**Week 6**

**Singly Linked list**

A linked list is a sequence of data elements, which are connected together via links. Each data element contains a connection to another data element in the form of a pointer. Python does not have linked lists in its standard library.

We implement the concept of linked lists using the concept of nodes. We create a Node object and create another class to use this object. We pass the appropriate values through the node object to point to the next data elements.

The basic linked list operations are:

* **Traversal** – Access the nodes of the list.
* **Insertion**– Adds a new node to an existing linked list.
* **Deletion**– Removes a node from an existing linked list.
* **Search**– Finds a particular element in the linked list.

**Traverse a Linked List**

Accessing the nodes of a linked list in order to process it is called**traversing**a linked list. Normally we use the traverse operation to display the contents or to search for an element in the linked list.

**Python code to create a Node Class**

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

self.dataval = dataval

self.nextval = None

**Algorithm to create a node in linked list**

Step 1: create a node object

Step 2: initialize the data

Step 3: set the link to NULL

**Algorithm to traverse the linked list**

Step 1: Initialize PTR = HEAD

Step 2: Repeat Steps 3 and 4 **while** PTR != **NULL**

Step 3: Apply process to PTR -> DATA

Step 4: SET PTR = PTR->NEXT

[**ENDOF** LOOP]

Step 5: EXIT

The below program creates the linked list with three data elements and also traverses the linked list. Traversing the list means visiting each node in the list in some order. Singly linked lists can be traversed in only forward direction starting form the first data element. We simply print the value of the next data element by assigning the pointer of the next node to the current data element.

**Python code for Creation of Linked list**

class Node:

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

self.dataval = dataval

self.nextval = None

class SLinkedList:

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

self.headval = None

def listprint(self):

printval = self.headval

while printval is not None:

print (printval.dataval)

printval = printval.nextval

list = SLinkedList()

list.headval = Node("Mon")

e2 = Node("Tue")

e3 = Node("Wed")

# Link first Node to second node

list.headval.nextval = e2

# Link second Node to third node

e2.nextval = e3

#print the list

list.listprint()

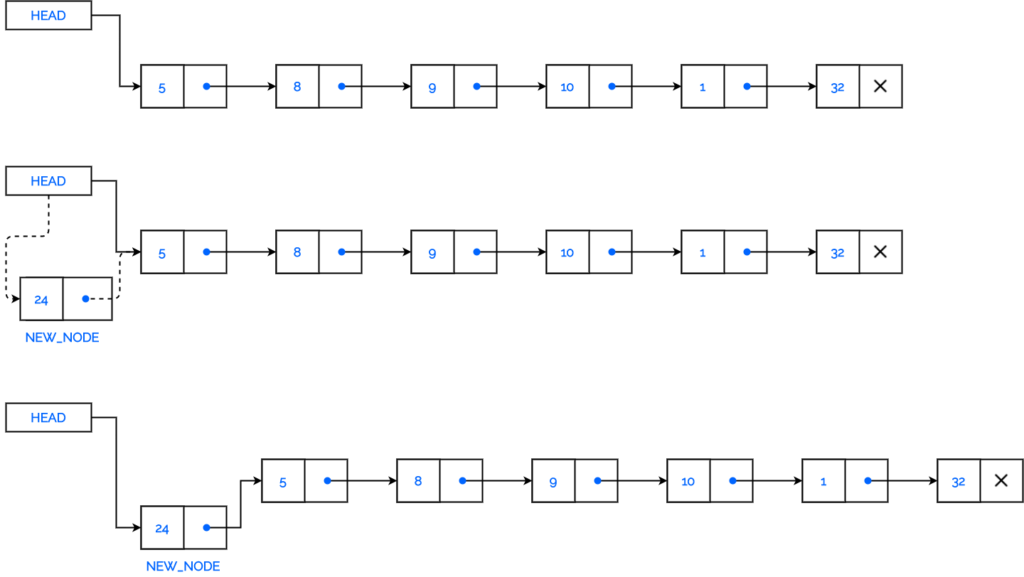
**Insertion in a Linked List**

Inserting element in the linked list involves reassigning the pointers from the existing nodes to the newly inserted node. Depending on whether the new data element is getting inserted at the beginning or at the middle or at the end of the linked list, we have the below scenarios.

**Inserting at the Beginning**

New node becomes the head of the linked list. The next pointer of the new data node points to the current head of the linked list. So the current head of the linked list becomes the second data element and the linked list

Consider the linked list shown in the figure. Suppose we want to create a new node with data 24 and add it as the first node of the list i.e.,Inserting node at the begining. The linked list will be modified as follows.



