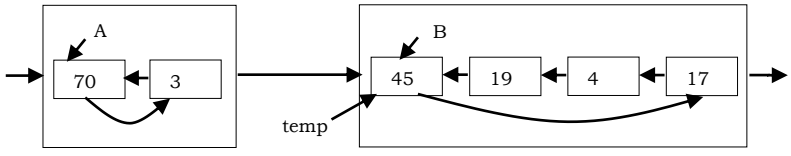

3. Let the next pointer of the node, which will be shifted (the tail node of *A*), point to the tail node of *B*.



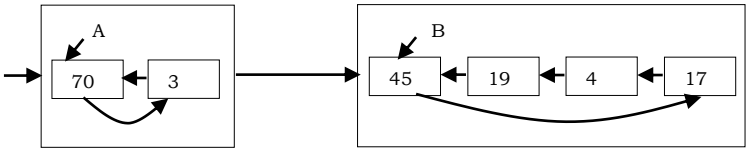

4. Let the next pointer of the head node of *B* point to the node temp points to.




5. Finally, set the head pointer of *B* to point to the node *temp* points to. Now the node temp points to becomes the new head node of *B*.



6. *temp* pointer can be thrown away. We have completed the shift operation to move the original tail node of $A$ to become the new head node of $B$.




## Performance

With unrolled linked lists, there are a couple of advantages, one in speed and one in space. First, if the number of elements in each block is appropriately sized (e.g., at most the size of one cache line), we get noticeably better cache performance from the improved memory locality. Second, since we have O($n/m$) links, where $n$ is the number of elements in the unrolled linked list and $m$ is the number of elements we can store in any block, we can also save an appreciable amount of space, which is particularly noticeable if each element is small.

## Comparing Doubly Linked Lists and Unrolled Linked Lists

To compare the overhead for an unrolled list, elements in doubly linked list implementations consist of data, a pointer to the next node, and a pointer to the previous node in the list, as shown below.

```
class Node:
    # If data is not given by user,its taken as None
    def __init__(self, data=None, next=None, prev=None):
        self.data = data
        self.next = next
        self.prev = prev
```

Assuming we have 4 byte pointers, each node is going to take 8 bytes. But the allocation overhead for the node could be anywhere between 8 and 16 bytes. Let's go with the best case and assume it will be 8 bytes. So, if we want to store 1K items in this list, we are going to have 16KB of overhead.

Now, let's think about an unrolled linked list node (let us call it *LinkedBlock*). It will look something like this:

```
class LinkedBlock:
    def __init__(self, nextBlock=None, blockHead=None):
        self.next = nextBlock
        self.head = blockHead
        self.nodeCount = 0
```

Therefore, allocating a single node (12 bytes + 8 bytes of overhead) with an array of 100 elements (400 bytes + 8 bytes of overhead) will now cost 428 bytes, or 4.28 bytes per element. Thinking about our 1K items from above, it would take about 4.2KB of overhead, which is close to 4x better than our original list. Even if the list becomes severely fragmented and the item arrays are only 1/2 full on average, this is still an improvement. Also, note that we can tune the array size to whatever gets us the best overhead for our application.

## Implementation

```
#Node of a Singly Linked List
class Node:
  #constructor
  def __init__(self):
            self.value = None
            self.next = None

#Node of a Singly Linked List
class LinkedBlock:
  #constructor
  def __init__(self):
            self.head = None
            self.next = None
            nodeCount = 0

blockSize = 2
blockHead = None

#create an empty block
```

```
def newLinkedBlock():
  block=LinkedBlock()
  block.next=None
  block.head=None
  block.nodeCount=0
  return block
#create a node
def newNode(value):
  temp=Node()
  temp.next=None
  temp.value=value
  return temp
def searchElements(blockHead, k):
  #find the block
  j=(k+blockSize-1)//blockSize #k-th node is in the j-th block
  p=blockHead
  j -= 1
  while(j):
        p=p.next
        j -= 1
  fLinkedBlock=p
  #find the node
  q=p.head
  k=k%blockSize
  if(k==0):
        k=blockSize
  k = p.nodeCount+1-k
  k -= 1
  while (k):
        q=q.next
        k -= 1
  fNode=q
  return fLinkedBlock, fNode
#start shift operation from block *p
def shift(A):
  B = A
  global blockHead
  while(A.nodeCount > blockSize): #if this block still have to shift
        if(A.next==None): #reach the end. A little different
              A.next=newLinkedBlock()
              B=A.next
              temp=A.head.next
              A.head.next=A.head.next.next
              B.head=temp
              temp.next=temp
              A.nodeCount -= 1
              B.nodeCount += 1
        else:
              B=A.next
              temp=A.head.next
              A.head.next=A.head.next.next
              temp.next=B.head.next
              B.head.next=temp
              B.head=temp
              A.nodeCount -= 1
              B.nodeCount += 1
  A=B
def addElement(k, x):
  global blockHead
  r = newLinkedBlock()
```

```
    p = Node()
    if(blockHead == None): #initial, first node and block
            blockHead=newLinkedBlock()
            blockHead.head=newNode(x)
            blockHead.head.next=blockHead.head
            blockHead.nodeCount += 1
    else:
            if(k==0): #special case for k=0.
                    p=blockHead.head
                    q=p.next
                    p.next=newNode(x)
                    p.next.next=q
                    blockHead.head=p.next
                    blockHead.nodeCount += 1
                    shift(blockHead)
            else:
                    r, p = searchElements(blockHead, k)
                    q = p
                    while(q.next != p):
                            q=q.next
                    q.next=newNode(x)
                    q.next.next=p
                    r.nodeCount += 1
                    shift(r)
     return blockHead
def searchElement(blockHead, k):
   q, p = searchElements(blockHead, k)
   return p.value
blockHead = addElement(0,11)
blockHead = addElement(0,21)
blockHead = addElement(1,19)
blockHead = addElement(1,23)
blockHead = addElement(2,16)
blockHead = addElement(2,35)
searchElement(blockHead, 1)
```

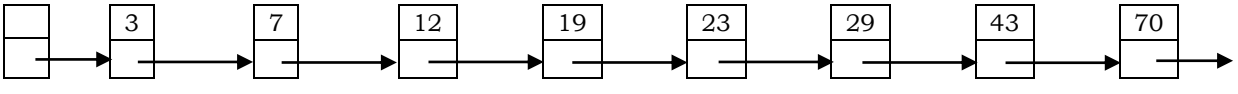
# 3.11 Skip Lists

Binary trees can be used for representing abstract data types such as dictionaries and ordered lists. They work well when the elements are inserted in a random order. Some sequences of operations, such as inserting the elements in order, produce degenerate data structures that give very poor performance. If it were possible to randomly permute the list of items to be inserted, trees would work well with high probability for any input sequence. In most cases queries must be answered on-line, so randomly permuting the input is impractical. Balanced tree algorithms re-arrange the tree as operations are performed to maintain certain balance conditions and assure good performance.

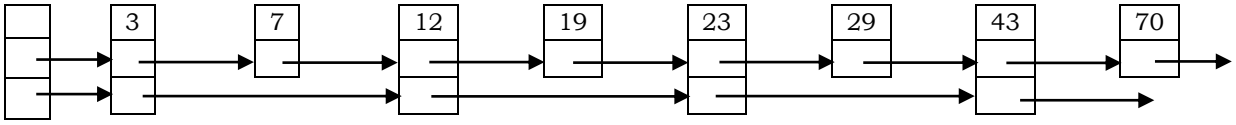
Skip list is a data structure that can be used as an alternative to balanced binary trees (refer to *Trees* chapter). As compared to a binary tree, skip lists allow quick search, insertion and deletion of elements. This is achieved by using probabilistic balancing rather than strictly enforce balancing. It is basically a linked list with additional pointers such that intermediate nodes can be skipped. It uses a random number generator to make some decisions.

In an ordinary sorted linked list, search, insert, and delete are in O(*n*) because the list must be scanned node-by-node from the head to find the relevant node. If somehow we could scan down the list in bigger steps (skip down, as it were), we would reduce the cost of scanning. This is the fundamental idea behind Skip Lists.

