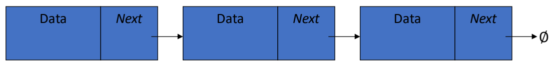
Linked List

Introduction

- Linked list is simply a chain of nodes.
- A node is comprised of two items:
  - data and
  - the pointer to the next data node, commonly referred to as the next reference.
- The end of the list is defined by a null next reference.

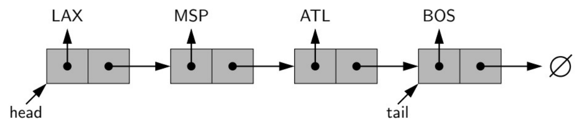


Introduction

- Types of linked lists:
  - Singly linked list
  - Double linked list
  - Circular linked list
- We will first look at singly linked list.
- Some slides are picked from this link: https://www.tutorialspoint.com/python_data_structure/python_linked_lists.htm

Singly Linked List

- The list instance maintains a member named **head** that identifies the first node of the list, and in some applications another member named tail that identifies the last node of the list.
- The None object is denoted as Ø
- Example below is that of a singly linked list whose elements are strings indicating airport codes.

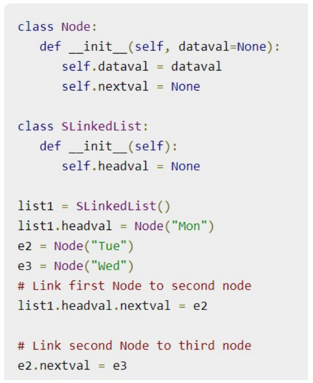


Singly Linked List

- A linked list's representation in memory relies on the collaboration of many objects → Nodes + the LinkedList
- Each **node is represented as a unique object**, with that instance storing a reference to its element and a reference to the next node (or None).
- Another object represents the linked list as a whole.
- The linked list instance must keep a reference to the head of the list.
- Without an explicit reference to the head, there would be no way to locate that node (or indirectly, any others).

Implementing Linked List in Python

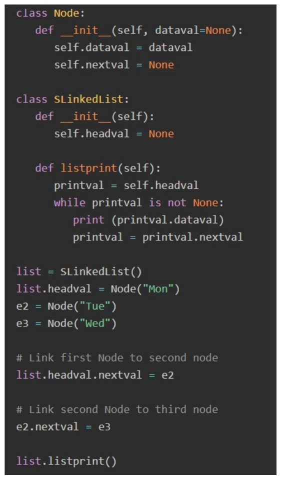
- Python does not have linked lists in its standard library.
- We implement the concept of linked lists using the concept of nodes
- We create a Node object and create another class to use this node object.
- We pass the appropriate values through the node object to point the to the next data elements.



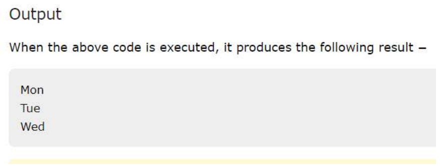
```
class Node:
   def __init__(self, dataval=None):
      self.dataval = dataval
      self.nextval = None

class SLinkedList:
   def __init__(self):
      self.headval = None

list1 = SLinkedList()
list1.headval = Node("Mon")
e2 = Node("Tue")
e3 = Node("Wed")
# Link first Node to second node
list1.headval.nextval = e2

# Link second Node to third node
e2.nextval = e3
```

Traversing a Linked List

- Singly linked lists can be traversed in only forward direction starting from 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.



```
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

list.listprint()
```



Output

When the above code is executed, it produces the following result –

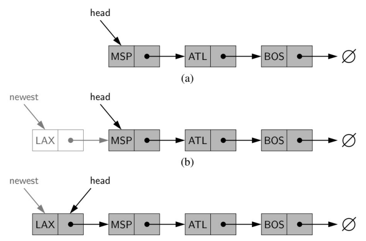
```
Mon
Tue
Wed
```

Inserting a Node to 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.

