II - I

DATA STRUCTURES

Linked Lists

Module 3 QB Solutions

Hitesh • Rishi • Ujjwal

MODULE 3

PART A

1) Write a program to count the number of occurrences of an element in the linked list without using recursion.

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

class LinkedList:
    def __init__(self):
        self.head = None
        self.last_node = None

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

    def display(self):
```

```
        current = self.head
        while current:
            print(current.data, end = ' ')
            current = current.next

    def count(self, key):
        current = self.head

        count = 0
        while current:
            if current.data == key:
                count = count + 1
            current = current.next

        return count

a_llist = LinkedList()
for data in [5, 1, 3, 5, 5, 15, 4, 9, 2]:
    a_llist.append(data)
print('The linked list: ', end = '')
a_llist.display()
print()

key = int(input('Enter data item: '))
count = a_llist.count(key)
print('{0} occurs {1} time(s) in the list.'.format(key,
count))
```

2) Write a program to find the intersection and union of two linked lists.

```
# node for the linked list
class Node:
    def __init__(self, value=None, next=None):
        self.value = value
        self.next = next  # reference to the next node
# implementation of linked list
class LinkedList:
    def __init__(self):
        self.head = None
        self.length = 0  # total number of elements in the
linked list
    # adds a node at the start of the linked list
    def insert(self, value):
        node = Node(value, self.head)  # create a node
        self.head = node
        self.length = 1  # increment the length
# traverse the complete list and print each node's value
def displayList(ll):
    temp = ll
    while temp:
        print(temp.value, end="\t")
        temp = temp.next
    print("")
# check if a node with the given value exists in the linked
list
def itemExists(ll, value):
    temp = ll
```

```
    # iterate through the list
    while temp:
        if temp.value == value:  # element is found
            return True
        temp = temp.next
    return False
def union(ll1, ll2):
    result = LinkedList()  # create a new linked list
    # iterate over the first linked list and add its elements
in the result list
    temp = ll1
    while temp:
        result.insert(temp.value)
        temp = temp.next
    # iterate over the 2nd linked list and add those elements
in the result list that are not in result already
    temp = ll2
    while temp:
        if not itemExists(result.head, temp.value):
            result.insert(temp.value)
        temp = temp.next
    return result.head
def intersection(ll1, ll2):
    result = LinkedList()  # create a new linked list
    # iterate over the first linked list
    temp = ll1
    while temp:
        if itemExists(
            ll2, temp.value):  # if the current node's value
```

```
exists in the second linked list, then add it to the result
list
            result.insert(temp.value)
        temp = temp.next
    return result.head
# create linked list 1
ll1 = LinkedList()
ll1.insert(2)
ll1.insert(6)
ll1.insert(4)
ll1.insert(3)
# create linked list 2
ll2 = LinkedList()
ll2.insert(11)
ll2.insert(3)
ll2.insert(2)
union_ll = union(ll1.head, ll2.head)
intersection_ll = intersection(ll1.head, ll2.head)
print("Linked list 1")
displayList(ll1.head)
print("Linked list 2")
displayList(ll2.head)
print("Union of linked list 1 and 2")
displayList(union_ll)
print("Intersection of linked list 1 and 2")
displayList(intersection_ll)
```

3) Write a program to search for an element in the linked list without using recursion

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

class LinkedList:
    def __init__(self):
        self.head = None
        self.last_node = None

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

    def display(self):
        current = self.head
        while current is not None:
            print(current.data, end = ' ')
            current = current.next

    def find_index(self, key):
        current = self.head

        index = 0
        while current:
```

```
            if current.data == key:
                return index
            current = current.next
            index = index + 1

        return -1

a_llist = LinkedList()
for data in [4, -3, 1, 0, 9, 11]:
    a_llist.append(data)
print('The linked list: ', end = '')
a_llist.display()
print()

key = int(input('What data item would you like to search for?
'))
index = a_llist.find_index(key)
if index == -1:
    print(str(key) + ' was not found.')
else:
    print(str(key) + ' is at index ' + str(index) + '.')
```

4) Write a program to swap nodes in a linked list without swapping data?

```
class LinkedList(object):
    def __init__(self):
        self.head = None
```

```
# head of list
class Node(object):
    def __init__(self, d):
        self.data = d
        self.next = None

# Function to swap Nodes x and y in linked list by
# changing links
def swapNodes(self, x, y):

    # Nothing to do if x and y are same
    if x == y:
        return

    # Search for x (keep track of prevX and CurrX)
    prevX = None
    currX = self.head
    while currX != None and currX.data != x:
        prevX = currX
        currX = currX.next

    # Search for y (keep track of prevY and currY)
    prevY = None
    currY = self.head
    while currY != None and currY.data != y:
        prevY = currY
        currY = currY.next

    # If either x or y is not present, nothing to do
```

```
        if currX == None or currY == None:
            return
        # If x is not head of linked list
        if prevX != None:
            prevX.next = currY
        else: #make y the new head
            self.head = currY

        # If y is not head of linked list
        if prevY != None:
            prevY.next = currX
        else: # make x the new head
            self.head = currX

        # Swap next pointers
        temp = currX.next
        currX.next = currY.next
        currY.next = temp

    # Function to add Node at beginning of list.
    def push(self, new_data):

        # 1. alloc the Node and put the data
        new_Node = self.Node(new_data)

        # 2. Make next of new Node as head
        new_Node.next = self.head

        # 3. Move the head to point to new Node
```

```
        self.head = new_Node

    # This function prints contents of linked list starting
    # from the given Node
    def printList(self):
        tNode = self.head
        while tNode != None:
            print tNode.data,
            tNode = tNode.next

# Driver program to test above function
llist = LinkedList()

# The constructed linked list is:
# 1->2->3->4->5->6->7
llist.push(7)
llist.push(6)
llist.push(5)
llist.push(4)
llist.push(3)
llist.push(2)
llist.push(1)
print "Linked list before calling swapNodes() "
llist.printList()
llist.swapNodes(4, 3)
print "
Linked list after calling swapNodes() "
llist.printList()
```

5) Write a program to print the middle most node of a linked list.

