**REPORT ON LINKED LISTS**

**PROGRAMMING DATA STRUCTURES AND ALGORITHMS**

Course-Work 02

National Institute of Business Management

Kurunegala

HDSE 23.2F

26/09/2023

**KUHDSE23.2F-001 – E.M.C.S.B. Ekanayaka**

**KUHDSE23.2F-002 - R.M.C.N. Rathnayaka**

**KUHDSE23.2F-003 – M.S.F. Shazna**

**KUHDSE23.2F-004 – N.P.U.L. Nugawela**

**Contents**

[INTRODUCTION 3](#_Toc145926622)

[What is a Linked List? 3](#_Toc145926623)

[TYPES OF LINKED LISTS 4](#_Toc145926624)

[1.Singly Linked List 4](#_Toc145926625)

[2. Doubly Linked List 5](#_Toc145926626)

[3. Circular Linked List 6](#_Toc145926627)

[EXAMPLES OF LINKED LISTS 7](#_Toc145926628)

[1. Singly Linked List 7](#_Toc145926629)

[2. Doubly Linked List 9](#_Toc145926630)

[3.Circular Linked List 12](#_Toc145926631)

[Circular Singly Linked List 12](#_Toc145926632)

[Circular Doubly Linked List 13](#_Toc145926633)

[IMPLEMENTATION 15](#_Toc145926634)

[Insert 15](#_Toc145926635)

[Singly Linked List 15](#_Toc145926636)

[Doubly Linked List 16](#_Toc145926637)

[Circular Singly Linked List 17](#_Toc145926638)

[Circular Doubly Linked List 18](#_Toc145926639)

[Delete 19](#_Toc145926640)

[Singly Linked List 19](#_Toc145926641)

[Doubly Linked List 20](#_Toc145926642)

[Circular Singly Linked List 21](#_Toc145926643)

[Circular Doubly Linked List 22](#_Toc145926644)

# **INTRODUCTION**

Linked lists are fundamental data structures used in computer science and programming.

They provide a flexible way to store and manipulate data, making them essential in various applications and algorithms.

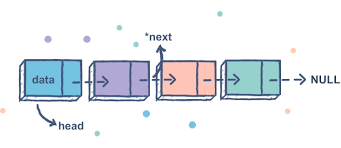
This report aims to provide an overview of linked lists, their types, operations, advantages, disadvantages, and common use cases.

## What is a Linked List?

A linked list is a linear data structure consisting of a sequence of elements, each of which is called a node.

Unlike arrays, where elements are stored in contiguous memory locations, linked list elements are connected via pointers, forming a chain-like structure.

Each node contains two parts: the data and a reference (or pointer) to the next node in the sequence.



# **TYPES OF LINKED LISTS**

### **1.Singly Linked List**

* This kind of linked list is unidirectional, that means it can only be traversed in one direction from initial node called **Head** to last node called **Tail**.
* A single node contains data and a pointer to the next node which helps in maintaining the structure of the list. Here, each node points to the next node in the sequence, but there is no pointer to the previous node.
* The first node (Head) helps to the access every other element in the list.
* The last node (Tail) points to NULL which determines the end of the list.

|  |  |
| --- | --- |
| **Advantages** | **Disadvantages** |
| Simple Implementation | Limited Backward Traversal |
| Memory and Space Efficiency | Extra Memory Overhead |
| Flexibility in Traversal | Difficulty in finding the last element |
| Dynamic Size | Costly Deletion Operation |



### **2. Doubly Linked List**

* This is kind of a linked list is bidirectional which means navigation allows traversal in both directions.
* Doubly Linked List contains a link element called **First** and **Last**.
* Each link carries a data field and a link field called next.
* Each link is linked with its next link using its next link.
* Each link is linked with its previous link using its previous link.
* A doubly linked list extends the functionality of a singly linked list by including pointers to both the next node called **Next** and previous node called **Prev**.
* The last link carries a link as NJLL to mark the end of the list.

|  |  |
| --- | --- |
| **Advantages** | **Disadvantages** |
| Bidirectional Traversal | Increased Memory Usage |
| Enhanced flexibility in implementations | Complexity in Implementation |
| Improved error handling | Slower Traversal |
| Simplified Reversal of elements | Reduced Space Efficiency |

