Doubly Linked List

In [22]:
```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

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

# Adding data elements
    def insert_begin(self, data):
        NewNode = Node(data)
        if self.head!=None:
            NewNode.next = self.head
            self.head.prev=NewNode
            self.head = NewNode
        else:
            self.head=NewNode

    def insert_end(self,data):
        NewNode = Node(data)
        last_node=self.head
        while last_node.next:
            last_node=last_node.next
        NewNode.prev=last_node
        last_node.next=NewNode
    def insert_index(self,index,data):
        NewNode = Node(data)
        if index==0:
            self.insert_begin(data)
        else:
            prev_node=self.head
            c_p=0
            while c_p<index-1 and prev_node.next:
                prev_node=prev_node.next
                c_p+=1
            NewNode.next=prev_node.next
            prev_node.next.prev=NewNode
            prev_node.next=NewNode
            NewNode.prev=prev_node
#Deletion
    def delete_begin(self):
        if self.head!=None:
            self.head.next.prev=None
            self.head = self.head.next

    def delete_end(self):
        last_node=self.head
        while last_node.next.next:
            last_node=last_node.next
        #last_node.next.prev=None[Not mandatory]
        last_node.next=None
    def delete_index(self,index):
```

```
            prev_node=self.head
            count=0
            while count<index-1 and prev_node.next:
                prev_node=prev_node.next
                count+=1
            #prev_node.next.prev=None   [Not mandatory]
            #prev_node.next.next=None   [Not Mandatory]
            prev_node.next=prev_node.next.next
            prev_node.next.prev=prev_node

# Print the Doubly Linked list
    def display(self):
        current_node=self.head
        while current_node:
            print(current_node.data,end='-->')
            current_node=current_node.next
        print('\n')
    def bw_display(self):
            last_node=self.head
            while last_node.next:
                last_node=last_node.next
            temp=last_node
            while temp.prev!=None:
                print(temp.data,end='-->')
                temp=temp.prev
            print(temp.data)

            print('\n')
# size of linked list
    def size(self):
        current_node=self.head
        count=0
        while current_node:
            current_node=current_node.next
            count+=1
        return count

dl = doubly_linked_list()
dl.insert_begin(2)
dl.insert_begin(20)
dl.insert_begin(12)
dl.insert_begin(34)
dl.insert_end(100)
dl.insert_end(200)
dl.insert_index(1,111)
dl.insert_index(0,222)

dl.insert_index(2,333)
dl.delete_begin()
dl.delete_end()
dl.delete_index(0)
dl.display()
dl.bw_display()
dl.size()
```

```
34-->111-->12-->20-->2-->100-->

100-->2-->20-->12-->111-->34
```

Out[22]: 6

Circular Singly Linked List

In [21]:
```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class CircularLinkedList:
    def __init__(self):
        self.head = None
    def get_LastNode(self):
        last_node=self.head
        while last_node.next!=self.head:
            last_node=last_node.next
        return last_node

    def insert_begin(self,data):
        NewNode=Node(data)
        if self.head==None:
            self.head=NewNode
            NewNode.next=self.head
        else:
            last_node=self.get_LastNode()
            last_node.next=NewNode
            NewNode.next=self.head
            self.head=NewNode
    def insert_end(self,data):
        NewNode=Node(data)
        if self.head==None:
            self.insert_begin(data)
        else:
            last_node=self.get_LastNode()
            last_node.next=NewNode
            NewNode.next=self.head
    def insert_index(self,index,data):
        NewNode=Node(data)
        if index==0:
            self.insert_begin(data)
        else:
            prev_node=self.head
            c_p=0
            while c_p<index-1 and prev_node.next:
                prev_node=prev_node.next
                c_p+=1
            NewNode.next=prev_node.next
            prev_node.next=NewNode
# Deleting
    def del_begin(self):
        if self.head!=None:
            self.head=self.head.next
            last_node=self.get_LastNode()
            last_node.next=self.head
    def del_end(self):
        last_node=self.head
        while last_node.next.next!=self.head:
```

```
            last_node=last_node.next
        last_node.next=self.head
    def del_index(self,index):
        if index==0:
            self.del_begin()
        else:
            prev_node=self.head
            c_p=0

            while c_p<index-1:
                prev_node=prev_node.next
                c_p+=1
            ele=prev_node.next.data# storing the data to be deleted
            prev_node.next=prev_node.next.next
            print(f"Node deleted at index {index} with data {ele}")


    def display(self):
        temp=current_node=self.head
        while current_node.next!=temp:
            print(current_node.data,end='==>')
            current_node=current_node.next
        print(current_node.data)

cl=CircularLinkedList()
cl.insert_begin(10)
cl.insert_begin(20)
cl.insert_begin(30)
cl.insert_begin(40)
cl.insert_end(50)
cl.insert_end(70)
cl.insert_index(0,100)
cl.del_begin()
cl.del_end()
cl.del_end()
cl.display()
# cl.del_index(1)
cl.del_index(0)

cl.display()
```

```
40==>30==>20==>10==>50
30==>20==>10==>50==>40
```

