Linked lists

Unit:2

Data Structures

Course Details
(B Tech 3rd Sem)

Sanchi Kaushik
Assistant Professor
Computer Science and Engineering(AIML)

Syllabus

UNIT-2

Linked lists -Self-referential structure, Singly Linked List, Doubly Linked List, Circular Linked List.

Operations on a Linked List: Insertion, Deletion, Traversal, Reversal, Searching Polynomial Representation and Addition Subtraction & Multiplications of Polynomials.

NIET Greater Noida
GET FUTURE READY

Contents

- Linked lists
- Singly Linked List, Doubly Linked List, Circular Linked List
- Operations on a Linked List: Insertion, Deletion, Traversal, Reversal, Searching
- Polynomial Representation and Addition Subtraction & Multiplications of Polynomials.

Objective of Unit

Objective of the course is to make students able to:

1. Learn the basic types for data structure, implementation and application.
2. Know the strength and weakness of different data structures.

3. Use the appropriate data structure in context of solution of given problem.

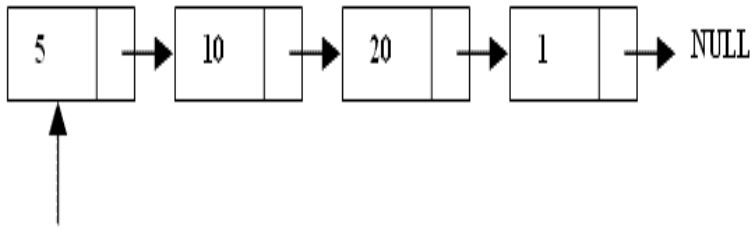
4. Develop programming skills which require to solve given problem.

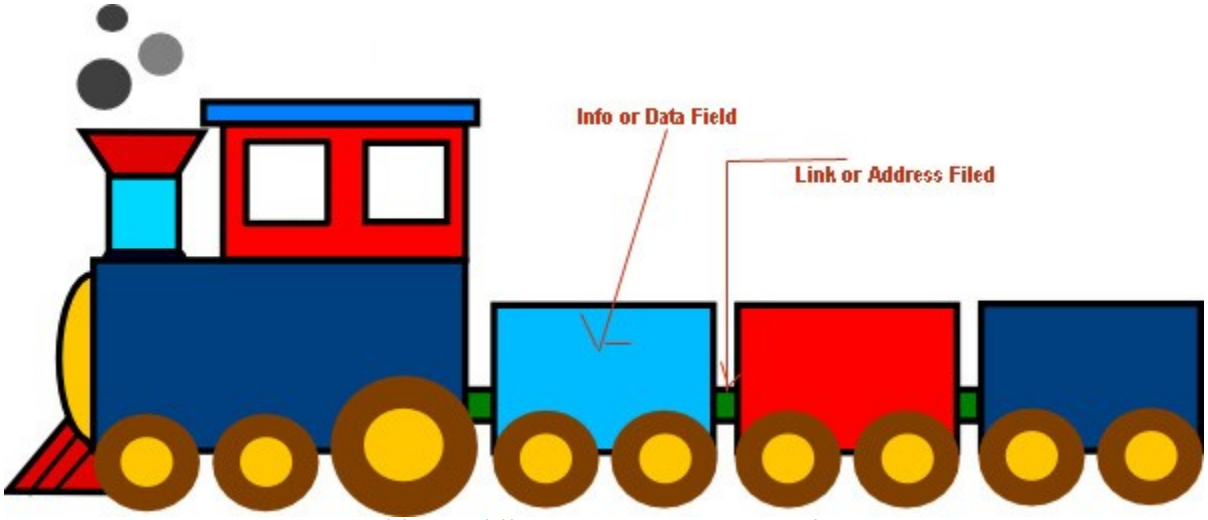
Course Outcome

CO3: Compare and contrast the advantages and disadvantages of linked lists over arrays and implement operations on different types of linked list.

Linked Lists

- A linked list is a linear data structure.
- Nodes make up linked lists.
- Nodes are structures made up of data and a pointer to another node.
- Usually the pointer is called next.



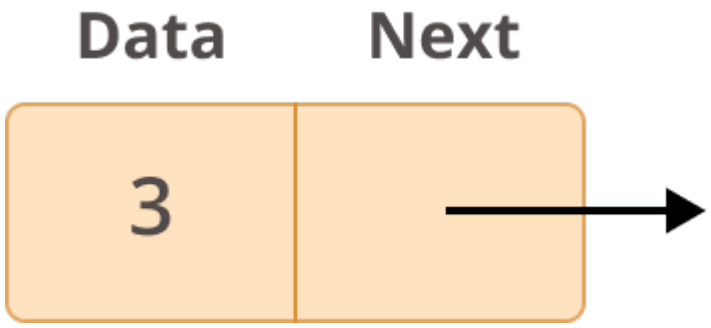


Linked List

- The elements of a linked list are not stored in adjacent memory locations as in arrays.
- It is a linear collection of data elements, called nodes, where the linear order is implemented by means of pointers.

Linked List

- In a linear or single-linked list, a node is connected to the next node by a single link.
- A node in this type of linked list contains two types of fields
  - data: which holds a list element
  - next: which stores a link (i.e. pointer) to the next node in the list.



Why Linked List?

Arrays can be used to store linear data of similar types, but arrays have the following limitations.

- The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.
- Inserting a new element in an array of elements is expensive because the room has to be created for the new elements and to create room existing elements have to be shifted.

Introduction to Link List(CO3)

- **Uses of Linked Lists**
- Due to their **dynamic size allocation** and ease of insertion/deletion, linked lists are applied in a lot of use cases.
- They’re used to implement a lot of complex data structures like the **adjacency list** in graphs.
- They are used for **lifecycle management** in operating systems.
- A playlist in a music application is implemented using a doubly linked list.
- **Blockchain**, a complex data structure that is

Properties of Linked list

- The nodes in a linked list are not stored contiguously in the memory.
- You don't have to shift any element in the list.
- Memory for each node can be allocated dynamically whenever the need arises.
- The size of a linked list can grow or shrink dynamically.

Operations on Linked List

- Creation:
  - This operation is used to create a linked list
- Insertion / Deletion:-
  - At/From the beginning of the linked list
  - At/From the end of the linked list
  - At/From the specified position in a linked list
- Traversing:
  - Traversing may be either forward or backward
- Searching:
  - Finding an element in a linked list

Arrays Vs Linked Lists

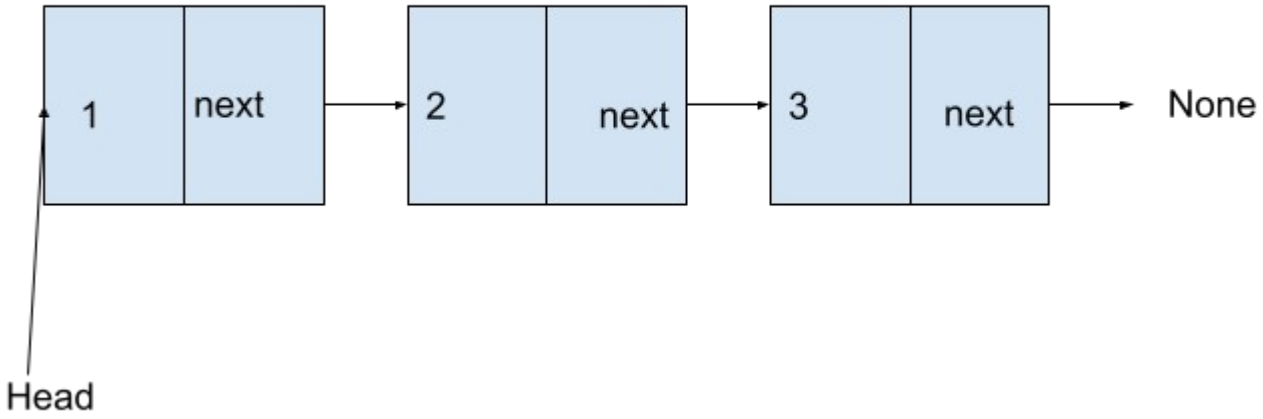
Arrays	Linked list
Fixed size: Resizing is expensive	Dynamic size
Insertions and Deletions are inefficient: Elements are usually shifted	Insertions and Deletions are efficient: No shifting
Random access i.e., efficient indexing	No random access → Not suitable for operations requiring accessing elements by index such as sorting
No memory waste if the array is full or almost full; otherwise may result in much memory waste.	Since memory is allocated dynamically(acc. to our need) there is no waste of memory.
Sequential access is faster [Reason: Elements in contiguous memory locations]	Sequential access is slow [Reason: Elements not in contiguous memory locations]

Types of Linked List

- Singly Linked List
- Doubly linked list
- Circular linked list

Singly Linked List

- In this type of linked list each node contains two fields one is data field which is used to store the data items and another is next field that is used to point the next node in the list.