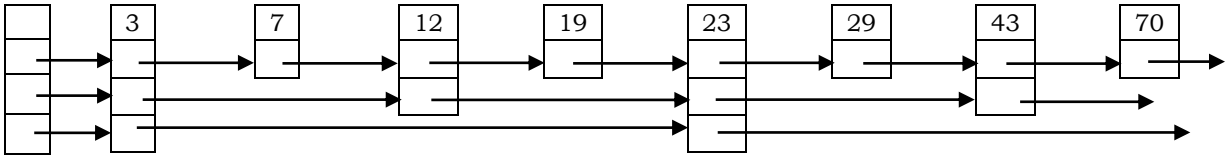
## Skip Lists with One Level



Skip Lists with Two Levels




Skip Lists with Three Levels




This section gives algorithms to search for, insert and delete elements in a dictionary or symbol table. The Search operation returns the contents of the value associated with the desired key or failure if the key is not present. The Insert operation associates a specified key with a new value (inserting the key if it had not already been present). The Delete operation deletes the specified key. It is easy to support additional operations such as "find the minimum key" or "find the next key".

Each element is represented by a node, the level of which is chosen randomly when the node is inserted without regard for the number of elements in the data structure. A level i node has i forward pointers, indexed 1 through i. We do not need to store the level of a node in the node. Levels are capped at some appropriate constant $MaxLevel$. The level of a list is the maximum level currently in the list (or 1 if the list is empty). The header of a list has forward pointers at levels one through MaxLevel. The forward pointers of the header at levels higher than the current maximum level of the list point to NULL.

## Initialization

An element NIL is allocated and given a key greater than any legal key. All levels of all skip lists are terminated with NIL. A new list is initialized so that the level of the list is equal to 1 and all forward pointers of the list's header point to NIL.

## Search for an element

We search for an element by traversing forward pointers that do not overshoot the node containing the element being searched for. When no more progress can be made at the current level of forward pointers, the search moves down to the next level. When we can make no more progress at level 1, we must be immediately in front of the node that contains the desired element (if it is in the list).

## Insertion and Deletion Algorithms

To insert or delete a node, we simply search and splice. A vector update is maintained so that when the search is complete (and we are ready to perform the splice), update[i] contains a pointer to the rightmost node of level i or higher that is to the left of the location of the insertion/deletion. If an insertion generates a node with a level greater than the previous maximum level of the list, we update the maximum level of the list and initialize the appropriate portions of the update vector. After each deletion, we check if we have deleted the maximum element of the list and if so, decrease the maximum level of the list.

## Choosing a Random Level

Initially, we discussed a probability distribution where half of the nodes that have level i pointers also have level i+1 pointers. To get away from magic constants, we say that a fraction p of the nodes with level i pointers also have level i+1 pointers. (for our original discussion, p = 1/2). Levels are generated randomly by an algorithm. Levels are generated without reference to the number of elements in the list

## Performance

In a simple linked list that consists of $n$ elements, to perform a search $n$ comparisons are required in the worst case. If a second pointer pointing two nodes ahead is added to every node, the number of comparisons goes down to $n/2 + 1$ in the worst case. Adding one more pointer to every fourth node and making them point to the fourth node ahead reduces the number of comparisons to $\lceil n/2 \rceil + 2$. If this strategy is continued so that every

node with $i$ pointers points to $2 * i - 1$ nodes ahead, O($logn$) performance is obtained and the number of pointers has only doubled ($n + n/2 + n/4 + n/8 + n/16 + \ldots. = 2n$).

The find, insert, and remove operations on ordinary binary search trees are efficient, O($logn$), when the input data is random; but less efficient, O($n$), when the input data is ordered. Skip List performance for these same operations and for any data set is about as good as that of randomly-built binary search trees - namely O($logn$).

## Comparing Skip Lists and Unrolled Linked Lists

In simple terms, Skip Lists are sorted linked lists with two differences:

- The nodes in an ordinary list have one next reference. The nodes in a Skip List have many *next* references (also called $forward$ references).
- The number of $forward$ references for a given node is determined probabilistically.

We speak of a Skip List node having levels, one level per forward reference. The number of levels in a node is called the *size* of the node. In an ordinary sorted list, insert, remove, and find operations require sequential traversal of the list. This results in O($n$) performance per operation. Skip Lists allow intermediate nodes in the list to be skipped during a traversal - resulting in an expected performance of O($logn$) per operation.

## Implementation

```
import random
import math
class Node(object):
    def __init__(self, data, level=0):
        self.data = data
        self.next = [None] * level

    def __str__(self):
        return "Node(%s,%s)" % (self.data, len(self.next))
    __repr__ = __str__

class SkipList(object):
    def __init__(self, max_level=8):
        self.max_level = max_level
        n = Node(None, max_level)
        self.head = n
        self.verbose = False

    def randomLevel(self, max_level):
        num = random.randint(1, 2**max_level - 1)
        lognum = math.log(num, 2)
        level = int(math.floor(lognum))
        return max_level - level

    def updateList(self, data):
        update = [None] * (self.max_level)
        n = self.head
        self._n_traverse = 0

        level = self.max_level - 1
        while level >= 0:
            if self.verbose and \
                n.next[level] != None and n.next[level].data >= data:
                print 'DROP down from level', level + 1
            while n.next[level] != None and n.next[level].data < data:
                self._n_traverse += 1
                if self.verbose:
                    print 'AT level', level, 'data', n.next[level].data
                n = n.next[level]
            update[level] = n
            level -= 1

        return update

    def find(self, data, update=None):
        if update is None:
            update = self.updateList(data)
        if len(update) > 0:
```

```
            candidate = update[0].next[0]
            if candidate != None and candidate.data == data:
                return candidate
        return None

    def insertNode(self, data, level=None):
        if level is None:
            level = self.randomLevel(self.max_level)
        node = Node(data, level)
        update = self.updateList(data)
        if self.find(data, update) == None:
            for i in range(level):
                node.next[i] = update[i].next[i]
                update[i].next[i] = node
def printLevel(sl, level):
    print 'level %d:' % level,
    node = sl.head.next[level]
    while node:
        print node.data, '=>',
        node = node.next[level]
    print 'END'
x = SkipList(4)
for i in range(0, 20, 2):
    x.insertNode(i)
printLevel(x, 0)
printLevel(x, 1)
printLevel(x, 2)
```

## 3.12 Linked Lists: Problems & Solutions

**Problem-1** Implement Stack using Linked List.

**Solution:** Refer to *Stacks* chapter.

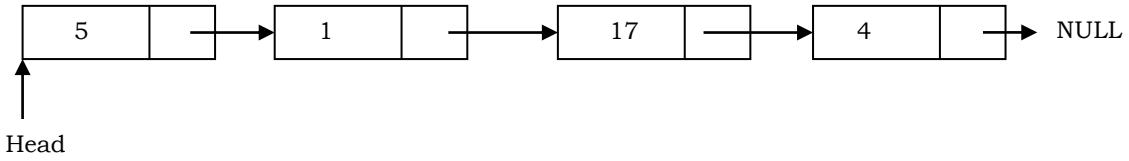
**Problem-2** Find $n^{th}$ node from the end of a Linked List.

**Solution: Brute-Force Method:** Start with the first node and count the number of nodes present after that node. If the number of nodes is $< n-1$ then return saying "fewer number of nodes in the list". If the number of nodes is $> n-1$ then go to next node. Continue this until the numbers of nodes after current node are $n-1$.

Time Complexity: O($n^2$), for scanning the remaining list (from current node) for each node.
Space Complexity: O(1).

**Problem-3** Can we improve the complexity of Problem-2?

**Solution: Yes**, using hash table. As an example consider the following list.




In this approach, create a hash table whose entries are < *position of node, node address* >. That means, key is the position of the node in the list and value is the address of that node.

| Position in List | Address of Node |
|---|---|
| 1 | Address of 5 node |
| 2 | Address of 1 node |
| 3 | Address of 17 node |
| 4 | Address of 4 node |

By the time we traverse the complete list (for creating the hash table), we can find the list length. Let us say the list length is $M$. To find $n^{th}$ from the end of linked list, we can convert this to $M-n+1^{th}$ from the beginning. Since we already know the length of the list, it is just a matter of returning $M-n+1^{th}$ key value from the hash table.

Time Complexity: Time for creating the hash table, $T(m) = O(m)$.
Space Complexity: Since we need to create a hash table of size $m$, $O(m)$.

**Problem-4** Can we use Problem-3 approach for solving Problem-2 without creating the hash table?

**Solution: Yes**. If we observe the Problem-3 solution, what we are actually doing is finding the size of the linked list. That means we are using the hash table to find the size of the linked list. We can find the length of the linked list just by starting at the head node and traversing the list. So, we can find the length of the list without creating the hash table. After finding the length, compute $M - n + 1$ and with one more scan we can get the $M - n + 1^{th}$ node from the beginning. This solution needs two scans: one for finding the length of the list and the other for finding $M - n + 1^{th}$ node from the beginning.