Circular Doubly Linked List

In [2]:

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


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

    def insert_begin(self,data):
```

```
            NewNode=Node(data)
            if self.head==None:
                self.head=NewNode
                NewNode.next=self.head
                NewNode.prev=self.head
            else:
                last_node=self.get_LastNode()
                last_node.next=NewNode
                NewNode.prev=last_node
                NewNode.next=self.head
                self.head.prev=NewNode
                self.head=NewNode

    def insert_end(self,data):
        NewNode=Node(data)
        if self.head==None:
            self.insert_begin(data)
        else:
            last_node=self.get_LastNode()
            last_node.next=NewNode
            NewNode.prev=last_node
            self.head.prev=NewNode
            NewNode.next=self.head

    def insert_index(self,index,data):
        NewNode=Node(data)
        if index==0:
            self.insert_begin(data)
        else:
            prev_node=self.get_PrevNode(index)
            prev_node.next.prev=NewNode
            NewNode.next=prev_node.next
            NewNode.prev=prev_node
            prev_node.next=NewNode

    def get_LastNode(self):
        last_node=self.head
        while last_node.next!=self.head:
            last_node=last_node.next
        return last_node

    def get_PrevNode(self,index):
        prev_node=self.head
        c_p=0
        while c_p<index-1 and prev_node.next!=self.head:
            prev_node=prev_node.next
            c_p+=1
        return prev_node
# Forward Traversal
    def display(self):
        temp=current_node=self.head
        while current_node.next!=temp:
            print(current_node.data,end='==>')
            current_node=current_node.next
        print(current_node.data)
# Backward Traversal
    def r_display(self):
        last_node=self.get_LastNode()
        temp=last_node
        while last_node.prev!=temp:
```

```
            print(last_node.data,end='==>')
            last_node=last_node.prev
        print(last_node.data)

dcl=DoublyCircularLL()
dcl.insert_begin(10)
dcl.insert_begin(20)
dcl.insert_begin(30)
dcl.insert_end(40)
dcl.insert_end(50)
dcl.insert_end(60)

dcl.insert_index(3,100)
dcl.display()
dcl.r_display()
```

```
30==>20==>10==>100==>40==>50==>60
60==>50==>40==>100==>10==>20==>30
```

Stack Implementation using linkedlist

In [25]:

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

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

    def isEmpty(self):
        return self.head==None
    def push(self,data):
        NewNode=Node(data)
        if self.isEmpty():
            self.head=NewNode
        else:
            NewNode.next=self.head
            self.head=NewNode
    def pop(self):
        if self.isEmpty():
            raise Exception("Queue Underflow")
            return
        else:
            temp=self.head
            self.head=self.head.next
            return temp.data
    def display(self):
        current_node=self.head
        while current_node:
            print(current_node.data,end='==>')
            current_node=current_node.next
    def top(self):
        print("Top element of the stack is ", self.head.data)
        return self.head.data

s=Stack()
s.push(30)
```

```
s.push(20)
s.push(10)
s.pop()
s.pop()
s.pop()
# s.pop()
s.push(20)
s.display()
```

```
20==>
```