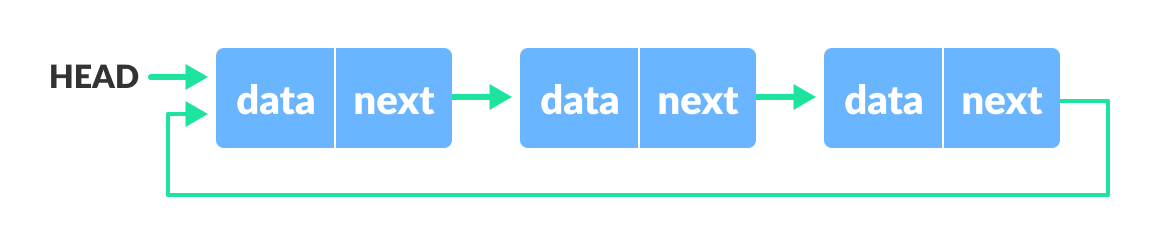


### **3. Circular Linked List**

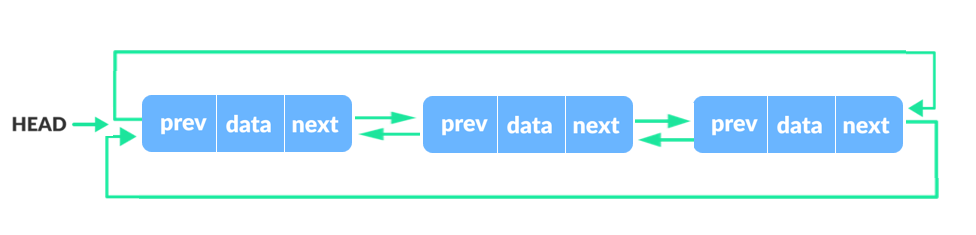
* The Circular linked list is where all the nodes are connected to form a circle.
* The first node called **Head** and last node called **Tail** are connected to each other forming a loop.
* There is no NULL at the end.
* This can be useful in applications where continuous cyclic access is required, such as scheduling algorithms.
* There are two types of circular linked lists:

1. **Circular Singly linked list**

The last node of the list contains a pointer to the first node of the list. We traverse the circular singly linked list until we reach the same node where we started. The circular singly linked list has no beginning or end. No null value is present in the next part of any of the nodes.



1. **Circular Doubly linked list**

Two consecutive elements are linked by the previous and next pointer and the last node points to the first node by the next pointer and also the first node points to the last node by the previous pointer.

|  |  |
| --- | --- |
| **Advantages** | **Disadvantages** |
| Accessibility of a node | Possibility of an infinite loop |
| Easy traversal | Complex compared to singly linked lists |
| Efficient memory usage | Harder to find the end of the list |
| Implementation of circular buffer | More memory is needed because it happens in run-time |

## EXAMPLES OF LINKED LISTS

### **Singly Linked List**

**A singly linked list executed using Python**

Here a single node is created since linking several nodes gives us a complete list.

For this, 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.

Here the value assigned to the node is 3.

class Node:

**# constructor**

def \_\_init\_\_(self, data = None, next=None):

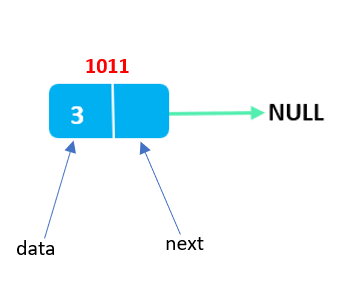
self.data = data

self.next = next

**# Creating a single node**

first = Node(3)

print(first.data)



Here a singly linked list is made with two nodes: `node1` containing 3 and `node2` containing 2. `node1` points to `node2`, forming the list with `node1` as the head.

class Node:

def \_\_init\_\_(self, data=None, next\_node=None):

self.data = data

self.next = next\_node

class LinkedList:

def \_\_init\_\_(self):

self.head = None

**# Creating nodes with values**

node2 = Node(2)

node1 = Node(3)

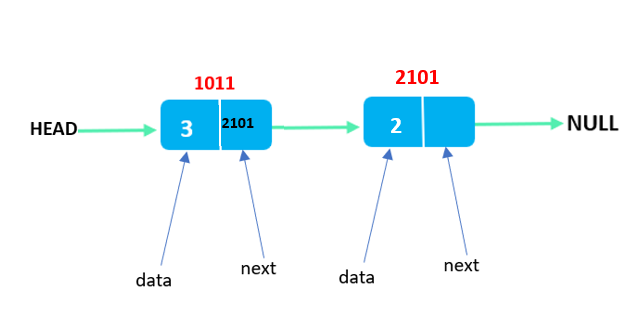
**# Linking nodes together**

node1.next = node2

**# Creating a linked list with the head pointing to the first node**

LL = LinkedList()