Time Complexity: Time for finding the length + Time for finding the $M\text{-}n + 1^{th}$ node from the beginning. Therefore, $T(n = O(n) + O(n) \approx O(n)$.

Space Complexity: O(1). Hence, no need to create the hash table.

**Problem-5** Can we solve Problem-2 in one scan?

**Solution: Yes**. **Efficient Approach**: Use two pointers $pNthNode$ and $pTemp$. Initially, both point to head node of the list. $pNthNode$ starts moving only after $pTemp$ has made $n$ moves. From there both move forward until $pTemp$ reaches the end of the list. As a result $pNthNode$ points to $n^{th}$ node from the end of the linked list.

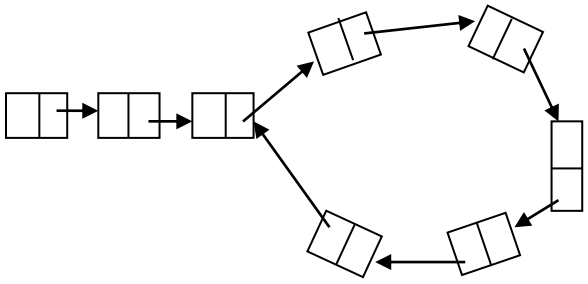
**Note:** At any point of time both move one node at a time.

```
def nthNodeFromEnd( self, n ):
    if 0 > n:
        return None
    # count k units from the self.head.
    temp = self.head
    count = 0
    while count < n and None != temp:
      temp = temp.next
      count += 1
    # if the LinkedList does not contain k elements, return None
    if count < n or None == temp:
      return None
    # keeping tab on the nth element from temp, slide temp until
    # temp equals self.tail. Then return the nth element.
    nth = self.head
    while None != temp.next:
      temp = temp.next
      nth = nth.next
    return nth
```

Time Complexity: $O(n)$. Space Complexity: O(1).

**Problem-6** Check whether the given linked list is either NULL-terminated or ends in a cycle (cyclic).

**Solution: Brute-Force Approach.** As an example, consider the following linked list which has a loop in it. The difference between this list and the regular list is that, in this list, there are two nodes whose next pointers are the same. In regular singly linked lists (without a loop) each node's next pointer is unique. That means the repetition of next pointers indicates the existence of a loop.



One simple and brute force way of solving this is, start with the first node and see whether there is any node whose next pointer is the current node's address. If there is a node with the same address then that indicates that some other node is pointing to the current node and we can say a loop exists.

Continue this process for all the nodes of the linked list.

**Does this method work?** As per the algorithm, we are checking for the next pointer addresses, but how do we find the end of the linked list (otherwise we will end up in an infinite loop)?

**Note:** If we start with a node in a loop, this method may work depending on the size of the loop.

**Problem-7** Can we use the hashing technique for solving Problem-6?

**Solution: Yes**. Using Hash Tables we can solve this problem.

**Algorithm:**

- Traverse the linked list nodes one by one.
- Check if the address of the node is available in the hash table or not.
- If it is already available in the hash table, that indicates that we are visiting the node that was already visited. This is possible only if the given linked list has a loop in it.
- If the address of the node is not available in the hash table, insert that node's address into the hash table.
- Continue this process until we reach the end of the linked list *or* we find the loop.

Time Complexity*:* O($n$) for scanning the linked list. Note that we are doing a scan of only the input.
Space Complexity*:* O($n$) for hash table.

**Problem-8** Can we solve Problem-6 using the sorting technique?

**Algorithm:**

- Traverse the linked list nodes one by one and take all the next pointer values into an array.
- Sort the array that has the next node pointers.
- If there is a loop in the linked list, definitely two next node pointers will be pointing to the same node.
- After sorting if there is a loop in the list, the nodes whose next pointers are the same will end up adjacent in the sorted list.
- If any such pair exists in the sorted list then we say the linked list has a loop in it.

Time Complexity*:* O($nlogn$) for sorting the next pointers array.
Space Complexity*:* O($n$) for the next pointers array.

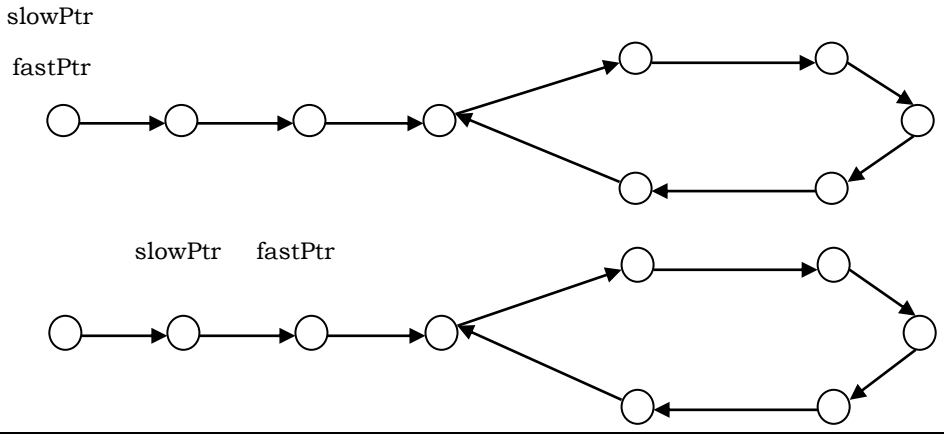
**Problem with the above algorithm:** The above algorithm works only if we can find the length of the list. But if the list has a loop then we may end up in an infinite loop. Due to this reason the algorithm fails.

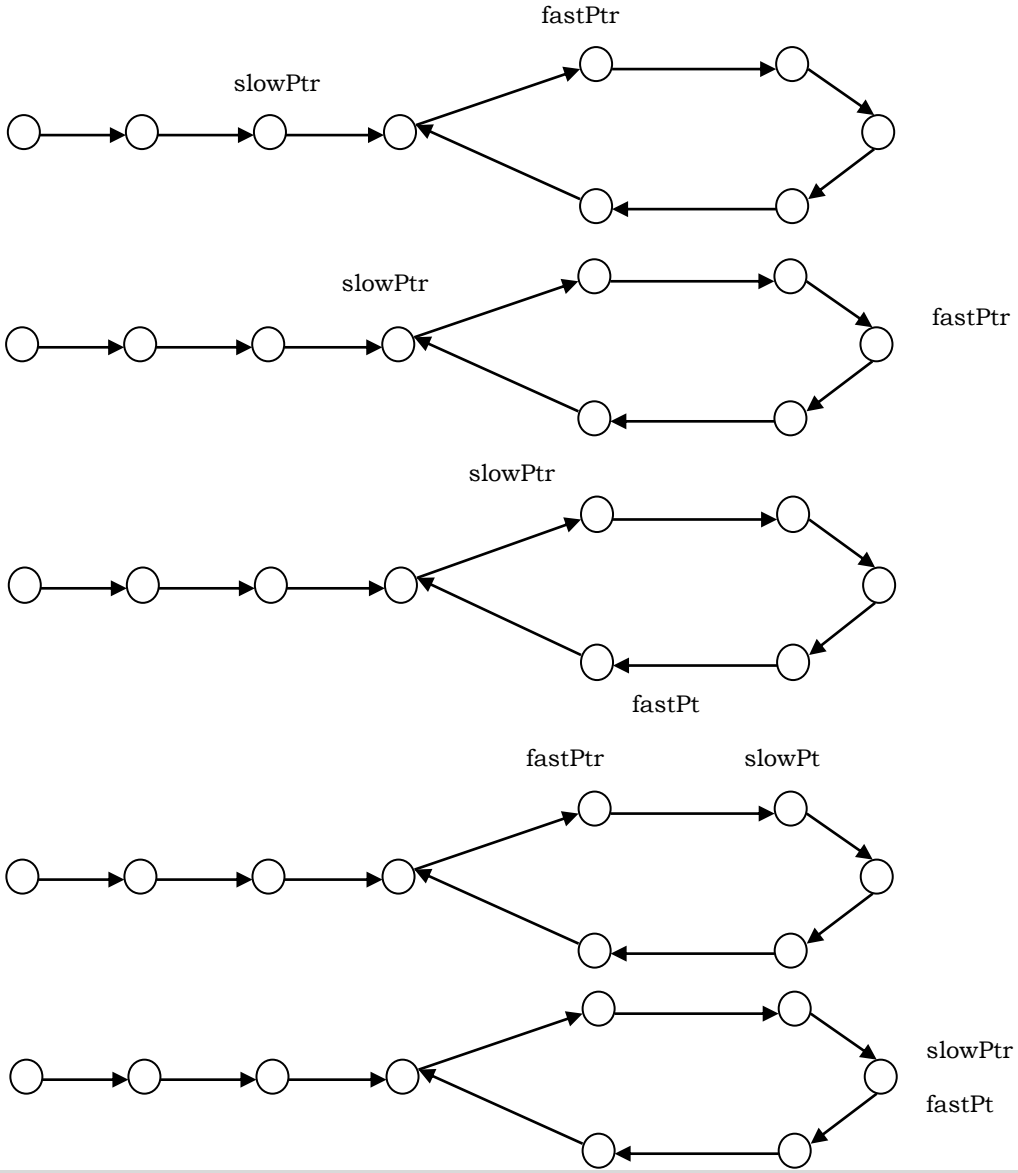
**Problem-9** Can we solve the Problem-6 in O($n$)?