Introduction to Link List(CO3)

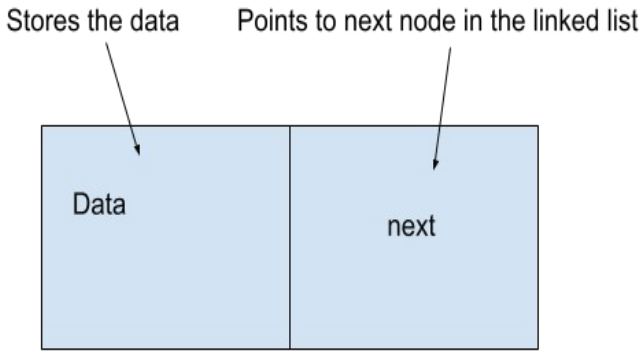
Operations performed on Linked List:-

1. Creating a List
2. Inserting an element in a list
3. Deleting an element from a list
4. Searching a list
5. Reversing a list

Creating a single Node:-

For single node creation, we make a Node class that holds some data and a single pointer next, that will be used to point to the next Node type object in the Linked List.

```
# A single node of a singly linked list
class Node:
  # constructor
  def __init__(self, data):
    self.data = data
    self.next = None
# Creating a single node
first = Node(3)
print(first.data)
```



Creation of Linked list

```
class node:
    def __init__(self, data ):
            self.data = data
            self.next = None
# A Linked List class with a single head node
class LinkedList:
   def __init__(self):
      self.head = None
```

Traversing a linked list

To traverse a linked list in python, we will start from the head, print the data and move to the next node until we reach None i.e. end of the linked list.

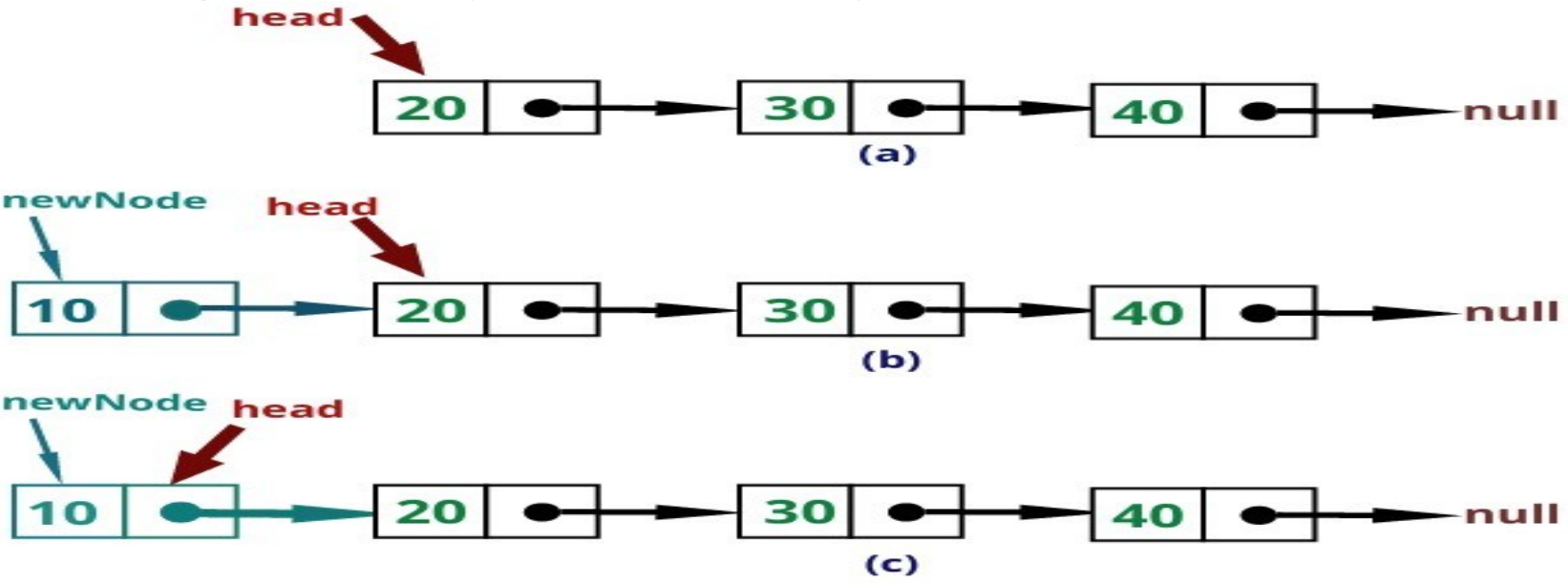
```
def traverse(self):
    temp=self.head
    while temp!=None:
        print(temp.data)
        temp=temp.next
```

Inserting an Element

- While inserting an element or a node in a linked list, we have to do following things:
  - Allocate a node
  - Assign a data to info field of the node.
  - Adjust a pointer
- We can insert an element in following places
  - At the beginning of the linked list
  - At the end of the linked list
  - At the specified position in a linked list

Inserting at the Beginning

1. Allocate memory for new node
2. Store data
3. Change next of new node to point to head
4. Change head to point to recently created node



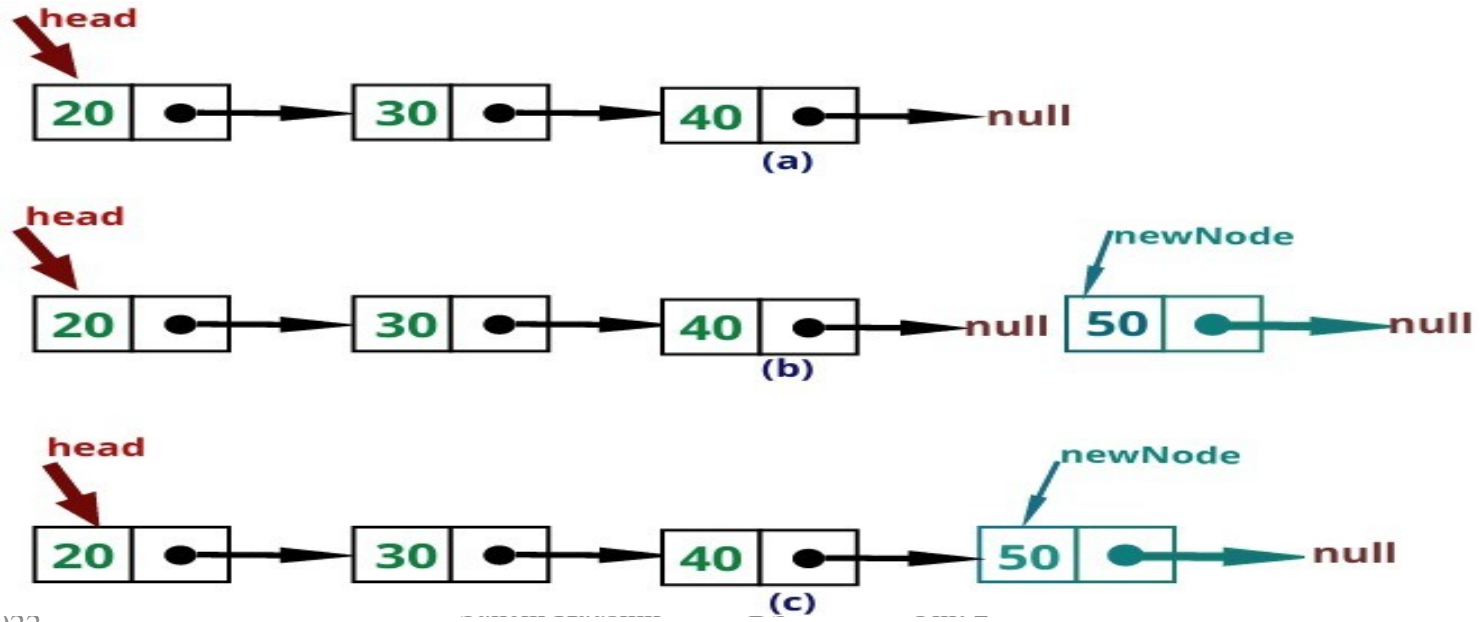
Inserting At Beginning of the list

```
def insertAtBeginning(self,data):
    newnode=Node(data)
     if self.head==None:
        self.head=newnode
    else:
        newnode.next=self.head
        self.head=newnode
```

