# ASSIGNMENT

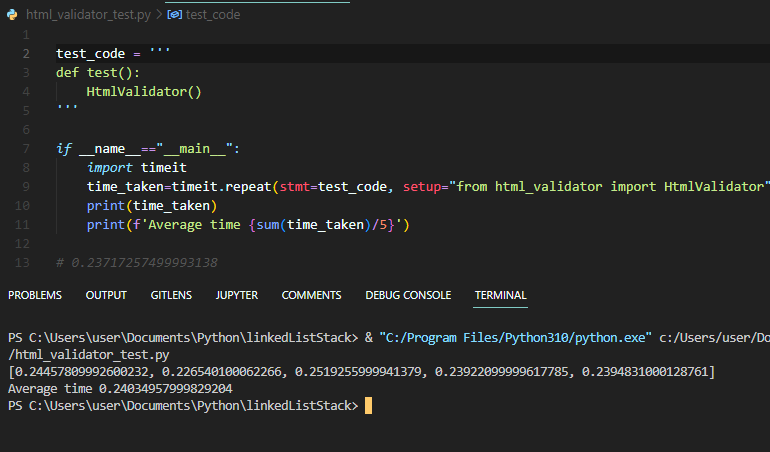
## BENCHMARK ANALYSIS

### QUESTION A

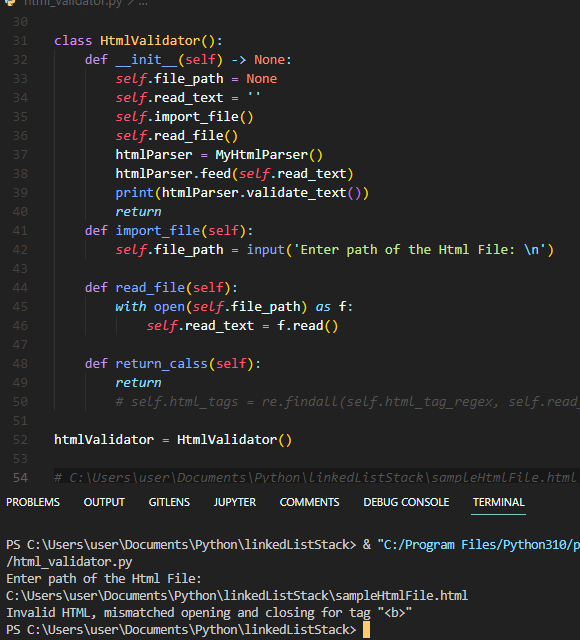
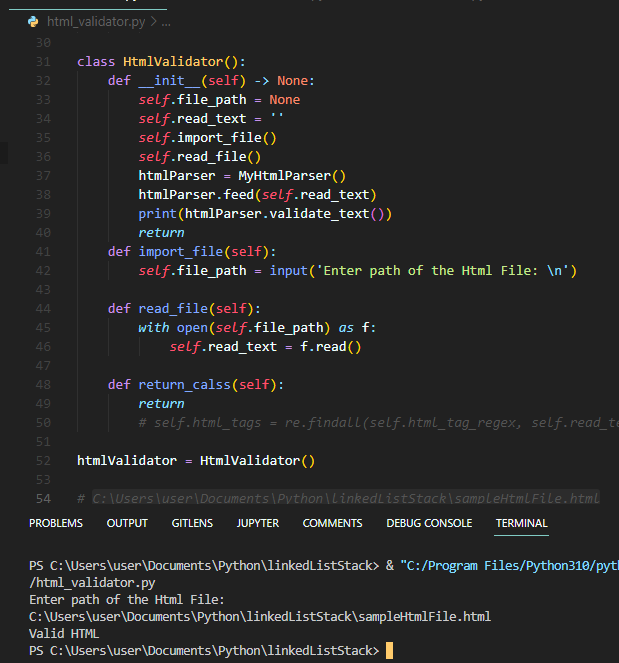
#### HTML VALIDATOR TEST RUNS

|  |  |
| --- | --- |
| tEST nO. | TIME OF EXECUTION |
|  | 0.24457809992600232 |
|  | 0.226540100062266 |
|  | 0.2519255999941379 |
|  | 0.23922099999617785 |
|  | 0.2394831000128761 |
| Average | **0.24034957999829204** |

