**SSN College of Engineering**

Department of Information Technology

UIT2201 — Programming and Data Structures

2022 – 2023

**Exercise — 10**

U. Pranaav | IT-B | 3122225002093

1)

I. AIM:

To design and implement Linked List with the following operations.

• Isempty

• Display

• Find

• append

• Insert by pos

• Delete by pos

• Insert by specifying previous value.

• Delete by specifying previous value.

II. CODE:

# -\*- coding: utf-8 -\*-

'''

This module provides a class for implementation of the singly linked

list created using a Node class. This is a part of the

exercises given under the course UIT2201 (Programming and Data

Structures).

In this source code I have executed my own logic. The code

follows good coding practices.

Your comments and suggestions are welcome.

Created on Wed June 7 2023

Revised on Wed June 10 2023

Original Author: U. Pranaav <pranaav2210205@ssn.edu.in>

'''

*class* Node:

    \_\_slots\_\_ = ['item', 'next']

*def* \_\_init\_\_(*self*,*item*=None,*next*=None):

        '''

        Node class represents a node in a singly linked list.

        Args:

            item: The item/value stored in the node.

            next: Reference to the next node.

        '''

*self*.item = *item*

*self*.next = *next*

*class* SinglyLinkedList:

    '''

    The SinglyLinkedList class represents a singly linked list.

    Attributes:

        head (Node): The head node of the linked list.

        tail (Node): The tail node of the linked list.

    Methods:

        \_\_init\_\_(): Initializes an empty linked list with a head and tail node.

        isempty(): Checks if the linked list is empty.

        append(ele): Appends an element to the end of the linked list.

        \_\_str\_\_(): Returns a string representation of the linked list.

        find(ele): Searches for an element in the linked list and returns itsposition.

        \_\_contains\_\_(ele): Checks if an element is present in the linked list.

        findprev(ind): Finds the previous node at a given index in the linked list.

        pop(ind): Removes and returns the element at the given index in the linked list.

    '''

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

        '''

        Initializes an empty linked list with a head and tail node.

        '''

*self*.head = *self*.tail = Node()

*def* isempty(*self*):

        '''

        Checks if the linked list is empty.

        Returns:

            True if the linked list is empty, False otherwise.

        '''

        return *self*.head == *self*.tail

*def* append(*self*, *ele*):

        '''

        Appends an element to the end of the linked list.

        Args:

            ele: The element to be appended.

        '''

        temp = Node(*ele*)

*self*.tail.next = temp

*self*.tail = temp

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

        '''

        Returns a string representation of the linked list.

        Returns:

            A string representation of the linked list.

        '''

        pos = *self*.head.next

        s = '['

        while pos is not None:

            s += str(pos.item) + ','

            pos = pos.next

        if s[-1] == ',':

            s = s[:-1]

        return s + ']'

*def* find(*self*, *ele*):

        '''

        Searches for an element in the linked list and returns its position.

        Args:

            ele: The element to search for.

        Returns:

            The position/index of the element if found, -1 otherwise.

        '''

        pos = *self*.head.next

        ct = 0

        while pos is not None:

            if pos.item == *ele*:

                return ct

            ct += 1

            pos = pos.next

        return -1

*def* \_\_contains\_\_(*self*, *ele*):

        '''

        Checks if an element is present in the linked list.

        Args:

            ele: The element to check for.

        Returns:

            True if the element is present, False otherwise.

        '''

        pos = *self*.head.next

        while pos is not None:

            if pos.item == *ele*:

                return True

            pos = pos.next

        return False

*def* findprev(*self*, *ind*):

        '''

        Finds the previous node at a given index in the linked list.

        Args:

            ind: The index for which to find the previous node.

        Returns:

            The previous node at the given index.

        '''

        if *ind* > 0:

            pos = *self*.head.next

            for i in range(*ind* - 1):

                pos = pos.next

            return pos

        else:

            return *self*.head

*def* pop(*self*, *ind*):

        '''

        Removes and returns the element at the given index in the linked list.

        Args:

            ind: The index of the element to be removed.

        Returns:

            The element removed from the linked list.

        Raises:

            Exception: If the linked list is empty.

        '''

        if *self*.isempty():

            raise Exception("Empty list")

        else:

            pos = *self*.findprev(*ind*)

            pop\_item = pos.next.item

            pos.next = pos.next.next

        return pop\_item

*def* insert(*self*, *ind*, *ele*):

        '''

        Inserts an element at the given index in the linked list.

        Args:

            ind: The index at which the element should be inserted.

            ele: The element to be inserted.