Introduction to Link List(CO3)

Inserting At End of the list

1. Allocate memory for new node
2. Store data
3. Traverse to last node
4. Change next of last node to recently created node



Introduction to Link List(CO3)

Inserting At End of the list

```
def insertAtEnd(self,data):
        newnode=Node(data)
        if self.head==None:
            self.head=newnode
        else:
            temp=self.head
            while temp.next!=None:
                temp=temp.next
            temp.next=newnode
```

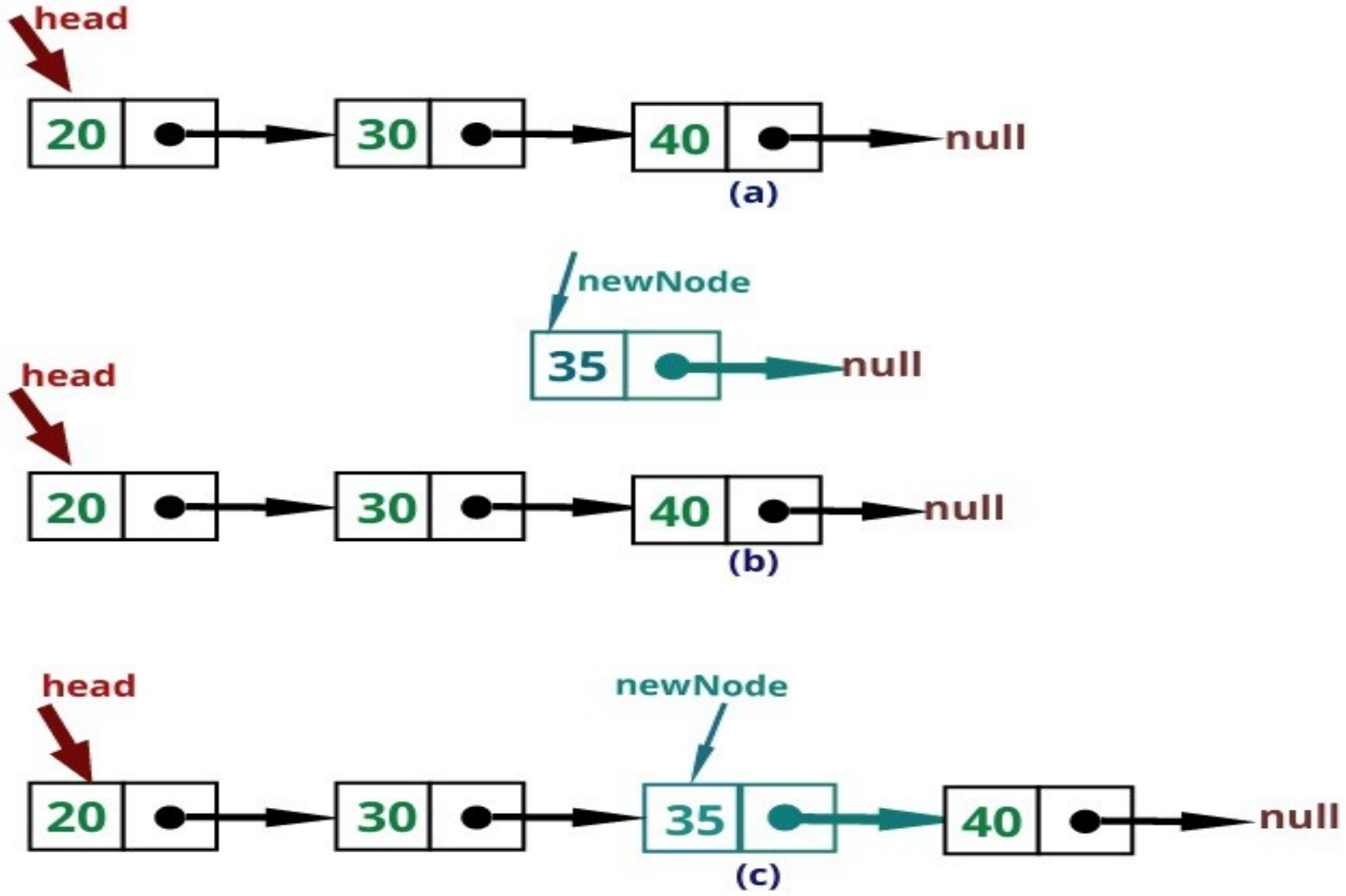
Introduction to Link List(CO3)

Inserting At Specific location in the list (After a Node)

1. Allocate memory and store data for new node
2. Traverse to node just before the required position of new node
3. Change next pointers to include new node in between

Introduction to Link List(CO3)

Inserting At Specific location in the list (After a Node)



Introduction to Link List(CO3)

Inserting At Specific location in the list (After a Node)

```
def insertAtGivenPosition(self,data,position):
    newnode=Node(data)
    count=1
    temp=self.head
    while count<position-1 and temp!=None:
        temp=temp.next
        count+=1
    newnode.next=temp.next
    temp.next=newnode
```

Introduction to Link List(CO3)

Deletion

In a single linked list, the deletion operation can be performed in three ways. They are as follows...

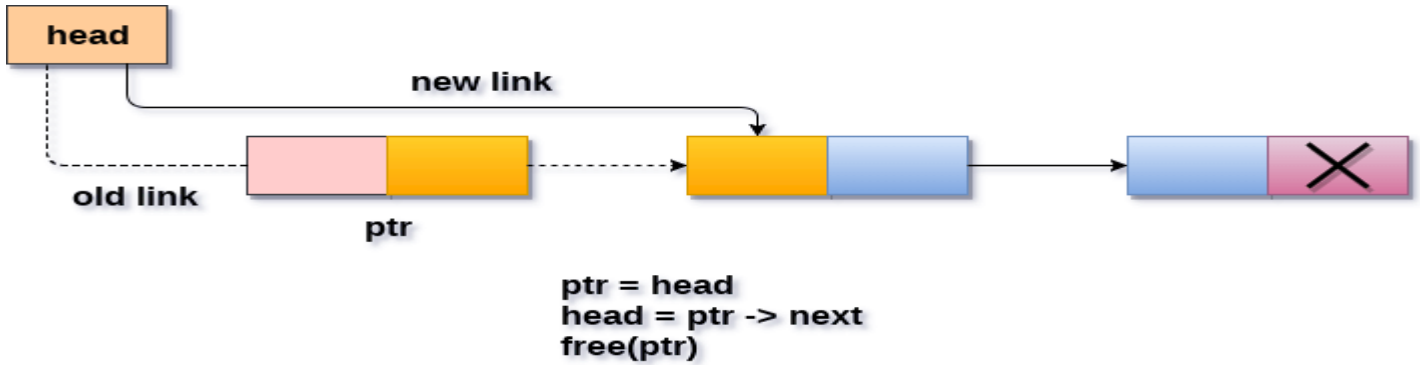
1.Deleting from Beginning of the list

2.Deleting from End of the list

3.Deleting a Specific Node

Deleting from Beginning of the list :-

To delete the first node of a linked list, we will first check if the head of the linked list is pointing to None, if yes then we will raise an exception using python try except with a message that the linked list is empty. Otherwise, we will delete the current node referred by head and move the head pointer to the next node.