**Solution: Yes**. **Efficient Approach (Memoryless Approach):** This problem was solved by *Floyd*. The solution is named the Floyd cycle finding algorithm. It uses *two* pointers moving at different speeds to walk the linked list. Once they enter the loop they are expected to meet, which denotes that there is a loop. This works because the only way a faster moving pointer would point to the same location as a slower moving pointer is if somehow the entire list or a part of it is circular. Think of a tortoise and a hare running on a track. The faster running hare will catch up with the tortoise if they are running in a loop.

As an example, consider the following example and trace out the Floyd algorithm. From the diagrams below we can see that after the final step they are meeting at some point in the loop which may not be the starting point of the loop.

**Note:** *slowPtr* (*tortoise*) moves one pointer at a time and *fastPtr* (*hare*) moves two pointers at a time.






```
def detectCycle(self):
        fastPtr = self.head
        slowPtr = self.head

        while (fastPtr and slowPtr):
            fastPtr = fastPtr.getNext()
            if (fastPtr == slowPtr):
                    return True

            if fastPtr == None:
                    return False

            fastPtr = fastPtr.getNext()
            if (fastPtr == slowPtr):
                    return True

            slowPtr = slowPtr.getNext()
```

Time Complexity: O($n$). Space Complexity: O(1).

**Problem-10** We are given a pointer to the first element of a linked list $L$. There are two possibilities for $L$, it either ends (snake) or its last element points back to one of the earlier elements in the list (snail). Give an algorithm that tests whether a given list $L$ is a snake or a snail.

**Solution:** It is the same as Problem-6.

**Problem-11** Check whether the given linked list is NULL-terminated or not. If there is a cycle find the start node of the loop.

**Solution:** The solution is an extension to the solution in Problem-9. After finding the loop in the linked list, we initialize the *slowPtr* to the head of the linked list. From that point onwards both *slowPtr* and *fastPtr* move only one node at a time. The point at which they meet is the start of the loop. Generally we use this method for removing the loops. Let $x$ and y be travelers such that y is walking twice as fast as $x$ (i.e. $y = 2x$). Further, let s be the place where $x$ and $y$ first started walking at the same time. Then when x and y meet again, the distance from $s$ to the start of the loop is the exact same distance from the present meeting place of $x$ and $y$ to the start of the loop.

```
def detectCycleStart( self ) :
    if None == self.head or None == self.head.next:
      return None

    # slow and fast both started at head after one step,
    # slow is at self.head.next and fast is at self.head.next.next
    slow = self.head.next
    fast = slow.next
    # each keep walking until they meet again.
    while slow != fast:
      slow = slow.next
      try:
        fast = fast.next.next
      except AttributeError:
        return None # no cycle if NoneType reached

    # from self.head to beginning of loop is same as from fast to beginning of loop
    slow = self.head
    while slow != fast:
      slow = slow.next
      fast = fast.next

    return slow # beginning of loop
```

Time Complexity: O($n$). Space Complexity: O(1).

**Problem-12** From the previous discussion and problems we understand that the meeting of tortoise and hare concludes the existence of the loop, but how does moving the tortoise to the beginning of the linked list while keeping the hare at the meeting place, followed by moving both one step at a time, make them meet at the starting point of the cycle?

**Solution:** This problem is at the heart of number theory. In the Floyd cycle finding algorithm, notice that the tortoise and the hare will meet when they are $n \times L$, where $L$ is the loop length. Furthermore, the tortoise is at the midpoint between the hare and the beginning of the sequence because of the way they move. Therefore the tortoise is $n \times L$ away from the beginning of the sequence as well.

If we move both one step at a time, from the position of the tortoise and from the start of the sequence, we know that they will meet as soon as both are in the loop, since they are $n \times L$, a multiple of the loop length, apart. One of them is already in the loop, so we just move the other one in single step until it enters the loop, keeping the other $n \times L$ away from it at all times.

**Problem-13** In Floyd cycle finding algorithm, does it work if we use steps 2 and 3 instead of 1 and 2?

**Solution: Yes**, but the complexity might be high. Trace out an example.

**Problem-14** Check whether the given linked list is NULL-terminated. If there is a cycle, find the length of the loop.

**Solution:** This solution is also an extension of the basic cycle detection problem. After finding the loop in the linked list, keep the *slowPtr* as it is. The *fastPtr* keeps on moving until it again comes back to *slowPtr*. While moving *fastPtr*, use a counter variable which increments at the rate of 1.

```
def findLoopLength( self ):
    if None == self.head or None == self.head.next:
      return 0
    # slow and fast both started at head after one step,
    # slow is at self.head.next and fast is at self.head.next.next
    slow = self.head.next
    fast = slow.next
```

```
    # each keep walking until they meet again.
    while slow != fast:
      slow = slow.next
      try:
        fast = fast.next.next
      except AttributeError:
        return 0 # no cycle if NoneType reached
    loopLength = 0
    slow = slow.next
    while slow != fast:
      slow = slow.next
      loopLength = loopLength + 1
    return loopLength
```

Time Complexity: O($n$). Space Complexity: O(1).

**Problem-15** Insert a node in a sorted linked list.

**Solution:** Traverse the list and find a position for the element and insert it.

```
def orderedInsert(self,item):
    current = self.head
    previous = None
    stop = False
    while current != None and not stop:
        if current.getData() > item:
            stop = True
        else:
            previous = current
            current = current.getNext()
    temp = Node(item)
    if previous == None:
        temp.setNext(self.head)
        self.head = temp
    else:
        temp.setNext(current)
        previous.setNext(temp)
```

Time Complexity: O($n$). Space Complexity: O(1).

**Problem-16** Reverse a singly linked list.

**Solution:** This algorithm reverses this singly linked list in place, in O($n$). The function uses three pointers to walk the list and reverse link direction between each pair of nodes.

```
# Iterative version
def reverseList(self):
        last = None
        current = self.head
        while(current is not None):
                nextNode = current.getNext()
                current.setNext(last)
                last = current
                current = nextNode
        self.head = last
```

Time Complexity: O($n$). Space Complexity: O(1).

**Recursive version**: We can find it easier to start from the bottom up, by asking and answering tiny questions (this is the approach in The Little Lisper):

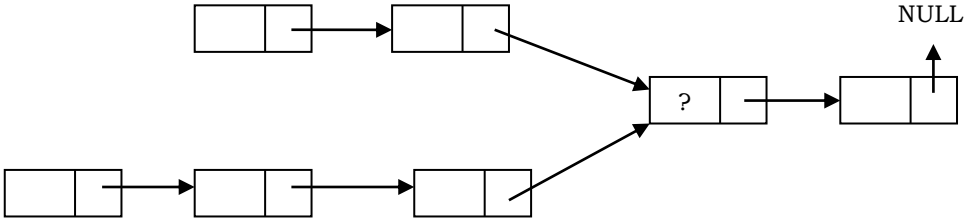
- What is the reverse of NULL (the empty list)? NULL.
- What is the reverse of a one element list? The element itself.
- What is the reverse of an $n$ element list? The reverse of the second element followed by the first element.

```
def reverseRecursive( self, n ) :
        if None != n:
```

```
        right = n.getNext()
        if self.head != n:
                n.setNext(self.head)
                self.head = n
        else:
                n.setNext(None)
        self.reverseRecursive( right )
```

Time Complexity: O($n$). Space Complexity: O($n$), for recursive stack.