Algorithm to inserting the node at the begining

Step 1: Create NEW\_NODE

Step 2: SET NEW\_NODE -> DATA = VAL

Step 3 :SET NEW\_NODE -> NEXT = HEAD

Step 4: SET HEAD = NEW\_NODE

Step 5: EXIT

**Python program to Insert node at the begining**

class Node:

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

self.dataval = dataval

self.nextval = None

class SLinkedList:

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

self.headval = None

# Print the linked list

def listprint(self):

printval = self.headval

while printval is not None:

print (printval.dataval)

printval = printval.nextval

def AtBegining(self,newdata):

NewNode = Node(newdata)

# Update the new nodes next val to existing node

NewNode.nextval = self.headval

self.headval = NewNode

#Driver program

list = SLinkedList()

list.headval = Node("Mon")

e2 = Node("Tue")

e3 = Node("Wed")

list.headval.nextval = e2

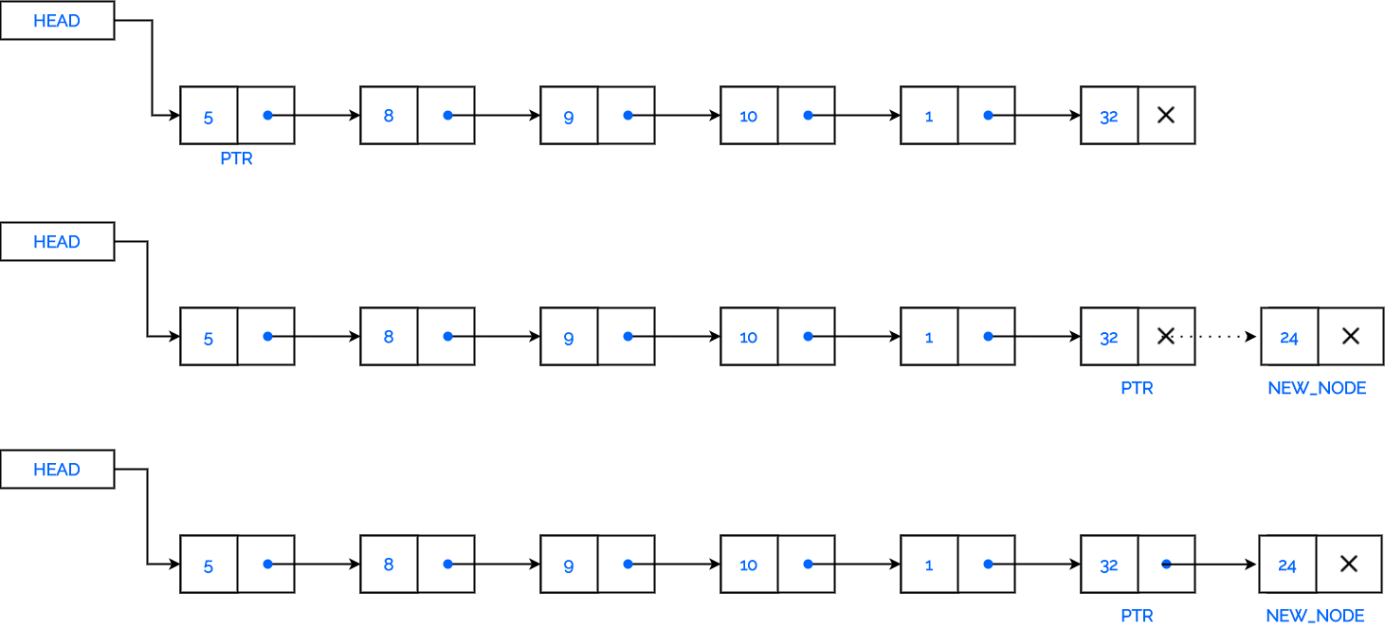
e2.nextval = e3

list.AtBegining("Sun")

list.listprint()

**Inserting at the End**

Suppose we want to add a new node with data 24 as the last node of the list. Then the linked list will be modified as follows.

****

This involves pointing the next pointer of the current last node of the linked list to the new data node. So the current last node of the linked list becomes the second last data node and the new node becomes the last node of the linked list.