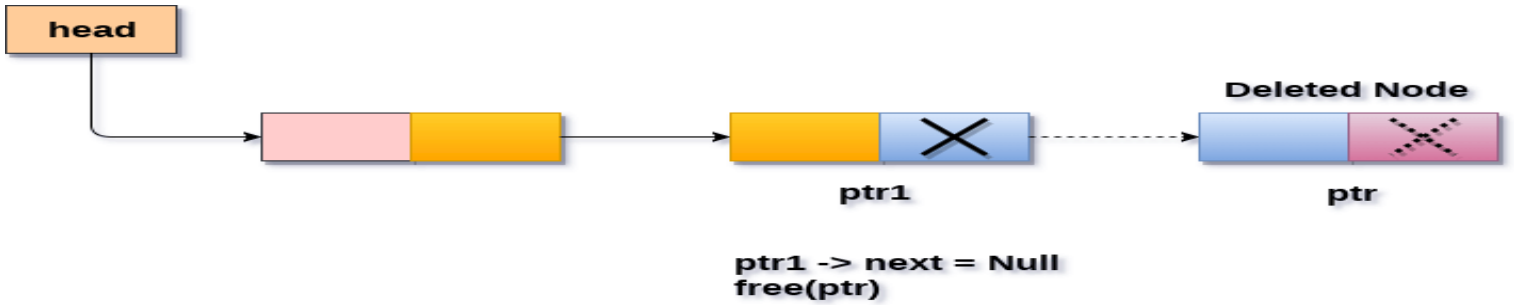
Deleting a node from the beginning

Deleting from Beginning of the list :-

```
def delFromBeginning(self):
    try:
        if self.head==None:
            raise Exception("Empty Linked List")
    else:
        temp=self.head
        self.head=temp.next
        temp.next=None
    except Exception as e:
        print(str(e))
```

Deleting from End of the list

To delete the last node of the linked list, we will traverse each node in the linked list and check if the next pointer of the next node of current node points to None,if yes then the next node of current node is the last node and it will get deleted.



Deleting a node from the last

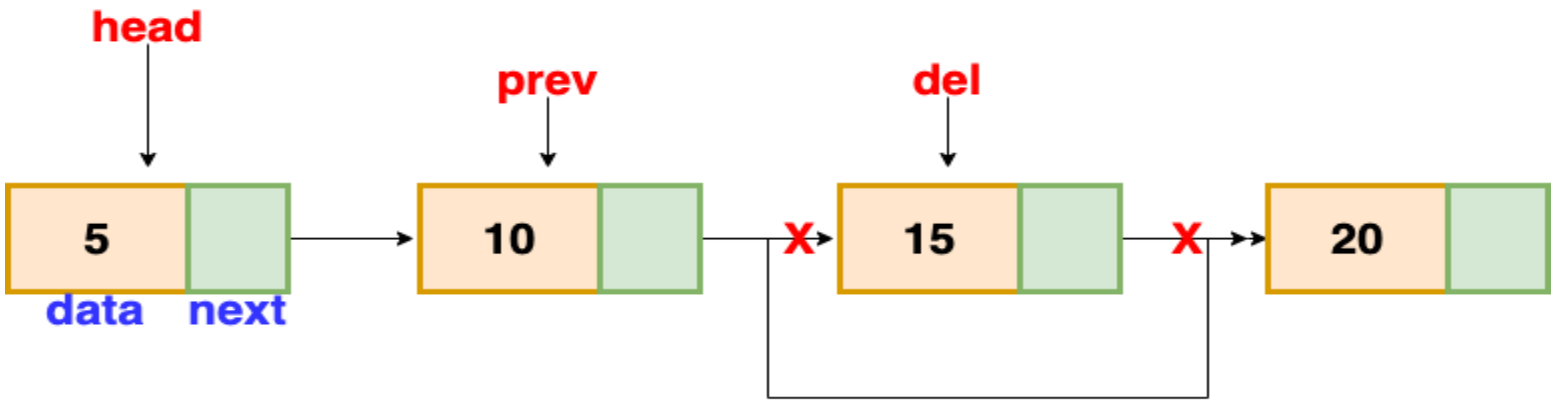
Introduction to Link List(CO3)

Deleting from End of the list

```
def delFromEnd(self):
        try:
            if self.head==None:
            raise Exception("Empty Linked List")
        else:
             temp=self.head
             prev=None
            while temp.next!=None:
                    prev=temp
                    temp=temp.next
prev.next=temp.next
            del temp
       except Exception as e:
             print(str(e))
```

Deleting a Specific Node from the list

To delete a node in between the linked list, at every node we will check if the position of next node is the node to be deleted, if yes, we will delete the next node and assign the next reference to the next node of the node being deleted.



Deleting a Node in Linked List

Introduction to Link List(CO3)

Deleting a Specific Node from the list

```
def delAtPos(self,position):
    try:
        if self.head==None:
            raise Exception("Empty Linked List")
        else:
            temp=self.head
            prev=None
            count=1
            while temp!=None and count<position:
                prev=temp
                temp=temp.next
                count+=1
            prev.next=temp.next
            del temp
    except Exception as e:
        print(str(e))
```

Introduction to Link List(CO3)

Searching an item in a linked list:-

we can search an element on a linked list using a loop using the following steps. We are finding item on a linked list.

1. Make head as the current node.
2. Run a loop until the current node is NULL because the last element points to NULL.
3. In each iteration, check if the key of the node is equal to item.

    If it the key matches the item, return true otherwise return false.

Introduction to Link List(CO3)

Searching an item in a linked list:-

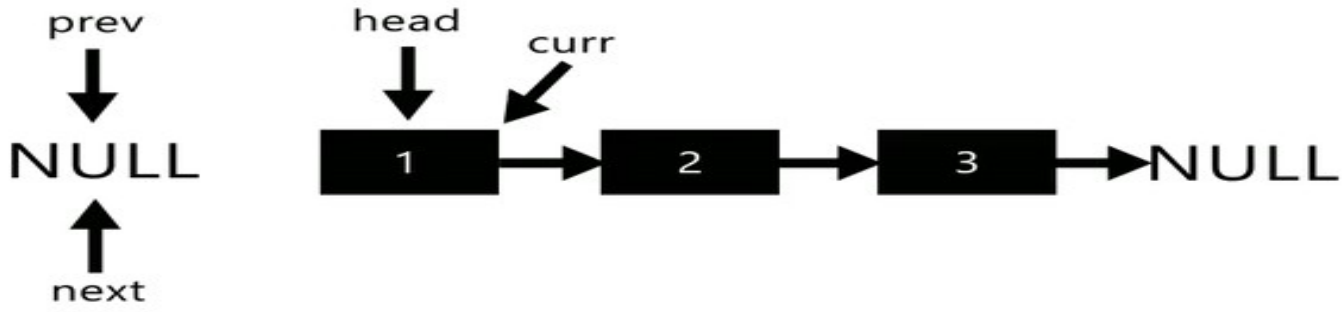
```
def search(self, key):
        current = self.head
         index = 0
        while current:
            if current.data == key:
                return index
            current = current.next
            index = index + 1
        return -1
```

Reverse a linked list

1.Initialize three pointers prev as NULL, curr as head and next as NULL.
2.Iterate through the linked list. In loop, do following.

```
// Before changing next of current,
// store next node
      next = curr->next
// Now change next of current
// This is where actual reversing happens
      curr->next = prev
// Move prev and curr one step forward
      prev = curr
      curr = next
```

Reverse a linked list




```
while (current != NULL)
      {
            next    = current->next;
            current->next = prev;
            prev = current;
            current = next;
      }
      *head_ref = prev;
```

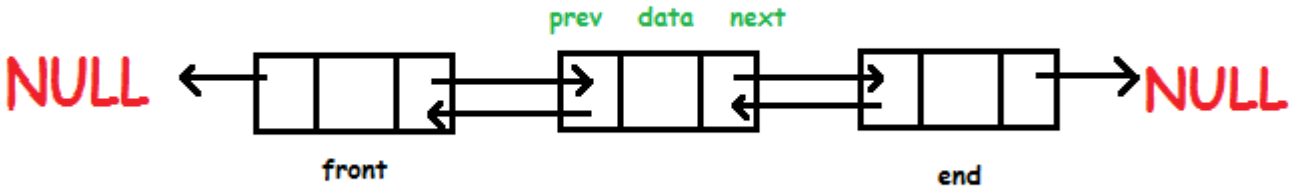
Introduction to Link List(CO3)

Sort a linked list

```
def sort(self):
    temp = self.head
    index = None
    if(self.head == None):
        return
    else:
        while(temp != None):
            index = temp.next
            while(index != None):
                if(temp.data > index.data):
                    a = temp.data
                    temp.data = index.data
                    index.data = a
                index = index.next
            temp = temp.next
```

Doubly Linked List

Doubly linked list is a type of linked list in which each node apart from storing its data has two links. The first link points to the previous node in the list and the second link points to the next node in the list. The first node of the list has its previous link pointing to NULL similarly the last node of the list has its next node



The two links help us to traverse the list in both backward and forward direction. But storing an extra link requires some extra space.

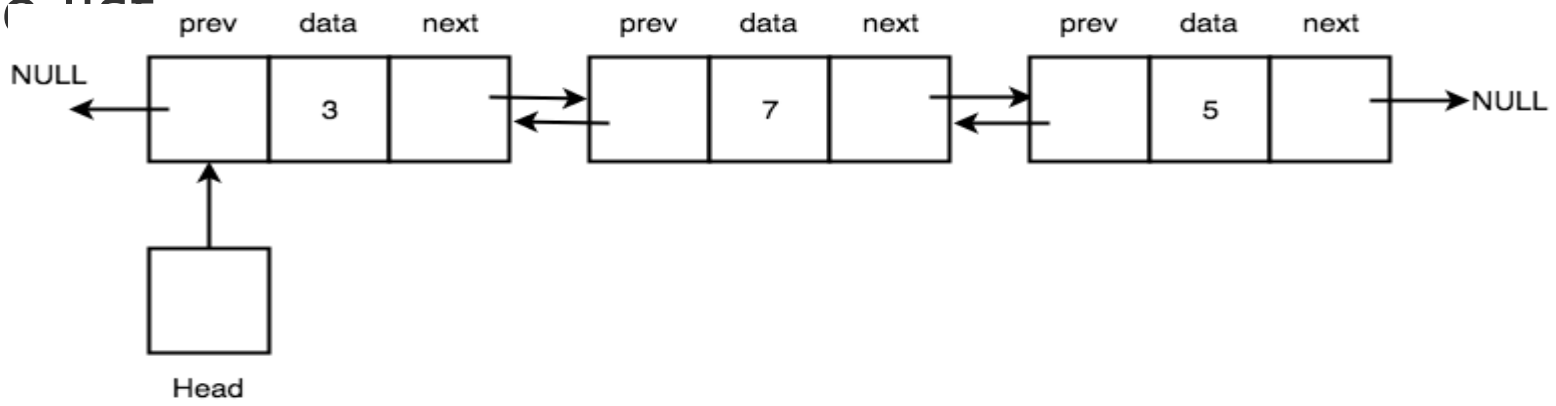
Link List(CO3)

A Doubly Linked list is also made up of nodes, but as compared to the node used in the Singly linked list, node in case of the doubly linked list has 3 parts:

1.prev: It is a pointer that points to the previous node in the list.

2.data: It holds the actual data.

3.next: It is a pointer that points to the next node in the list.



Link List(CO3)

Advantages of Doubly Linked List

1. Traversal can be done on either side means both in forward as well as backward.

2. Deletion Operation is more efficient if the pointer to delete node is given.

Disadvantages of Linked List

1. Since it requires extra pointer that is the previous pointer to store previous node reference.

2. After each operation like insertion-deletion, it requires an extra pointer that is a previous pointer which needs to be maintained.

Characteristics of Doubly Linked Lists

1. **Ordered** - Data elements are stored in a certain order
2. **Mutable**- Data elements can be added/ deleted/ modified
3. **Dynamic** - The length of the Linked List is dynamic
4. **Traversable** - Unlike Singly Linked Lists, Doubly Linked Lists can be traversed in both directions enabled by the two pointers available
5. Can store **data of any type**
6. Unlike Singly Linked Lists, **deletion of a node is easier** to implement using the left-hand node-link

Operations on Double Linked List

In a double linked list, we perform the following operations...

1. Creation
2. Insertion
3. Deletion
4. Display

Creating a single Node:-

For single node creation, we make a Node class that holds some data and a single pointer next, that will be used to point to the next Node type object in the Linked List.

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

Link List(CO3)

Creating a Doubly Linked List:-

```
class doubly_linked_list:
        def __init__(self):
            self.head = None
 def create(self, NewVal):
      NewNode = Node(NewVal)
      if self.head is None:
         NewNode.prev = None
         self.head = NewNode
      else:
         temp = self.head
         while (temp.next is not None):
            temp = temp.next
         temp.next = NewNode
         NewNode.prev = temp
```

Traversing a Doubly linked list

To traverse a linked list in python, we will start from the head, print the data and move to the next node until we reach None i.e. end of the linked list.

```
def display(self):
    temp=self.head
    while temp!=None:
        print(temp.data,"<-->",end=" ")
        temp=temp.next
```

Insertion

In a double linked list, the insertion operation can be performed in three ways as follows...

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the double linked list...

- **Step 1 -** Create a **newNode** with given value and **newNode → previous** as **NULL**.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then, assign **NULL** to **newNode → next** and **newNode** to **head**.
- **Step 4 -** If it is **not Empty** then, assign**newNode → next=head, head->prev = NULL** and **head= newNode**.

Inserting At Beginning of the list

```
def insertatbegining(self, data):
        new_node = Node(data)
        if self.head is None:
                self.head = new_node
         else:
                new_node.next = self.head
                self.head.prev= new_node
                self.head = new_node
```

Inserting At End of the list

- **Step 1 -** Create a **newNode** with given value and **newNode → next** as **NULL**.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty**, then assign **NULL** to **newNode → previous** and **newNode** to **head**.
- **Step 4 -** If it is **not Empty**, then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next** is equal to **NULL**).
- **Step 6 -** Assign **newNode** to **temp → next** and **temp** to **newNode → previous**.

Inserting At End of the list

```
def insertatend(self, data):
        newnode=Node(data)
        if self.head is None:
            self.head = newnode
        else:
            temp = self.head
            while temp.next is not None:
                temp=temp.next
            temp.next = newnode
            newnode.prev = temp
```

Inserting At Specific location in the list (After a Node)

```
def insertafternode(self, x, data):
    newnode=Node(data)
    if self.head is None:
        print("List is empty")
    else:
        temp = self.head
        while temp is not None:
            if temp.data == x:
                break
            temp = temp.next
        if temp is None:
            print("item not in the list")
        else:
            newnode.prev = temp
            newnode.next = temp.next
            if temp.next is not None:
                temp.next.prev = newnode
            temp.next = newnode
```

Deletion

In a double linked list, the deletion operation can be performed in three ways as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

Deleting from Beginning of the list

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is not Empty then, define a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Check whether list is having only one node (**temp → previous** is equal to **temp → next**)
- **Step 5 -** If it is **TRUE**, then set **head = NULL** and delete **temp** (Setting **Empty** list conditions)
- **Step 6 -** If it is **FALSE**, then assign **head=temp → next**,

  head → previous =NULL and delete **temp**.

Deleting from Beginning of the list

```
def deleteatstart(self):
    if self.head is None:
        print("The list has no element to delete")
    if self.head.next is None:
        self.head = None
    self.head = self.head.next
    self.head.prev = None
```

Deleting from End of the list

- **Step 1 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 2 -** If it is **Empty**, then display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- **Step 3 -** If it is not Empty then, define a Node pointer **'temp'** and initialize with **head**.
- **Step 4 -** Check whether list has only one Node (**temp → previous** and **temp → next** both are **NULL**)
- **Step 5 -** If it is **TRUE**, then assign **NULL** to **head** and delete **temp**. And terminate from the function. (Setting **Empty** list condition)
- **Step 6** - If it is FALSE, then keep moving temp until it reaches to the last node in the list. (until temp → next is equal to NULL)
- **Step 7** - Assign NULL to temp → previous → next and delete temp.

Deleting from End of the list

```
def deleteatend(self):
    if self.head is None:
        print("The list has no element to delete")
        return
    if self.head.next is None:
        self.head = None
    temp = self.head
    while temp.next is not None:
        temp = temp.next
    temp.prev.next = None
```

Deleting a Specific Node from the list

```
def deletenodebyvalue(self, x):
    if self.head is None:
        print("empty list")
        return

    if self.head.data== x:
        self.head = self.head.next
        self.head.prev = None
        return
    temp = self.head
    while temp.next is not None:
        if temp.data == x:
            break;
        temp = temp.next
    if temp.next is not None:
        temp.prev.next = temp.next
        temp.next.prev = temp.prev
    else:
        if temp.data == x:
            temp.prev.next = None
        else:
            print("Element not found")
```

Link List(CO3)

Search a Specific Node from the Doubly Linked list

```
def search(self,data):
    temp = self.head
    c=1
    while temp:
        if temp.data==data:
            break

        temp = temp.next
        c+=1
    if temp==None:
        print("The given data doesnt exist:")
        return -1
    return c
```

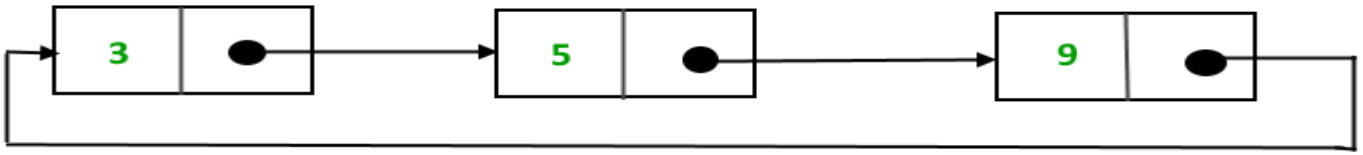
Link List(CO3)

Reverse a Doubly Linked List

```
def reverse(self):
    if self.head is None:
        print("The list has no element to reverse")
        return
    temp = self.head
    q= temp.next
    temp.next = None
    temp.prev = q
    while q is not None:
        q.prev = q.next
        q.next = temp
        temp = q
        q = q.prev
    self.head = temp
```

Circular linked list

- Circular linked list is a variation of linked list in which the first elements points to the next element and the last element points to the first element.
- Both singly and doubly linked list can be made into a circular linked list. Circular linked list can be used to help traverse the same list again and again if needed.