**Test Results using TimeIt**



**Program Output Screen Capture**

****

### QUESTION B

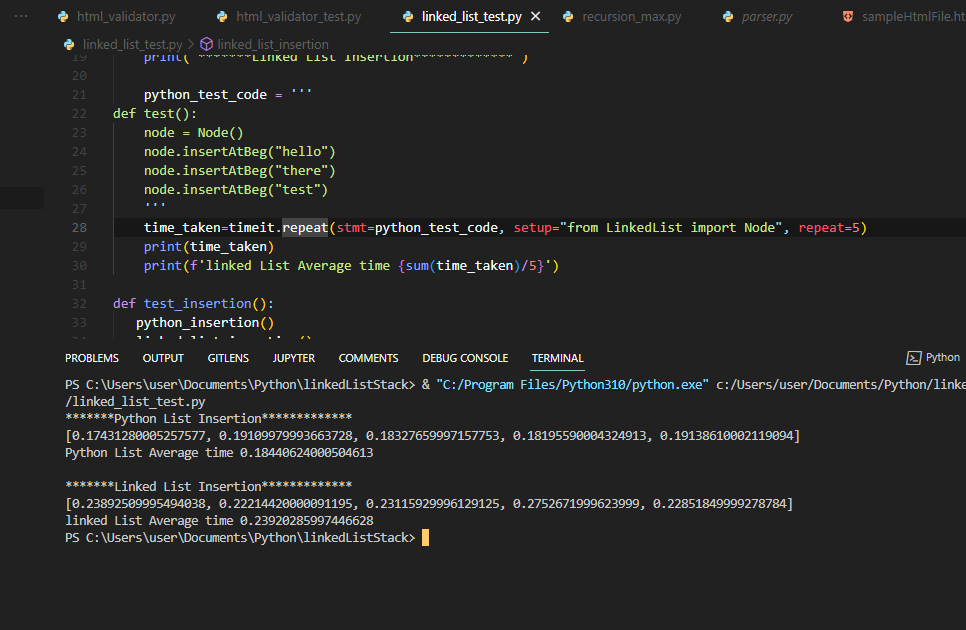
#### PYTHON LIST COMPARISON TO A LIST IMPLEMENTED USING A LINKED LIST

**TEST ONE.**

***Testing INSERTION of an element to both the lists***

|  |  |  |
| --- | --- | --- |
| tEST nO. | Python list | linked list |
|  | 0.17431280005257577 | 0.23892509995494038 |
|  | 0.19109979993663728 | 0.22214420000091195 |
|  | 0.18327659997157753 | 0.23115929996129125 |
|  | 0.18195590004324913 | 0.2752671999623999 |
|  | 0.19138610002119094 | 0.22851849999278784 |
| Average | **0.18440624000504613** | **0.23920285997446628** |

**Test Runs**

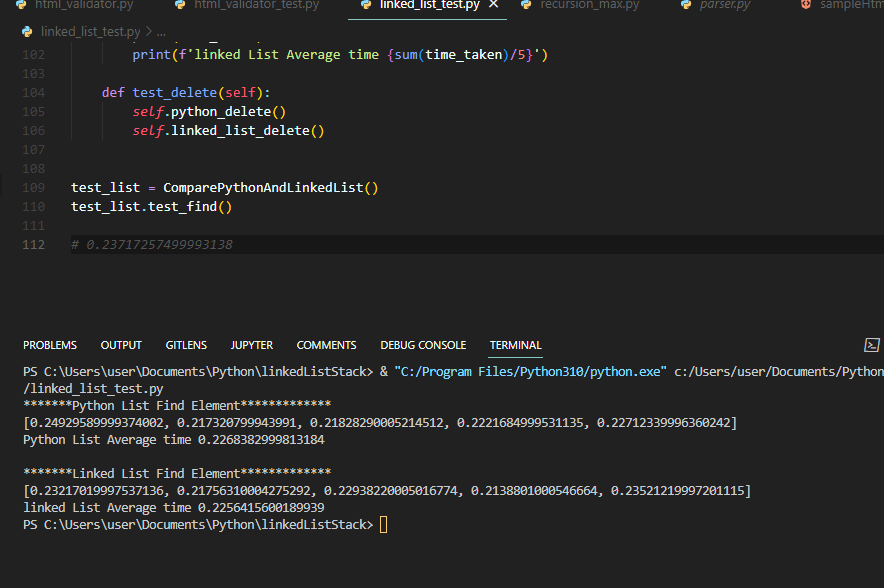


**TEST TWO.**

***Testing FINDING OR SEARCHING for an element from both the lists***

|  |  |  |
| --- | --- | --- |
| tEST nO. | Python list | linked list |
|  | 0.24929589999374002 | 0.23217019997537136 |
|  | 0.217320799943991 | 0.21756310004275292 |
|  | 0.21828290005214512 | 0.22938220005016774 |
|  | 0.2221684999531135 | 0.2138801000546664 |
|  | 0.22712339996360242 | 0.23521219997201115 |
| Average | **0.2268382999813184** | **0.2256415600189939** |