LL.head = node1



The places where singly linked lists are used is:

1. **Implementing stacks and queues:** Singly linked lists can be used to implement stacks and queues. In a stack, elements are added and removed from one end of the list, while in a queue, elements are added at one end and removed from the other end of the list.
2. **Navigation in web browsers:** Singly linked lists can be used to store the browsing history in web browsers. Each URL visited is stored as a node in the list, with the next pointer pointing to the next URL visited.
3. **Navigation of images in social media:** Just like playlist of song, singly linked list is used in image viewer in which each image represents a node and we can view one image after the other.

### **Doubly Linked List**

**A doubly linked list executed using Python**

Here a single node is created since linking several nodes gives us a complete list.

For this, we make a Node class that holds some data and two pointers next and prev, that will be used to point to the next Node and the previous node type object in the Linked List.

Here the value assigned to the node is 3.

**# Creating a node class**

class Node:

def \_\_init\_\_(self, data):

self.data = data **#adding an element to the node**

self.next = None **# not linked with any other node**

self.prev = None **# not be linked in either direction**

class DoublyLinkedList:

def \_\_init\_\_(self):

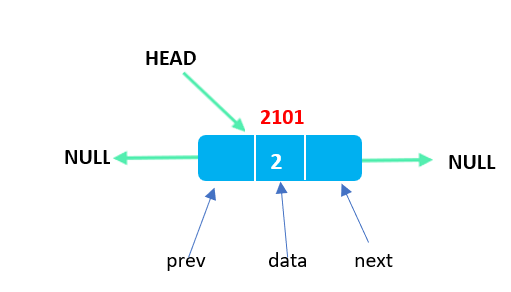
self.head = None **# Initially there are no elements in the list**

**# Creating a doubly linked list**

doubly\_linked\_list = DoublyLinkedList()

**# Assigning the value 2 to the head**

doubly\_linked\_list.head = Node(2)



Here define a `Node` class for individual list elements, each equipped with data, a `next` pointer to the next node, and a `prev` pointer to the previous node.

The `DoublyLinkedList` class acts as the container for the linked list, initially with an empty `head`.

Three nodes, `node\_1`, `node\_2`, and `node\_3`, are created, linked sequentially, and `node\_1` is designated as the `head`. The code then displays the doubly linked list by traversing it from the `head`.

class Node:

def \_\_init\_\_(self, data):

self.data = data

self.next = None

self.prev = None

class DoublyLinkedList:

def \_\_init\_\_(self):

self.head = None

**# Creating a doubly linked list**

doubly\_linked\_list = DoublyLinkedList()

**# Creating nodes with values**

node\_1 = Node(2)

node\_2 = Node(1)

node\_3 = Node(3)

**# Linking nodes together**

node\_1.next = node\_2 **# Link node\_1 to node\_2**

node\_2.prev = node\_1

node\_2.next = node\_3

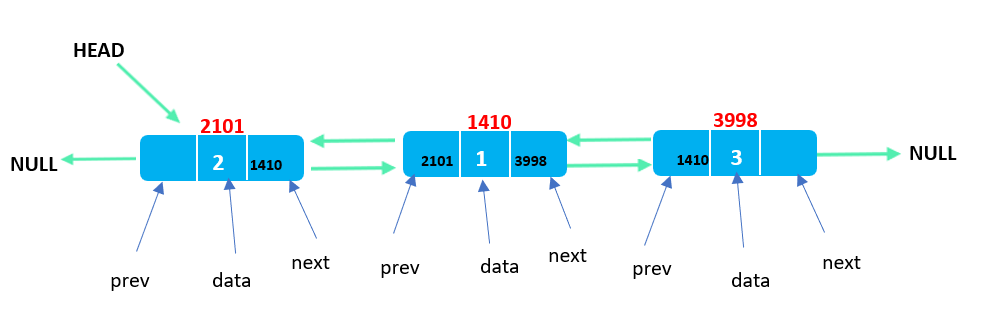
node\_3.prev = node\_2

**# Setting node\_1 as the head of the list**

doubly\_linked\_list.head = node\_1

**# Displaying the list**

current = doubly\_linked\_list.head



The places where doubly linked lists are used is:

1. **Undo/Redo Functionality:** Doubly linked lists are commonly employed to implement undo and redo functionality in various applications. Each state change is stored as a node, enabling users to move both forward and backward through the history of changes.
2. **Text Editors**: Text editors can use doubly linked lists to represent text documents. Each line or paragraph is stored as a node, allowing users to navigate through the document bidirectionally for editing and viewing purposes.
3. **Music Playlist**: Doubly linked lists can be used to create music playlists. Each song represents a node, enabling users to move through the playlist in both directions, facilitating playback and management.
4. **Task** **Management**: In task management applications, doubly linked lists can be employed to represent tasks. Users can navigate through their task lists in both directions, making it convenient for marking tasks as completed or revisiting previous tasks.

### **3.Circular Linked List**

### **Circular Singly Linked List**

**A Circular Singly linked list executed using Python**

**# Add the first node.**

first = Node(10)

**# Linking with head node**

MyList.head = first

**# Linking next of the node with head to make it circular**

first.next = MyList.head

**# Add the second node.**

second = Node(20)

**# Linking with the first node**

second.next = MyList.head

**# Update the head to point to the second node**

MyList.head = second

**# Add the third node.**

third = Node(30)

**# Linking with the second node**

third.next = MyList.head

**# Update the next pointer of the last node**

first.next = third

Initially, an empty linked list, `MyList`, is instantiated.

Three nodes, each containing values 10, 20, and 30, are then added.

The `next` pointers of these nodes are intended to create a circular linkage, where the last node points back to the head node, forming a loop.

**class Node:**

def \_\_init\_\_(self, data):

self.data = data

self.next = None

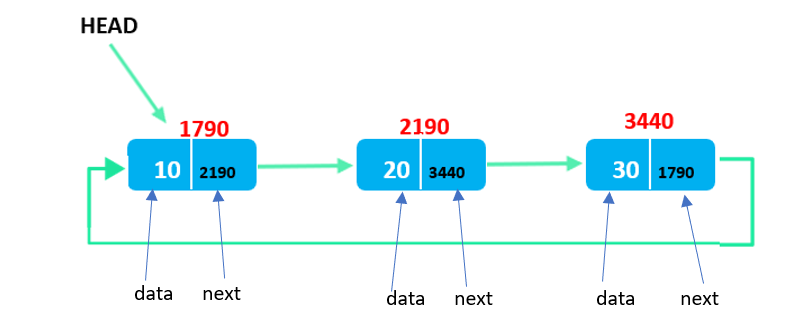
**class LinkedList:**

def \_\_init\_\_(self):

self.head = None

**# Create an empty LinkedList**

MyList = LinkedList()



### **Circular Doubly Linked List**

**A Circular doubly linked list executed using Python**

**# Add first node.**

first = Node(10)

**# linking with head node**

MyList.head = first

**# linking next of the node with head**

first.next = MyList.head

**# linking prev of the head**

MyList.head.prev = first

**# Add second node.**

second = Node(20)

first.next = second

second.prev = first

second.next = MyList.head

MyList.head.prev = second

**# Add third node.**

third = Node(30)

second.next = third

third.prev = second

third.next = MyList.head

**# linking prev of the head**

MyList.head.prev = third

This code constructs a doubly linked circular linked list by defining nodes with values 10, 20, and 30.

It ensures that the "next" and "prev" pointers are correctly updated to establish the circular structure.

This results in a doubly linked list where each node is linked both forward and backward, forming a closed loop.

class Node:

def \_\_init\_\_(self, data):

self.data = data

self.next = None

self.prev = None

class LinkedList:

**# constructor to create an empty LinkedList**

def \_\_init\_\_(self):

self.head = None

**# Create an empty LinkedList**

MyList = LinkedList()

