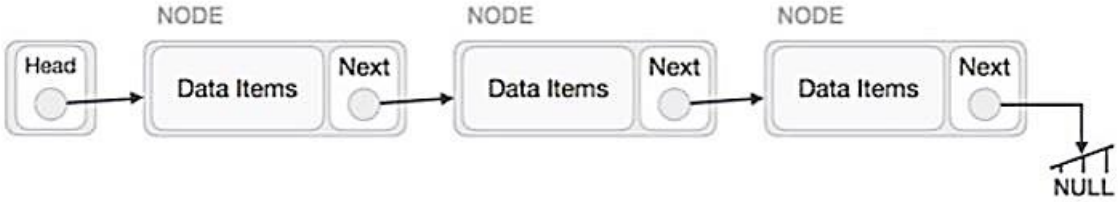
➢ Linked List: -

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

- **Link** − Each link of a linked list can store a data called an element.
- **Next** − Each link of a linked list contains a link to the next link called Next.
- **LinkedList** − A Linked List contains the connection link to the first link called First.

Linked list Representation: -

Linked list can be visualized as a chain of nodes, where every node points to the next node.



As per the above illustration, following are the important points to be considered.

- Linked List contains a link element called first.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list

Types of linked list: -

Following are the various types of linked list.

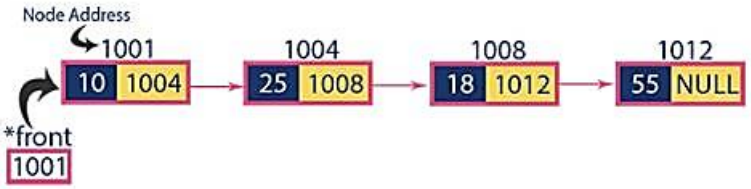
- **Singly Linked List** − Item navigation is forward only.
- **Doubly Linked List** − Items can be navigated forward and backward.
- **Circular Linked List** − Last item contains link of the first element as next and the first element has a link to the last element as previous.

➢ Singly linked list: -

Singly linked list can be defined as the collection of ordered set of elements. A node in the singly linked list consists of two parts: data part and link part. Data part of the node stores actual information that is to be represented by the node while the link part of the node stores the address of next node in the sequence.

One way chain or singly linked list can be traversed only in one direction. In other words, we can say that each node contains only next pointer, therefore we can not traverse the list in the reverse direction.

- In a single linked list, the address of the first node is always stored in a reference node known as "front" (Sometimes it is also known as "head").
- Always next part (reference part) of the last node must be NULL.



Operation on Singly Linked List: -

1. **Insertion: -** the insertion is categorized into the following categories.

   I. Insertion at beginning of the linked list: -

   It involves inserting any element at the front of the list. We just need to a few link adjustments to make the new node as the head of the list

   - Step 1 - Create a **newNode** with given value.
   - Step 2 - Check whether list is **Empty** (**head** == **None**)
   - Step 3 - If it is **Empty** then, set **newNode→next** = **None** and **head** = **newNode**
   - Step 4 – If it is **Not Empty** then, set **newNode→next** = **head** and **head** = **newNode**

   Program for insert at beginning of linked list: -

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

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

    def printList(self):
        if self.head is None:
            print("\nlinked list is empty")
        else:
            n = self.head
            while n is not None:
                print(n.data,end =" ")
                n = n.next

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

list1  = linkedlist()
list1.add_begin(51)
list1.add_begin(60)
list1.printList()
```

   Output = 60, 51

II. Insertion at end of the linked list: -

It involves insertion at the last of the linked list. The new node can be inserted as the only node in the list or it can be inserted as the last one. Different logics are implemented in each scenario.

- Step 1 - Create a **newNode** with given value and **newNode → next** as **None.**
- Step 2 - Check whether list is **Empty (head == None).**
- Step 3 - If it is **Empty** then, set **head = newNode**.
- 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 **None**).
- Step 6 - Set **temp → next** = **newNode**.

Program for insert at the end of linked list: -

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

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

    def printList(self):
        if self.head is None:
            print("\nlinked list is empty")
        else:
            n = self.head
            while n is not None:
                print(n.data,end =" ")
                n = n.next
    def add_end(self,data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
        else:
            temp = self.head
            while temp.next is not None:
                temp = temp.next
            temp.next = new_node

list1  = linkedlist()
list1.add_end(51)
list1.add_end(60)
list1.printList()
```

Output = 51 60

III. Insertion after specified node: -

It involves insertion after the specified node of the linked list. We need to skip the desired number of nodes in order to reach the node after which the new node will be inserted.

- Step 1 - Create a **newNode** with given value.
- Step 2 - Check whether list is **Empty (head == None)**
- Step 3 - If it is Empty then, set **newNode → next** = **None** and **head** = **newNode**

- 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 node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).
- Step 6 - Every time check whether **temp** is reached to 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 - Finally, Set **'newNode → next = temp → next'** and **'temp → next = newNode'**