Queue implementation using linked list

In [44]:

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

class Queue:
    def __init__(self):
        self.head = None
    def isEmpty(self):
        return self.head==None
    def Enqueue(self,data):
        #Insert at end
        NewNode=Node(data)
        if self.head==None:
            self.head=NewNode
        else:
            last_node=self.get_LastNode()
            last_node.next=NewNode
    def Dequeue(self):
        if not self.isEmpty():
                temp=self.head.data
                self.head=self.head.next
                return temp
        else:
            raise Exception("Queue underflow!")

    def get_LastNode(self):
        last_node=self.head
        while last_node.next!=None:
            last_node=last_node.next
        return last_node
    def display(self):
        current_node=self.head
        while current_node:
            print(current_node.data,end='==>')
            current_node=current_node.next
    def front(self):
        print("Front element of the queue is ", self.head.data)
        return self.head.data
q=Queue()
q.Enqueue(10)
q.Enqueue(20)
q.Enqueue(30)
q.Dequeue()
q.front()
```

```
q.display()
```

```
Front element of the queue is  20
20==>30==>
```

Stack implementation using Singly LL

In [ ]:

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

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

    def is_empty(self):
        return self.head is None

    def push(self, data):
        new_node = Node(data)
        new_node.next = self.head
        self.head=new_node

    def pop(self):
        if self.is_empty():
            print("Stack Underflow!")
            return None
        data = self.head.data
        self.head = self.head.next
        return data

    def display(self):
        c_n=self.head
        while c_n:
            print(c_n.data,end='\n')
            c_n=c_n.next

def display_menu():
    print('\n')
    print("1. push")
    print("2. pop")
    print("3. Exit")

stack = Stack()

while True:
    display_menu()
    choice = int(input("Enter your choice: "))

    if choice == 1:
        data = input("Enter the data to Push into stack: ")
        stack.push(data)
        stack.display()
    elif choice == 2:
        data = stack.pop()
```

```
        if data is not None:
            print("ele popped out is ", data,'\n')
            print("Updated stack is:\n")
            stack.display()
        else:
            print("stack is empty")
    elif choice == 3:
        print("Exiting...")
        break
    else:
        print("Invalid choice. Please try again.")
```

1. push
2. pop
3. Exit

Menu based Queue implementation using Singly LL

In [ ]:

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


class Queue:
    def __init__(self):
        self.head = None
        self.tail = None

    def is_empty(self):
        return self.head is None
    def getLastNode(self):
        c_n=self.head
        while c_n.next:
            c_n=c_n.next
        return c_n

    def enqueue(self, data):
        new_node = Node(data)
        if self.is_empty():
            self.head = new_node
        else:
            last_node=self.getLastNode()
            last_node.next = new_node

    def dequeue(self):
        if self.is_empty():
            print("Queue Underflow!")
            return None

        data = self.head.data
        self.head = self.head.next

        return data
```

```
    def display(self):
        c_n=self.head
        while c_n:
            print(c_n.data,end='==>')
            c_n=c_n.next



def display_menu():
    print('\n')
    print("1. Enqueue")
    print("2. Dequeue")
    print("3. Exit")


queue = Queue()

while True:
    display_menu()
    choice = int(input("Enter your choice: "))

    if choice == 1:
        data = input("Enter the data to enqueue: ")
        queue.enqueue(data)
        queue.display()


    elif choice == 2:
        data = queue.dequeue()
        if data is not None:
            print("Dequeued", data,'\n')
            print("Updated list is:")
            queue.display()
        else:
            print("Queue is empty")
    elif choice == 3:
        print("Exiting...")
        break
    else:
        print("Invalid choice. Please try again.")
```

In [ ]: