CS532 Homework 1

Archana Machireddy

Question 1

class Element:

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

self.key = key

self.next = None

self.prev = None

class DoublyLinkedList:

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

self.head = None

def insert(self, x):

x.next = self.head

if self.head is not None:

self.head.prev = x

self.head = x

x.prev = None

def delete(self, x):

if x.prev is not None:

x.prev.next = x.next

else:

self.head = x.next

if x.next is not None:

x.next.prev = x.prev

def search(self, k):

x = self.head

while x is not None and x.key != k:

x = x.next

if x == None:

return False

else:

return x

Question 2

class SinglyLinkedList:

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

self.head = None

def insert(self,x):

# Inserting in the beginning of the list

x.next = self.head

self.head = x

def delete(self, x):

# If list is empty

if self.head == None:

return

# If head node is the element being searched

if self.head == x:

self.head = x.next

return

# Search in the rest of the list and find the element

# before the one to be deleted and change its 'next' link

data = self.head

while data is not None:

if data == x: # First loop will be false (head node)

prev.next = data.next

return

else:

prev = data

data = data.next

def search(self, k):

x = self.head

while x is not None and x.key != k:

x = x.next

if x == None:

return False

else:

return x

The delete operation in singly linked list differs substantially from the one in doubly linked list. In doubly linked list the element being deleted has pointers to both the previous and the next elements making it easier to delete by just modifying these two pointers. In contrast, the singly linked list only has a pointer to its next element. In order to delete an element we need to find its previous element to modify the pointer of the previous element to point to the element next to the element being deleted.

In case only the value of the key to be deleted is given, both the singly and doubly linked lists have to be searched for the key, and the time complexity is O(n) in worst case in both scenarios. But, in case we are given the pointer to the element that has to be deleted, doubly linked list can delete it with time complexity of O(1) as the previous and next pointer are readily available. But in singly linked list even if we have a pointer to the element to be deleted, the list still has to be traversed to find its previous element, making the time complexity O(n).

Question 3

def \_\_str\_\_(self):

x = self.head

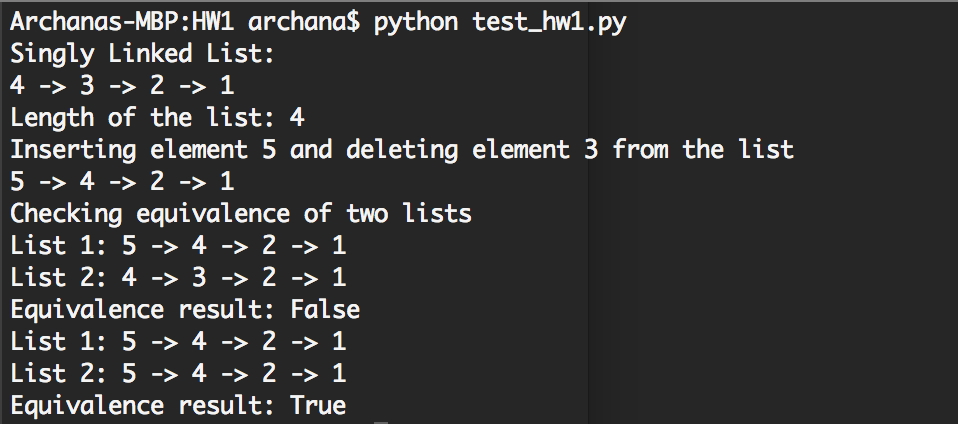
list\_contents=[]

while x is not None:

list\_contents.append(str(x.key))

x = x.next

return " -> ".join(list\_contents)



Question 4

def \_\_len\_\_(self):

if self.head == None:

return 0

count = 0

x = self.head

while x is not None:

count = count+1

x = x.next

return count

Question 5

def \_\_eq\_\_(self,other):

x = self.head

y = other.head

while x is not None and y is not None:

if x.key != y.key:

return False

else:

x = x.next

y = y.next

if x is None and y is None:

return True

Question 6

class CircularDoublyLinkedList:

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

self.sentinel = Element(None)

self.sentinel.next = self.sentinel

self.sentinel.prev = self.sentinel

def insert(self, x):

x.next = self.sentinel.next

self.sentinel.next.prev = x

self.sentinel.next = x

x.prev = self.sentinel

def delete(self,x):

x.prev.next = x.next

x.next.prev = x.prev

def search(self, k):

x = self.sentinel.next

while x != self.sentinel and x.key != k:

x = x.next

if x.key == k:

return x

else:

return None

def \_\_str\_\_(self):

x = self.sentinel.next

list\_contents=[]

while x is not self.sentinel:

list\_contents.append(str(x.key))

x = x.next

return " -> ".join(list\_contents)

def \_\_len\_\_(self):

if self.sentinel.next == self.sentinel:

return 0

count = 0

x = self.sentinel.next

while x is not self.sentinel:

count = count+1

x = x.next

return count

