Data Structures and Algorithms – Assignment 1



**Name: Ashish Prem AM.EN.U4CSE19309**

**Array & Linked List**

1. Write a Python program that uses functions to perform the following operations on array
   1. Creation
   2. Insertion (at start, at end, using index, based on value)
   3. Deletion (at start, at end, using index, based on value)
   4. Traversal
   5. Searching an element. (based on value, based on index)

import array

arr = array.array("i" , [ 1 , 2 , 3 ])

arr.insert(0 , 21)

arr.insert(2,7)

arr.append(5)

for i in range(0,6):

print(arr[i] , end=" ")

arr.pop(2)

arr.pop()

arr.remove(21)

print()

for i in range(0,3):

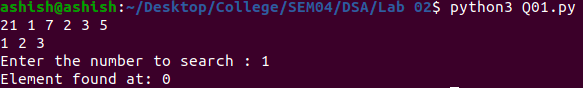
print(arr[i] , end=" ")

n = int(input("\nEnter the number to search : "))

for i in range(0,3):

if arr[i] == n:

print("Element found at: "+ str(arr.index(n)))



1. Write a Python program that uses functions to perform the following operations on singly linked list.
   1. Creation
   2. Insertion (as first node, as last node, in between node)
   3. Deletion (first node, last node, in between node)
   4. Traversal
   5. Searching an element.

class Node:

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

*self*.data = data

*self*.ref = None

class LinkedList:

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

*self*.head = None

*#traversal*

def print\_LL(self):

if *self*.head is None:

print("List is empty")

else:

n = *self*.head

while n is not None:

print(n.data)

n = n.ref

def add\_begin(self,data):

new\_node = Node(data)

new\_node.ref = *self*.head

*self*.head = new\_node

def add\_end(self,data):

new\_node = Node(data)

if *self*.head is None:

*self*.head = new\_node

else:

n = *self*.head

while n.ref is not None:

n = n.ref

n.ref = new\_node

def after\_node(self,data,x):

n = *self*.head

while n is not None:

if x == n.data:

break

n = n.ref

if n is None:

print("Node not present in Linked List")

else:

new\_node = Node(data)

new\_node.ref = n.ref

n.ref = new\_node

def delete\_begin(self):

if *self*.head is None:

print("Linked list is empty")

else:

*self*.head = *self*.head.ref

def delete\_end(self):

if *self*.head is None:

print("Linked list is empty")

elif *self*.head.ref is None:

*self*.head = None

else:

n = *self*.head

while n.ref.ref is not None:

n = n.ref

n.ref = None

def delete\_value(self,x):

if *self*.head is None:

print("LL is empty")

return

elif x == *self*.head.data:

*self*.head = *self*.head.ref

n = *self*.head

while n.ref is not None:

if n.ref.data == x:

break

n = n.ref

if n.ref is None:

print("Node is not present!")

else:

n.ref = n.ref.ref

LL1 = LinkedList()

LL1.add\_begin(200)

LL1.after\_node(60,200)

LL1.after\_node(50,60)

LL1.add\_end(20)

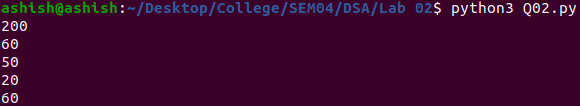
LL1.print\_LL()

LL1.delete\_begin()

LL1.delete\_end()

LL1.delete\_value(50)

LL1.print\_LL()



1. Write a Python program that uses functions to perform the following operations on doubly linked list
   1. Creation
   2. Insertion (as first node, as last node, in between node)
   3. Deletion (first node, last node, in between node)
   4. Traversal
   5. Searching an element.

class Node:

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

*self*.data = data

*self*.pref = None

*self*.nref = None

class DLL:

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

*self*.head = None

def print\_ll(self):

if *self*.head is None:

print("Dll is empty!!")

else:

n = *self*.head

while n is not None:

print(n.data, "-->", end=" ")

n = n.nref

def print\_ll\_rev(self):

if *self*.head is None:

print("Dll is empty!!")

else:

print()

n = *self*.head

while n.nref is not None:

n = n.nref

while n is not None:

print(n.data, "-->", end=" ")

n = n.pref

def insert\_empty(self, data):

if *self*.head is None:

new\_node = Node(data)

*self*.head = new\_node

else:

print("Linked List is not empty")

def add\_begin(self, data):

new\_node = Node(data)

if *self*.head is None:

*self*.head = new\_node

new\_node.nref = *self*.head

*self*.head.pref = new\_node

*self*.head = new\_node

def add\_end(self, data):

new\_node = Node(data)

if *self*.head is None:

*self*.head = new\_node

n = *self*.head

while n.nref is not None:

n = n.nref

n.nref = new\_node

new\_node.pref = n

def add\_after(self, data, x):

if *self*.head is None:

print("LL is empty")

else:

n = *self*.head

while n is not None:

if x == n.data:

break

n = n.nref

if n is None:

print("Node is not present")

else:

new\_node = Node(data)

new\_node.nref = n.nref

new\_node.pref = n

if n.nref is not None:

n.nref.pref = new\_node

n.nref = new\_node

def add\_before(self, data, x):

if *self*.head is None:

print("LL is empty")

else:

n = *self*.head

while n is not None:

if x == n.data:

break

n = n.nref

if n is None:

print("Node is not present")

else:

new\_node = Node(data)

new\_node.nref = n

new\_node.pref = n.pref

if n.pref is not None:

n.pref.nref = new\_node

else:

*self*.head = new\_node

n.pref = new\_node

def del\_begin(self):

if *self*.head is None:

print("List is empty")

return

elif *self*.head.nref is None:

*self*.head = None

print("Only one node. Deleted!!")

else:

*self*.head = *self*.head.nref

*self*.head.pref = None

def del\_end(self):

if *self*.head is None:

print("List is empty")

return

elif *self*.head.nref is None:

*self*.head = None

print("Only one node. Deleted!!!")

else:

n = *self*.head

while n.nref is not None:

n = n.nref

n.pref.nref = None

def del\_value(self,x):

if *self*.head is None:

print("List is empty")

return

elif *self*.head.nref is None:

if x == *self*.head.data:

*self*.head = None

else:

print("X is not present in LL")

return

if *self*.head.data == x:

*self*.head = *self*.head.nref

*self*.head.pref = None

return

n = *self*.head

while n.nref is not None:

if x == n.data:

break

n = n.nref

if n.nref is not None:

n.nref.pref = n.pref

n.pref.nref = n.nref

else:

if n.data == x:

n.pref.nref = None

dl1 = DLL()

dl1.insert\_empty(20)

dl1.add\_begin(30)

dl1.add\_end(40)

dl1.add\_after(50, 30)

dl1.add\_before(200, 40)

dl1.print\_ll()

dl1.print\_ll\_rev()

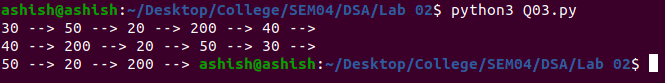
dl1.del\_begin()

dl1.del\_end()

dl1.del\_value(40)

print()

dl1.print\_ll()



1. Write a Python program that uses functions to perform the following operations on circular linked list
   1. Creation
   2. Insertion
   3. Deletion
   4. Traversal
   5. Searching an element.

class Node:

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

*self*.data = data

*self*.ref = None

class CircularList:

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

*self*.head = None

def print\_CL(self):

if *self*.head is None:

print("List is empty")

else:

n = *self*.head

while n is not None:

print(n.data, " --> ", end="")

n = n.ref

if n == *self*.head:

break

def add\_begin(self, data):

new\_node = Node(data)

*self*.head = new\_node

new\_node.ref = *self*.head

def add\_end(self, data):

new\_node = Node(data)

if *self*.head is None:

*self*.head = new\_node

new\_node.ref = *self*.head

elif *self*.head.ref is *self*.head:

*self*.head.ref = new\_node

new\_node.ref = *self*.head

else:

n = *self*.head

while n.ref is not *self*.head:

n = n.ref

n.ref = new\_node

new\_node.ref = *self*.head

def after\_node(self, data, x):

n = *self*.head

while n.ref != *self*.head:

if x == n.data:

break

n = n.ref

if n is None:

print("Node not present in Linked List")

else:

new\_node = Node(data)

new\_node.ref = n.ref

n.ref = new\_node

def delete\_begin(self):

if *self*.head is None:

print("Linked list is empty")

else:

*self*.head = *self*.head.ref

def delete\_end(self):

if *self*.head is None:

print("Linked list is empty")

elif *self*.head.ref is None:

*self*.head = None

else:

n = *self*.head

while n.ref.ref is not *self*.head:

n = n.ref

n.ref = *self*.head

def delete\_value(self, x):

if *self*.head is None:

print("LL is empty")

return

elif x == *self*.head.data:

*self*.head = *self*.head.ref

n = *self*.head

while n.ref is not *self*.head:

if n.ref.data == x:

break

n = n.ref

if n.ref is *self*.head:

print("Node is not present!")

else:

n.ref = n.ref.ref

C1 = CircularList()

C1.add\_begin(40)

C1.after\_node(60, 40)

C1.print\_CL()

C1.add\_end(80)

C1.add\_end(100)

C1.print\_CL()

print()

C1.delete\_begin()

C1.delete\_value(60)

C1.delete\_end()

C1.print\_CL()

****

1. Write a Python program to remove duplicates from an unsorted linked list.

class Node:

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

*self*.data = data

*self*.ref = None

class LinkedList:

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

*self*.head = None

def print\_LL(self):

if *self*.head is None:

print("List is empty")

else:

n = *self*.head

while n is not None:

print(n.data)

n = n.ref

def add\_begin(self, data):

new\_node = Node(data)

new\_node.ref = *self*.head

*self*.head = new\_node

def add\_end(self, data):

new\_node = Node(data)

if *self*.head is None:

*self*.head = new\_node

else:

n = *self*.head

while n.ref is not None:

n = n.ref

n.ref = new\_node

def after\_node(self, data, x):

n = *self*.head

while n is not None:

if x == n.data:

break

n = n.ref

if n is None:

print("Node not present in Linked List")

else:

new\_node = Node(data)

new\_node.ref = n.ref

n.ref = new\_node

def delete\_begin(self):

if *self*.head is None:

print("Linked list is empty")

else:

*self*.head = *self*.head.ref

def delete\_end(self):

if *self*.head is None:

print("Linked list is empty")

elif *self*.head.ref is None:

*self*.head = None

else:

n = *self*.head

while n.ref.ref is not None:

n = n.ref

n.ref = None

def delete\_value(self, x):

if *self*.head is None:

print("LL is empty")

return

elif x == *self*.head.data:

*self*.head = *self*.head.ref

n = *self*.head

while n.ref is not None:

if n.ref.data == x:

break

n = n.ref

if n.ref is None:

print("Node is not present!")

else:

n.ref = n.ref.ref

def remove\_duplicates(self):

current = *self*.head

temp = *self*.head

while current is not None:

while temp.ref is not None:

if temp.ref.data == current.data:

temp.ref = temp.ref.ref

else:

temp = temp.ref

temp = current.ref

current = current.ref

LL1 = LinkedList()

LL1.add\_begin(200)

LL1.after\_node(60, 200)

LL1.after\_node(80, 60)

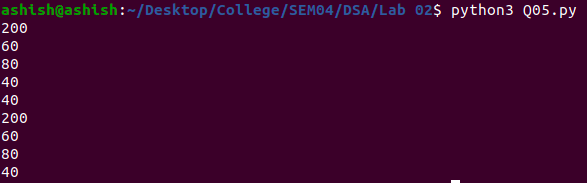
LL1.add\_end(40)

LL1.add\_end(40)

LL1.print\_LL()

LL1.remove\_duplicates()

LL1.print\_LL()



1. Write a Python program to implement an algorithm to find the kth to the last element of a singly linked list.

class Node:

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

*self*.data = data

*self*.ref = None

class LinkedList:

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

*self*.head = None

def print\_LL(self):

if *self*.head is None:

print("List is empty")

else:

n = *self*.head

while n is not None:

print(n.data)

n = n.ref

def add\_begin(self,data):

new\_node = Node(data)

new\_node.ref = *self*.head

*self*.head = new\_node

def add\_end(self,data):

new\_node = Node(data)

if *self*.head is None:

*self*.head = new\_node

else:

n = *self*.head

while n.ref is not None:

n = n.ref

n.ref = new\_node

def after\_node(self,data,x):

n = *self*.head

while n is not None:

if x == n.data:

break

n = n.ref

if n is None:

print("Node not present in Linked List")

else:

new\_node = Node(data)

new\_node.ref = n.ref

n.ref = new\_node

def delete\_begin(self):

if *self*.head is None:

print("Linked list is empty")

else:

*self*.head = *self*.head.ref

def delete\_end(self):

if *self*.head is None:

print("Linked list is empty")

elif *self*.head.ref is None:

*self*.head = None

else:

n = *self*.head

while n.ref.ref is not None:

n = n.ref

n.ref = None

def delete\_value(self,x):

if *self*.head is None:

print("LL is empty")

return

elif x == *self*.head.data:

*self*.head = *self*.head.ref

n = *self*.head

while n.ref is not None:

if n.ref.data == x:

break

n = n.ref

if n.ref is None:

print("Node is not present!")

else:

n.ref = n.ref.ref

def k\_value(self,x):

if *self*.head is None:

print("LL is empty")

return

elif x == *self*.head.data:

print("Element found",*self*.head.data)

n = *self*.head

while n.ref is not None:

if n.ref.data == x:

break

n = n.ref

if n.ref is None:

print("Element is not present!")

else:

while n.ref is not None:

print(n.ref.data," --> ",end="")

n = n.ref

LL1 = LinkedList()

LL1.add\_begin(200)

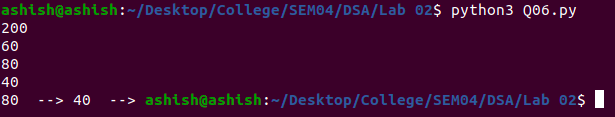
LL1.after\_node(60,200)

LL1.after\_node(80,60)

LL1.add\_end(40)

LL1.print\_LL()

LL1.k\_value(80)



7.Write a Python program to detect if a linked list has a loop in it.

class Node:

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

*self*.data = data

*self*.ref = None

class CircularList:

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

*self*.head = None

def print\_CL(self):

if *self*.head is None:

print("List is empty")

else:

n = *self*.head

while n is not None:

print(n.data, " --> ", end="")

n = n.ref

if n == *self*.head:

break

def add\_begin(self, data):

new\_node = Node(data)

*self*.head = new\_node

new\_node.ref = *self*.head

def add\_end(self, data):

new\_node = Node(data)

if *self*.head is None:

*self*.head = new\_node

new\_node.ref = *self*.head

elif *self*.head.ref is *self*.head:

*self*.head.ref = new\_node

new\_node.ref = *self*.head

else:

n = *self*.head

while n.ref is not *self*.head:

n = n.ref

n.ref = new\_node

new\_node.ref = *self*.head

def after\_node(self, data, x):

n = *self*.head

while n.ref != *self*.head:

if x == n.data:

break

n = n.ref

if n is None:

print("Node not present in Linked List")

else:

new\_node = Node(data)

new\_node.ref = n.ref

n.ref = new\_node

def delete\_begin(self):

if *self*.head is None:

print("Linked list is empty")

else:

*self*.head = *self*.head.ref

def delete\_end(self):

if *self*.head is None:

print("Linked list is empty")

elif *self*.head.ref is None:

*self*.head = None

else:

n = *self*.head

while n.ref.ref is not *self*.head:

n = n.ref

n.ref = *self*.head

def delete\_value(self, x):

if *self*.head is None:

print("LL is empty")

return

elif x == *self*.head.data:

*self*.head = *self*.head.ref

n = *self*.head

while n.ref is not *self*.head:

if n.ref.data == x:

break

n = n.ref

if n.ref is *self*.head:

print("Node is not present!")

else:

n.ref = n.ref.ref

def find\_loop(self):

if *self*.head is None:

print("No loop. LL is empty!!")

elif *self*.head.ref == *self*.head.data:

print("Loop found")

else:

n = *self*.head

while n.ref is not *self*.head:

if n.ref.ref is *self*.head:

print("Loop found")

break

else:

n = n.ref

C1 = CircularList()

C1.add\_begin(40)

C1.after\_node(60, 40)

C1.print\_CL()

C1.add\_end(80)

C1.add\_end(100)

C1.print\_CL()

C1.find\_loop()