**Problem-17** Suppose there are two singly linked lists both of which intersect at some point and become a single linked list. The head or start pointers of both the lists are known, but the intersecting node is not known. Also, the number of nodes in each of the lists before they intersect is unknown and may be different in each list. $List1$ may have $n$ nodes before it reaches the intersection point, and $List2$ might have $m$ nodes before it reaches the intersection point where $m$ and $n$ may be $m = n, m < n$ or $m > n$. Give an algorithm for finding the merging point.




**Solution: Brute-Force Approach:** One easy solution is to compare every node pointer in the first list with every other node pointer in the second list by which the matching node pointers will lead us to the intersecting node. But, the time complexity in this case will be O($mn$) which will be high.

Time Complexity: O($mn$). Space Complexity: O(1).

**Problem-18** Can we solve Problem-17 using the sorting technique?

**Solution: No.** Consider the following algorithm which is based on sorting and see why this algorithm fails.

**Algorithm:**

- Take first list node pointers and keep them in some array and sort them.
- Take second list node pointers and keep them in some array and sort them.
- After sorting, use two indexes: one for the first sorted array and the other for the second sorted array.
- Start comparing values at the indexes and increment the index according to whichever has the lower value (increment only if the values are not equal).
- At any point, if we are able to find two indexes whose values are the same, then that indicates that those two nodes are pointing to the same node and we return that node.

Time Complexity: Time for sorting lists + Time for scanning (for comparing)
= O($mlogm$) +O($nlogn$) +O($m + n$) We need to consider the one that gives the maximum value.

Space Complexity: O(1).

**Any problem with the above algorithm? Yes**. In the algorithm, we are storing all the node pointers of both the lists and sorting. But we are forgetting the fact that there can be many repeated elements. This is because after the merging point, all node pointers are the same for both the lists. The algorithm works fine only in one case and it is when both lists have the ending node at their merge point.

**Problem-19** Can we solve Problem-17 using hash tables?

**Solution: Yes.**

**Algorithm:**

- Select a list which has less number of nodes (If we do not know the lengths beforehand then select one list randomly).
- Now, traverse the other list and for each node pointer of this list check whether the same node pointer exists in the hash table.
- If there is a merge point for the given lists then we will definitely encounter the node pointer in the hash table.

```
def findIntersectingNode( self, list1, list2 ):
    intersect = {}
    t = list1
```

```
        while None != t:
                intersect[t] = True
                t = t.getNext()
        # first duplicate is intersection
        t = list2
        while None != t:
                if None != intersect.get( t ):
                        return t
                t = t.getNext()
        return None
```

Time Complexity: Time for creating the hash table + Time for scanning the second list = O($m$) + O($n$) (or O($n$) + O($m$), depending on which list we select for creating the hash table. But in both cases the time complexity is the same.
Space Complexity: O($n$) or O($m$).

**Problem-20** Can we use stacks for solving Problem-17?

**Solution: Yes.**

**Algorithm:**

- Create two stacks: one for the first list and one for the second list.
- Traverse the first list and push all the node addresses onto the first stack.
- Traverse the second list and push all the node addresses onto the second stack.
- Now both stacks contain the node address of the corresponding lists.
- Now compare the top node address of both stacks.
- If they are the same, take the top elements from both the stacks and keep them in some temporary variable (since both node addresses are node, it is enough if we use one temporary variable).
- Continue this process until the top node addresses of the stacks are not the same.
- This point is the one where the lists merge into a single list.
- Return the value of the temporary variable.

Time Complexity: O($m + n$), for scanning both the lists.
Space Complexity: O($m + n$), for creating two stacks for both the lists.

**Problem-21** Is there any other way of solving Problem-17?

**Solution**: **Yes**. Using "finding the first repeating number" approach in an array (for algorithm refer *Searching* chapter).

**Algorithm:**

- Create an array $A$ and keep all the next pointers of both the lists in the array.
- In the array find the first repeating element [Refer to *Searching* chapter for algorithm].
- The first repeating number indicates the merging point of both the lists.

Time Complexity: O($m + n$). Space Complexity: O($m + n$).

**Problem-22** Can we still think of finding an alternative solution for Problem-17?

**Solution: Yes**. By combining sorting and search techniques we can reduce the complexity.

**Algorithm:**

- Create an array $A$ and keep all the next pointers of the first list in the array.
- Sort these array elements.
- Then, for each of the second list elements, search in the sorted array (let us assume that we are using binary search which gives O($logn$)).
- Since we are scanning the second list one by one, the first repeating element that appears in the array is nothing but the merging point.

Time Complexity: Time for sorting + Time for searching = O($Max(mlogm, nlogn)$).
Space Complexity: O($Max(m, n)$).

**Problem-23** Can we improve the complexity for Problem-17?

**Solution: Yes.**

**Efficient Approach:**

- Find lengths (L1 and L2) of both lists -- O($n$) + O($m$) = O($max(m, n)$).
- Take the difference $d$ of the lengths -- O(1).
- Make $d$ steps in longer list -- O($d$).

- Step in both lists in parallel until links to next node match -- O($min(m,n)$).
- Total time complexity = O($max(m,n)$).
- Space Complexity = O(1).

```
def getIntersectionNode(self, list1, list2):
    currentList1,currentList2 = list1,list2
    list1Len,list2Len = 0,0
    while currentList1 is not None:
        list1Len += 1
        currentList1 = currentList1.next
    while currentList2 is not None:
        list2Len += 1
        currentList2 = currentList2.next
    currentList1,currentList2 = list1,list2
    if list1Len > list2Len:
        for i in range(list1Len-list2Len):
            currentList1 = currentList1.next
    elif list2Len > list1Len:
        for i in range(list2Len-list1Len):
            currentList2 = currentList2.next
    while currentList2 != currentList1:
        currentList2 = currentList2.next
        currentList1 = currentList1.next
    return currentList1
```

**Problem-24** How will you find the middle of the linked list?

**Solution: Brute-Force Approach:** For each of the node counts how many nodes are there in the list and see whether it is the middle.

Time Complexity: O($n^2$). Space Complexity: O(1).

**Problem-25** Can we improve the complexity of Problem-24?

**Solution: Yes.**

**Algorithm:**

- Traverse the list and find the length of the list.
- After finding the length, again scan the list and locate $n/2$ node from the beginning.

Time Complexity: Time for finding the length of the list + Time for locating middle node = O($n$) + O($n$) ≈ O($n$).
Space Complexity: O(1).

**Problem-26** Can we use the hash table for solving Problem-24?

**Solution: Yes.** The reasoning is the same as that of Problem-3.

Time Complexity: Time for creating the hash table. Therefore, $T(n)$ = O($n$).
Space Complexity: O($n$). Since we need to create a hash table of size $n$.

**Problem-27** Can we solve Problem-24 just in one scan?

**Solution: Efficient Approach:** Use two pointers. Move one pointer at twice the speed of the second. When the first pointer reaches the end of the list, the second pointer will be pointing to the middle node.

**Note:** If the list has an even number of nodes, the middle node will be of $\lfloor n/2 \rfloor$.

```
def findMiddleNode( self) :
    fastPtr = self.head
    slowPtr = self.head
    while (fastPtr != None):
        fastPtr = fastPtr.getNext()
        if (fastPtr == None):
            return slowPtr
        fastPtr = fastPtr.getNext()
        slowPtr = slowPtr.getNext()
    return slowPtr
```

Time Complexity: O($n$). Space Complexity: O(1).

**Problem-28** How will you display a linked list from the end?

**Solution:** Traverse recursively till the end of the linked list. While coming back, start printing the elements. It is natural to express many list operations using recursive methods. For example, the following is a recursive algorithm for printing a list backwards:

1. Separate the list into two pieces: the first node (called the head); and the rest (called the tail).
2. Print the tail backward.
3. Print the head.

Of course, Step 2, the recursive call, assumes that we have a way of printing a list backward.

```
def printListFromEnd( self, list) :
        if list == None:
            return
        head = list
        tail = list.getNext()
        self.printListFromEnd(tail)
        print head.getData(),
if __name__ == "__main__":
    linkedlst = LinkedList()
    linkedlst.insertAtEnd(1)
    linkedlst.insertAtEnd(2)
    linkedlst.insertAtEnd(3)
    linkedlst.insertAtEnd(4)
    linkedlst.printList()
    linkedlst.printListFromEnd(linkedlst.head)
```

Time Complexity: O($n$). Space Complexity: O($n$)→ for Stack.