**HEAD**



**2101**



**NULL**

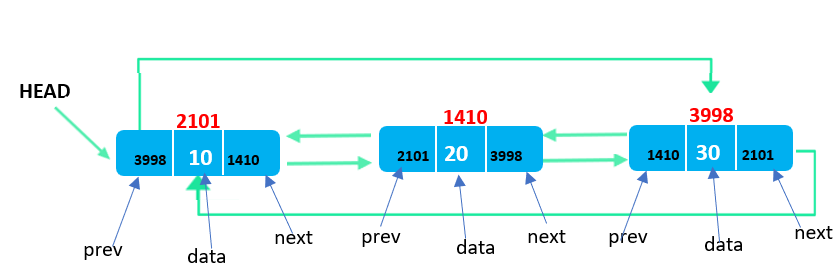
**NULL**

**2**

prev

next

data



The places where circular singly linked lists are used is:

1. **Music and Playlist Management:** Circular singly linked lists can be employed to manage playlists in media players. Songs are represented as nodes, and the "next" pointer links each song to the next one in the playlist. When you reach the end of the playlist, it loops back to the beginning, creating a seamless music playback experience.
2. **File Systems**: In some file systems, circular singly linked lists are used to manage free disk space. Each free block is represented as a node, and when space is allocated, the block is removed from the list. When the space is freed, the block is added back to the list. The circular structure ensures efficient space allocation and reuse.
3. **Game Development**: Circular singly linked lists can be used in game development for managing game elements that need to cycle through a predefined sequence. For example, in a game with power-ups, the power-ups can be organized in a circular list, allowing players to cycle through them.

The places where circular doubly linked lists are used is:

1. **Undo/Redo Functionality**: In software applications, like text editors or graphic design software, where users need to undo or redo actions, doubly circular linked lists can be employed. Each node represents a state or action, and users can navigate backward (undo) or forward (redo) through the list of states.
2. **Cache Management:** Doubly circular linked lists are used in caching mechanisms, such as the LRU (Least Recently Used) cache. In an LRU cache, the most recently used items are moved to the front of the list, while the least recently used items are at the end. When the cache reaches its capacity, items from the end of the list (the least recently used) are removed.
3. **Navigation in Circular Data Structures**: In data structures like circular queues, where elements are processed in a circular manner, doubly circular linked lists are beneficial. Each node can represent an element in the queue, and forward and backward traversal enables efficient processing.

## IMPLEMENTATION

## Insert

### **Singly Linked List**

**def** display(self):

**#Node current will point to head**

        current = self.head;

**if**(self.head == None):

**print**("List is empty");

**return**;

**print**("Nodes of singly linked list: ");

**while**(current != None):

**#Prints each node by incrementing pointer**

**print**(current.data),

            current = current.next;

sList = SinglyLinkedList();

**#Add nodes to the list**

sList.addNode(1);

sList.addNode(2);

sList.addNode(3);

sList.addNode(4);

**#Displays the nodes present in the list**

sList.display();

**class** Node:

**def** \_\_init\_\_(self,data):

        self.data = data;

        self.next = None;

**class** SinglyLinkedList:

**def** \_\_init\_\_(self):

        self.head = None;

        self.tail = None;

**#addNode() will add a new node to the list**

**def** addNode(self, data):

**#Create a new node**

        newNode = Node(data);

**#Checks if the list is empty**

**if**(self.head == None):

**#If list is empty, both head and tail will point to new node**

            self.head = newNode;

            self.tail = newNode;

**else**:

            self.tail.next = newNode;

**#newNode will become new tail of the list**

            self.tail = newNode;

**#display() will display all the nodes present**

**in the list**

OUTPUT IS:

**1 2 3 4**

### **Doubly Linked List**

**#Add newNode as new head of the list**

**else**:

**#head's previous node will be newNode**

            self.head.previous = newNode;

**#newNode's next node will be head**

            newNode.next = self.head;

**#newNode's previous will point to None**

            newNode.previous = None;

**#newNode will become new head**

            self.head = newNode;

**def** display(self):

**#Node current will point to head**

        current = self.head;

**if**(self.head == None):

**print**("List is empty");

**return**;

**print**("Adding a node to the start of the list: ");

**while**(current != None):

**print**(current.data),

            current = current.next;

**print**();

dList = InsertStart();

**#Adding  to the list**

dList.addAtStart(1);

dList.display();

dList.addAtStart(2);

dList.display();

dList.addAtStart(3);

dList.display();

dList.addAtStart(4);

dList.display();

dList.addAtStart(5);

dList.display();

**class** Node:

**def** \_\_init\_\_(self,data):

        self.data = data;

        self.previous = None;

        self.next = None;