        '''

        pos = *self*.findprev(*ind*)

        temp = Node(*ele*)

        temp\_pos = pos.next.next

        pos.next = temp

        pos.next.next = temp\_pos

*def* remove(*self*, *ele*):

        '''

        Removes the first occurrence of the given element from the linked list.

        Args:

            ele: The element to be removed.

        Raises:

            Exception: If the element is not found in the list.

        '''

        prev = *self*.findprev(*ele*)

        delnode = prev.next

        if delnode is None:

            raise Exception("Element is not in list!")

        prev.next = delnode.next

*self*.size -= 1

#driver code

if \_\_name\_\_ == '\_\_main\_\_':

    #this part of the code will only be run when the function is called directly

    #it will not be executed when it is imported as a module

    a = SinglyLinkedList()

    a.append(5)

    a.append(6)

    a.append(8)

    a.append(10)

    a.append(12)

    a.append(40)

    a.append(50)

    a.append(60)

    a.insert(7,90)

    print("Current linked list is :")

    print(a)

    print()

    print("Now popping randomly.")

    print(a.pop(7))

    print(a)

    print()

    print(a.pop(2))

    print(a)

    print()

III. OUTPUT:

Current linked list is :

[5,6,8,10,12,40,50,90]

Now popping randomly.

90

[5,6,8,10,12,40,50]

8

[5,6,10,12,40,50]

2)

I. AIM:

To design and implement Linked stack.

II. CODE:

# -\*- coding: utf-8 -\*-

'''

This module provides a class for implementation of the linked array

created using a Node that inherits the Node. This is a part of the

exercises given under the course UIT2201 (Programming and Data

Structures).

In this source code I have executed my own logic. The code

follows good coding practices.

Your comments and suggestions are welcome.

Created on Wed June 10 2023

Revised on Wed June 12 2023

Original Author: U. Pranaav <pranaav2210205@ssn.edu.in>

'''

*class* Node:

    \_\_slots\_\_ = ['item', 'next']

*def* \_\_init\_\_(*self*,*item*=None,*next*=None):

        '''

        Node class represents a node in a singly linked list.

        Args:

            item: The item/value stored in the Dnode.

            next: Reference to the next Dnode.

        '''

*self*.item = *item*

*self*.next = *next*

*class* LinkedArray:

    '''

    This is the implementation of a linked array using linked lists.

    Methods:

        init(): Initializes an empty LinkedArray object.

        push(ele): Pushes an element onto the top of the stack.

        pop(): Removes and returns the element from the top of the stack.

        peek(): Returns the element from the top of the stack without removing it.

        isempty(): Checks if the stack is empty.

        str(): Returns a string representation of the stack.

    '''

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

        '''

        Initializes an empty LinkedArray object.

        '''

*self*.top = Node()

*self*.size = 0

*def* push(*self*, *ele*):

        '''

        Pushes an element onto the top of the stack.

        Args:

            ele: The element to be pushed onto the stack.

        '''

        temp = Node(*ele*)

        pos = *self*.top.next

*self*.top.next = temp

        temp.next = pos

*self*.size += 1

*def* pop(*self*):

        '''

        Removes and returns the element from the top of the stack.

        Returns:

            The element removed from the top of the stack.

        Raises:

            Exception: If the stack is empty (stack underflow).

        '''

        if *self*.isempty():

            raise Exception("Stack underflow")

        else:

            del\_val = *self*.top.next.item

*self*.top.next = *self*.top.next.next

*self*.size -= 1

            return del\_val

*def* peek(*self*):

        '''

        Returns the element from the top of the stack without removing it.

        Returns:

            The element from the top of the stack.

        Raises:

            Exception: If the stack is empty.

        '''

        if *self*.isempty():

            raise Exception("Stack is empty")

        else:

            return *self*.top.next.item

*def* isempty(*self*):

        '''

        Checks if the stack is empty.

        Returns:

            True if the stack is empty, False otherwise.

        '''

        return *self*.top.next is None

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

        '''

        Returns a string representation of the stack.

        Returns:

            A string representation of the stack.