Advantages of Circular linked lists:

1. Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.
2. Circular lists are useful in applications to repeatedly go around the list.
3. Circular Doubly Linked Lists are used for implementation of advanced data structures like Fibonacci Heap.

Disadvantages of Circular linked list

1. Depending on the implementation, inserting at start of list would require doing a search for last node which could be expensive.
2. Finding end of the list and loop control is harder (no NULL's to mark the beginning and end).

Operations on singly circular linked list

- Creation
- Insertion
- Deletion
- Display

Circular Link List(CO3)

Creation of Circular Linked List:

```
def create(self,data):
        newnode=node(data)
        if self.head==None:
            self.head=newnode
            newnode.next=newnode
        else:
            temp=self.head
            while temp.next!=self.head:
                temp=temp.next
            newnode.next=self.head
            self.head=newnode
            temp.next=self.head
```

Displaying a circular Linked List

Step 1 - Check whether list is **Empty** (**head** == **NULL**)

Step 2 - If it is **Empty**, then display **'List is Empty!!!'** and terminate the function.

Step 3 - If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.

Step 4 - Keep displaying **temp → data** with an arrow (**--->**)

until **temp** reaches to the last node

Step 5 - Finally display **temp → data** with arrow pointing to **head → data**.

Circular Link List(CO3)

Display Circular Linked List:

```
def display(self):
        current = self.head;
        if self.head is None:
              print("List is empty")
        else:
            print("Nodes of the circular linked list: ")
        #Prints each node by incrementing pointer.
            print(current.data)
            while(current.next != self.head):
                current = current.next
                print(current.data)
```

Insertion

- Insertion can be of three types.
  - Insert at first
  - Insert at last
  - Insert after constant
- Note: insertion after constant in circular and linear linked list is exact same .

Circular Link List(CO3)

Inserting At Beginning of the list

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**)
- **Step 3 -** If it is **Empty** then,

set **head** = **newNode** and **newNode→next** = **head** .

- **Step 4 -** If it is **Not Empty** then, define a Node pointer '**temp**' and initialize with '**head**'.
- **Step 5 -** Keep moving the '**temp**' to its next node until it reaches to the last node (until '**temp → next** == **head**').
- **Step 6 -** Set '**newNode → next** =**head**',

'**head** = **newNode**' and '**temp → next** = **head**'.

Circular Link List(CO3)

Inserting At Beginning of the list

```
def insertatbegining(self, newElement):
        newNode = node(newElement)
        if(self.head == None):
            self.head = newNode
            newNode.next = self.head
        else:
            temp = self.head
            while(temp.next != self.head):
                temp = temp.next
            temp.next = newNode
            newNode.next = self.head
            self.head = newNode
```

Circular Link List(CO3)

Inserting At End of the list

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether list is **Empty** (**head** == **NULL**).
- **Step 3 -** If it is **Empty** then, set **head** = **newNode** and **newNode → next** = **head**.
- **Step 4 -** If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- **Step 5 -** Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next** == **head**).
- **Step 6 -** Set **temp → next** = **newNode** and **newNode → next** = **head**.

Circular Link List(CO3)

Inserting At End of the list

```
def insertatend(self, data):
    newNode = node(data)
    if(self.head == None):
        self.head = newNode
        newNode.next = self.head
    else:
        temp = self.head
        while(temp.next != self.head):
            temp = temp.next
        temp.next = newNode
        newNode.next = self.head
```

Circular Link List(CO3)

Inserting At Specific location in the list (After a Node)

Refer the program in file

#Insert at position in circular Linked List

Circular Link List(CO3)

Inserting At Specific location in the list (After a Node)

- **Step 6 -** Every time check whether **temp** is reached to the last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.
- **Step 7 -** If **temp** is reached to the exact node after which we want to insert the newNode then check whether it is last node (temp → next == head).
- **Step 8 -** If **temp** is last node then set **temp → next** = **newNode** and **newNode → next** = **head**.
- **Step 8 -** If **temp** is not last node then set **newNode → next** = **temp → next** and **temp → next** = **newNode**

Deletion :

- Deletion can be of three types.
  - Delete from front
  - Delete from last
  - Deletion from mid

- Note: deletion from mid in circular and linear linked list is exact same .

Circular Link List(CO3)

Deleting from Beginning of the list

Step 1 - Check whether list is **Empty** (**head** == **NULL**)
Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
Step 3 - If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize both '**temp1**' and '**temp2**' with **head**.
Step 4 - Check whether list is having only one node (**temp1 → next** == **head**)
Step 5 - If it is **TRUE** then set **head** = **NULL** and delete **temp1** (Setting **Empty** list conditions)
Step 6 - If it is **FALSE** move the **temp1** until it reaches to the last node. (until **temp1 → next** == **head**)
Step 7 - Then set **head** = **temp2 → next**, **temp1 → next** = **head** and delete **temp2**.

Deleting from Beginning of the list

```
def deleteatfirst(self):
    if(self.head != None):
        if(self.head.next == self.head):
            self.head = None
        else:
            temp = self.head
            firstNode = self.head
            while(temp.next != self.head):
                temp = temp.next
            self.head = self.head.next
            temp.next = self.head
            firstNode = None
```

Circular Link List(CO3)

Deleting from End of the list

Step 1 - Check whether list is **Empty** (**head** == **NULL**)
Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
Step 3 - If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **head**.
Step 4 - Check whether list has only one Node (**temp1 → next** == **head**)
Step 5 - If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate from the function. (Setting **Empty** list condition)
Step 6 - If it is **FALSE**. Then, set '**temp2 = temp1** ' and move **temp1** to its next node. Repeat the same until **temp1** reaches to the last node in the list. (until **temp1 → next** == **head**)
Step 7 - Set **temp2 → next** = **head** and delete **temp1**.

Circular Link List(CO3)

Deleting from End of the list

```
def deleteatend(self):
    if(self.head != None):
        if(self.head.next == self.head):
            self.head = None
        else:
            temp = self.head
            while(temp.next.next != self.head):
                temp = temp.next
            lastNode = temp.next
            temp.next = self.head
            lastNode = None
```

Circular Link List(CO3)

Deleting at position:-

```
def deleteatposition(self, position):
    nodeToDelete = self.head
    temp = self.head
    NoOfElements = 0
    if(temp != None):
        NoOfElements += 1
        temp = temp.next
    while(temp != self.head):
        NoOfElements += 1
        temp = temp.next
    if(position < 1 or position > NoOfElements):
        print("\nInavalid position.")
    elif (position == 1):
        if(self.head.next == self.head):
            self.head = None
        else:
            while(temp.next != self.head):
                temp = temp.next
            self.head = self.head.next
            temp.next = self.head
            nodeToDelete = None
    else:
        temp = self.head
        for i in range(1, position-1):
            temp = temp.next
        nodeToDelete = temp.next
        temp.next = temp.next.next
        nodeToDelete = None
```

Circular Link List(CO3)

Search a node in circular Linked list:

```
def search(self, searchValue):
    temp = self.head
    found = 0
    i = 0
    if(temp != None):
        while (True):
            i += 1
            if(temp.data == searchValue):
                found += 1
                break
            temp = temp.next
            if(temp == self.head):
                break
        if(found == 1):
            print(searchValue,"is found at index =", i)
        else:
            print(searchValue,"is not found in the list.")
    else:
        print("The list is empty.")
```

Applications of linked list in computer science –

1. Implementation of stacks and queues.

2. Implementation of graphs : Adjacency list representation of graphs is most popular which is uses linked list to store adjacent vertices.