**class** InsertStart:

**#Represent the head and tail of the doubly linked list**

**def** \_\_init\_\_(self):

        self.head = None;

        self.tail = None;

**#addAtStart() will add a node to the starting of the list**

**def** addAtStart(self, data):

**#Create a new node**

        newNode = Node(data);

**#If list is empty**

**if**(self.head == None):

**#Both head and tail will point to newNode**

            self.head = self.tail = newNode;

**#head's previous will point to None**

            self.head.previous = None;

**#tail's next will point to None, as it is the last**

**node of the list**

            self.tail.next = None;

OUTPUT IS:

Adding a node to the start of the list**: 5 4 3 2 1**

### **Circular Singly Linked List**

**class** Node:

**def** \_\_init\_\_(self, data):

self.data = data

self.next = None

**class** LinkedList:

def \_\_init\_\_(self):

self.head = None

**#Add new element at the start of the list**

**def** push\_front(self, newElement):

newNode = Node(newElement)

**if**(self.head == None):

self.head = newNode

newNode.next = self.head

return

**else:**

temp = self.head

**while**(temp.next != self.head):

temp = temp.next

temp.next = newNode

newNode.next = self.head

self.head = newNode

**#display the content of the list**

**def** PrintList(self):

temp = self.head

**if**(temp != None):

print("The list contains:", end=" ")

**while** (True):

print(temp.data, end=" ")

temp = temp.next

**if**(temp == self.head):

**break**

print()

**else:**

print("The list is empty.")

**# test the code**

MyList = LinkedList()

**#Add three elements at the start of the list.**

MyList.push\_front(10)

MyList.push\_front(20)

MyList.push\_front(30)

MyList.PrintList()

OUTPUT IS:

**30 20 10**

### **Circular Doubly Linked List**

**class** Node:

**def** \_\_init\_\_(self, data):

self.data = data

self.next = None

self.prev = None

**class** LinkedList:

**def** \_\_init\_\_(self):

self.head = None

**#Add new element at the start of the list**

**def** push\_front(self, newElement):

newNode = Node(newElement)

**if**(self.head == None):

self.head = newNode

newNode.next = self.head

newNode.prev = self.head

return

**else:**

temp = self.head

**while**(temp.next != self.head):

temp = temp.next

temp.next = newNode

NewNode.prev = temp

newNode.next = self.head

self.head.prev = newNode

self.head = newNode

**#display the content of the list**

**def** PrintList(self):

temp = self.head

**if**(temp != None):

print("The list contains:", end=" ")

**while** (True):

print(temp.data, end=" ")

temp = temp.next

**if**(temp == self.head):

**break**

print()

**else:**

print("The list is empty.")

**# test the code**

MyList = LinkedList()

**#Add three elements at the start of the list.**

MyList.push\_front(10)

MyList.push\_front(20)

MyList.push\_front(30)

MyList.PrintList()

OUTPUT IS:

**30 20 10**

## Delete

### **Singly Linked List**

**class** Node:

**def** \_\_init\_\_(self, data):

self.data = data

self.next = None

**class** LinkedList:

**def** \_\_init\_\_(self):

self.head = None

**#Add new element at the end of the list**

**def** push\_back(self, newElement):

newNode = Node(newElement)

**if**(self.head == None):

self.head = newNode

return

**else:**

temp = self.head

**while**(temp.next != None):

temp = temp.next

temp.next = newNode

**#Delete first node of the list**

**def** pop\_front(self):

**if**(self.head != None):

temp = self.head

self.head = self.head.next

temp = None

**#display the content of the list**

**def** PrintList(self):

temp = self.head

**if**(temp != None):

print("The list contains:", end=" ")

**while** (temp != None):

print(temp.data, end=" ")

temp = temp.next

print()

**else:**

print("The list is empty.")

MyList = LinkedList()

**#Add four elements in the list**.

MyList.push\_back(10)

MyList.push\_back(20)

MyList.push\_back(30)

MyList.push\_back(40)

MyList.PrintList()

**#Delete the first node**

MyList.pop\_front()

MyList.PrintList()

OUTPUT IS:

List contains: **10 20 30 40**

New list contains: **20 30 40**

### **Doubly Linked List**

**if**(self.head != None):

self.head.prev = None

**#display the content of the list**

**def** PrintList(self):

temp = self.head

**if**(temp != None):

print("The list contains:", end=" ")

**while** (temp != None):

print(temp.data, end=" ")

temp = temp.next

print()

**else:**

print("The list is empty.")