**Test Runs**

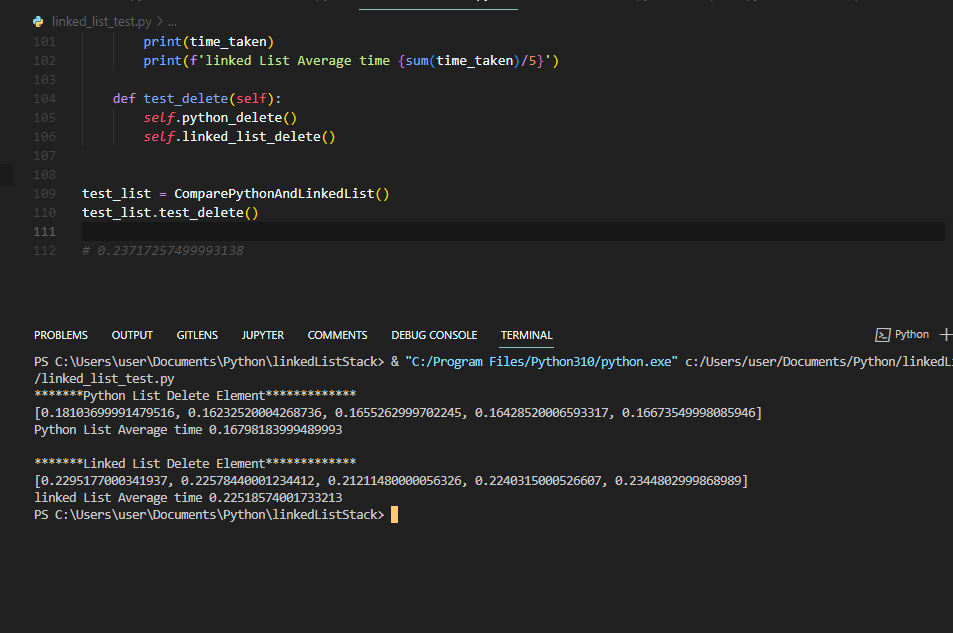


**TEST THREE.**

***Testing DELETION an element from both the lists***

|  |  |  |
| --- | --- | --- |
| tEST nO. | Python list | linked list |
|  | 0.18103699991479516 | 0.2295177000341937 |
|  | 0.16232520004268736 | 0.22578440001234412 |
|  | 0.1655262999702245 | 0.21211480000056326 |
|  | 0.16428520006593317 | 0.2240315000526607 |
|  | 0.16673549998085946 | 0.2344802999868989 |
| Average | **0.16798183999489993** | **0.22518574001733213** |

**Test Run**

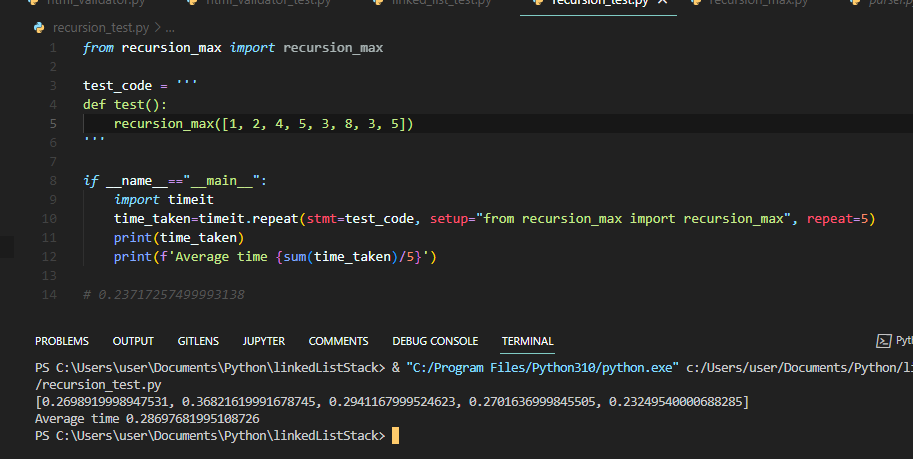


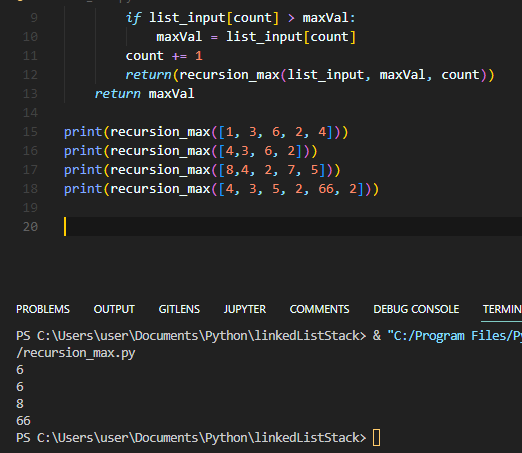
### QUESTION C

#### PROGRAM THAT CONTAINS A RECURSIVE PYTHON FUNCTION TO FIND THE MAXIMUM VALUES IN A LIST WITHOUT USING A LOOP

|  |  |
| --- | --- |
| tEST nO. | TIME OF EXECUTION |
|  | 0.2698919998947531 |
|  | 0.36821619991678745 |
|  | 0.2941167999524623 |
|  | 0.2701636999845505 |
|  | 0.23249540000688285 |
| Average | **0.28697681995108726** |

**Test Runs**





## TIME COMPLEXITY ANALYSIS

### QUESTION A

#### HTML VALIDATOR QUESTION

We solve this problem by the use of a Stack ADT

In the html validator question, we have to traverse through the whole document that may be of length (n).

In the case where we meet an opening tag, then we push the tag to the stack, a process that takes constant time O(1) since we always be adding to the top of the stack.

In the case where we meet a closing tag, then we pop if the immediate tag on top of the stack is the opening tag of the closing time. This operation also takes a constant time of O(1).

Therefore:

Traverse the document: O(n)

Push to the top of the stack: O(1)

Pop from the top of the stack: O(1)

Thus:

### QUESTION B

#### COMPARISION BETWEEN PYTHON LIST AND LINKED LIST ADT

* **Insertion**

**Python list**

Inserting an element at any position in a python list would take a worst case scenario of a linear time that is time complexity since we may be inserting the element at the end of the list.

We may also have a problem of increasing the size of the list incase more data is to be inserted.

**Linked list**

Insert at the beginning

Since we have the pointer to the head of the linked list, to insert an element to the head of the linked list will just involve changing the pointer of the head to the new Node and updating the new Node to point to the previous first Node.

This operation takes constant time of

Insert at the end or in between

To Insert an element to the end of the linked list would need us to traverse the whole linked list if it is a singly linked list and that would take a linear time of the size of the linked list that is and of it is a doubly linked list since we already have the tail pointing to the last Element.

To insert at a position, It would take in worst case since it maybe at the last position.

* **Finding an element**

**Python list**

Finding an element at any position in a python list takes constant time that is time complexity.

Since a python list is like a contiguous block of memory with addresses, getting an element involves just accessing its memory block that it’s stored at.

**Linked list**

To find an in a linked list, we would need to traverse the whole linked list if it is a singly linked list and that would take a linear time of the size of the linked list that is and of it is a doubly linked list since we already have the tail pointing to the last Element.

To get at a position, It would take in worst case since it maybe at the last position and Best case of ) that is if it’s the first element

* **Deleting an element**

**Python list**

Deleting an element at any position in a python list would take a worst case scenario of a linear time that is time complexity since we may be deleting the element at the end of the list.

We may also have a problem of decreasing the size of the list in order to free memory.

**Linked list**

Deleting at the beginning

Since we have the pointer to the head of the linked list, to delete an element to the head of the linked list will just involve changing the pointer of the head to the next.

This operation takes constant time of

Delete at the end or in between

To delet an element to the end of the linked list would need us to traverse the whole linked list if it is a singly linked list and that would take a linear time of the size of the linked list that is and of it is a doubly linked list since we already have the tail pointing to the last Element.

To delete at a position, It would take in worst case since it maybe at the last position.