Program for insert after at specific node: -

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

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

    def printList(self):
        if self.head is None:
            print("\nlinked list is empty")
        else:
            n = self.head
            while n is not None:
                print(n.data,end =" ")
                n = n.next

    def between_list(self,data,position):
        new_node = Node(data)
        if self.head is None:
            new_node.next = None
            self.head = new_node
        else:
            temp = self.head
            while temp is not None:
                if position == temp.data:
                    break
                temp = temp.next
            else:
                print("Given node is not found in the list!!! Insertion not possible!!!")
            try:
                new_node = Node(data)
                new_node.next = temp.next
                temp.next = new_node
            except:
                pass

list1  = linkedlist()
list1.between_list(51,30)
list1.between_list(60,51)
list1.printList()
```

Output = 51 60

2. **Deletion:** -the operation is categorized into the following categories.
   I. **Deletion at beginning of linked list: -**

It involves deletion of a node from the beginning of the list. This is the simplest operation among all. It just needs a few adjustments in the node pointers.

- Step 1 - Check whether list is **Empty (head == None)**
- 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 → next == None)**
- Step 5 - If it is **TRUE** then set **head** = **None** and delete **temp** (Setting **Empty** list Conditions)
- Step 6 - If it is **FALSE** then set **head = temp → next,** and delete **temp**.

Program for deletion at beginning of linked list: -

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

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

    def printList(self):
        if self.head is None:
            print("\nlinked list is empty")
        else:
            n = self.head
            while n is not None:
                print(n.data,end =" ")
                n = n.next
    def begin_delete(self):
        if self.head is None:
            print("\n linked list is empty")
        else:
            temp = self.head
            if temp.next == None:
                self.head = None
            else:
                self.head = self.head.next

list1  = linkedlist()
list1.begin_delete()
list1.printList()
```

II. Deletion at the end of linked list: -

It involves deleting the last node of the list. The list can either be empty or full. Different logic is implemented for the different scenarios.

Step 1 - Check whether list is **Empty (head == None)**
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 pointers '**temp**' and initialize '**temp**' with **head**.
Step 4 - Check whether list has only one Node (**temp→ next == None)**
Step 5 - If it is **TRUE**. Then, set **head = None** and delete **temp**. And terminate the function. (Setting **Empty** list condition)
Step 6 - If it is **FALSE**. Then, Keep moving the **temp** to its **next** node →**next** until it reaches to **temp → next→ next** is equal to **None**).
Step 7 - Finally, set **temp → next** = **None** and delete **temp**

Program for deletion at the end of linked list: -

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

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

    def printList(self):
        if self.head is None:
            print("\nlinked list is empty")
        else:
            n = self.head
            while n is not None:
                print(n.data,end =" ")
                n = n.next
    def end_delete(self):
        if self.head is None:
            print("\nlinked list is empty")
        else:
            n = self.head
            while n.next.next is not None:
                n = n.next
            n.next = None
list1  = linkedlist()
list1.end_delete()
list1.printList()
```

III. Deletion after specified node: -

It involves deleting the node after the specified node in the list. we need to skip the desired number of nodes to reach the node after which the node will be deleted. This requires traversing through the list.

- Step 1 - Check whether list is **Empty (head == None)**
- 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 5 - Keep moving the **temp** to its next node until it reaches to exact node to be deleted (until **temp→ data** is equal to **location**, here location is the node value after which we want to insert the newNode). And every time set **temp** to **temp→ next** before moving the 'temp ' to its next node.
- Step 6 - Every time check whether **temp** is reached to last node or not. If it is reached to last node then display **'Given node is not found in the list!!! deletion not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.
- Step 7 - Finally, **Set temp→ next to temp→ next→ next**

Program for deletion after specified node: -

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

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

    def printList(self):
        if self.head is None:
            print("\nlinked list is empty")
        else:
            n = self.head
            while n is not None:
                print(n.data,end =" ")
                n = n.next

    def delete_by_value(self ,x):
        if self.head is None:
            print( 'List is Empty!!! Deletion is not possible')
        else:
            temp = self.head
            while temp is not None:
                if x == temp.data:
                    break
                temp = temp.next
            else:
                print("\n node is not present")
            try:
                temp.next = temp.next.next
            except:
                pass

list1  = linkedlist()
list1.delete_by_value(5)
list1.printList()
```

3. **Traversing: -**
   In traversing, we simply visit each node of the list at least once in order to perform some specific operation on it, for example, printing data part of each node present in the list.

   - Step 1 - Check whether list is Empty (head == None)
   - 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

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

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

    def printList(self):
        if self.head is None:
            print("\nlinked list is empty")
        else:
            n = self.head
            while n is not None:
                print(n.data,end ="--->")
                n = n.next

list1  = linkedlist()
list1.printList()
```

   Program for traversing: -

4. **Searching: -**
   In searching, we match each element of the list with the given element. If the element is found on any of the location, then location of that element is returned otherwise null is returned.

   - Step 1 - Check whether list is **Empty (head == None)**
   - Step 2 - If it is **Empty** then, display '**List is Empty!!! Searching 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 5 - Keep moving the **temp** to its next node until it reaches to exact node to be search (until **temp→ data** is equal to **location**, here location is the node value which we want to search) and if present display '**Node is present'** And every time set **temp** to **temp→ next** before moving the 'temp ' to its next node.
- Step 6 - Every time check whether **temp** is reached to last node or not. If it is reached to last node then display **'Given node is not found in the list!!!'** and terminate the function. Otherwise move the **temp** to next node.

Program for searching: -

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

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

    def search(self,x):
        if self.head is None:
            print( 'List is Empty!!! searching is not possible')
        else:
            temp = self.head
            while temp is not None:
                if temp.data == x:
                    print("Node is present")
                    break
                temp = temp.next
            else:
                print("node is not present")

list1  = linkedlist()
list1.search(40)
```

➢ Application: -

- It is used to implement stacks and queues which are like fundamental needs throughout computer science.
- To prevent the collision between the data in the hash map, we use a singly linked list.
- If we ever noticed the functioning of a casual notepad, it also uses a singly linked list to perform undo or redo or deleting functions.
- We can think of its use in a photo viewer for having look at photos continuously in a slide show.
- In the system of train, the idea is like a singly linked list, as if you want to add a Boggie, either you have to take a new boggie to add at last or you must spot a place in between boggies and add it.