        '''

        pos = *self*.top.next

        s = ''

        while pos is not None:

            d = str(pos.item)

            s += d[::-1] + ','

            pos = pos.next

        if s[-1] == ',':

            s = s[:-1]

        return '[' + s[::-1] + ']'

#driver code

if \_\_name\_\_ == '\_\_main\_\_':

    #this part of the code will only be run when the function is called directly

    #it will not be executed when it is imported as a module

    a = LinkedArray()

    a.push(10)

    a.push(20)

    a.push(30)

    a.push(40)

    a.push(50)

    a.push(60)

    print(*f*"Current array is : {a}")

    print(*f*"Pop element is : {a.pop()}")

    print(a)

    print()

    print(*f*"Pop element is : {a.pop()}")

    print(a)

    print()

    print(*f*"Pop element is : {a.pop()}")

    print(a)

    print()

    print(*f*"Final element is : {a.peek()}")

    print()

    print(*f*"Checking if stack is empty : {a.isempty()}")

    print()

III. OUTPUT:

Current array is : [10,20,30,40,50,60]

Pop element is : 60

[10,20,30,40,50]

Pop element is : 50

[10,20,30,40]

Pop element is : 40

[10,20,30]

Final element is : 30

Checking if stack is empty : False

3)

I. AIM:

To design and implement a linked queue.

II. CODE:

# -\*- coding: utf-8 -\*-

'''

This module provides a class for implementation of the singly linked

queue created using a Node class. This is a part of the

exercises given under the course UIT2201 (Programming and Data

Structures).

In this source code I have executed my own logic. The code

follows good coding practices.

Your comments and suggestions are welcome.

Created on Wed June 7 2023

Revised on Wed June 10 2023

Original Author: U. Pranaav <pranaav2210205@ssn.edu.in>

'''

*class* Node:

    \_\_slots\_\_ = ['item', 'next']

*def* \_\_init\_\_(*self*,*item*=None,*next*=None):

        '''

        Node class represents a node in a singly linked queue.

        Args:

            item: The item/value stored in the node.

            next: Reference to the next node.

        '''

*self*.item = *item*

*self*.next = *next*

*class* SinglyLinkedQueue:

    '''

    The SinglyLinkedQueue class represents a singly linked queue.

    Attributes:

        front (Node): The front node of the linked queue.

        rear (Node): The rear node of the linked queue.

    Methods:

        \_\_init\_\_(): Initializes an empty linked queue with a front and rear node.

        isempty(): Checks if the linked queue is empty.

        enqueue(ele): enqueues an element to the end of the linked queue.

        \_\_str\_\_(): Returns a string representation of the linked queue.

        find(ele): Searches for an element in the linked queue and returns itsposition.

        \_\_contains\_\_(ele): Checks if an element is present in the linked queue.

        findprev(ind): Finds the previous node at a given index in the linked queue.

        dequeue(): Removes element at the front of linked queue.

    '''

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

        '''

        Initializes an empty linked queue with a front and rear node.

        '''

*self*.front = *self*.rear = Node()

*def* isempty(*self*):

        '''

        Checks if the linked queue is empty.

        Returns:

            True if the linked queue is empty, False otherwise.

        '''

        return *self*.front.item == None

*def* enqueue(*self*, *ele*):

        '''

        enqueues an element to the end of the linked queue.

        Args:

            ele: The element to be enqueued.

        '''

        if not(*self*.isempty()):

            temp = Node(*ele*)

*self*.rear.next = temp

*self*.rear = temp

        else:

*self*.rear.item = *ele*

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

        '''

        Returns a string representation of the linked queue.

        Returns:

            A string representation of the linked queue.

        '''

        pos = *self*.front

        s = '['

        while pos is not None:

            s += str(pos.item) + ','

            pos = pos.next

        if s[-1] == ',':

            s = s[:-1]

        return s + ']'

*def* find(*self*, *ele*):

        '''

        Searches for an element in the linked queue and returns its position.

        Args:

            ele: The element to search for.

        Returns:

            The position/index of the element if found, -1 otherwise.

        '''

        pos = *self*.front.next

        ct = 0

        while pos is not None:

            if pos.item == *ele*:

                return ct

            ct += 1

            pos = pos.next

        return -1

*def* \_\_contains\_\_(*self*, *ele*):

        '''

        Checks if an element is present in the linked queue.

        Args:

            ele: The element to check for.

        Returns:

            True if the element is present, False otherwise.

        '''

        pos = *self*.front.next

        while pos is not None:

            if pos.item == *ele*:

                return True

            pos = pos.next

        return False

*def* findprev(*self*, *ind*):

        '''

        Finds the previous node at a given index in the linked queue.

        Args:

            ind: The index for which to find the previous node.

        Returns:

            The previous node at the given index.

        '''

        if *ind* > 0:

            pos = *self*.front.next

            for i in range(*ind* - 1):

                pos = pos.next

            return pos

        else:

            return *self*.front

*def* dequeue(*self*):

        '''

        Removes and returns the element at the given index in the linked queue.

        Args:

            ind: The index of the element to be removed.

        Returns:

            The element removed from the linked queue.