### QUESTION C

#### PROGRAM THAT CONTAINS A RECURSIVE PYTHON FUNCTION TO FIND THE MAXIMUM VALUES IN A LIST WITHOUT USING A LOOP

For a recursive function, we may need to traverse the whole list recursively and compare the elements while maintaining the largest value that has been found.

If the size of the array is

## DISCUSSION

### QUESTION A

#### HTML VALIDATOR TEST

In this question, I used a stack ADT to insert opening tags and subsequently pop the tags if I find matching ending tags.

The algorithm uses linear time of for traversing and reading the whole html document and finding patterns matching html opening and closing tags with exception of self-closing tags.

I also stripped off the elbow brackets around the tags to facilitate for easy comparison.

With linear time execution, it can be said that the algorithm is efficient and is determined with the length of the html document.

### QUESTION B

#### PYTHON LIST COMPARISON TO A LIST IMPLEMENTED USING A LINKED LIST

A linked list is an Abstract Data Type that brings in the efficiency of being able to insert data in constant time of.

With a singly linked list, we can have a node which has the data and the pointer to the next node.

|  |  |
| --- | --- |
| Data | Next |

Node

A Doubly linked list has a node the data, pointer to the next node and pointer to the previous node.

|  |  |  |
| --- | --- | --- |
| Previous | Data | Next |

Node

There are other more types like Cyclic linked list in which the last nodes next pointer points to the first node etc.

Linked list can be able to grow gracefully with the influx of more data or input, a feature that an array is not good at as an array requires another space to be allocated for the next data.

This makes a linked list faster than an array in some operations like;

Insertion and finding nth element.

### QUESTION C

#### PROGRAM THAT CONTAINS A RECURSIVE PYTHON FUNCTION TO FIND THE MAXIMUM VALUES IN A LIST WITHOUT USING A LOOP

Using recursion to find the largest value in an array that is not ordered is would consume more memory but would take the same time as the iterative approach.

If the recursive function is not terminated once the base case is reached, then the recursive case may execute until the system is out of memory as each time the recursive function is called, then a memory space is allocated for it.

## PYTHON SOURCE CODE

### QUESTION A

#### HTML VALIDATOR QUESTION

*Stack.py*

*from*  LinkedList *import* Node

class Stack():

    def \_\_init\_\_(self, data=None) -> None:

*self*.head = None

*self*.length = 0

*if* data:

*for* data *in* data:

*self*.push(data)

    def push(self, data):

*"""*

*Worst case Time Complexity is O(1) since we are inserting an element on top of the stack*

*Space Complexity is O(1) since we are creating one temporary variable*

*"""*

        node = Node()

        node.setData(data)

        node.setNext(*self*.head)

*self*.head = node

*self*.length += 1

    def pop(self):

*"""*

*Worst case Time Complexity is O(1) since we are removing an element on top of the stack*

*Space Complexity is O(1) since we are creating one temporary variable*

*"""*

*if* *self*.head == None:

*raise* IndexError

*else*:

            data = *self*.head.getData()

*self*.head = *self*.head.getNext()

*self*.length -= 1

*return*(data)

    def peek(self):

*if* *self*.head == None:

*raise* IndexError

*return* *self*.head.getData()

    def all(self):

        all\_list: list = []

*if* *self*.head == None:

*raise* IndexError

        current = *self*.head

        index: int = 0

*while* index < *self*.length:

            all\_list.append(current.getData())

            current = current.getNext()

            index += 1

*return* all\_list

*# ourList = ["first", "second", "third", "fouth"]*

*# stack = Stack(ourList)*

*# stack.peek()*

*# stack.pop()*

*# print(stack.all())*

*html\_validator.py*

*from* Stack *import* Stack

*from* html.parser *import*  HTMLParser

class MyHtmlParser(HTMLParser):

    def \_\_init\_\_(self, \*, convert\_charrefs: bool = ...) -> None:

        super().\_\_init\_\_(convert\_charrefs=convert\_charrefs)

*self*.rawdata = ''

*self*.stack = Stack()

*self*.unclosed\_tag = ''

    def handle\_starttag(self, tag, attrs):

*if* not tag in ['meta']:

*self*.stack.push(tag)

*return*(f"Pushed Start Tag: {tag}")

    def handle\_endtag(self, tag):

*if* tag == *self*.stack.peek():

*self*.stack.pop()

