

Lab-VI

Hasan Amin

(374866)

CS-250

Data Structures and Algorithms

School of Natural Sciences

# SLL Class:

|  |
| --- |
| **Code** |

class Node:

    def \_\_init\_\_(self, value):

        self.data = value

        self.nextnode = None

class SLL:

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

        self.head = Node(data)

    def isEmpty(self):  # Test for empty list

        return self.head is None  # True if & only if no 1st Link

    def traverse(self, nextnode):   # i give it the next node for the particular node and it returns me that particular node

        currnode = self.head

        while currnode.nextnode != nextnode:

            currnode = currnode.nextnode

        return currnode

    def get\_previous\_node(self, node: Node):

        currnode = self.head

        while currnode.nextnode != node:

            currnode = currnode.nextnode

        return currnode

    def traverse\_by\_index(self, index):

        currnode = self.head

        for i in range(index):

            currnode = currnode.nextnode

        return currnode

    def append(self, data):

        new\_node = Node(data)

        last\_node = self.traverse(None)

        last\_node.nextnode = new\_node

    def insert(self, index, data):

        new\_node = Node(data)

        if index > self.count() + 1:

            raise ValueError("List index out of range")

        that\_node = self.traverse\_by\_index(index)

        if that\_node != self.head:

            prev\_node = self.get\_previous\_node(that\_node)

            prev\_node.nextnode = new\_node

            new\_node.nextnode = that\_node

        else:

            new\_node.nextnode = that\_node

            self.head = new\_node

    def deleteFirst(self):

        if self.isEmpty():  # Empty list? Raise an exception

            raise Exception("Cannot delete first of empty list")

        curr\_node = self.head

        self.head = self.head.nextnode

        return curr\_node

    def deleteLast(self):

        curr\_node = self.head

        if curr\_node == None or curr\_node.nextnode == None:

            self.head = None

            return curr\_node

        temp = self.head

        while temp.nextnode is not None:

            current = temp

            temp = temp.nextnode

        current.nextnode = None

        return temp.data

    def SearchElement(self, val):  # search element in linked list

        print(f"Searching for element {val}")

        if self.isEmpty():

            print("nothing to remove,list is empty")

            return

        temp = self.head

        while temp.nextnode is not None:

            if temp.data == val:

                print(f"Value Found {temp.data}")

                return True

            temp = temp.nextnode

        if temp.data == val:

            print(f"\nElement Found {temp.data}")

        else:

            return False

    def count(self): # count the number of nodes

        currnode = self.head

        counter = 0

        while currnode is not None:

            counter += 1

            currnode = currnode.nextnode

        return counter

    def unique(self):

        if self.isEmpty():

            return  # No duplicates to remove from an empty list

        current = self.head

        while current is not None:

            # Compare the current node's data with subsequent nodes

            runner = current

            while runner.nextnode is not None:

                if runner.nextnode.data == current.data:

                    # Remove the duplicate node

                    runner.nextnode = runner.nextnode.nextnode

                else:

                    runner = runner.nextnode

            current = current.nextnode

    def removeEvens(self):

        sll2 = SLL(self.head.data)

        currnode = self.head

        self.remove(self.head.data)

        while currnode.nextnode.nextnode is not None:

            currnode = currnode.nextnode.nextnode

            sll2.append(currnode.data)

            self.remove(currnode.data)

            if currnode.nextnode is None:

                break

        return sll2

    def groupOddEvens(self):

        even = self.removeEvens()

        currnode = self.head

        while currnode.nextnode is not None:

            currnode = currnode.nextnode

        currnode.nextnode = even.head

    def count50s(self):

        c = 0

        currnode = self.head

        while currnode is not None:

            if currnode.data == 50:

                c += 1

            currnode = currnode.nextnode

        return c

    def remove(self, info):

        currnode = self.head

        if currnode is None:

            raise ValueError(f"{info} not found in list")

        if currnode.data == info:

            self.head = currnode.nextnode

        else:

            while currnode.nextnode is not None and currnode.nextnode.data != info:

                currnode = currnode.nextnode

            if currnode.nextnode and currnode.nextnode.data == info:

                currnode.nextnode = currnode.nextnode.nextnode

            else:

                raise ValueError(f"{info} not found in list")

    def pop(self, index): # removing node from a specific location/index

        if self.isEmpty():  # Empty list? Raise an exception

            raise ValueError("List is Empty")

        elif index > len(self):

            raise ValueError("list index out of range")

        else:

            toDelete = self.traverse\_by\_index(index)

            self.remove(toDelete.data)

    def destroy(self):

        currnode = self.head

        while currnode is not None:

            temp = currnode.nextnode

            del currnode

            currnode = temp

        self.head = None  # Set the head to None to indicate an empty list

    def \_\_len\_\_(self) -> int:

        return self.count()

    def \_\_str\_\_(self):  # Build a string representation

        result = "["  # Enclose list in square brackets

        cur\_node = self.head  # Start with first link

        while cur\_node is not None:  # Keep going until no more links

            if len(result) > 1:  # After first link,

                result += " --> "  # separate links with right arrowhead

            result += str(cur\_node.data)  # Append string version of link

            cur\_node = cur\_node.nextnode  # Move on to nextnode link

        return result + "]"  # Close with square bracket

    def \_\_eq\_\_(self, obj: object) -> bool:

        if (

            len(self) != len(obj)

            or (self is None and obj is not None)

            or (self is not None and obj is None)

        ):

            return False

        curr1 = self.head

        curr2 = obj.head

        while curr1.data is not None and curr2.data is not None:

            if curr1.data != curr2.data:

                return False

            curr2 = curr2.nextnode

            curr1 = curr1.nextnode

        return True

    def \_\_ne\_\_(self, obj: object) -> bool:

        return not self.\_\_eq\_\_(obj)

# Class Method Testing:

|  |
| --- |
| Code |
| from SinglyLinkedList import SLL  sll=SLL("abc")  sll.append("abc")  sll.append("abc")  sll.append("aec")  sll.append("axc")  sll.append("ahc")  sll.append(50)  sll.append(50)  sll.append(50)  sll.append(50)  sll.append(234)  sll.insert(0,53)  sll.insert(3,53)  sll.insert(2,100)  print(f"Linked List: {sll}   Count of 50: {sll.count50s()}")  sll.unique()  print(f"Duplicates removed: {sll}")  sll.groupOddEvens()  print(f"OddEvens Grouped: {sll}")  sll.remove("axc")  print(f'Removed "axc" from my SLL: {sll}')  sll.pop(0)  print(f'Popped the node at 0th index from my SLL: {sll}')  mysll2=SLL()  mysll2=sll.removeEvens()  print(f"Odd Nodes:{sll} Even Nodes: {mysll2}")  mysll2.destroy()  print(f"Destroyed My Even Nodes SLL:{mysll2}") |

|  |
| --- |
| Output |
|  |