**# test the code**

MyList = LinkedList()

**#Add four elements in the list.**

MyList.push\_back(10)

MyList.push\_back(20)

MyList.push\_back(30)

MyList.push\_back(40)

MyList.PrintList()

**#Delete the first node**

MyList.pop\_front()

MyList.PrintList()

**class** Node:

**def** \_\_init\_\_(self, data):

self.data = data

self.next = None

self.prev = None

**class** LinkedList:

**def** \_\_init\_\_(self):

self.head = None

**#Add new element at the end of the list**

**def** push\_back(self, newElement):

newNode = Node(newElement)

if(self.head == None):

self.head = newNode

return

**else:**

temp = self.head

**while**(temp.next != None):

temp = temp.next

temp.next = newNode

newNode.prev = temp

**#Delete first node of the list**

**def** pop\_front(self):

**if**(self.head != None):

temp = self.head

self.head = self.head.next

temp = None

OUTPUT IS:

List contains: **10 20 30 40**

New list contains: **20 30 40**

### **Circular Singly Linked List**

**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

**#display the content of the list**

**def** PrintList(self):

temp = self.head

**if**(temp != None):

print("The list contains:", end=" ")

**while** (True):

print(temp.data, end=" ")

temp = temp.next

**if**(temp == self.head):

break

print()

**else:**

print("The list is empty.")

MyList = LinkedList()

**#Add four elements in the list.**

MyList.push\_back(10)

MyList.push\_back(20)

MyList.push\_back(30)

MyList.push\_back(40)

MyList.PrintList()

**#Delete the first node**

MyList.pop\_front()

MyList.PrintList()

**class** Node:

**def** \_\_init\_\_(self, data):

self.data = data

self.next = None

**class** LinkedList:

**def** \_\_init\_\_(self):

self.head = None

**#Add new element at the end of the list**

**def** push\_back(self, newElement):

newNode = Node(newElement)

**if**(self.head == None):

self.head = newNode

newNode.next = self.head

return

**else:**

temp = self.head

**while**(temp.next != self.head):

temp = temp.next

temp.next = newNode

newNode.next = self.head

**#Delete first node of the list**

**def** pop\_front(self):

**if**(self.head != None):

**if**(self.head.next == self.head):

self.head = None

OUTPUT IS:

List contains: **10 20 30 40**

New list contains: **20 30 40**

### **Circular Doubly Linked List**

**else:**

temp = self.head

firstNode = self.head

**while**(temp.next != self.head):

temp = temp.next

self.head = self.head.next

self.head.prev = temp

temp.next = self.head

firstNode = None

**#display the content of the list**

**def** PrintList(self):

temp = self.head

**if**(temp != None):

print("The list contains:", end=" ")

**while** (True):

print(temp.data, end=" ")

temp = temp.next

**if**(temp == self.head):

break

print()

**else:**

print("The list is empty.")

MyList = LinkedList()

**#Add three elements at the end of the list.**

MyList.push\_back(10)

MyList.push\_back(20)

MyList.push\_back(30)

MyList.push\_back(40)

MyList.PrintList()

**#Delete the first node**

MyList.pop\_front()

MyList.PrintList()

**class** Node:

**def** \_\_init\_\_(self, data):

self.data = data

self.next = None

self.prev = None

**class** LinkedList:

**def** \_\_init\_\_(self):

self.head = None

**#Add new element at the end of the list**

**def** push\_back(self, newElement):

newNode = Node(newElement)

**if**(self.head == None):

self.head = newNode

newNode.next = self.head

newNode.prev = self.head

return

**else:**

temp = self.head

**while**(temp.next != self.head):

temp = temp.next

temp.next = newNode

newNode.next = self.head

newNode.prev = temp

self.head.prev = newNode

**#Delete first node of the list**

**def** pop\_front(self):

**if**(self.head != None):

**if**(self.head.next == self.head):

self.head = None

OUTPUT IS:

List contains: **10 20 30 40**

New list contains: **20 30 40**

In conclusion, linked lists are fundamental data structures that play a crucial role in computer science and programming. They offer flexibility in data storage and manipulation, making them essential for various applications and algorithms. Linked lists come in different forms, including singly linked lists, doubly linked lists, and circular linked lists, each with its own advantages and use cases.