*return*(f"Popped End Tag: {tag}")

*else*:

*if* *self*.unclosed\_tag == '':

*self*.unclosed\_tag = *self*.stack.peek()

    def validate\_text(self):

*if* *self*.unclosed\_tag == '':

*return* 'Valid HTML'

*else*:

*return* f'Invalid HTML, mismatched opening and closing for tag "<{*self*.unclosed\_tag}>"'

class HtmlValidator():

    def \_\_init\_\_(self) -> None:

*self*.file\_path = None

*self*.read\_text = ''

*self*.import\_file()

*self*.read\_file()

        htmlParser = MyHtmlParser()

        htmlParser.feed(*self*.read\_text)

        print(htmlParser.validate\_text())

*return*

    def import\_file(self):

*self*.file\_path = input('Enter path of the Html File: \n')

    def read\_file(self):

*with* open(*self*.file\_path) *as* f:

*self*.read\_text = f.read()

    def return\_calss(self):

*return*

*# self.html\_tags = re.findall(self.html\_tag\_regex, self.read\_text)*

htmlValidator = HtmlValidator()

*html\_validator\_test.py*

test\_code = '''

def test():

    HtmlValidator()

'''

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

*import* timeit

    time\_taken=timeit.repeat(stmt=test\_code, setup="from html\_validator import HtmlValidator", repeat=5)

    print(time\_taken)

    print(f'Average time {sum(time\_taken)/5}')

*# 0.23717257499993138*

### QUESTION B

#### PYTHON LIST COMPARISON TO A LIST IMPLEMENTED USING A LINKED LIST

*Linked\_List.py*

class Node():

    def \_\_init\_\_(self) -> None:

*self*.length = 0

*self*.head= None

*self*.data= None

*self*.next= None

    def setData(self, data):

*self*.data = data

    def getData(self):

*return* *self*.data

    def setNext(self, next):

*self*.next = next

    def getNext(self):

*return* *self*.next

    def getAll(self):

*"""*

*Worst case Time Complexity is O(n) since we are traversing the whole node data structure*

*Space Complexity is O(1) since we are creating one temporary variable*

*"""*

        list\_node = []

*if* *self*.head == None:

*raise* ValueError

        current: Node = *self*.head

        index: int = 0

*while* index < *self*.length:

            list\_node.append(current.getData())

            current = current.getNext()

            index += 1

*return* list\_node

    def insertAtBeg(self, data):

*"""*

*Worst case Time Complexity is O(1) since we are inserting at the beginning of the linkedList*

*Space Complexity is O(1) since we are creating one temporary variable*

*"""*

        newNode = Node()

        newNode.setData(data)

*if* *self*.length != 0:

*self*.head = newNode

*else*:

            newNode.setNext(*self*.head)

*self*.head = newNode

*self*.length += 1

    def insertAtEnd(self, data):

*"""*

*Worst case Time Complexity is O(n) since we are inserting at the end of the linkedList that is we are traversing*

*the whole list inorder for insertion*

*Space Complexity is O(1) since we are creating one temporary variable*

*"""*

        newNode = Node()

        newNode.setData(data)

        current = *self*.head

*while* current.getNext() != None:

            current = *self*.getNext()

        current.setNext(newNode)

*self*.length +=1

    def insertAtPos(self, data, pos):

*"""*

*Worst case Time Complexity is O(n) since we may insert at the last position of the linkedList that is we are traversing*

*the whole list inorder for insertion*

*Space Complexity is O(1) since we are creating one temporary variable*

*"""*

*if* pos> *self*.length or pos < 0:

*return* None

*else*:

*if* pos == 0:

*self*.insertAtBeg(data)

*else*:

                newNode = Node()

                newNode.setData(data)

                count = 0

                current = *self*.head

*while* count < pos -1:

                    count +=1

                    current = current.getNext()

                newNode.setNext(current.getNext)

                current.setNext(newNode)

*self*.length +=1

    def delAtBeg(self):

*"""*

*Worst case Time Complexity is O(1) since we are deleting at the first position of the linked list*

*Space Complexity is O(1) since we are creating one temporary storage for the node to be deleted that is later*

*dropped*

*"""*

*if* *self*.length == 0:

*raise* IndexError

*else*:

*self*.head = *self*.head.getNext()

*self*.length -= 1

    def delAtEnd(self):