Step 1: Create NEW\_NODE

Step 2: SET NEW\_NODE -> DATA = VAL

Step 3: SET NEW\_NODE -> NEXT = **None**

Step 4: SET PTR = HEAD

Step 5: Repeat Step 6 **while** PTR ->NEXT != **NULL**

Step 6: SET PTR = PTR -> NEXT

[**ENDOF** LOOP]

Step 7: SET PTR -> NEXT = NEW\_NODE

Step 8: EXIT

**Python code to insert node at end**

#Inserting node at the end

class Node:

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

self.dataval = dataval

self.nextval = None

class SLinkedList:

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

self.headval = None

# Function to add newnode

def AtEnd(self, newdata):

NewNode = Node(newdata)

if self.headval is None:

self.headval = NewNode

return

last = self.headval

#Traverse to last node

while(last.nextval):

last = last.nextval

#Connect the new node at end

last.nextval=NewNode

# Print the linked list

def listprint(self):

printval = self.headval

whileprintval is not None:

print (printval.dataval)

printval = printval.nextval

#Driver program

list = SLinkedList()

list.headval = Node("Mon")

e2 = Node("Tue")

e3 = Node("Wed")

list.headval.nextval = e2

e2.nextval = e3

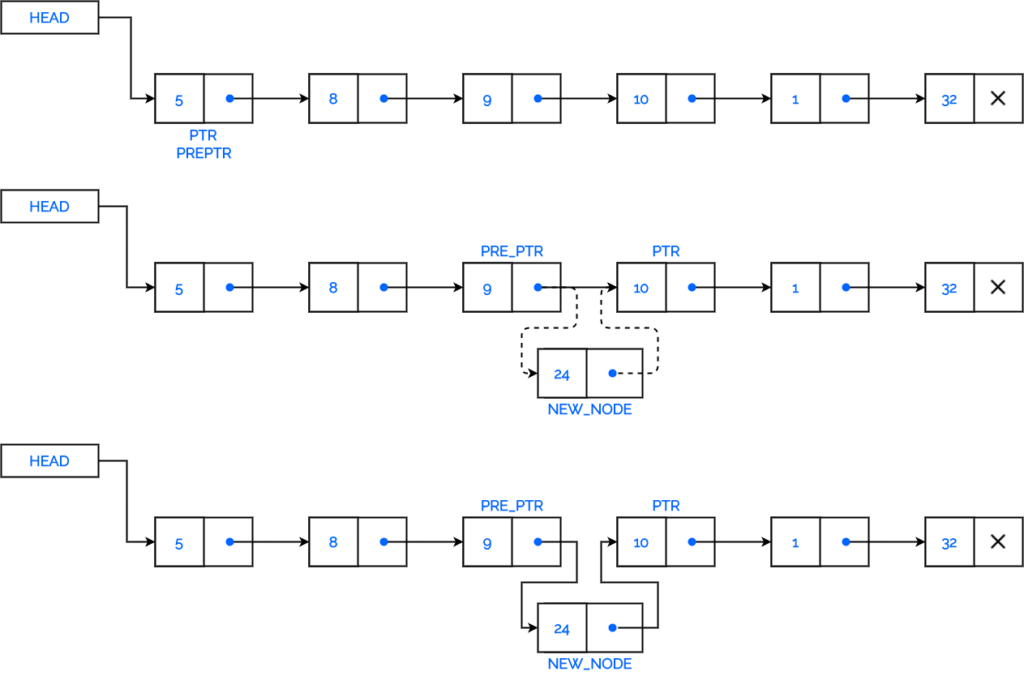
list.AtEnd("Thu")

list.listprint()

**Inserting in between two Data Nodes**

This involves changing the pointer of a specific node to point to the new node. That is possible by passing in both the new node and the existing node after which the new node will be inserted. So we define function inside class which will change the next pointer of the new node to the next pointer of existing node after which the new node will be inserted. Then assign the new node to next pointer of the existing node.

Suppose we want to add a new node with value 24 after the node having data 9. These changes will be done in the linked list.



**Algorithm to insert after given POS\_VAL**

Step 1: Create NEW\_NODE

Step 2: SET NEW\_NODE -> DATA = VAL

Step 3: SET NEW\_NODE -> NEXT = **None**

Step 4: SET PTR = HEAD

Step 5: Repeat Step 6 **while** PTR ->NEXT != **POS\_VAL**

Step 6: PTR = PTR->NEXT

Step 7: SET NEW\_NODE->NEXT = PTR->NEXT

Step 8: SET PTR -> NEXT = NEW\_NODE

Step 9 : EXIT

#Python program to insert inbetween

class Node:

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

self.dataval = dataval

self.nextval = None

class SLinkedList:

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

self.headval = None

# Function to add node

def Inbetween(self,key,newdata):

NewNode = Node(newdata)

if self.headval is None:

self.headval = NewNode

return

last = self.headval

print("inside",last.dataval)

#Traverse to Key node

while((last is not None) and (last.dataval != key)):

last = last.nextval

#Connect the new node at end

if( last is not None):

NewNode.nextval = last.nextval

last.nextval = NewNode

else:

print("data missing")

# Print the linked list

def listprint(self):

printval = self.headval

while printval is not None:

print(printval.dataval)

printval = printval.nextval

list = SLinkedList()

list.headval = Node("Mon")

e2 = Node("Tue")

e3 = Node("Thu")

list.headval.nextval = e2

e2.nextval = e3

list.listprint()

list.Inbetween("Tue","Fri")

list.listprint()

**Removing an Item**

We can remove an existing node using the key for that node. In the below program we locate the previous node of the node which is to be deleted.Then, point the next pointer of this node to the next node of the node to be deleted.