        Raises:

            Exception: If the linked queue is empty.

        '''

        if *self*.isempty():

            raise Exception("Empty queue")

        else:

            dequeue\_item = *self*.front.item

*self*.front = *self*.front.next

        return dequeue\_item

#driver code

if \_\_name\_\_ == '\_\_main\_\_':

    #this part of the code will only be run when the function is called directly

    #it will not be executed when it is imported as a module

    a = SinglyLinkedQueue()

    a.enqueue(5)

    a.enqueue(6)

    a.enqueue(8)

    a.enqueue(10)

    a.enqueue(12)

    a.enqueue(40)

    a.enqueue(50)

    a.enqueue(60)

    print("Current linked queue is :")

    print(a)

    print()

    print("Now dequeueing.")

    print(*f*"Dequeue element is : {a.dequeue()}")

    print(*f*"Queue is : {a}")

    print()

    print(*f*"Dequeue element is : {a.dequeue()}")

    print(*f*"Queue is : {a}")

    print()

    print(*f*"Dequeue element is : {a.dequeue()}")

    print(*f*"Queue is : {a}")

    print()

    print(*f*"Dequeue element is : {a.dequeue()}")

    print(*f*"Queue is : {a}")

    print()

    print(*f*"Dequeue element is : {a.dequeue()}")

    print(*f*"Queue is : {a}")

    print()

    print(*f*"Dequeue element is : {a.dequeue()}")

    print(*f*"Queue is : {a}")

    print()

    print(*f*"Dequeue element is : {a.dequeue()}")

    print(*f*"Queue is : {a}")

    print()

    print(*f*"Dequeue element is : {a.dequeue()}")

    print(*f*"Queue is : {a}")

    print()

III. OUTPUT:

Current linked queue is :

[5,6,8,10,12,40,50,60]

Now dequeueing.

Dequeue element is : 5

Queue is : [6,8,10,12,40,50,60]

Dequeue element is : 6

Queue is : [8,10,12,40,50,60]

Dequeue element is : 8

Queue is : [10,12,40,50,60]

Dequeue element is : 10

Queue is : [12,40,50,60]

Dequeue element is : 12

Queue is : [40,50,60]

Dequeue element is : 40

Queue is : [50,60]

Dequeue element is : 50

Queue is : [60]

Dequeue element is : 60

Queue is : []

4)

I. AIM:

To design and implement a doubly linked list ADT capable of reverse display and reversed traversal.

II. CODE:

# -\*- coding: utf-8 -\*-

'''

This module provides a class for implementation of the doubly linked

list created using a DNode that inherits the Node. This is a part of the

exercises given under the course UIT2201 (Programming and Data

Structures).

In this source code I have executed my own logic. The code

follows good coding practices.

Your comments and suggestions are welcome.

Created on Wed June 7 2023

Revised on Wed June 10 2023

Original Author: U. Pranaav <pranaav2210205@ssn.edu.in>

'''

*class* Node:

    \_\_slots\_\_ = ['item', 'next']

*def* \_\_init\_\_(*self*,*item*=None,*next*=None):

        '''

        Node class represents a node in a singly linked list.

        Args:

            item: The item/value stored in the Dnode.

            next: Reference to the next Dnode.

        '''

*self*.item = *item*

*self*.next = *next*

*class* DNode(Node):

    \_\_slots\_\_ = ['item', 'next', 'prev']

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

        '''

        DNode class represents a Dnode in a singly linked list.

        Args:

            item: The item/value stored in the node.

            next: Reference to the next node.

            prev: Reference to the previous node.

        '''

        super().\_\_init\_\_(*item*,*next*)

*self*.prev = *prev*

*class* doublylinkedlist:

    '''

    This class represents a doubly linked list data structure.

    Attributes:

        head: A reference to the head (first) Dnode in the linked list.

        tail: A reference to the tail (last) Dnode in the linked list.

        size: The number of elements in the linked list.

    Methods:

        \_\_init\_\_(): Initializes an empty doubly linked list.

        append(ele): Adds an element to the end of the linked list.

        insert(ind, ele): Inserts an element at the specified index in the

        linked list.

        findprev(ind): Returns the Dnode before the specified index in the

        linked list.

        \_\_str\_\_(): Returns a string representation of the linked list.

        reverse\_display(): Returns a string representation of the linked list

        in reverse order.

        pop(ind=None): Removes and returns the element at the specified index

        in the linked list.

        \_\_len\_\_(): Returns the number of elements in the linked list.

    '''

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

        '''

        Initializes an empty doubly linked list.

        The head and tail Dnodes are created, and the size is set to 0.

        '''

*self*.head = *self*.tail = DNode()

*self*.size = 0

*def* append(*self*, *ele*):

        '''

        Adds an element to the end of the linked list.

        Args:

            ele: The element to be added to the linked list.

        '''

        temp = DNode(*ele*)

*self*.tail.next = temp

        temp.prev = *self*.tail

*self*.tail = temp

*self*.size += 1

*def* insert(*self*, *ind*, *ele*):

        '''

        Inserts an element at the specified index in the linked list.

        Args:

            ind: The index at which to insert the element.

            ele: The element to be inserted.

        Raises:

            Exception: If the index is invalid.

        '''

        if *ind* > 0 and *ind* < *self*.size - 2:

            pos = *self*.findprev(*ind*)

            temp = DNode(*ele*)

            pos.next.prev = temp

            temp.next = pos.next

            pos.next = temp

            temp.prev = pos

*self*.size += 1

        elif *ind* == *self*.size - 1:

*self*.append(*ele*)

        else:

            raise Exception("Invalid index")

*def* findprev(*self*, *ind*):

        '''

        Returns the Dnode before the specified index in the linked list.

        Args:

            ind: The index of the Dnode to find.

        Returns:

            The Dnode before the specified index.

        Raises:

            Exception: If the index is invalid.

        '''

        if *ind* < *self*.size:

            if *ind* > 0:

                pos = *self*.head.next

                for i in range(*ind* - 1):

                    pos = pos.next

                return pos

            else:

                return *self*.head

        else:

            raise Exception("Invalid index")

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

        '''

        Returns a string representation of the linked list.

        Returns:

            A string representation of the linked list.

        '''

        pos = *self*.head.next

        s = '['

        while pos is not None:

            s += str(pos.item) + ','

            pos = pos.next

        if s[-1] == ',':

            s = s[:-1]

        return s + ']'

*def* reverse\_display(*self*):

        '''

        Returns a string representation of the linked list in reverse order.

        Returns:

            A string representation of the linked list in reverse order.

        '''

        pos = *self*.tail

        s = '['

        while pos is not None and pos.item is not None:

            s += str(pos.item) + ','

            pos = pos.prev

        if s[-1] == ',':

            s = s[:-1]

        return s + ']'

*def* pop(*self*, *ind*=None):

        '''

        Removes and returns the element at the specified index in the linked list.

        If no index is provided, the element at the end of the linked list is

        removed.

        Args:

            ind: The index of the element to remove. Default is None.

        Returns:

            The removed element.

        Raises:

            Exception: If the index is invalid.

        '''

        if *ind* is None:

            pos = *self*.tail.prev

            pop\_val = pos.next.item

*self*.tail.prev = None

            pos.next = None

*self*.tail = pos

*self*.size -= 1

            return pop\_val

        else:

            pos = *self*.findprev(*ind*)

            pop\_val = pos.next.item

            pos.next.next.prev = pos

            pos.next = pos.next.next

*self*.size -= 1

            return pop\_val

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

        '''

        Returns the number of elements in the linked list.

        Returns:

            The number of elements in the linked list.

        '''

        return *self*.size

#driver code

if \_\_name\_\_ == '\_\_main\_\_':

    #this part of the code will only be run when the function is called directly

    #it will not be executed when it is imported as a module

    a = doublylinkedlist()

    a.append(10)

    a.append(20)

    a.append(30)

    a.append(40)

    a.append(50)

    a.append(60)

    a.insert(1,10000)

    print(a)

    print()

    a.insert(1,20000)

    print(a)

    print()

    a.insert(1,30000)

    print(a)

    print()

    a.insert(1,40000)

    print(a)

    print()

    print(*f*"Pop element is : {a.pop()}")

    print(a)

    print()

    print(*f*"Pop element is : {a.pop(2)}")

    print(a)

    print()

    print(*f*"Pop element is : {a.pop(4)}")

    print(a)

    print()

    print(a.reverse\_display())

III. OUTPUT:

Current list is : [10,10000,20,30,40,50,60]

Current list is : [10,20000,10000,20,30,40,50,60]

Current list is : [10,30000,20000,10000,20,30,40,50,60]

Current list is : [10,40000,30000,20000,10000,20,30,40,50,60]

Pop element is : 60

Current list is : [10,40000,30000,20000,10000,20,30,40,50]

Pop element is : 30000

Current list is : [10,40000,20000,10000,20,30,40,50]

Pop element is : 20

Current list is : [10,40000,20000,10000,30,40,50]

Now printing the reverse display

[50,40,30,10000,20000,40000,10]