*"""*

*Worst case Time Complexity is O(n) since we are deleting at the end of the linkedList that is we are traversing*

*the whole list inorder for insertion*

*Space Complexity is O(1) since we are creating one temporary variable*

*"""*

*if* *self*.length == 0:

*raise* IndexError

*else*:

            current = *self*.head

            index = 0

*while* index < *self*.length - 1:

                index += 1

                current = current.getNext()

            current.setNext(None)

*self*.length -= 1

    def delNode(self, node):

*"""*

*Worst case Time Complexity is O(n) since we are deleting at the end of the linkedList that is we are traversing*

*the whole list inorder for insertion*

*Space Complexity is O(1) since we are creating one temporary variable*

*"""*

*if* *self*.length == 0:

*raise* ValueError("Empty List")

*else*:

            current = *self*.head

            previous = None

            found = False

*while* not found:

*if* current == node:

                    found= True

*elif* current is None:

*raise*(ValueError(f'Linked List does not have a node {node}'))

*else*:

                    previous = current

                    current = current.getNext()

*if* previous is None:

*self*.head = current.getNext()

*else*:

                previous.setNext(current.getNext())

*self*.length -= 1

    def delVal(self, value):

*"""*

*Worst case Time Complexity is O(n) since we are deleting at the end of the linkedList that is we are traversing*

*the whole list inorder for insertion*

*Space Complexity is O(1) since we are creating one temporary variable*

*"""*

        currnode= *self*.head

        prevnode= *self*.head

*while* currnode.getNext() != None:

*if* currnode.data == value:

                prevnode.setNext(currnode.getNext())

*self*.length -= 1

*return*

*else*:

                prevnode = currnode

                currnode = currnode.getNext()

*raise* ValueError

    def delAtPos(self, pos):

*"""*

*Worst case Time Complexity is O(n) since we are deleting at the end of the linkedList that is we are traversing*

*the whole list inorder for insertion*

*Space Complexity is O(1) since we are creating one temporary variable*

*"""*

        index = 0

        currnode = *self*.head

        prevnode = *self*.head

*if* *self*.length == 0:

*raise* IndexError

*else*:

*if* pos < 0:

                pos = (*self*.length-1) + pos

*if* pos > *self*.length -1 or pos < 0:

*raise* IndexError

*else*:

*while* prevnode.next != None or index < pos:

                    index += 1

*if* index == pos:

                        prevnode.setNext(currnode.getNext())

*self*.length -= 1

*return*

*else*:

                        prevnode = currnode

                        currnode = currnode.getNext()

    def indexOf(self, value):

*"""*

*Worst case Time Complexity is O(n) since we can find an element at the end of the node*

*Space Complexity is O(1) since we are creating one temporary variable*

*"""*

        currnode= *self*.head

        index = 0

*while* currnode.getNext() != None:

*if* currnode.data == value:

*return* index

*else*:

                currnode = currnode.getNext()

                index += 1

*raise* ValueError

    def valueOf(self, pos):

*"""*

*Worst case Time Complexity is O(n) since we can find an element at the end of the node*

*Space Complexity is O(1) since we are creating one temporary variable*

*"""*

        index = 0

        currnode = *self*.head

*if* *self*.length == 0:

*raise* IndexError

*else*:

*if* pos < 0:

                pos = (*self*.length-1) + pos

*if* pos > *self*.length -1 or pos < 0:

*raise* IndexError

*else*:

*while* currnode.next != None or index < pos:

                    index += 1

*if* index == pos:

*return*(currnode.getData())

*else*:

                        currnode = currnode.getNext()

*linked\_list\_test.py*

*import* timeit

*from* LinkedList *import* Node

class ComparePythonAndLinkedList():

    def \_\_init\_\_(self) -> None:

*pass*

    def python\_insertion(self):

        print("\*\*\*\*\*\*\*Python List Insertion\*\*\*\*\*\*\*\*\*\*\*\*\*")

        python\_test\_code = '''

def test():

    listElement = []

    list.append("hello")

    list.append("there")

    list.append("test")

        '''

        time\_taken=timeit.repeat(stmt=python\_test\_code,  repeat=5)

        print(time\_taken)

        print(f'Python List Average time {sum(time\_taken)/5}\n')

    def linked\_list\_insertion(self):

        print("\*\*\*\*\*\*\*Linked List Insertion\*\*\*\*\*\*\*\*\*\*\*\*\*")

        python\_test\_code = '''

def test():

    node = Node()

    node.insertAtBeg("hello")

    node.insertAtBeg("there")

    node.insertAtBeg("test")