3. Dynamic memory allocation : We use linked list of free blocks.

4. Maintaining directory of names.

5. Manipulation of polynomials by storing constants in the node of linked list .

5. representing sparse matrices.

Polynomials

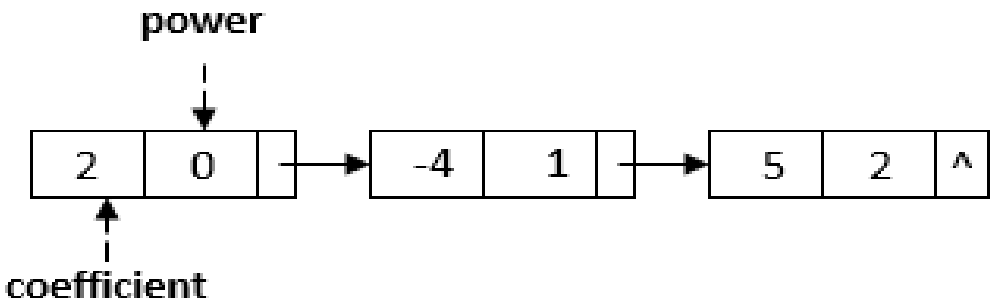
Polynomials are the algebraic expressions which consist of variables and coefficients.

Example -

$10x^2 + 26x$, here 10 and 26 are coefficients and 2, 1 is its exponential value.

Represent a Polynomial with a Linked List

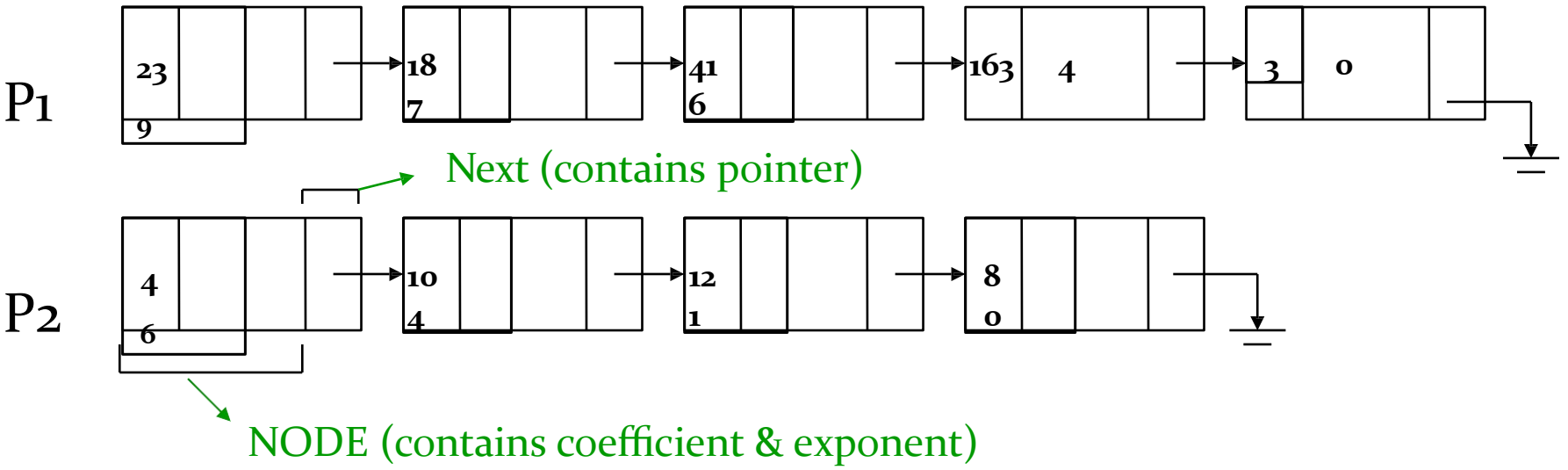
We can use a linked list to represent a polynomial. **In the linked list, each node has two data fields: coefficient and power**. Therefore, each node represents a term of a polynomial. For example, we can represent the polynomial with a linked list:



Link List(CO3)

-

- **Linked list Implementation:**
- **$p1(x) = 23x^9 + 18x^7 + 41x^6 + 163x^4 + 3$**
- **$p2(x) = 4x^6 + 10x^4 + 12x + 8$**



Addition of Polynomials

Example:

Input:

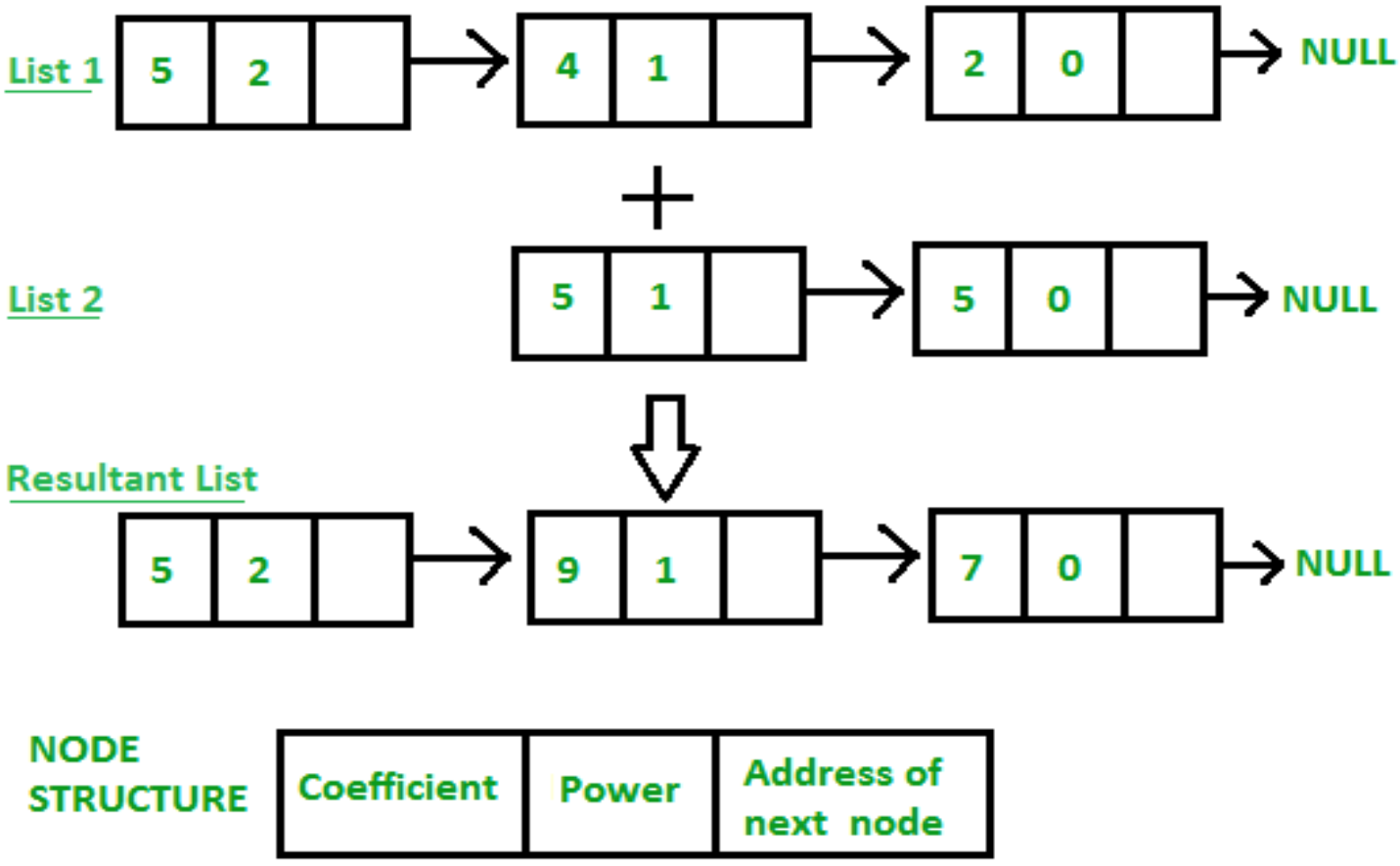
1st number = $5x^2 + 4x^1 + 2x^0$

2nd number = $-5x^1 - 5x^0$

Output:

$5x^2-1x^1-3x^0$

Addition of Polynomials



Multiplication of two polynomials using Linked list

Input:

Poly1: 3x^2 + 5x^1 + 6,

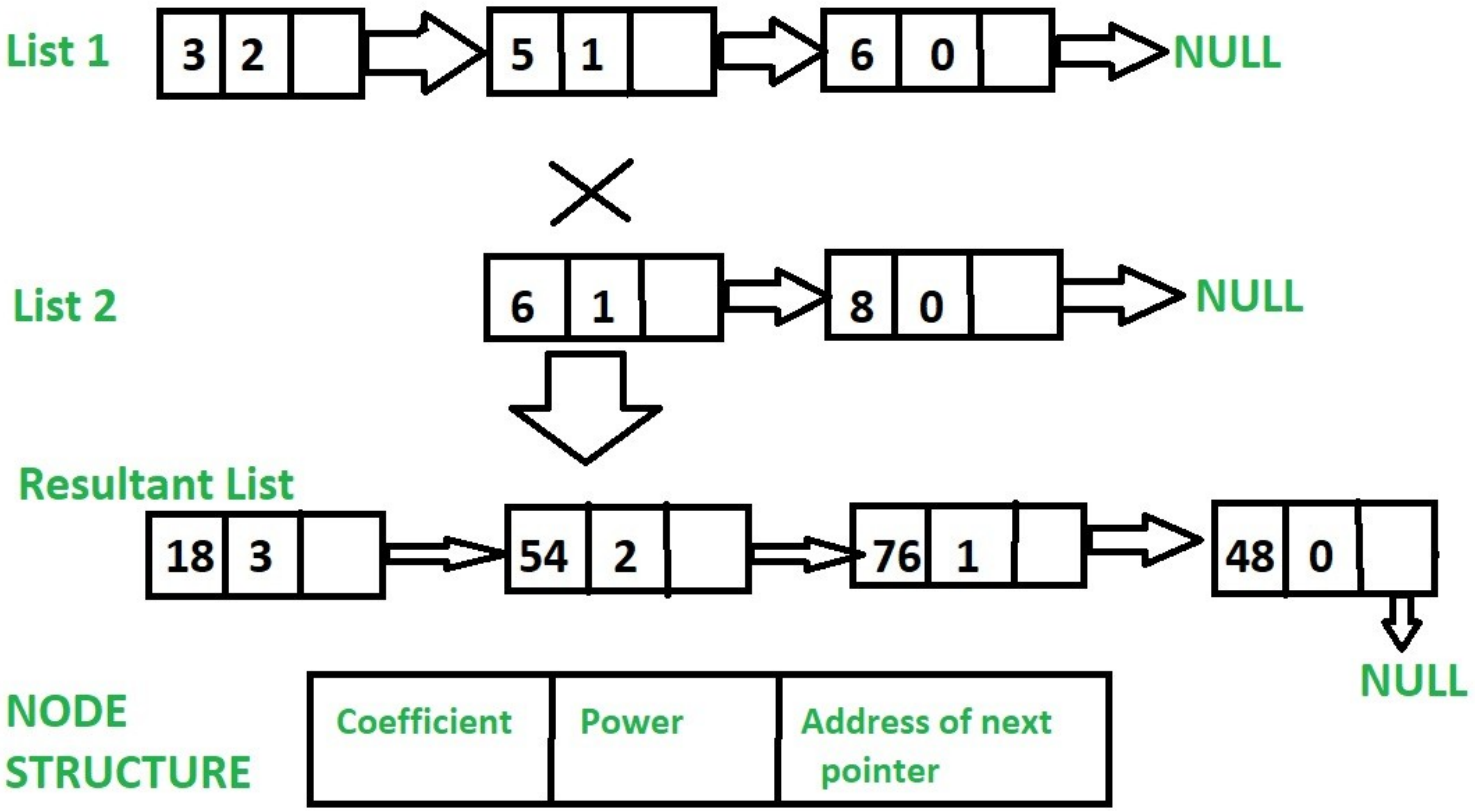
Poly2: 6x^1 + 8

Output:

18x^3 + 54x^2 + 76x^1 + 48

On multiplying each element of 1st polynomial with elements of 2nd polynomial, we get 18x^3 + 24x^2 + 30x^2 + 40x^1 + 36x^1 + 48 On adding values with same power of x, 18x^3 + 54x^2 + 76x^1 + 48

Multiplication of two polynomials using Linked list



Daily Quiz

Q3. Describe what is Node in link list? And name the types of Linked Lists?

Q4. Mention what is the difference between Linear Array and Linked List?

Q5. Mention what are the applications of Linked Lists?

Q6. What does the dummy header in linked list contain?

Q7. Mention what is the difference between singly and doubly linked lists?

Q8. Mention what is the biggest advantage of linked lists?

Q9. Mention how to insert a new node in linked list where free node will be available?

Expected Questions for University Exam

Q1. Write a program in c to delete a specific element in single linked list. Double linked list takes more space than single linked list for storing one extra address. Under what condition, could a double linked list more beneficial than single linked list.

Q2. What is a doubly linked list? How is it different from the single linked list?

Q3. What are the advantages and disadvantages of array over linked list?

Q4 . Write a program to implement linear linked list, showing all the operations that can be performed on a linked list.

Q5. Implement Doubly Circular link list and insert an element at a given position in the doubly circular link list.

Old Question Papers

https://drive.google.com/open?id=15fAkaWQ5ccZRZPzwlP4PBh1LxcPp4VAd

References

1. **Fundamentals of Data Structures in C** by Horowitz, Sahni and Anderson-Freed.
2. **Data Structures Through C in Depth** by S.K Srivastava, Deepali Srivastava.
3. **Data Structure using c** by R Krishanamoorty.
4. **Data Structures** by Lipschutz, Schaum's Outline Series, Tata McGraw-hill Education (India) Pvt. Ltd.
5. **Data Structure Using C** by Reema Thareja, Oxford Higher Education.
6. **Data Structure Using C** by AK Sharma, Pearson Education India.

MCQ

1. Which of the following is not a disadvantage to the usage of array?

a) Fixed size

b) There are chances of wastage of memory space if elements inserted in an array are lesser than the allocated size

c) Insertion based on position

d) Accessing elements at specified positions

MCQ

2. Which of these is not an application of a linked list?

a) To implement file systems

b) For separate chaining in hash-tables

c) To implement non-binary trees

d) Random Access of elements

MCQ

3.A linear collection of data elements where the linear node is given by means of pointer is called?

A. linked list

B. node list

C. primitive list

D. None of these

MCQ

4. Linked lists are not suitable to for the implementation of?

A. Insertion sort

B. Radix sort

C. Polynomial manipulation

D. Binary search

MCQ

5. In circular linked list, insertion of node requires modification of?

A. One pointer

B. Two pointer

C. Three pointer

D. None

MCQ

6. In a circular linked list

a) Components are all linked together in some sequential manner.

b) There is no beginning and no end.

c) Components are arranged hierarchically.

d) Forward and backward traversal within the list is permitted.

MCQ

7. **A linear collection of data elements where the linear node is given by means of pointer is called?**

a) Linked list

b) Node list

c) Primitive list

d) None

MCQ

8. **Which of the following operations is performed more efficiently by doubly linked list than by singly linked list?**

a) Deleting a node whose location in given

b) Searching of an unsorted list for a given item

c) Inverting a node after the node with given location

d) Traversing a list to process each node

MCQ

9. **In linked list each node contain minimum of two fields. One field is data field to store the data second field is?**

a) Pointer to character

b) Pointer to integer

c) Pointer to node

d) Node

MCQ

10. **The concatenation of two list can performed in O(1) time. Which of the following variation of linked list can be used?**

a) Singly linked list

b) Doubly linked list

c) Circular doubly linked list

d) Array implementation of list

Old Question Papers

https://drive.google.com/open?id=15fAkaWQ5ccZRZPzwlP4PBh1LxcPp4VAd

References

[1] Aaron M. Tenenbaum, Yedidyah Langsam and Moshe J. Augenstein, "Data Structures Using C and C++", PHI Learning Private Limited, Delhi India
[2] Horowitz and Sahani, "Fundamentals of Data Structures", Galgotia Publications Pvt Ltd Delhi India.
[3] Lipschutz, "Data Structures" Schaum's Outline Series, Tata McGraw-hill Education (India) Pvt. Ltd.
[4] Thareja, "Data Structure Using C" Oxford Higher Education.
[5] AK Sharma, "Data Structure Using C", Pearson Education India.

Thank you