**Delete a Node from the beginning of a Linked list**

Suppose we want to delete a node from the beginning of the linked list. The list has to be modified as follows:

**Algorithm : Remove node at the begining**

Step 1: **IF** HEAD = **NULL**

Write Empty list

EXIT

Step 2: SET PTR = HEAD

Step 3: SET HEAD = HEAD -> NEXT

Step 4: EXIT

**Program to delete node at the begining**

class Node:

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

self.data = data

self.next = None

class SLinkedList:

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

self.head = None

def Atbegining(self, data\_in):

NewNode = Node(data\_in)

NewNode.next = self.head

self.head = NewNode

# Function to remove node

def RemoveNode(self, Removekey):

Ptr = self.head

if (Ptr is None):

print("Empty List")

return

self.head = Ptr.next

def Listprint(self):

printval = self.head

if(printval == None):

print("Empty")

while (printval):

print(printval.data),

printval = printval.next

llist = SLinkedList()

llist.Atbegining("Mon")

llist.Atbegining("Tue")

llist.Atbegining("Wed")

llist.Atbegining("Thu")

print("List before deleting")

llist.Listprint()

llist.RemoveNode("Mon")

print("List before deleting")

llist.Listprint()

**Searching an Item**

We can search for an item in the existing list. Traverse thru the list by comparing the data in the node with the key to search. If found display “Key found “ otherwise display “key not Found”

**Algorithm**: **Search for the given KEY**

Step 1: SET PTR = HEAD

Step 2: Repeat Step 3 **while** PTR ->NEXT is not None and PTR->DATA != KEY

Step 3: PTR = PTR->NEXT

Step 4: if PTR is None

Display “Key Not Found”

else

Display “Key Found”

Step 5 : EXIT

(For Program refer Lab programs)

**Linked list Iterators:**

Iterator in python is an object that is used to iterate over iterable objects like lists, tuples, dicts, and sets. The iterator object is initialized using the **iter()**method. It uses the **next()** method for iteration.

1. **\_\_iter(iterable)\_\_** method that is called for the initialization of an iterator. This returns an iterator object
2. **next ( \_\_next\_\_ in Python 3)** The next method returns the next value for the iterable. When we use a for loop to traverse any iterable object, internally it uses the iter() method to get an iterator object which further uses next() method to iterate over. This method raises a Stop Iteration to signal the end of the iteration.

How an iterator really works in python

# Sample built-in iterators

# Iterating over a list

print("List Iteration")

l = ["geeks", "for", "geeks"]

for i in l:

    print(i)

# Iterating over a tuple (immutable)

print("\nTuple Iteration")

t = ("geeks", "for", "geeks")

for i in t:

    print(i)

# Iterating over a String

print("\nString Iteration")

s = "Geeks"

for i in s :

    print(i)

# Iterating over dictionary

print("\nDictionary Iteration")

d = dict()

d['xyz'] = 123

d['abc'] = 345

for i in d :

    print("%s  %d" %(i, d[i]))

**Output :**

List Iteration

geeks

for

geeks

Tuple Iteration

geeks

for

geeks

String Iteration

G

e

e

k

s

Dictionary Iteration

xyz 123

abc 345

**Iterators in Linked list**

To make Linked list iterable we need to add \_\_iter\_\_ method so that, we can step through each node one by one.

#Insertion at the beginning in a Singly Linked List

class Node:

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

self.value = value

self.next = None

class SinglyLinkedList:

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

self.head = None

#Function to add node in the beginning

def atBeg(self, value):

new\_node = Node(value)

new\_node.next = self.head

self.head = new\_node

#Iterator Function

def \_\_iter\_\_(self):

node = self.head

while node:

yield node

node = node.next

#Initially, create a list with 1, 2, 3 as data

sll=SinglyLinkedList()

sll.atBeg(1)

sll.atBeg(2)

sll.atBeg(3)

#Using iterators for printing

for node in sll:

print(node.value, end = "->")

The time complexity for initializing a singly linked list is **O(1).**The space complexity is **O(1)** as no additional memory is required to initialize the linked list.