**Problem-29** Check whether the given Linked List length is even or odd?

**Solution:** Use a $2x$ pointer. Take a pointer that moves at $2x$ [two nodes at a time]. At the end, if the length is even, then the pointer will be NULL; otherwise it will point to the last node.

```
def isLinkedListLengthEven(self):
  current = self.head
   while current != None and current.getNext()!= None:
     current = current.getNext().getNext()
     if current == None:
            return 1
     return 0
```

Time Complexity: O($\lfloor n/2 \rfloor$) ≈O($n$). Space Complexity: O(1).

**Problem-30** If the head of a linked list is pointing to $kth$ element, then how will you get the elements before $kth$ element?

**Solution:** Use Memory Efficient Linked Lists [XOR Linked Lists].

**Problem-31** Given two sorted Linked Lists, how to merge them into the third list in sorted order?

**Solution:** Assume the sizes of lists are $m$ and $n$.

```
def mergeTwoLists(self, list1, list2):
    temp = Node()
    pointer = temp
    while list1 !=None and list2 !=None:
        if list1.getData()<list2.getData():
            pointer.setNext(list1)
            list1 = list1.getNext()
        else:
            pointer.setNext(list2)
            list2 = list2.getNext()
        pointer = pointer.getNext()
    if list1 == None:
        pointer.setNext(list2)
    else:
        pointer.setNext(list1)
    return temp.getNext()
```

Time Complexity: O($n + m$), where $n$ and $m$ are lengths of two lists.

**Problem-32** Reverse the linked list in pairs. If you have a linked list that holds $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow X$, then after the function has been called the linked list would hold $2 \rightarrow 1 \rightarrow 4 \rightarrow 3 \rightarrow X$.

**Solution:**

```
def reverseInPairs( self ) :
  temp = self.head
  while None != temp and None != temp.getNext():
    self.swapData( temp, temp.getNext() )
    temp = temp.getNext().getNext()
def swapData( self, a, b ):
  tmp = a.getData()
  a.setData(b.getData())
  b.setData(tmp)
```

Time Complexity – O($n$). Space Complexity: O(1).

**Problem-33** Given a binary tree convert it to doubly linked list.

**Solution:** Refer *Trees* chapter.

**Problem-34** How do we sort the Linked Lists?

**Solution:** Refer *Sorting* chapter.

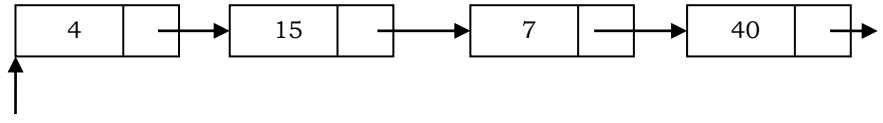
**Problem-35** Split a Circular Linked List into two equal parts. If the number of nodes in the list are odd then make first list one node extra than second list.

**Solution:**

**Algorithm:**

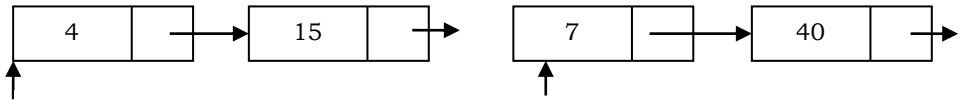
- Store the mid and last pointers of the linked list using Floyd cycle finding algorithm.
- Set head pointers of the two linked lists.

As an example, consider the following linked list.




Head

After the split, the above list will look like:




Head middle

```
def splitList(head):
   fast = head
   slow = head
   while fast != None and fast.getNext() != None:
       slow = slow.getNext()
       fast = fast.getNext()
      fast = fast.getNext()
   middle = slow.getNext()
   slow.setNext(None)
   return head, middle
```

Time Complexity: O($n$). Space Complexity: O(1).

**Problem-36** If we want to concatenate two linked lists, which of the following gives O(1) complexity?

1) Singly linked lists
2) Doubly linked lists
3) Circular doubly linked lists

**Solution:** Circular Doubly Linked Lists. This is because for singly and doubly linked lists, we need to traverse the first list till the end and append the second list. But in the case of circular doubly linked lists we don't have to traverse the lists.

**Problem-37** How will you check if the linked list is palindrome or not?

**Solution:**

**Algorithm:**

1. Get the middle of the linked list.
2. Reverse the second half of the linked list.
3. Compare the first half and second half.
4. Construct the original linked list by reversing the second half again and attaching it back to the first half.

Time Complexity: O($n$). Space Complexity: O(1).

**Problem-38** For a given $K$ value ($K > 0$) reverse blocks of $K$ nodes in a list.
**Example**: Input: 1 2 3 4 5 6 7 8 9 10. Output for different $K$ values:
For $K = 2$: 2 1 4 3 6 5 8 7 10 9 For $K = 3$: 3 2 1 6 5 4 9 8 7 10 For $K = 4$: 4 3 2 1 8 7 6 5 9 10

**Solution:**

**Algorithm:** This is an extension of swapping nodes in a linked list.

1) Check if remaining list has $K$ nodes.
    a. If yes get the pointer of $K + 1^{th}$ node.
    b. Else return.
2) Reverse first $K$ nodes.
3) Set next of last node (after reversal) to $K + 1^{th}$ node.
4) Move to $K + 1^{th}$ node.
5) Go to step 1.
6) $K - 1^{th}$ node of first $K$ nodes becomes the new head if available. Otherwise, we can return the head.

```
def reverseKBlock(self, head, k):
    temp = Node(0);
    temp.setNext(head)
    previous = temp
    while True:
        begin = previous.getNext()
        end = previous
        for i in range(0,k):
            end = end.getNext()
            if end == None:
                return temp.getNext()
        nextBlock = end.getNext()
        self.reverseList(begin,end)
        previous.setNext(end)
        begin.setNext(nextBlock)
        previous = begin
def reverseList(self, start, end):
        alreadyReversed = start
        actual = start
        nextNode = start.getNext()
        while actual != end:
            actual = nextNode
            nextNode = nextNode.getNext()
            actual.setNext(alreadyReversed)
            alreadyReversed = actual
```

**Problem-39** Is it possible to get O(1) access time for Linked Lists?

**Solution: Yes**. Create a linked list and at the same time keep it in a hash table. For $n$ elements we have to keep all the elements in a hash table which gives a preprocessing time of O($n$). To read any element we require only constant time O(1) and to read $n$ elements we require $n * 1$ unit of time = $n$ units. Hence by using amortized analysis we can say that element access can be performed within O(1) time.

Time Complexity – O(1) [Amortized]. Space Complexity - O($n$) for Hash Table.

**Problem-40** **Josephus Circle:** Flavius Josephus was a famous Jewish historian of the first century, at the time of the destruction of the Second Temple. According to legend, during the Jewish-Roman war he was trapped in a cave with a group of forty soldiers surrounded by Romans. Preferring death to capture, the Jews decided to form a circle and, proceeding around it, to kill every third person remaining until no one was left. Josephus found the safe spot in the circle and thus stayed alive. Write a function josephus(n,m) that returns a list of $n$ people, numbered from 0 to $n-1$, in the order in which they are executed, every $m^{th}$person in turn, with the sole survivor as the last person in the list. That mean, find which person will be the last one remaining (with rank 1).

**Solution:** Assume the input is a circular linked list with $n$ nodes and each node has a number (range 1 to $n$) associated with it. The head node has number 1 as data.

```
def getJosephusPosition(n, m):
    class Node:
        def __init__(self, data = None, next = None):
            self.setData(data)
            self.setNext(next)
        #method for setting the data field of the node
        def setData(self,data):
            self.data = data
       #method for getting the data field of the node
        def getData(self):
            return self.data
        #method for setting the next field of the node
        def setNext(self,next):
            self.next = next
        #method for getting the next field of the node
        def getNext(self):
            return self.next
        #returns true if the node points to another node
        def hasNext(self):
            return self.next != None
    answer = []
    # initialize circular linked list
    head = Node(0)
    prev = head
    for n in range(1, n):
        currentNode = Node(n)
        prev.setNext(currentNode)
        prev = currentNode
        prev.setNext(head) # set the last node to point to the front (circular list)
    # extract items from linked list in proper order
    currentNode = head
    counter = 0
    while currentNode.getNext() != currentNode:
        counter += 1
        if counter == m:
            counter = 0
            prev.setNext(currentNode.next)
            answer.append(currentNode.getData())
        else:
            prev = currentNode
        currentNode = currentNode.getNext()
    answer.append(currentNode.getData())
    return answer
print str(getJosephusPosition(6, 3))
```

**Problem-41** Given a linked list consists of data, a next pointer and also a random pointer which points to a random node of the list. Give an algorithm for cloning the list.

