**Module 18. Linked Lists**

**Learning Objectives**

* Understand the concept of linked lists and their importance in data structures.
* Implement a basic singly linked list in Python, including creating nodes and linking them together.
* Demonstrate how to perform basic operations on a singly linked list, such as insertion, deletion, and traversal.
* Explore the concept of a doubly linked list and its advantages over a singly linked list.
* Implement a doubly linked list in Python, including maintaining backward and forward pointers.
* Understand the concept of a circular linked list and its applications.
* Implement a circular linked list in Python, including handling traversal and insertion/deletion operations.

# **Linked Lists**

A linked list is composed of nodes, where each node contains an element, and each node is connected to its next neighbor.

A black background with a black square

Description automatically generated with medium confidence

The figure above shows that there are three nodes in the linked list.

And a node can be defined as below:

class Node:

    def \_\_init\_\_(self, element):

        self.elmenet = element

        self.next = None

The variable *head* denotes the first node in the list, while the variable *tail* denotes the last node in the list. If the list is empty, both *head* and *tail* are set to **None**.

1. How it works?

When a list is empty, both the *head* and *tail* variables point to **None**.

A black background with a black square

Description automatically generated with medium confidence

In order to add nodes into a list, first of all create the first node:

A black background with blue text

Description automatically generated

The *head* variable points to Node 1, and similarly, the *tail* variable also points to Node 1.

Blue arrows pointing to a black background

Description automatically generated

head = Node("Atlanta")

tail = head

Next, create the second node and add it to the list.

A black background with blue text

Description automatically generated

Node 2 is succeeded by Node 1, updating the *next* pointer of Node 1 and the tail as follows: The *next* pointer of Node 1 now directs to Node 2, and likewise, the *tail* variable also points to Node 2.

A black background with blue text

Description automatically generated

tail.next = Node("Macon")

tail = tail.next

Next, create the third node and add it to the list.

A black background with blue text

Description automatically generated

Node 3 is followed by Node 2, updating the *next* pointer of Node 2 and the tail as follows: The *next* pointer of Node 2 now points to Node 3, and similarly, the *tail* variable also points to Node 3.

A black background with a black square

Description automatically generated with medium confidence

tail.next = Node("Dallas")

tail = tail.next

To traverse elements in the list, you can follow these steps:

1. Start from the *head* of the list.
2. Access the element of the current node.
3. Move to the next node using the *next* pointer.
4. Repeat steps 2 and 3 until you reach a node where the *next* pointer is None, indicating the end of the list.

current = head

while current != None:

    print(current.elmenet)

    current = current.next

1. Basic operations of a linked List

* addFirst: Add a node to the head of the list
* addLast: Add a node to the tail of the list
* insert: Add a node at the specified index
* removeFirst: Remove the first node in the list
* removeLast: Remove the last node in the list
* removeAt: Remove the node at the specified index

1. Method Implementation
2. addFirst

When adding a node to the current list at the first position, please follow this procedure:

A black background with a black square

Description automatically generated with medium confidence

Initially, generate a new node, for instance, the "Chicago" node.

newNode = Node(e)

Insert the new node, "Chicago", at the beginning of the list before the first node, "Atlanta".

newNode.next = self.\_\_head

A black background with a black square

Description automatically generated with medium confidence

The *head* variable is updated to point from the "Atlanta" node to the "Chicago" node.

self.\_\_head = newNode

A black background with a black square

Description automatically generated with medium confidence

If the new node is added to an empty list, then it becomes the only node in the list.

if self.\_\_tail == None:   
 self.\_\_tail = self.\_\_head

A blue arrows pointing to a black background

Description automatically generated

1. addLast

When adding a node to the current list at the last position, please follow this procedure:

A black background with blue text

Description automatically generated

Initially, generate a new node, for instance, the "Chicago" node.

newNode = Node(e)

If the new node is added to an empty list, then it becomes the only node in the list.

A blue arrows pointing to a black background

Description automatically generated

if self.\_\_tail == None:   
 self.\_\_head = self.\_\_tail = newNode

Otherwise, insert the new node, "Chicago", after the last node, "Dallas".

self.\_\_tail.next = newNode

A black background with blue text

Description automatically generated

And then, update the *tail* variable to point from the node "Dallas" to the new node "Chicago".

self.\_\_tail = self.\_\_tail.next

A black screen with blue text

Description automatically generated

1. insert(index, e)

When inserting a node to the current list at the specified index, please follow this procedure:

A black background with blue text

Description automatically generated

An index of 0 indicates that the new node is added to the first position in the list.

if index == 0:  
 self.addFirst(e) # Insert first

If the index is greater than or equal to the number of nodes in the list, it indicates that the new node is added to the last position in the list.

elif index >= self.\_\_size:

    self.addLast(e) # Insert last

When inserting a new node in the middle of the list, declare a variable, *current*, and point it to the *head* node.

current = self.\_\_head

A black screen with blue text

Description automatically generated

Iterate through the nodes using a loop while moving to the previous node until the desired index is reached.

for i in range(1, index):

    current = current.next

A screen shot of a computer

Description automatically generated

And declare a variable, *temp* and point it to the next node of the current node.

temp = current.next

A black screen with blue text

Description automatically generated

Link the new node, “Chicago” to the *current.next* node.

current.next = Node(e)

A black background with blue text and arrows

Description automatically generated

Finally, link the next node of the *current.next* to the temp, “Dallas” node.

(current.next).next = temp

A black background with blue text and arrows

Description automatically generated

1. removeFirst

When removing the first node in the current list, please follow this procedure:

A black background with a black square

Description automatically generated with medium confidence

If the list is empty, then return **None** which means nothing to remove.

if self.\_\_size == 0:  
    return None

If the list is not empty, declare a temporary variable, *temp* and assign it to the first node.

temp = self.\_\_head

A black background with a black square

Description automatically generated with medium confidence

Move the *head* from the “Chicago” node to the “Atlanta” node.

self.\_\_head = self.\_\_head.next

A black background with a black square

Description automatically generated with medium confidence

Return *temp.element*, “Chicago”.

Once the first node is removed, if the *head* variable points to **None**, indicating an empty list, then the *tail* variable should also be set to **None**.

if self.\_\_head == None:

    self.\_\_tail = None

1. removeLast

When removing the last node in the current list, please follow this procedure:

A black background with a black square

Description automatically generated with medium confidence

If the list is empty, then return **None** which means nothing to remove.

if self.\_\_size == 0:  
    return None

If there is only one node in the list, declare a temporary variable, *temp* and assign it to the first node.

temp = self.\_\_head

A blue arrow pointing to a black background

Description automatically generated

Then assign **None** to both the *head* and *tail* variables.

self.\_\_head = self.\_\_tail = None

A black background with a black square

Description automatically generated with medium confidence

If there are multiple nodes in the list, declare a variable named *current* and point it to the *head* node.

current = self.\_\_head

A black background with a black square

Description automatically generated with medium confidence

Then iterate through the nodes using a loop, while moving to the previous node of the target node to be removed.

for i in range(self.\_\_size - 2):

    current = current.next

A black background with a black square

Description automatically generated with medium confidence

Declare a temporary variable named *temp* and assign it to the last node, which is "Dallas".

temp = self.\_\_tail

A black background with a black square

Description automatically generated with medium confidence

Set the current node, "Macon", as the tail, and assign **None** to the *tail* 's next pointer (*tail.next*).

self.\_\_tail = current  
self.\_\_tail.next = None

A black background with a black square

Description automatically generated with medium confidence

Finally, return *temp.element*, “Dallas”.

1. removeAt(index)

When removing a node in the current list at the specified index, please follow this procedure:

A black background with a black square

Description automatically generated with medium confidence

If the index is out of range of the list, return **None**.

if index < 0 or index >= self.\_\_size:  
    return None

If the index is 0, it indicates that the first node is to be removed.

elif index == 0:  
    return self.removeFirst()

If the index indicates the last node, remove the last node.

elif index == self.\_\_size - 1:

    return self.removeLast()

Otherwise, declare a temporary variable named *previous* and assign it to the *head* node, which is "Chicago".

A black background with a black square

Description automatically generated with medium confidence

Iterate through the nodes using a loop, while moving to the previous node of the target node to be removed.

A screen shot of a computer

Description automatically generated

Declare a temporary variable named *current* and assign it to the target node to be removed, which is "Macon".

A screenshot of a computer

Description automatically generated

Connect the *previous.next* to *current.next*.

A graph of a function

Description automatically generated with medium confidence

Finally, return *current.element*, “Macon”.

1. Linked List Implementation

class LinkedList:

    def \_\_init\_\_(self):

        self.\_\_head = None

        self.\_\_tail = None

        self.\_\_size = 0

    def getFirst(self):

        if self.\_\_size == 0:

            return None

        else:

            return self.\_\_head.element

    def getLast(self):

        if self.\_\_size == 0:

            return None

        else:

            return self.\_\_tail.element

    def addFirst(self, e):

        newNode = Node(e)

        newNode.next = self.\_\_head

        self.\_\_head = newNode

        self.\_\_size += 1

        if self.\_\_tail == None:

            self.\_\_tail = self.\_\_head

    def addLast(self, e):

        newNode = Node(e)

        if self.\_\_tail == None:

            self.\_\_head = self.\_\_tail = newNode

        else:

            self.\_\_tail.next = newNode

            self.\_\_tail = self.\_\_tail.next

        self.\_\_size += 1

    def insert(self, index, e):

        if index == 0:

            self.addFirst(e)

        elif index >= self.\_\_size:

            self.addLast(e)

        else:

            current = self.\_\_head

            for i in range(1, index):

                current = current.next

            temp = current.next

            current.next = Node(e)

            (current.next).next = temp

            self.\_\_size += 1

    def removeFirst(self):

        if self.\_\_size == 0:

            return None

        else:

            temp = self.\_\_head

            self.\_\_head = self.\_\_head.next

            self.\_\_size -= 1

            if self.\_\_head == None:

                self.\_\_tail = None

            return temp.element

    def removeLast(self):

        if self.\_\_size == 0:

            return None

        elif self.\_\_size == 1:

            temp = self.\_\_head

            self.\_\_head = self.\_\_tail = None

            self.\_\_size = 0

            return temp.element

        else:

            current = self.\_\_head

            for i in range(self.\_\_size - 2):

                current = current.next

            temp = self.\_\_tail

            self.\_\_tail = current

            self.\_\_tail.next = None

            self.\_\_size -= 1

            return temp.element

    def removeAt(self, index):

        if index < 0 or index >= self.\_\_size:

            return None

        elif index == 0:

            return self.removeFirst()

        elif index == self.\_\_size - 1:

            return self.removeLast()

        else:

            previous = self.\_\_head

            for i in range(1, index):

                previous = previous.next

            current = previous.next

            previous.next = current.next

            self.\_\_size -= 1

            return current.element

    def isEmpty(self):

        return self.\_\_size == 0

    def getSize(self):

        return self.\_\_size

    def \_\_str\_\_(self):

        result = "["

        current = self.\_\_head

        for i in range(self.\_\_size):

            result += str(current.element)

            current = current.next

            if current != None:

                result += ", "

            else:

                result += "]"

        return result

    def clear(self):

        self.\_\_head = self.\_\_tail = None

    def \_\_iter\_\_(self):

        return LinkedListIterator(self.\_\_head)

# The Node class

class Node:

    def \_\_init\_\_(self, e):

        self.element = e

        self.next = None

class LinkedListIterator:

    def \_\_init\_\_(self, head):

        self.current = head

    def \_\_next\_\_(self):

        if self.current == None:

            raise StopIteration

        else:

            element = self.current.element

            self.current = self.current.next

            return element

(Test Program)

from LinkedList import LinkedList

city = LinkedList()

# Add nodes to the list

city.add("Atlanta")

print("(1)", city)

city.insert(0, "Chicago")

print("(2)", city)

city.add("Macon")

print("(3)", city)

city.addLast("Dallas")

print("(4)", city)

city.insert(2, "New York")

print("(5)", city)

city.insert(5, "Boston")

print("(6)", city)

city.insert(0, "LA")

print("(7)", city)

# Remove nodes from the list

city.removeAt(0)

print("(8)", city)

city.removeAt(2)

print("(9)", city)

city.removeAt(city.getSize() - 1)

print("(10)", city)

(Output)

(1) [Atlanta]

(2) [Chicago, Atlanta]

(3) [Chicago, Atlanta, Macon]

(4) [Chicago, Atlanta, Macon, Dallas]

(5) [Chicago, Atlanta, New York, Macon, Dallas]

(6) [Chicago, Atlanta, New York, Macon, Dallas, Boston]

(7) [LA, Chicago, Atlanta, New York, Macon, Dallas, Boston]

(8) [Chicago, Atlanta, New York, Macon, Dallas, Boston]

(9) [Chicago, Atlanta, Macon, Dallas, Boston]

(10) [Chicago, Atlanta, Macon, Dallas]

# **Circular Linked Lists**

A circular linked list is a variant of a linked list where the first and last nodes are interconnected, creating a loop or circle within the list structure.

A black background with a black square

Description automatically generated with medium confidence

The next node of the last node is indicating to the first node.

1. Circular Linked List Implementation

class CircularLinkedList:

    def \_\_init\_\_(self):

        self.\_\_head = None

        self.\_\_tail = None

        self.\_\_size = 0

    def getFirst(self):

        if self.\_\_size == 0:

            return None

        else:

            return self.\_\_head.element

    def getLast(self):

        if self.\_\_size == 0:

            return None

        else:

            return self.\_\_tail.element

    def addFirst(self, e):

        newNode = Node(e)

        if self.\_\_size == 0:

            self.\_\_head = self.\_\_tail = newNode

            self.\_\_tail.next = self.\_\_head

        else:

            newNode.next = self.\_\_head

            self.\_\_head = newNode

            self.\_\_tail.next = self.\_\_head

        self.\_\_size += 1

    def addLast(self, e):

        newNode = Node(e)

      if self.\_\_size == 0:

            self.\_\_head = self.\_\_tail = newNode

            self.\_\_tail.next = self.\_\_head

        else:

            self.\_\_tail.next = newNode

            self.\_\_tail = newNode

            self.\_\_tail.next = self.\_\_head

        self.\_\_size += 1

    def insert(self, index, e):

        if index == 0:

            self.addFirst(e)

        elif index >= self.\_\_size:

            self.addLast(e)

        else:

            current = self.\_\_head

            for i in range(1, index):

                current = current.next

            newNode = Node(e)

            newNode.next = current.next

            current.next = newNode

            self.\_\_size += 1

    def removeFirst(self):

        if self.\_\_size == 0:

            return None

        else:

            temp = self.\_\_head

            self.\_\_head = self.\_\_head.next

            self.\_\_tail.next = self.\_\_head

            self.\_\_size -= 1

            if self.\_\_size == 0:

                self.\_\_tail = None

            return temp.element

    def removeLast(self):

        if self.\_\_size == 0:

            return None

        elif self.\_\_size == 1:

            temp = self.\_\_head

            self.\_\_head = self.\_\_tail = None

            self.\_\_size = 0

            return temp.element

        else:

            current = self.\_\_head

          for i in range(self.\_\_size - 2):

                current = current.next

            temp = self.\_\_tail

            self.\_\_tail = current

            self.\_\_tail.next = self.\_\_head

            self.\_\_size -= 1

            return temp.element

    def removeAt(self, index):

        if index < 0 or index >= self.\_\_size:

            return None

        elif index == 0:

            return self.removeFirst()

        elif index == self.\_\_size - 1:

            return self.removeLast()

        else:

            previous = self.\_\_head

            for i in range(1, index):

                previous = previous.next

            current = previous.next

            previous.next = current.next

            self.\_\_size -= 1

            return current.element

    def isEmpty(self):

        return self.\_\_size == 0

    def getSize(self):

        return self.\_\_size

    def \_\_str\_\_(self):

        result = "["

        current = self.\_\_head

        for i in range(self.\_\_size):

            result += str(current.element)

            current = current.next

            if current != self.\_\_head:

                result += ", "

            else:

                result += "]"

        return result

    def clear(self):

        self.\_\_head = self.\_\_tail = None

        self.\_\_size = 0

    def \_\_iter\_\_(self):

        return CircularLinkedListIterator(self.\_\_head)

# The Node class

class Node:

    def \_\_init\_\_(self, e):

        self.element = e

        self.next = None

class CircularLinkedListIterator:

    def \_\_init\_\_(self, head):

        self.current = head

    def \_\_next\_\_(self):

        if self.current is None:

            raise StopIteration

        else:

            element = self.current.element

            self.current = self.current.next

            return element

# **Doubly Linked Lists**

A doubly linked list consists of nodes equipped with two pointers: one pointing to the next node and the other pointing to the previous node. These pointers are commonly referred to as the forward pointer and the backward pointer, respectively. As a result, traversal of a doubly linked list can be performed both forward and backward.

A blue text on a black background

Description automatically generated

1. Doubly Linked List Implementation

class DoublyLinkedList:

    def \_\_init\_\_(self):

        self.\_\_head = None

        self.\_\_tail = None

        self.\_\_size = 0

    def getFirst(self):

        if self.\_\_size == 0:

            return None

        else:

            return self.\_\_head.element

    def getLast(self):

        if self.\_\_size == 0:

            return None

        else:

            return self.\_\_tail.element

    def addFirst(self, e):

        newNode = Node(e)

        newNode.next = self.\_\_head

        newNode.prev = None

        if self.\_\_head:

            self.\_\_head.prev = newNode

        self.\_\_head = newNode

        if self.\_\_tail is None:

            self.\_\_tail = newNode

        self.\_\_size += 1

    def addLast(self, e):

        newNode = Node(e)

        newNode.prev = self.\_\_tail

        newNode.next = None

        if self.\_\_tail:

            self.\_\_tail.next = newNode

        self.\_\_tail = newNode

        if self.\_\_head is None:

            self.\_\_head = newNode

        self.\_\_size += 1

    def insert(self, index, e):

        if index == 0:

            self.addFirst(e)

        elif index >= self.\_\_size:

            self.addLast(e)

        else:

            current = self.\_\_head

            for i in range(1, index):

                current = current.next

            newNode = Node(e)

            newNode.next = current.next

            newNode.prev = current

            current.next.prev = newNode

            current.next = newNode

            self.\_\_size += 1

    def removeFirst(self):

        if self.\_\_size == 0:

            return None

        else:

            temp = self.\_\_head

            self.\_\_head = self.\_\_head.next

            if self.\_\_head:

                self.\_\_head.prev = None

            self.\_\_size -= 1

            if self.\_\_head is None:

                self.\_\_tail = None

            return temp.element

    def removeLast(self):

        if self.\_\_size == 0:

            return None

        elif self.\_\_size == 1:

            temp = self.\_\_head

            self.\_\_head = self.\_\_tail = None

            self.\_\_size = 0

            return temp.element

        else:

            temp = self.\_\_tail

            self.\_\_tail = self.\_\_tail.prev

            self.\_\_tail.next = None

            self.\_\_size -= 1

            return temp.element

    def removeAt(self, index):

        if index < 0 or index >= self.\_\_size:

            return None

        elif index == 0:

            return self.removeFirst()

        elif index == self.\_\_size - 1:

            return self.removeLast()

        else:

            current = self.\_\_head

            for i in range(index):

                current = current.next

            temp = current

            current.prev.next = current.next

            current.next.prev = current.prev

            self.\_\_size -= 1

            return temp.element

    def isEmpty(self):

        return self.\_\_size == 0

    def getSize(self):

        return self.\_\_size

    def \_\_str\_\_(self):

        result = "["

        current = self.\_\_head

        while current:

            result += str(current.element)

            current = current.next

            if current:

                result += ", "

            else:

                result += "]"

        return result

    def clear(self):

        self.\_\_head = self.\_\_tail = None

    def \_\_iter\_\_(self):

        return LinkedListIterator(self.\_\_head)

# The modified Node class

class Node:

    def \_\_init\_\_(self, e):

        self.element = e

        self.next = None

        self.prev = None

class LinkedListIterator:

    def \_\_init\_\_(self, head):

        self.current = head

    def \_\_next\_\_(self):

        if self.current is None:

            raise StopIteration

        else:

            element = self.current.element

            self.current = self.current.next

            return element

**Summary**

1. A linked list is a fundamental data structure used in computer science to store and manage collections of elements. Unlike arrays, which store elements in contiguous memory locations, linked lists consist of nodes, each containing a data element and a reference (or pointer) to the next node in the sequence.
2. This arrangement allows for dynamic memory allocation and efficient insertion and deletion operations, as nodes can be easily added or removed without requiring contiguous memory blocks.
3. Linked lists come in various forms, including singly linked lists, doubly linked lists, and circular linked lists. In a singly linked list, each node has a reference to the next node in the sequence, while in a doubly linked list, each node has references to both the next and previous nodes. Circular linked lists form a closed loop where the last node points back to the first node.

**Programming Exercises**

**Exercise 1: Reverse a Linked List**

Write a function to reverse a singly linked list in place. Ensure that the original structure of the list is modified.

**Exercise 2: Remove Nth Node From End of List**

Write a function to remove the nth node from the end of a linked list and return its head.

**Exercise 3: Find the Middle of a Linked List**

Write a function to find the middle node of a singly linked list. If the list contains an even number of nodes, return the second middle node.

**Exercise 4: Remove Nth Node From End of List**

Write a function to remove the nth node from the end of a singly linked list and return the head of the modified list. Assume n is always valid and does not exceed the length of the list.

**Exercise 5: Merge Two Sorted Linked Lists**

Write a method to merge two sorted singly linked lists into a single sorted linked list.