        '''

        time\_taken=timeit.repeat(stmt=python\_test\_code, setup="from LinkedList import Node", repeat=5)

        print(time\_taken)

        print(f'linked List Average time {sum(time\_taken)/5}')

    def test\_insertion(self):

*self*.python\_insertion()

*self*.linked\_list\_insertion()

    def python\_find(self):

        print("\*\*\*\*\*\*\*Python List Find Element\*\*\*\*\*\*\*\*\*\*\*\*\*")

        list = []

        python\_test\_code = '''

def test():

    listElement = []

    list.append("hello")

    list.append("there")

    list.append("test")

    list.index(test)

        '''

        time\_taken=timeit.repeat(stmt=python\_test\_code,  repeat=5)

        print(time\_taken)

        print(f'Python List Average time {sum(time\_taken)/5}\n')

    def linked\_list\_find(self):

        print("\*\*\*\*\*\*\*Linked List Find Element\*\*\*\*\*\*\*\*\*\*\*\*\*")

        python\_test\_code = '''

def test():

    node = Node()

    node.insertAtBeg("hello")

    node.insertAtBeg("there")

    node.insertAtBeg("test")

    node.indexOf("test")

        '''

        time\_taken=timeit.repeat(stmt=python\_test\_code, setup="from LinkedList import Node", repeat=5)

        print(time\_taken)

        print(f'linked List Average time {sum(time\_taken)/5}')

    def test\_find(self):

*self*.python\_find()

*self*.linked\_list\_find()

    def python\_delete(self):

        print("\*\*\*\*\*\*\*Python List Delete Element\*\*\*\*\*\*\*\*\*\*\*\*\*")

        list = []

        python\_test\_code = '''

def test():

    listElement = []

    list.append("hello")

    list.append("there")

    list.append("test")

    listElement.remove("test")

        '''

        time\_taken=timeit.repeat(stmt=python\_test\_code,  repeat=5)

        print(time\_taken)

        print(f'Python List Average time {sum(time\_taken)/5}\n')

    def linked\_list\_delete(self):

        print("\*\*\*\*\*\*\*Linked List Delete Element\*\*\*\*\*\*\*\*\*\*\*\*\*")

        python\_test\_code = '''

def test():

    node = Node()

    node.insertAtBeg("hello")

    node.insertAtBeg("there")

    node.insertAtBeg("test")

    node.delVal("test")

        '''

        time\_taken=timeit.repeat(stmt=python\_test\_code, setup="from LinkedList import Node", repeat=5)

        print(time\_taken)

        print(f'linked List Average time {sum(time\_taken)/5}')

    def test\_delete(self):

*self*.python\_delete()

*self*.linked\_list\_delete()

test\_list = ComparePythonAndLinkedList()

test\_list.test\_delete()

*# 0.23717257499993138*

### QUESTION C

#### PROGRAM THAT CONTAINS A RECURSIVE PYTHON FUNCTION TO FIND THE MAXIMUM VALUES IN A LIST WITHOUT USING A LOOP

*Recursion\_max.py*

def recursion\_max(list\_input: list, maxVal: int=0, count:int=0):

*"""*

*Worst case Time Complexity is O(n) since we can find the last value at the end of the list*

*Space Complexity is O(n) since we are allocating memory space each time we call the recursive function*

*dropped*

*"""*

*if* count< len(list\_input):

*if* list\_input[count] > maxVal:

            maxVal = list\_input[count]

        count += 1

*return*(recursion\_max(list\_input, maxVal, count))

*return* maxVal

print(recursion\_max([1, 3, 6, 2, 4]))

print(recursion\_max([4,3, 6, 2]))

print(recursion\_max([8,4, 2, 7, 5]))

print(recursion\_max([4, 3, 5, 2, 66, 2]))

*recursion\_test.py*

*from* recursion\_max *import* recursion\_max

test\_code = '''

def test():

    recursion\_max([1, 2, 4, 5, 3, 8, 3, 5])

'''

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

*import* timeit

    time\_taken=timeit.repeat(stmt=test\_code, setup="from recursion\_max import recursion\_max", repeat=5)

    print(time\_taken)

    print(f'Average time {sum(time\_taken)/5}')

*# 0.23717257499993138*