**Solution:** We can use a hash table to associate newly created nodes with the instances of node in the given list.

**Algorithm:**

- Scan the original list and for each node $X$, create a new node $Y$ with data of $X$, then store the pair $(X, Y)$ in hash table using $X$ as a key. Note that during this scan set $Y \rightarrow next$ and $Y \rightarrow random$ to $NULL$ and we will fix them in the next scan.
- Now for each node $X$ in the original list we have a copy $Y$ stored in our hash table. We scan the original list again and set the pointers building the new list.

```
class Node:
    def __init__(self, data):
            self.setData(data)
            self.setNext(None)
            self.setRand(None)
    #method for setting the data field of the node
    def setData(self,data):
            self.data = data
    #method for getting the data field of the node
    def getData(self):
            return self.data
    #method for setting the next field of the node
    def setNext(self,next):
            self.next = next
    #method for setting the next field of the node
    def setRand(self,rand):
            self.rand = rand
    #method for getting the next field of the node
    def getRand(self):
            return self.rand
    #method for getting the next field of the node
    def getNext(self):
            return self.next
    #returns true if the node points to another node
    def hasNext(self):
       return self.next != None
    def cloneLinkedList(old):
       if not old:
            return
       old_copy = old
       root = Node(old.getData())
       prev = root
       temp = None
       old = old.getNext()
       mapping = {}
       while old:
            temp = Node(old.getData())
            mapping[old] = temp
            prev.setNext(temp)
            prev = temp
            old = old.getNext()
       old = old_copy
       temp = root
       while old:
            temp.setRand(mapping[old.rand])
            temp = temp.getNext()
            old = old.getNext()
       return root
```

Time Complexity: O($n$). Space Complexity: O($n$).

**Problem-42** Can we solve Problem-41 without any extra space?

**Solution: Yes**.

```
# Definition for singly-linked list with a random pointer.
class RandomListNode:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.random = None

class Solution:
    # @param head, a RandomListNode
    # @return a RandomListNode
    def copyRandomList(self, head):
        if None == head:
            return None
        save_list = [ ]
        p1 = head
        while None != p1:
            save_list.append(p1)
            p1 = p1.next

        new_head = RandomListNode(-1)
        new_head.next = head
        first = new_head
        second = head
        copyHead = RandomListNode(-1)
        copyFirst = copyHead
        copySecond = None

        while None != first:
            copySecond = RandomListNode(second.data) if None != second else None
            copyFirst.next = copySecond
            copyFirst = copyFirst.next
            first = first.next
            if None != second:
                second = second.next

        p1 = head
        p1_tail = head.next
        p2 = copyHead.next
        while None != p1:
            p1_tail = p1.next
            p1.next = p2
            p2.random = p1
            p1 = p1_tail
            p2 = p2.next
        p2 = copyHead.next
        while None != p2:
            p2.random = p2.random.random.next if None != p2.random.random else None
            p2 = p2.next
        len_save_list = len(save_list)
        for i in range(0, len_save_list - 1):
            save_list[i].next = save_list[i + 1]
        save_list[len_save_list - 1].next = None
        return copyHead.next
```

Time Complexity: $O(3n) \approx O(n)$. Space Complexity: $O(1)$.

**Problem-43** Given a linked list with even and odd numbers, create an algorithm for making changes to the list in such a way that all even numbers appear at the beginning.

**Solution**: To solve this problem, we can use the splitting logic. While traversing the list, split the linked list into two: one contains all even nodes and the other contains all odd nodes. Now, to get the final list, we can simply append the odd node linked list after the even node linked list.

To split the linked list, traverse the original linked list and move all odd nodes to a separate linked list of all odd nodes. At the end of the loop, the original list will have all the even nodes and the odd node list will have all the odd nodes. To keep the ordering of all nodes the same, we must insert all the odd nodes at the end of the odd node list.

Time Complexity: O($n$). Space Complexity: O(1).

**Problem-44** In a linked list with $n$ nodes, the time taken to insert an element after an element pointed by some pointer is

(A) O(1) (B) O($logn$) (C) O($n$) (D) O($n1ogn$)

**Solution**: A.

**Problem-45** **Find modular node:** Given a singly linked list, write a function to find the last element from the beginning whose $n\%k == 0$, where $n$ is the number of elements in the list and $k$ is an integer constant. For example, if $n = 19$ and $k = 3$ then we should return $18^{th}$node.

**Solution:** For this problem the value of $n$ is not known in advance.

```
def modularNodeFromBegin(self, k):
        currentNode = self.head
        modularNode =  None
        i = 1
        if k <= 0:
                return None;
        while currentNode != None:
                if i%k == 0:
                        modularNode = currentNode
                i = i + 1
                currentNode = currentNode.getNext()
        print ( modularNode.getData())
```

Time Complexity: O($n$). Space Complexity: O(1).

**Problem-46** **Find modular node from the end:** Given a singly linked list, write a function to find the first from the end whose $n\%k == 0$, where $n$ is the number of elements in the list and $k$ is an integer constant. If $n = 19$ and $k = 3$ then we should return $16^{th}$ node.

**Solution:** For this problem the value of $n$ is not known in advance and it is the same as finding the $k^{th}$element from the end of the the linked list.

```
def modularNodeFromEnd(self, k):
        currentNode = self.head
        modularNode = self.head
        i = 0
        if k <= 0:
                return None;
        while i < k and currentNode != None:
                i = i + 1
                currentNode = currentNode.getNext()
        if currentNode == None:
                return
        while currentNode != None:
                modularNode = modularNode.getNext()
                currentNode = currentNode.getNext()
        print (modularNode.getData())
```

Time Complexity: O($n$). Space Complexity: O(1).

**Problem-47** **Find fractional node:** Given a singly linked list, write a function to find the $\frac{n}{k}th$ element, where $n$ is the number of elements in the list.

**Solution:** For this problem the value of $n$ is not known in advance.

```
def fractionalNode(self, k):
        fractionalNode = None
        currentNode = self.head
        i = 0
        if k <= 0:
                return None;
        while currentNode != None:
                if i%k == 0:
```

```
                    if fractionalNode == None:
                        fractionalNode = self.head
                    else:
                        fractionalNode = fractionalNode.getNext()
                i = i + 1
                currentNode = currentNode.getNext()
        print (fractionalNode.getData())
```

Time Complexity: O($n$). Space Complexity: O(1).

**Problem-48** **Find $\sqrt{n}^{th}$ node:** Given a singly linked list, write a function to find the $\sqrt{n}^{th}$ element, where $n$ is the number of elements in the list. Assume the value of $n$ is not known in advance.

**Solution:** For this problem the value of $n$ is not known in advance.

```
def sqrtNthNodes(self):
    sqrtNode = None
    currentNode = self.head
    i = j = 1
    while currentNode != None:
        if i == j * j:
            if sqrtNode == None:
                sqrtNode = self.head
            else:
                sqrtNode = sqrtNode.getNext()
            j = j + 1
        i = i + 1
        currentNode = currentNode.getNext()
    print (sqrtNode.getData())
```

Time Complexity: O($n$). Space Complexity: O(1).

**Problem-49** Given two lists List1 = $\{A_1, A_2, \ldots, A_n\}$ and List2 = $\{B_1, B_2, \ldots, B_m\}$ with data (both lists) in ascending order. Merge them into the third list in ascending order so that the merged list will be:

$\{A_1, B_1, A_2, B_2 \ldots\ldots A_m, B_m, A_{m+1} \ldots A_n\}$ if $n >= m$
$\{A_1, B_1, A_2, B_2 \ldots\ldots A_n, B_n, B_{n+1} \ldots B_m\}$ if $m >= n$

**Solution:**

```
def mergeTwoSortedLists(self, list1, list2):
    temp = Node(0)
    pointer = temp
    while list1 !=None and list2 !=None:
        if list1.getData()<list2.getData():
            pointer.setNext(list1)
            list1 = list1.getNext()
        else:
            pointer.setNext(list2)
            list2 = list2.getNext()
        pointer = pointer.getNext()
    if list1 == None:
        pointer.setNext(list2)
    else:
        pointer.setNext(list1)
    return temp.getNext()
```

Time Complexity: The *while* loop takes O($min(n,m)$) time as it will run for $min(n,m)$ times. The other steps run in O(1). Therefore the total time complexity is O($min(n,m)$).
Space Complexity: O(1).