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


class LinkedList:
    def __init__(self):
        self.head = None
        self.last_node = None

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


def print_middle(llist):
    current = llist.head
    length = 0
    while current:
        current = current.next
        length = length + 1

    current = llist.head
```

```
    for i in range((length - 1)//2):
        current = current.next

    if current:
        if length % 2 == 0:
            print('The two middle elements are {} and {}.'
                .format(current.data, current.next.data))
        else:
            print('The middle element is
{}.'.format(current.data))
    else:
        print('The list is empty.')


a_llist = LinkedList()

data_list = input('Please enter the elements in the linked
list: ').split()
for data in data_list:
    a_llist.append(int(data))

print_middle(a_llist)
```

6) Write a program to display node values in reverse order for a double linked list?

```
class Node:
```

```
    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None


class DoublyLinkedList:
     # Constructor for empty Doubly Linked List
    def __init__(self):
        self.head = None

    # Function reverse a Doubly Linked List
    def reverse(self):
        temp = None
        current = self.head

        # Swap next and prev for all nodes of
        # doubly linked list
        while current is not None:
            temp = current.prev
            current.prev = current.next
            current.next = temp
            current = current.prev

        # Before changing head, check for the cases like
        # empty list and list with only one node
        if temp is not None:
            self.head = temp.prev
```

```
    # Given a reference to the head of a list and an
    # integer,inserts a new node on the front of list
    def push(self, new_data):

        # 1. Allocates node
        # 2. Put the data in it
        new_node = Node(new_data)

        # 3. Make next of new node as head and
        # previous as None (already None)
        new_node.next = self.head

        # 4. change prev of head node to new_node
        if self.head is not None:
            self.head.prev = new_node

        # 5. move the head to point to the new node
        self.head = new_node

    def printList(self, node):
        while(node is not None):
            print node.data,
            node = node.next


# Driver code
dll = DoublyLinkedList()
dll.push(2)
```

```
dll.push(4)
dll.push(8)
dll.push(10)

print "\nOriginal Linked List"
dll.printList(dll.head)

# Reverse doubly linked list
dll.reverse()

print "\n Reversed Linked List"
dll.printList(dll.head)
```

7) Write a program to split a circular linked list into two halves?

```
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.next = None


# Class to create a new  Circular Linked list
class CircularLinkedList:

    # Constructor to create a empty circular linked list
    def __init__(self):
```

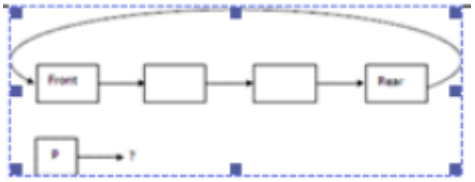
```
        self.head = None

    # Function to insert a node at the beginning of a
    # circular linked list
    def push(self, data):
        ptr1 = Node(data)
        temp = self.head

        ptr1.next = self.head

        # If linked list is not None then set the next of
        # last node
        if self.head is not None:
            while(temp.next != self.head):
                temp = temp.next
            temp.next = ptr1

        else:
            ptr1.next = ptr1 # For the first node

        self.head = ptr1

    # Function to print nodes in a given circular linked list
    def printList(self):
        temp = self.head
        if self.head is not None:
            while(True):
                print "%d" %(temp.data),
                temp = temp.next
```

```
            if (temp == self.head):
                break


    # Function to split a list (starting with head) into
    # two lists. head1 and head2 are the head nodes of the
    # two resultant linked lists
    def splitList(self, head1, head2):
        slow_ptr = self.head
        fast_ptr = self.head

        if self.head is None:
            return

        # If there are odd nodes in the circular list then
        # fast_ptr->next becomes head and for even nodes
        # fast_ptr->next->next becomes head
        while(fast_ptr.next != self.head and
            fast_ptr.next.next != self.head ):
            fast_ptr = fast_ptr.next.next
            slow_ptr = slow_ptr.next

        # If there are even elements in list then
        # move fast_ptr
        if fast_ptr.next.next == self.head:
            fast_ptr = fast_ptr.next

        # Set the head pointer of first half
        head1.head = self.head
```

```
        # Set the head pointer of second half
        if self.head.next != self.head:
            head2.head = slow_ptr.next

        # Make second half circular
        fast_ptr.next = slow_ptr.next

        # Make first half circular
        slow_ptr.next = self.head


# Driver program to test above functions

# Initialize lists as empty
head = CircularLinkedList()
head1 = CircularLinkedList()
head2 = CircularLinkedList()

head.push(12)
head.push(56)
head.push(2)
head.push(11)

print "Original Circular Linked List"
head.printList()

# Split the list
head.splitList(head1 , head2)
```

```
print "\nFirst Circular Linked List"
head1.printList()

print "\nSecond Circular Linked List"
head2.printList()
```

8) A circularly linked list is used to represent a Queue. A single variable p is used to access the Queue. Find the node to which p should point such that both the operations enQueue and deQueue can be performed in constant time?



As we cannot get rear from front in O(1), but if P is rear we can implement both enqueue and deQueue in O(1) because from rear we can get front in O(1). Below are sample functions. Shows The pointer points to the rear node : -

Enqueue :- Insert new node after rear, and make rear point to the newly inserted node :

Struct Node * New Node;

New Node → Next = Rear → Next,

Rear → Next = New Node;

Rear = New Node;

Dequeue :- Delete the front node, and make the second node the front node.

Rear → Next points to the front node.

Front → Next points to the second node.

Struct Node * Front;

Front = Rear → Next;

Rear → Next = Front → Next;

Free (front);

9) Define a node in a linked list? Explain the difference between creation of a single linked list node and double linked list node?

A node is a collection of two sub-elements or parts. A data part that stores the element and a next part that stores the link to the next node.

Single and double linked lists are two types of linked lists. The main difference between Single Linked List and Doubly Linked List is that a node in the single linked list stores the address of the next node while a node in a double linked list stores the address of the next node and the previous node.

10) Write a program to modify the linked list such that all even numbers appear before all the odd numbers in the modified linked list.

```
# Node class
class Node:

    # Function to initialise the node object
    def __init__(self, data):
        self.data = data # Assign data
```

```
        self.next =None

# Function to segregate even and odd nodes.
def segregateEvenOdd():
    global head

    # Starting node of list having
    # even values.
    evenStart = None

    # Ending node of even values list.
    evenEnd = None

    # Starting node of odd values list.
    oddStart = None

    # Ending node of odd values list.
    oddEnd = None

    # Node to traverse the list.
    currNode = head

    while(currNode != None):
        val = currNode.data

        # If current value is even, add
        # it to even values list.
        if(val % 2 == 0):
            if(evenStart == None):
```

```
            evenStart = currNode
            evenEnd = evenStart
        else:
            evenEnd . next = currNode
            evenEnd = evenEnd . next

    # If current value is odd, add
    # it to an odd values list.
    else:
        if(oddStart == None):
            oddStart = currNode
            oddEnd = oddStart
        else:
            oddEnd . next = currNode
            oddEnd = oddEnd . next

    # Move head pointer one step in
    # forward direction
    currNode = currNode . next

# If either odd list or even list is empty,
# no change is required as all elements
# are either even or odd.
if(oddStart == None or evenStart == None):
    return

# Add odd list after even list.
evenEnd . next = oddStart
oddEnd . next = None
```

```
    # Modify head pointer to
    # starting of even list.
    head = evenStart

''' UTILITY FUNCTIONS '''
''' Function to insert a node at the beginning '''
def push(new_data):

    global head
    # 1 & 2: Allocate the Node &
    #         Put in the data
    new_node = Node(new_data)

    # 3. Make next of new Node as head
    new_node.next = head

    # 4. Move the head to point to new Node
    head = new_node

''' Function to print nodes in a given linked list '''
def printList():
    global head
    node = head
    while (node != None):
        print(node.data, end = " ")
        node = node.next
    print()
```

```
''' Driver program to test above functions'''

''' Let us create a sample linked list as following
0.1.4.6.9.10.11 '''

push(11)
push(10)
push(9)
push(6)
push(4)
push(1)
push(0)

print("Original Linked list")
printList()

segregateEvenOdd()

print("Modified Linked list")
printList()
```

PART B

1. Write a program to implement the following operations of a single linked list: creating a list, List traversal.

```
class Node:
    # Function to initialize the node object
    def __init__(self, data):
        self.data = data  # Assign data
        self.next = None  # Initialize next as null
 # Linked List class contains a Node object
class LinkedList:
    # Function to initialize head
    def __init__(self):
        self.head = None
    # (TRAVERSAL)
    # starting from head
    def printList(self):
        temp = self.head
        while (temp):
            print (temp.data,end=' ')
            temp = temp.next
```

Since the question has only asked us to create and traverse, we will not be using the insert function add nodes. You can also add an insert function in the class and use them to push nodes into the list.
The code below can be used to add nodes without insertion function.

```
llist = LinkedList()
llist.head = Node(1)
second = Node(2)
third = Node(3)
llist.head.next = second; # Link first node with second
```

```
second.next = third; # Link second node with the third node
llist.printList()
```

OUTPUT

1 2 3

2. A node can be inserted at various places in a linked list. Write algorithms for inserting a new node in a single linked list at::

1) At the front of the linked list:

2) After a given node:

3) At the end of the linked list

Note: Commented text in the code is the Algorithm

Note: Add this function to the LinkedList class.

Front of list:

```
def push(self, new_data):

    # 1: Allocate the Node and Put in the data
    new_node = Node(new_data)

    # 2. Make next of new Node as head
    new_node.next = self.head

    # 3. Move the head to point to new Node
    self.head = new_node
```

After a Node:

```
def insertAfter(self, prev_node, new_data):

    # 1. check if the given prev_node exists
    if prev_node is None:
        return

    # 2. Create new node &
    # 3. Put in the data
    new_node = Node(new_data)

    # 4. Make next of new Node as next of prev_node
    new_node.next = prev_node.next

    # 5. make next of prev_node as new_node
    prev_node.next = new_node
```

End of list:

```
def append(self, new_data):

   # 1. Create a new node
   # 2. Put in the data
   # 3. Set next as None
   new_node = Node(new_data)

   # 4. If the Linked List is empty, then make the
   # new node as head
   if self.head is None:
        self.head = new_node
```

```
        return

    # 5. Else traverse till the last node
    last = self.head
    while (last.next):
        last = last.next

    # 6. Change the next of last node
    last.next =  new_node
```

3)Write a program to count the number of nodes present in a single linked list?

Note: Add this function to the LinkedList class.

Iterative method

```
def getCount(self):
        temp = self.head # Initialise temp
        count = 0 # Initialise count

        # Loop while end of linked list is not reached
        while (temp):
            count += 1
            temp = temp.next
        return count
print ("Count of nodes is :",llist.getCount())
```

Recursive method

```
def getCountRec(self, node):
        if (not node): # Base case
            return 0
        else:
            return 1 + self.getCountRec(node.next)

    # A wrapper over getCountRec()
    def getCount(self):
       return self.getCountRec(self.head)
print ("Count of nodes is :",llist.getCount())
```

4)Write a program to search for an element present in a single linked list?

Note: Add this function to the LinkedList class.

```
def search(self, x):
        # Initialize current to head
        current = self.head
        # loop till current not equal to None
        while current != None:
            if current.data == x:
                return "Present" # data found
            current = current.next
        return "Not Present"
print(llist.search(3))
```

5) Write a program to delete a node from the middle position of the single linked list?

```
class Node:
    def __init__(self,data):
        self.data = data;
        self.next = None;
class DeleteMid:
    #Represent the head and tail of the singly linked list
    def __init__(self):
        self.head = None;
        self.tail = None;
        self.size = 0;
    #deleteFromMid() will delete a node from the middle of
the list
    def deleteFromMid(self):
        #Checks if list is empty
        if(self.head == None):
            print("List is empty");
            return;
        else:
            #Store the mid position of the list
            count = (self.size//2) if(self.size % 2 == 0)
else((self.size+1)//2);
            #Checks whether head is equal to tail or not, if
yes then the list has only one node.
            if( self.head != self.tail ):
                #Initially, temp will point to head
                temp = self.head;
```

```
                current = None;

                #Current will point to node previous to
                #If temp is pointing to node 2 then current
will point to node 1.
                for i in range(0, count-1):
                    current = temp;
                    temp = temp.next;

                if(current != None):
#temp is the middle that needs to be removed.
#So, the current node will point to the node next to temp by
skipping temp.
                    current.next = temp.next;
                    #Delete temp
                    temp = None;
                #If current points to None then, head and
tail will point to nodes next to temp.
                else:
                    self.head = self.tail = temp.next;
                    #Delete temp
                    temp = None;
            #If the list contains only one element
            #then it will remove it and both head and tail
will point to None
            else:
                self.head = self.tail = None;
        self.size = self.size - 1;
```

6) Write a program to reverse a singly linked list of length n?

Note: ADD DISPLAY OPERATION AND MAKE A LINKED LIST TO COMPLETE THE PROGRAM

```
class Node:
    # Constructor to initialize the node object
    def __init__(self, data):
        self.data = data
        self.next = None
class LinkedList:
     # Function to initialize head
    def __init__(self):
        self.head = None
     # Function to reverse the linked list
    def reverse(self):
        prev = None
        current = self.head
        while(current is not None):
            next = current.next
            current.next = prev
            prev = current
            current = next
        self.head = prev
```

7) Write a program to implement the following operations of a double linked list: Creating a list Inserting a node at the beginning

class DoublyLinkedList:

```
# Constructor for empty Doubly Linked List
    def __init__(self):
        self.head = None

    # Given a reference to the head of a list and an
    # integer, inserts a new node on the front of list
    def push(self, new_data):

        # 1. Allocates node
        # 2. Put the data in it
        new_node = Node(new_data)

        # 3. Make next of new node as head and
        # previous as None (already None)
        new_node.next = self.head

        # 4. change prev of head node to new_node
        if self.head is not None:
            self.head.prev = new_node

        # 5. move the head to point to the new node
        self.head = new_node
```

8) Write a program to implement the following operations of a circular single linked list: Creating a list Deleting a node at the end

```
def addToEmpty(self, data):

       if (self.last != None):
           return self.last

       # Creating the newnode temp
       temp = Node(data)
       self.last = temp

       # Creating the link
```

```
        self.last.next = self.last
        return self.last
def DeleteLast(head):
    current = head
    temp = head
    previous = None

    # check if list doesn't have any node
    # if not then return
    if (head == None):
        print("\nList is empty")
        return None

    # check if list have single node
    # if yes then delete it and return
    if (current.next == current):
        head = None
        return None

    # move first node to last
    # previous
    while (current.next != head):
        previous = current
        current = current.next

    previous.next = current.next
    head = previous.next

    return head
```

9) Write a program to merge two sorted linked lists into a third linked list using recursion?

```
class Node:
    def __init__(self, data):
        self.data = data
```

```
        self.next = None


# Constructor to initialize the node object
class LinkedList:

    # Function to initialize head
    def __init__(self):
        self.head = None

    # Method to print linked list
    def printList(self):
        temp = self.head

        while temp :
            print(temp.data, end="->")
            temp = temp.next

    # Function to add nodes at the end.
    def append(self, new_data):
        new_node = Node(new_data)

        if self.head is None:
            self.head = new_node
            return
        last = self.head

        while last.next:
            last = last.next
        last.next = new_node


# Function to merge two sorted linked lists.
def mergeLists(head1, head2):

    # create a temp node NULL
    temp = None
```

```
    # List1 is empty then return List2
    if head1 is None:
        return head2

    # if List2 is empty then return List1
    if head2 is None:
        return head1

    # If List1's data is smaller or
    # equal to List2's data
    if head1.data <= head2.data:

        # assign temp to List1's data
        temp = head1

        # Again check List1's data is smaller or equal
List2's
        # data and call mergeLists function.
        temp.next = mergeLists(head1.next, head2)

    else:
        # If List2's data is greater than or equal List1's
        # data assign temp to head2
        temp = head2

        # Again check List2's data is greater or equal List's
        # data and call mergeLists function.
        temp.next = mergeLists(head1, head2.next)

    # return the temp list.
    return temp


# Driver Function
if __name__ == '__main__':
    # Create linked list :
    # 10->20->30->40->50
    list1 = LinkedList()
```

```
    list1.append(10)
    list1.append(20)
    list1.append(30)
    list1.append(40)
    list1.append(50)

    # Create linked list 2 :
    # 5->15->18->35->60
    list2 = LinkedList()
    list2.append(5)
    list2.append(15)
    list2.append(18)
    list2.append(35)
    list2.append(60)
    # Create linked list 3
    list3 = LinkedList()
    # Merging linked list 1 and linked list 2
    # in linked list 3
    list3.head = mergeLists(list1.head, list2.head)
    print(" Merged Linked List is : ", end="")
    list3.printList()
```

10) Write a function to delete a given node in a double linked list?

```
# Program to delete a node in a doubly-linked list

# for Garbage collection
import gc

# A node of the doubly linked list
class Node:

    # Constructor to create a new node
    def __init__(self, data):
        self.data = data
        self.next = None
```

```
        self.prev = None

class DoublyLinkedList:
     # Constructor for empty Doubly Linked List
    def __init__(self):
        self.head = None

   # Function to delete a node in a Doubly Linked List.
   # head_ref --> pointer to head node pointer.
   # dele --> pointer to node to be deleted

    def deleteNode(self, dele):

        # Base Case
        if self.head is None or dele is None:
            return

        # If node to be deleted is head node
        if self.head == dele:
            self.head = dele.next

        # Change next only if node to be deleted is NOT
        # the last node
        if dele.next is not None:
            dele.next.prev = dele.prev

        # Change prev only if node to be deleted is NOT
        # the first node
        if dele.prev is not None:
            dele.prev.next = dele.next
        # Finally, free the memory occupied by dele
        # Call python garbage collector
        gc.collect()

    # Given a reference to the head of a list and an
    # integer, inserts a new node on the front of list
    def push(self, new_data):
```

```
        # 1. Allocates node
        # 2. Put the data in it
        new_node = Node(new_data)

        # 3. Make next of new node as head and
        # previous as None (already None)
        new_node.next = self.head

        # 4. change prev of head node to new_node
        if self.head is not None:
            self.head.prev = new_node

        # 5. move the head to point to the new node
        self.head = new_node


    def printList(self, node):
        while(node is not None):
            print(node.data,end=' ')
            node = node.next

# Driver program to test the above functions

# Start with empty list
dll = DoublyLinkedList()

# Let us create the doubly linked list 10<->8<->4<->2
dll.push(2);
dll.push(4);
dll.push(8);
dll.push(10);

print ("\n Original Linked List",end=' ')
dll.printList(dll.head)

# delete nodes from doubly linked list
dll.deleteNode(dll.head)
dll.deleteNode(dll.head.next)
```

```
dll.deleteNode(dll.head.next)
# Modified linked list will be NULL<-8->NULL
print("\n Modified Linked List",end=' ')
dll.printList(dll.head)
```

Algorithm:

i) Let the node to be deleted be del.

ii) If the node to be deleted is the head node, then change the head pointer to the next current head.

if headnode == del then

    headnode = del.nextNode

iii)Set next of previous to del, if previous to del exists.

if del.nextNode != none

    del.nextNode.previousNode = del.previousNode

iv)Set prev of next to del, if next to del exists.

if del.previousNode != none

    del.previousNode.nextNode = del.next

11) Write an algorithm to insert new nodes at the beginning, at middle position and at the end of a Singly Linked List.

Insertion at beginning:

Algorithm :

1. if head = None, then

   new_node = node(data)

   self.head = new_node

   else:

      create object of node

2. nb = Node(data)

3. nb.next = self.head
4. self.head = nb

Insertion at end:

Algorithm:

```
# If the Linked List is empty, then make the  new node as
head
new_node = node(data)
if self.head is None:
          self.head = new_node  return
#Else traverse till the last node
temp = self.head
while temp.next is not None:
temp = temp.next
# Change the next of last node  temp.next =new_node
```

Insert a node at middle position:

Algorithm:

Step 1.if position is equal to 1

create a node and insert data in to the node

return

Step 2. pos = p

temp = sel.head

for i in range(pos-1):

move the pointer to the next node i.e., temp = temp.next

if pointer is None:

print position is not present in the list

else:

create a node i.e., np =node(data)

np.next = temp.next

temp.next= np

12) Discuss about implementation of queues using linked lists.

A queue data structure can be implemented using a linked list data structure. The queue which is implemented using a linked list can work for an unlimited number of values.

That means, a queue using a linked list can work for variable size of data (No need to fix the size at the beginning of the implementation). The Queue implemented using a linked list can organize as many data values as we want.

We will implement Queue using a linked list. Queues maintain two data pointers:

FRONT: to know first inserted element

REAR to know the last inserted element.

Queue is a very simple data structure, you only have to manipulate FRONT and REAR to get Queue property.

Basic Operation:

Following are basic operations of Queue:

Main Queue Operations:

1)EnQueue(): Inserts an element at the rear of the Queue.

2)DeQueue(): Remove and return the front element of the Queue.

Auxiliary Queue operation:

1) Front(): Display the data in front of the Queue.
2)QueueSize(): Returns the number of elements stored in the Queue.
3)display(): Display elements of the queue.

There are many different operations that we can implement.
For ex.:- Merging of Two Queues,Sorting of Queues etc..

EnQueue:
Inserting an element in Queue.

Pseudo Code:
Step 1 : Create a newNode with a given value and set 'newNode → next' to NULL.
Step 2 : Check whether queue is Empty (rear == NULL)
Step 3 : If it is Empty then, set front = newNode and rear = newNode.
Step 4 : If it is Not Empty then, set rear → next = newNode and rear = newNode.

DeQueue:
Deleting an element from Queue.

Pseudo Code:
Step 1 : Check whether the queue is Empty (front == NULL).
Step 2 : If it is Empty, then display "Queue is Empty!!! Deletion is not possible!!!" and terminate from the function
Step 3 : If it is Not Empty then, define a Node pointer 'temp' and set it to 'front'.

Step 4 : Then set 'front = front → next' and delete 'temp' (free(temp)).

Display:
To display all elements present in Queue, we will take one temp(you can give whatever name you want) pointer and make that to point front. Then we traverse up-to rear or we can say temp!=NULL using temp=temp→next (which will make our pointer move towards the rear) and print temp→data.

Pseudo Code:
Step 1 : Check whether the queue is Empty (front == NULL).
Step 2 : If it is Empty then, display 'Queue is Empty!!!' and terminate the function.
Step 3 : If it is Not Empty then, define a Node pointer 'temp' and initialize with front.
Step 4 : Display 'temp → data ⟶' and move it to the next node. Repeat the same until 'temp' reaches to 'rear' (temp → next != NULL).
Step 5 : Finally! Display 'temp → data ⟶ NULL'.

Time complexity of the DeQueue and EnQueue operation is O(1),Because there is no loop in those operations.
Time complexity of Display operation is O(n),Since we are traversing to print each n(number of elements) elements.
Space complexity is O(n),Where n is the number of elements.

13) Describe how a polynomial is represented using singly linked lists

A polynomial p(x) is the expression in variable x which is in the form (ax^n + bx^{n-1} + ... + cx+ k), where a, b, c ...., k falls in the category of real numbers and 'n' is a non negative integer, which is called the degree of polynomial.

An essential characteristic of the polynomial is that each term in the polynomial expression consists of two parts:
one is the coefficient
other is the exponent
Example:
$10x^2$ + 26x, here 10 and 26 are coefficients and 2, 1 is its exponential value.

Points to keep in mind while working with Polynomials:

The sign of each coefficient and exponent is stored within the coefficient and the exponent itself
The storage allocation for each term in the polynomial must be done in ascending and descending order of their exponent

In the linked representation of a polynomial, each term is considered a node. And such a node contains three fields.
i)Coefficient field
ii)Exponent field
iii)Link field
The coefficient field holds the value of the coefficient of term and the exponent field contains the exponent value of the term. And the link field contains the address of the next term of the polynomial.

14) Write an algorithm to add two polynomials using a linked list.

Algorithm:
1)Create a new linked list, newHead to store the resultant list.
2)Traverse both lists until one of them is null.
3)If any list is null, insert the remaining node of another list in the resultant list.
Otherwise compare the degree of both nodes, a (first listâ€™s node) and b (second list's node). Here three cases are possible:
a) If the degree of a and b is equal, we insert a new node in the resultant list with the coefficient equal to the sum of coefficients of a and b and the same degree.
b) If the degree of a is greater than b, we insert a new node in the resultant list with the coefficient and degree equal to that of a.
c) If the degree of b is greater than a, we insert a new node in the resultant list with the coefficient and degree equal to that of b.

15) Explain how a linked list can be used for representing polynomials using a suitable example.

Hint:i)Linked lists offer some important advantages over other linear data structures. Unlike arrays, they are a dynamic data structure, resizable at run-time. Also, the insertion and deletion operations are efficient and easily implemented.
ii)For any polynomial operation, such as addition or multiplication of polynomials, linked list representation is easier to deal with. Linked lists are useful for dynamic memory allocation.

>>> Write 13th answer for this question by including points which are included in hint

16) Explain about insertion and deletion operations on single linked lists. Write pseudo code for the same.

Inserting element in the linked list involves reassigning the pointers from the existing nodes to the newly inserted node. Depending on whether the new data element is getting inserted at the beginning or at the middle or at the end of the linked list, we have the below scenarios.

Inserting at the Beginning:

This involves pointing the next pointer of the new data node to the current head of the linked list. So the current head of the linked list becomes the second data element and the new node becomes the head of the linked list.

Example:

```
class Node:
   def __init__(self, dataval=None):
      self.dataval = dataval
      self.nextval = None

class SLinkedList:
   def __init__(self):
      self.headval = None
```

```
# Print the linked list
   def listprint(self):
      printval = self.headval
      while printval is not None:
         print (printval.dataval)
         printval = printval.nextval
   def AtBegining(self,newdata):
      NewNode = Node(newdata)

# Update the new nodes next val to existing node
   NewNode.nextval = self.headval
   self.headval = NewNode

list = SLinkedList()
list.headval = Node("Mon")
e2 = Node("Tue")
e3 = Node("Wed")

list.headval.nextval = e2
e2.nextval = e3

list.AtBegining("Sun")
list.listprint()
```

Inserting at the End:

This involves pointing the next pointer of the current last node of the linked list to the new data node. So the current last node of the linked

list becomes the second last data node and the new node becomes the last node of the linked list.

Example:

```
class Node:
   def __init__(self, dataval=None):
      self.dataval = dataval
      self.nextval = None
class SLinkedList:
   def __init__(self):
      self.headval = None
# Function to add newnode
   def AtEnd(self, newdata):
      NewNode = Node(newdata)
      if self.headval is None:
         self.headval = NewNode
         return
      laste = self.headval
      while(laste.nextval):
         laste = laste.nextval
      laste.nextval=NewNode
# Print the linked list
   def listprint(self):
      printval = self.headval
      while printval is not None:
         print (printval.dataval)
         printval = printval.nextval

list = SLinkedList()
```

```
list.headval = Node("Mon")
e2 = Node("Tue")
e3 = Node("Wed")

list.headval.nextval = e2
e2.nextval = e3

list.AtEnd("Thu")

list.listprint()
```

Inserting in between two Data Nodes:
This involves changing the pointer of a specific node to point to the new node. That is possible by passing in both the new node and the existing node after which the new node will be inserted. So we define an additional class which will change the next pointer of the new node to the next pointer of the middle node. Then assign the new node to the next pointer of the middle node.

Example:

```
class Node:
   def __init__(self, dataval=None):
      self.dataval = dataval
      self.nextval = None
class SLinkedList:
   def __init__(self):
      self.headval = None
```

```
# Function to add node
   def Inbetween(self,middle_node,newdata):
      if middle_node is None:
         print("The mentioned node is absent")
         return

      NewNode = Node(newdata)
      NewNode.nextval = middle_node.nextval
      middle_node.nextval = NewNode

# Print the linked list
   def listprint(self):
      printval = self.headval
      while printval is not None:
         print (printval.dataval)
         printval = printval.nextval

list = SLinkedList()
list.headval = Node("Mon")
e2 = Node("Tue")
e3 = Node("Thu")

list.headval.nextval = e2
e2.nextval = e3

list.Inbetween(list.headval.nextval,"Fri")

list.listprint()
```

Deletion at beginning:

Deletion in singly linked list at the end. There are two scenarios in which a node is deleted from the end of the linked list. There is only one node in the list and that needs to be deleted. There are more than one node in the list and the last node of the list will be deleted.

Example:

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

#class LinkedList
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.")

# 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()
```

Deletion at specified position:In this method, a node at the specified position in the linked list is deleted. For example - if the given List is 10→20→30 and the 2nd node is deleted, the Linked List becomes 10→20.

First, the specified position must be greater than equal to 1. If the specified position is 1 and head is not null, then make the head next as head and delete the previous head. Else, traverse to the node that is previous to the specified position. If the specified node and previous to the specified node are not null then adjust the link. In other case, the specified node will be already null. The below figure describes the process, if the deletion node is other than the head node.

Example:

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

#class Linked List
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 an element at the given position
def pop_at(self, position):
  if(position < 1):
    print("\nposition should be >= 1.")
  elif (position == 1 and self.head != None):
    nodeToDelete = self.head
    self.head = self.head.next
    nodeToDelete = None
  else:
    temp = self.head
    for i in range(1, position-1):
      if(temp != None):
        temp = temp.next
    if(temp != None and temp.next != None):
      nodeToDelete = temp.next
      temp.next = temp.next.next
```

```
        nodeToDelete = None
      else:
        print("\nThe node is already null.")

  #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 three elements at the end of the list.
MyList.push_back(10)
MyList.push_back(20)
MyList.push_back(30)
MyList.PrintList()

#Delete an element at position 2
MyList.pop_at(2)
MyList.PrintList()
```

```
#Delete an element at position 1
MyList.pop_at(1)
MyList.PrintList()
```

Deletion at last:Deleting the last node of the Linked List involves checking the head for empty. If it is not empty, then check the head next for empty. If the head next is empty, then release the head, else traverse to the second last node of the list. Then, link the next or second last node to NULL and delete the last node.

Example:

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

#class LinkedList
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 last node of the list
def pop_back(self):
  if(self.head != None):
    if(self.head.next == None):
      self.head = None
    else:
      temp = self.head
      while(temp.next.next != None):
        temp = temp.next
      lastNode = temp.next
      temp.next = None
      lastNode = 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 last node
MyList.pop_back()
MyList.PrintList()
```

17 & 18) Write an algorithm to delete an element from a doubly linked list.

Algorithm:
Let us suppose we have to delete node 'delete'

1) If the node to be deleted is the head node, then change the head pointer to the next current head.
Example: if node with data '10' is to be deleted then head will point to '20'

2) Set next or previous to delete, if previous to delete.
Example if 30 is to be deleted then next of 20 will point to 40

To delete node:

traverse linked list till we found the node to be deleted.

```
currNode = head
while currNode:
     if currNode.data == userData:
           currNode.prevNode.nextNode = currNode.nextNode
           currNode.nextNode.prevNode = currNode.prevNode
           del currNode
           break
     currNode = currNode.nextNode
```

19) Elaborate on how a linked list can be used to represent polynomials using a suitable example.

Answer same as 13th question answer

20) Given an ordered linked list whose node is represented by 'key' as information and 'next' as link field. Write a python program to implement deleting the number of nodes (consecutive) whose 'key' values are greater than or equal to 'Kmin' and less than 'Kmax'.

```
class newNode:
    def __init__(self, data):
        self.data = data
        self.left = self.right = None

# print the tree in LVR (Inorder traversal) way.
def Print(root):
    if (root != None):
```

```
        Print(root.left)
        print(root.data, end = " ")
        Print(root.right)

# Main function which truncates
# the binary tree.
def pruneUtil(root, k, Sum):

    # Base Case
    if (root == None):
        return None

    # Initialize left and right Sums as
    # Sum from root to this node
    # (including this node)
    lSum = [Sum[0] + (root.data)]
    rSum = [lSum[0]]

    # Recursively prune left and right
    # subtrees
    root.left = pruneUtil(root.left, k, lSum)
    root.right = pruneUtil(root.right, k, rSum)

    # Get the maximum of left and right Sums
    Sum[0] = max(lSum[0], rSum[0])

    # If maximum is smaller than k,
    # then this node must be deleted
    if (Sum[0] < k[0]):
```

```
        root = None
    return root
    elif(Sum[0]>k[0]):
         root=None
    return root

    else:
         return None

# A wrapper over pruneUtil()
def prune(root, k):
    Sum = [0]
    return pruneUtil(root, k, Sum)

# Driver Code
if __name__ == '__main__':
    k = [45]
    root = newNode(1)
    root.left = newNode(2)
    root.right = newNode(3)
    root.left.left = newNode(4)
    root.left.right = newNode(5)
    root.right.left = newNode(6)
    root.right.right = newNode(7)
    root.left.left.left = newNode(8)
    root.left.left.right = newNode(9)
    root.left.right.left = newNode(12)
    root.right.right.left = newNode(10)
    root.right.right.left.right = newNode(11)
```

```
    root.left.left.right.left = newNode(13)
    root.left.left.right.right = newNode(14)
    root.left.left.right.right.left = newNode(15)

    print("Tree before truncation")
    Print(root)
    print()
    root = prune(root, k) # k is 45

    print("Tree after truncation")
    Print(root)
```

PART C

1) Write the advantages and disadvantages of linked lists?

Advantages Of Linked List:

a) Dynamic data structure: A linked list is a dynamic arrangement so it can grow and shrink at runtime by allocating and deallocating memory. So there is no need to give the initial size of the linked list.
b) No memory wastage: In the Linked list, efficient memory utilization can be achieved since the size of the linked list increases or decreases at run time so there is no memory wastage and there is no need to pre-allocate the memory.

c) Implementation: Linear data structures like stack and queues are often easily implemented using a linked list.
d)Insertion and Deletion Operations: Insertion and deletion operations are quite easier in the linked list. There is no need to shift elements after the insertion or deletion of an element; only the address present in the next pointer needs to be updated.

Disadvantages Of Linked List:

a) Memory usage: More memory is required in the linked list as compared to an array. Because in a linked list, a pointer is also required to store the address of the next element and it requires extra memory for itself.
b) Traversal: In a Linked list traversal is more time-consuming as compared to an array. Direct access to an element is not possible in a linked list as in an array by index. For example, for accessing a node at position n, one has to traverse all the nodes before it.
c) Reverse Traversing: In a singly linked list reverse traversing is not possible, but in the case of a doubly-linked list, it can be possible as it contains a pointer to the previously connected nodes with each node. For performing this extra memory is required for the back pointer hence, there is a wastage of memory.
d)Random Access: Random access is not possible in a linked list due to its dynamic memory allocation.

2) List out types of linked lists and their applications?

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers. In simple words, a linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.

Types of Linked Lists:

- Singly Linked List.
- Doubly Linked List.
- Circular Linked List.
- Circular Doubly Linked List

Applications:
i) Implementation of stacks and queues
ii) Implementation of graphs : Adjacency list representation of graphs is most popular which uses linked lists to store adjacent vertices.
iii) Dynamic memory allocation : We use a linked list of free blocks.
iv) Maintaining directory of names
v) Performing arithmetic operations on long integers
vi) Manipulation of polynomials by storing constants in the node of linked list
vii) representing sparse matrices

3) Write the advantages of doubly linked lists over singly linked lists?

Following are advantages/disadvantages of doubly linked list over singly linked list.

1. A DLL can be traversed in both forward and backward direction.

2. The delete operation in DLL is more efficient if a pointer to the node to be deleted is given.
3. We can quickly insert a new node before a given node.

4) Write the applications of single and double linked lists?

Applications of Singly Linked List are as following:
i) It is used to implement stacks and queues which are like fundamental needs throughout computer science.
ii) To prevent the collision between the data in the hash map, we use a singly linked list.
iii) 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.
iv) We can think of its use in a photo viewer for having to look at photos continuously in a slide show.
v) In the system of train, the idea is like a singly linked list, as if you want to add a Boogie, either you have to take a new boggie to add at last or you must spot a place in between bogies and add it.

Applications of Doubly Linked List are as following:
i) Great use of the doubly linked list is in navigation systems, as it needs front and back navigation.
ii) In the browser when we want to use the back or next function to change the tab, it uses the concept of a doubly-linked list here.
iii) Again, it is easily possible to implement other data structures like a binary tree, hash tables, stack, etc.
iv) It is used in a music playing system where you can easily play the previous one or next one song as many times as one person wants to.

Basically it provides full flexibility to perform functions and make the system user-friendly.
v) In many operating systems, the thread scheduler (the thing that chooses what processes need to run at which times) maintains a doubly-linked list of all the processes running at any time. This makes it easy to move a process from one queue (say, the list of active processes that need a turn to run) into another queue (say, the list of processes that are blocked and waiting for something to release them).
vi) It is used in a famous game concept which is a deck of cards.

5) Find the time complexity to count the number of elements in a linked list?

To count the number of elements, you have to traverse through the entire list, hence complexity is O(n).

6) Define a circular single linked list?

A singly circular list consists of only one additional pointer named 'Next'. The 'Next' of each node, except the last node, will contain the address of the upcoming node. The last node's 'Next' will store the address of the first node.

7) Write any two operations that are performed more efficiently by a doubly linked list than a singly linked list?

Deleting a node whose location is given.
Reason:

We cannot travel in the singly linked list from the current node to the previous node, thus getting to node A is not possible from node B. From the start node. We can traverse back and forth from a node In the Doubly linked list. Thus, if we have a pointer to node B, we can move back to locate node A. Once we have the pointer node the way you suggested we can do the connection. All of this will take O() time. Hence, the correct answer is Deleting a node whose location is given.

8) Consider a single linked list, list out any two operations that can be implemented in O(1) time?

Could be implemented in O(1) time
since head,tail pointers is given

1) can insert node and get the address of that node into head pointer and next of that node contains next node address
2) As tail pointer is present new node address is stored in tail and even in last but 1 node of linked list
3) Could be deleted easily but pointing head to the next node
All this could be done with O(1)

(or)

i) Insertion at the front of the linked list
ii) Insertion at the end of the linked list
iii) Deletion of the front node of the linked list

9 &10)

```
def fun1(head):
     if(head == None):
          return
```

```
        fun1(head.next)
        print(head.data, end = “  “)
```

What does the following function do for a given Linked List?

fun1() prints the given Linked List in reverse manner. For Linked List 1→2→3→4→5, fun1() prints 5→4→3→2→1.

Prints all nodes of the linked list in reverse order.

11) Identify the operation which is difficult to perform in a circular single linked list?

Backward Direction Traversal is the difficult operation to do in a circular single linked list.

12) Write the asymptotic time complexity to insert an element at the second position in the linked list?

Thus the asymptotic time complexity to insert an element in the second position of the linked list is O (1).

13) Identify the variant of linked list in which none of the node contains a NULL pointer?

A list in which the last node points back to the header node is called a circular linked list. The chains do not indicate first or last nodes. In this case, external pointers provide a frame of reference because the last node of a circular linked list does not contain the NULL pointer.

14) In a circular linked list, how many pointers require modification if a node is inserted?

Two Pointers

15) Identify the searching technique for which linked lists are not suitable data structures?

Binary search of the data structure link lists are not suitable.

Binary search based on divide and conquer algorithm, determination of middle element is important. Binary search is usually fast and efficient for arrays because accessing the middle index between two given indices is easy and fast. But memory allocation for singly linked lists is dynamic and non-contiguous, which makes finding the middle element difficult. One approach could be using skip list; one could be traversing the linked list using one pointer.

A linked list is a linear collection of data elements, whose order is not given by their physical placement in memory. Instead, each element points to the next. It is a data structure consisting of a collection of nodes which together represent a sequence.

(or)

Not Suitable for:

Binary search: Because finding mid element itself takes O(n) time.

16) In the worst case, find the number of comparisons needed to search a singly linked list of length n for a given element?

Singly linked list has uni – directional flow, i.e., it has only one pointer for moving (the next pointer).
In the worst case, for searching an element in the singly linked list, we will have to traverse the whole list (the case when the required element is either the last element or is not present in the list).

So, in the worst case for a list of length n, we will have to go to each node for comparison and thus, we would be needing ‘n’ comparisons.

17) State the name of the data structure in which data elements are logically adjacent to each other?

Linked allocation

Detailed Explanation:
In linked allocation, each file is a linked list of disk blocks. The directory contains a pointer to the first and optionally the last block of the file. With linked allocation, each directory entry has a pointer to the first disk block of the file.

18) Write the disadvantages of doubly linked lists over singly linked lists?

Disadvantages over singly linked list:

Disadvantages of a Doubly Linked List compared to a singly linked list, each node stores an extra pointer which consumes extra memory. Operations require more time due to the overhead of handling extra pointers as compared to singly-linked lists. No random access of elements.

19) Write the time complexity of enqueue() and dequeued() operations of a linked list implementation of a linear queue?

Implemented properly, both the enque and deque operations on a queue implemented using a singly-linked list should be constant time, i.e., O(1).

20) Write an example of a non-contiguous data structure?

A noncontiguous data structure is a method of storing data in sectors of memory that are not adjoining.

Example:A linked list is an example of a non- contiguous data structure. Here, the nodes of the list are linked together using pointers stored in each node.