## Topics

1. Implement Node Class
2. Implement DoublyLinkedList Class
3. Implement Basic Methods of DoublyLinkedList

* isEmpty()
* size()
* first()
* last()
* addFirst()
* addLast()
* removeFirst()
* removeLast()

class Node:

"""Class representing a node in a doubly linked list."""

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

self.data = data # قيمة العقدة

self.prev = prev # الرابط للعقدة السابقة

self.next = next # الرابط للعقدة التالية

class DoublyLinkedList:

"""Class representing a doubly linked list with header and trailer."""

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

self.header = Node() # رأس القائمة (حارس البداية)

self.trailer = Node() # ذيل القائمة (حارس النهاية)

self.header.next = self.trailer

self.trailer.prev = self.header

self.\_size = 0 # عداد حجم القائمة

def isEmpty(self):

"""Check if the list is empty."""

return self.\_size == 0

def size(self):

"""Return the number of elements in the list."""

return self.\_size

def first(self):

"""Return the first element in the list."""

if self.isEmpty():

raise Exception("List is empty")

return self.header.next.data # القيمة الأولى بعد الرأس

def last(self):

"""Return the last element in the list."""

if self.isEmpty():

raise Exception("List is empty")

return self.trailer.prev.data # القيمة الأخيرة قبل الذيل

def addFirst(self, data):

"""Add a new node at the beginning of the list."""

new\_node = Node(data, self.header, self.header.next)

self.header.next.prev = new\_node

self.header.next = new\_node

self.\_size += 1

def addLast(self, data):

"""Add a new node at the end of the list."""

new\_node = Node(data, self.trailer.prev, self.trailer)

self.trailer.prev.next = new\_node

self.trailer.prev = new\_node

self.\_size += 1

def removeFirst(self):

"""Remove and return the first node in the list."""

if self.isEmpty():

raise Exception("List is empty")

to\_remove = self.header.next

self.header.next = to\_remove.next

to\_remove.next.prev = self.header

self.\_size -= 1

return to\_remove.data # القيمة المحذوفة

def removeLast(self):

"""Remove and return the last node in the list."""

if self.isEmpty():

raise Exception("List is empty")

to\_remove = self.trailer.prev

self.trailer.prev = to\_remove.prev

to\_remove.prev.next = self.trailer

self.\_size -= 1

return to\_remove.data # القيمة المحذوفة

# اختبار الكود

if \_\_name\_\_ == "\_\_main\_\_":

# إنشاء قائمة مزدوجة

dll = DoublyLinkedList()

# إضافة قيم

dll.addFirst(10)

dll.addLast(20)

dll.addLast(30)

# طباعة القيم

print("First:", dll.first()) # 10

print("Last:", dll.last()) # 30

print("Size:", dll.size()) # 3

# إزالة قيم

dll.removeFirst() # إزالة أول عنصر

dll.removeLast() # إزالة آخر عنصر

# التحقق من القائمة

print("First:", dll.first()) # 20

print("Size:", dll.size()) # 1

## Homework

1. Describe a method for finding the middle node of a doubly linked list with header and trailer sentinels by “link hopping,” and without relying on explicit knowledge of the size of the list. In the case of an even number of nodes, report the node slightly left of center as the “middle.”

def findMiddle(self):

if self.isEmpty():

return None

left\_pointer = self.header.next

right\_pointer = self.trailer.prev

while left\_pointer != right\_pointer and left\_pointer.next != right\_pointer:

left\_pointer = left\_pointer.next

right\_pointer = right\_pointer.prev

return left\_pointer # العقدة الوسطى

1. Give an implementation of the size( ) method for the DoublyLinkedList class, assuming that we did not maintain size as an instance variable.

def size(self):

count = 0

current = self.header.next

while current != self.trailer:

count += 1

current = current.next

return count

1. Implement the equals( ) method for the DoublyLinkedList class.

def equals(self, other):

if self.size() != other.size():

return False

node1 = self.header.next

node2 = other.header.next

while node1 != self.trailer and node2 != other.trailer:

if node1.data != node2.data:

return False

node1 = node1.next

node2 = node2.next

return True

1. Give an algorithm for concatenating two doubly linked lists L and M, with header and trailer sentinel nodes, into a single list L′.

def concatenate(self, other):

if self.isEmpty():

self.header.next = other.header.next

other.header.next.prev = self.header

elif not other.isEmpty():

self.trailer.prev.next = other.header.next

other.header.next.prev = self.trailer.prev

self.trailer = other.trailer

other.header = None

other.trailer = None

1. Our implementation of a doubly linked list relies on two sentinel nodes, header and trailer, but a single sentinel node that guards both ends of the list should suffice. Reimplement the DoublyLinkedList class using only one sentinel node.

class SingleSentinelDoublyLinkedList:

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

self.sentinel = Node(None)

self.sentinel.next = self.sentinel

self.sentinel.prev = self.sentinel

def isEmpty(self):

return self.sentinel.next == self.sentinel

def addFirst(self, data):

new\_node = Node(data, self.sentinel, self.sentinel.next)

self.sentinel.next.prev = new\_node

self.sentinel.next = new\_node

def addLast(self, data):

new\_node = Node(data, self.sentinel.prev, self.sentinel)

self.sentinel.prev.next = new\_node

self.sentinel.prev = new\_node

1. Implement a circular version of a doubly linked list, without any sentinels, that supports all the public behaviors of the original as well as two new update methods, rotate( ) and rotateBackward.

class CircularDoublyLinkedList:

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

self.header = None

def isEmpty(self):

return self.header is None

def addLast(self, data):

new\_node = Node(data)

if self.isEmpty():

self.header = new\_node

self.header.next = self.header

self.header.prev = self.header

else:

tail = self.header.prev

tail.next = new\_node

new\_node.prev = tail

new\_node.next = self.header

self.header.prev = new\_node

def rotate(self):

if not self.isEmpty():

self.header = self.header.next

def rotateBackward(self):

if not self.isEmpty():

self.header = self.header.prev

1. Implement the clone( ) method for the DoublyLinkedList class.

def clone(self):

new\_list = DoublyLinkedList()

current = self.header.next

while current != self.trailer:

new\_list.addLast(current.data)

current = current.next

return new\_list