Singly linked lists are simple to implement and memory-efficient, making them suitable for stacks, queues, and navigation in web browsers. Doubly linked lists provide bidirectional traversal, enabling applications like undo/redo functionality and LRU caching. Circular linked lists, whether singly or doubly, allow for seamless looping and are used in scenarios requiring cyclic behavior.

Linked lists are dynamic in size, making them adaptable to changing data needs, but they have trade-offs such as limited random access and memory overhead. Choosing the right type of linked list depends on the specific requirements of a problem.

Overall, linked lists are foundational data structures that form the building blocks for more complex data structures and algorithms, and a solid understanding of them is essential for any programmer or computer scientist.

**A MUSIC PLAYER DEVELOPED USING PYTHON**

Here a Doubly linked lists is used.

import tkinter as tk

import fnmatch

import os

from pygame import mixer

canvas = tk.Tk()

canvas.title("Music Player")

canvas.geometry("600x800")

canvas.config(bg='black')

rootpath = "C:\\Users\\shazna salman\\Desktop\\PDSA\\musicapp\\music"

pattern = "\*.mp3"

mixer.init()

prv\_img = tk.PhotoImage(file="prv.png")

stop\_img = tk.PhotoImage(file="stop.png")

play\_img = tk.PhotoImage(file="play.png")

pause\_img = tk.PhotoImage(file="pause.png")

next\_img = tk.PhotoImage(file="next.png")

def select(event=None):

if listBox.curselection():

selected\_song = listBox.get(listBox.curselection())

label.config(text=selected\_song)

mixer.music.load(os.path.join(rootpath, selected\_song))

mixer.music.play()

def stop():

mixer.music.stop()

listBox.select\_clear(0, 'end')

def pla\_next():

current\_selection = listBox.curselection()

if current\_selection:

next\_song\_index = current\_selection[0] + 1

if next\_song\_index < listBox.size():

next\_song\_name = listBox.get(next\_song\_index)

label.config(text=next\_song\_name)

mixer.music.load(os.path.join(rootpath, next\_song\_name))

mixer.music.play()

listBox.select\_clear(0, 'end')

listBox.select\_set(next\_song\_index)

def pla\_prv():

current\_selection = listBox.curselection()

if current\_selection:

prev\_song\_index = current\_selection[0] - 1

if prev\_song\_index >= 0:

prev\_song\_name = listBox.get(prev\_song\_index)

label.config(text=prev\_song\_name)

mixer.music.load(os.path.join(rootpath, prev\_song\_name))

mixer.music.play()

listBox.select\_clear(0, 'end')

listBox.select\_set(prev\_song\_index)

def pause\_song():

if pauseButton["text"] == "Pause":

mixer.music.pause()

pauseButton["text"] = "Play"

else:

mixer.music.unpause()

pauseButton["text"] = "Pause"

listBox = tk.Listbox(canvas, fg="cyan", bg="black", width=100, font=('ds-digital', 14))

listBox.pack(padx=15, pady=15)

listBox.bind('<<ListboxSelect>>', select)

label = tk.Label(canvas, text='', bg='black', fg='yellow', font=('ds-digital', 14))

label.pack(pady=15)

top = tk.Frame(canvas, bg="black")

top.pack(padx=10, pady=5, anchor='center')

prevButton = tk.Button(canvas, text="Prev", image=prv\_img, bg='black', borderwidth=0, command=pla\_prv)

prevButton.pack(pady=15, in\_=top, side='left')

stopButton = tk.Button(canvas, text="Stop", image=stop\_img, bg='black', borderwidth=0, command=stop)

stopButton.pack(pady=15, in\_=top, side='left')

playButton = tk.Button(canvas, text="Play", image=play\_img, bg='black', borderwidth=0, command=select)

playButton.pack(pady=15, in\_=top, side='left')

pauseButton = tk.Button(canvas, text="Pause", image=pause\_img, bg='black', borderwidth=0, command=pause\_song)

pauseButton.pack(pady=15, in\_=top, side='left')

nextButton = tk.Button(canvas, text="Next", image=next\_img, bg='black',

borderwidth=0, command=pla\_next)

nextButton.pack(pady=15, in\_=top, side='left')

for root, dirs, files in os.walk(rootpath):

for filename in fnmatch.filter(files, pattern):

listBox.insert('end', filename)

canvas.mainloop()