**Problem-50** Median in an infinite series of integers

**Solution:** Median is the middle number in a sorted list of numbers (if we have an odd number of elements). If we have an even number of elements, the median is the average of two middle numbers in a sorted list of numbers.

We can solve this problem with linked lists (with both sorted and unsorted linked lists).

*First*, let us try with an *unsorted* linked list. In an unsorted linked list, we can insert the element either at the head or at the tail. The disadvantage with this approach is that finding the median takes O($n$). Also, the insertion operation takes O(1).

Now, let us try with a *sorted* linked list. We can find the median in O(1) time if we keep track of the middle elements. Insertion to a particular location is also O(1) in any linked list. But, finding the right location to insert is not O($logn$) as in a sorted array, it is instead O($n$) because we can't perform binary search in a linked list even if it is sorted.

So, using a sorted linked list isn't worth the effort as insertion is O($n$) and finding median is O(1), the same as the sorted array. In the sorted array the insertion is linear due to shifting, but here it's linear because we can't do a binary search in a linked list.

**Note**: For an efficient algorithm refer to the *Priority Queues and Heaps* chapter.

**Problem-51** Given a linked list, how do you modify it such that all the even numbers appear before all the odd numbers in the modified linked list?

**Solution**:

```
def exchangeEvenOddList(head):
    # initializing the odd and even list headers
    oddList = evenList =None
    # creating tail variables for both the list
    oddListEnd = evenListEnd = None
    itr=head
    if( head == None ):
        return
    else:
        while( itr != None ):
            if( itr.data % 2 == 0 ):
                if( evenList == NULL ):
                    # first even node
                    evenList = evenListEnd = itr
                else:
                    # inserting the node at the end of linked list
                    evenListEnd.next = itr
                    evenListEnd = itr
            else:
                if( oddList == NULL ):
                    # first odd node
                    oddList = oddListEnd = itr
                else:
                    # inserting the node at the end of linked list
                    oddListEnd.next = itr
                    oddListEnd = itr
        itr = itr.next
    evenListEnd.next = oddList
    return head
```

Time Complexity: O($n$). Space Complexity: O(1).

**Problem-52** Given two linked lists, each list node with one integer digit, add these two linked lists. The result should be stored in the third linked list. Also note that the head node contains the most significant digit of the number.

**Solution**: Since the integer addition starts from the least significant digit, we first need to visit the last node of both lists and add them up, create a new node to store the result, take care of the carry if any, and link the resulting node to the node which will be added to the second least significant node and continue.

First of all, we need to take into account the difference in the number of digits in the two numbers. So before starting recursion, we need to do some calculation and move the longer list pointer to the appropriate place so that we need the last node of both lists at the same time. The other thing we need to take care of is *carry*. If two digits add up to more than 10, we need to forward the *carry* to the next node and add it. If the most significant digit addition results in a *carry*, we need to create an extra node to store the carry.

The function below is actually a wrapper function which does all the housekeeping like calculating lengths of lists, calling recursive implementation, creating an extra node for the *carry* in the most significant digit, and adding any remaining nodes left in the longer list.

```
class AddingListNumbers:
    def addTwoNumbers(self, list1, list2):
        if list1 == None:
            return list2
        if list2 == None:
            return list1

        len1 = len2 = 0
        head = list1
        while head != None:
            len1 += 1
            head = head.next

        head = list2
        while head != None:
            len2 += 1
            head = head.next

        if len1 >= len2:
            longer = list1
            shorter = list2
        else:
            longer = list2;
            shorter = list1

        sum = None
        carry = 0

        while shorter != None:
            value = longer.data + shorter.data + carry
            carry = value / 10
            value -= carry * 10

            if sum == None:
                sum = Node(value)
                result = sum
            else:
                sum.next = Node(value)
                sum = sum.next

            longer = longer.next
            shorter = shorter.next

        while longer != None:
            value = longer.data + carry
            carry = value / 10
            value -= carry * 10

            sum.next = Node(value)
            sum = sum.next

            longer = longer.next

        if carry != 0:
            sum.next = Node(carry)
        return result
```

Time Complexity: O($max(List1\ length, List2\ length)$).
Space Complexity: O($min(List1\ length, List2\ length)$) for recursive stack.

**Note:**It can also be solved using stacks.

**Problem-53** Write code for finding the sum of all data values from linked list with recursion.

**Solution**: One of the basic operations we perform on linked lists (as we do with lists) is to iterate over them, processing alst their values. The following function computes the sum of the values in a linked list.

```
def linkedListSum(lst):
    sum = 0
    while lst != None:
```

```
        sum += lst.
        lst = lst.getNext()
    return sum
```

Lots of code that traverses (iterates over) linked lists looks similar. In class we will go over (hand simulate) how this code processes the linked list above, with the call *linkedListSum*(x) and see exactly how it is that we visit each node in the linked list and stop processing it at the end.

We can also define linked lists recursively and use such a definition to help us write functions that recursively process linked lists.

1) None is the smallest linked list: it contains no nodes
2) A list node whose next refers to a linked list is also linked list

So None is a linked list (of 0 values); a list node whose next is *None* is a linked list (of 1 value); a list node whose next is a list node whose next is *None* is a linked list (of 2 values); etc.

So, we can recursively process a linked list by processing its first node and then recursively processing the (one smaller) linked list they refer to; recursion ends at None (which is the base case: the smallest linked list). We can recursively compute the sum of linked list by

```
def linkedListSum(self, lst):
    if lst == None:
        return 0
    else:
        return lst.getData() + linkedListSum(lst.getNext())
```

An even simpler traversal of linked lists computes their length. Here are the iterative and recursive methods.

```
def listLength(lst):
    count = 0
    while lst != None:
        count += 1
        lst = lst.getNext()
    return count

def listLengthRecursive(lst):
    if lst == None:
        return 0
    else:
        return 1 + listLengthRecursive(lst.getNext())
```

These are simpler than the *linkedListSum* function: rather than adding the value of each list node, these add 1 to a count for each list node, ultimately computing the number of list nodes in the entire linked list: its length.

**Problem-54** Given a sorted linked list, write a program to remove duplicates from it.

**Solution**: Skip the repeated adjacent elements.

```
def deleteLinkedListDuplicates(self):
        current = self.head;
        while current != None and current.next !=None:
            if current.getData() == current.getNext().getData():
                    current.setNext(current.getNext().getNext())
            else:
                    current = current.getNext()
        return head
```

Time Complexity: O($n$). Space Complexity: O(1).

**Problem-55** Given a list, List1 = $\{A_1, A_2, \ldots A_{n-1}, A_n\}$ with data, reorder it to $\{A_1, A_n, A_2, A_{n-1}\ldots\ldots\}$ without using any extra space.

**Solution:** Split the list, reverse the latter half and merge.

```
# Definition for singly-linked list.
class Node:
        def __init__(self, x):
                self.data = x
                self.next = None
class reorderLists:
    def reverse(self,head):
        dummy = prev = Node(0)
        while head:
```

```
            next = head.next
            head.next = prev.next
            prev.next =head
            head = next
        return dummy.next
    def getMiddleNode(self,head):
        slow = fast = head
        while fast.next and fast.next.next:
            fast = fast.next.next
            slow = slow.next
        head = slow.next
        slow.next = None
        return head
def reorderList(self, head):
    if not head or not head.next:
        return head
    head2 = self.getMiddleNode(head)
    head2 = self.reverse(head2)
    p = head
    q= head2
    while q:
        qnext = q.next # store the next node since q will be moved
        q.next = p.next
        p.next = q
        p = q.next
        q = qnext
    return head
```

Time Complexity: O(*n*). Space Complexity: O(1).

**Problem-56** Which sorting algorithm is easily adaptable to singly linked lists?

**Solution:** Simple Insertion sort is easily adabtable to singly linked lists. To insert an element, the linked list is traversed until the proper position is found, or until the end of the list is reached. It is inserted into the list by merely adjusting the pointers without shifting any elements, unlike in the array. This reduces the time required for insertion but not the time required for searching for the proper position.