Inserting at the Beginning of a Linked List

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



9

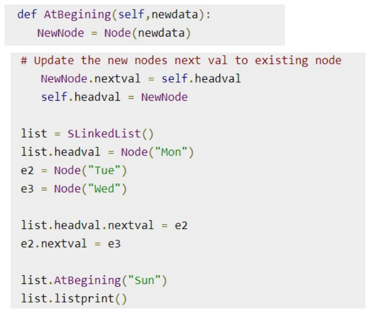
Inserting at the Beginning of a Linked List

```
Algorithm add_first(L, e):
    newest = Node(e)  {create new node instance storing reference to element e}
    newest.next = L.head  {set new node's next to reference the old head node}
    L.head = newest  {set variable head to reference the new node}
    L.size = L.size + 1  {increment the node count}
```

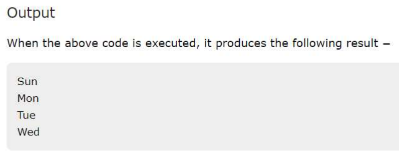
L – linked list that will have a new node added
e – data value to be added to the new node

10

Inserting at the Beginning of a Linked List



```
def AtBegining(self,newdata):
   NewNode = Node(newdata)

# Update the new nodes next val to existing node
   NewNode.nextval = self.headval
   self.headval = NewNode

list = SLinkedList()
list.headval = Node("Mon")
e2 = Node("Tue")
e3 = Node("Wed")

list.headval.nextval = e2
e2.nextval = e3

list.AtBegining("Sun")
list.listprint()
```



Output

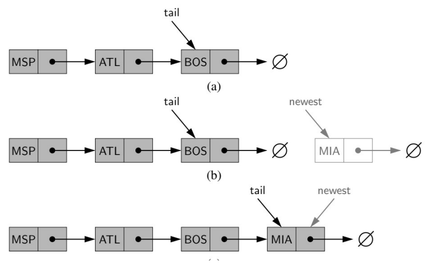
When the above code is executed, it produces the following result –

```
Sun
Mon
Tue
Wed
```

11

Inserting at the End

- 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.



12

Inserting at the End

```
Algorithm add_last(L,e):
    newest = Node(e)          {create new node instance storing reference to element e}
    newest.next = None        {set new node's next to reference the None object}
    L.tail.next = newest      {make old tail node point to new node}
    L.tail = newest           {set variable tail to reference the new node}
    L.size = L.size + 1       {increment the node count}
```

L – linked list that will have a new node added
e – data value to be added to the new node

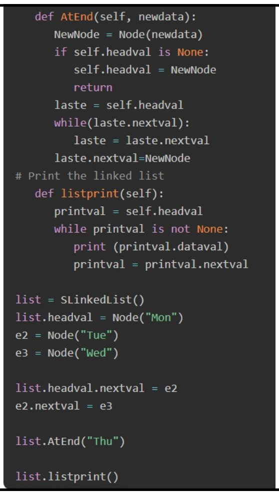
13

Inserting at the End

Output

When the above code is executed, it produces the following result –

```
Mon
Tue
Wed
Thu
```



```
   def AtEnd(self, newdata):
      NewNode = Node(newdata)
      if self.headval is None:
         self.headval = NewNode
         return
      laste = self.headval
      while(laste.nextval):
         laste = laste.nextval
      laste.nextval=NewNode
# 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("Wed")

list.headval.nextval = e2
e2.nextval = e3

list.AtEnd("Thu")

list.listprint()
```

14

Inserting between two 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 an additional class which will change the next pointer of the new node to the next pointer of middle node. Then assign the new node to next pointer of the middle node.

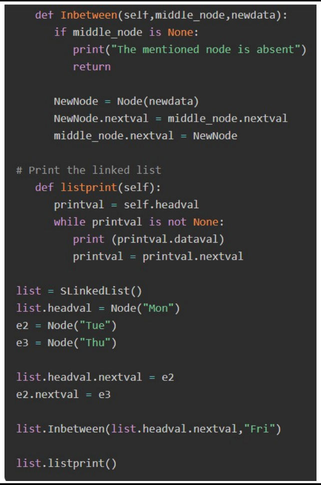
15

Inserting between two Nodes

Output

When the above code is executed, it produces the following result –

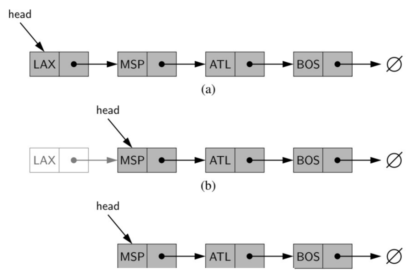
```
Mon
Tue
Fri
Thu
```



```
   def Inbetween(self,middle_node,newdata):
      if middle_node is None:
         print("The mentioned node is absent")
         return

      NewNode = Node(newdata)
      NewNode.nextval = middle_node.nextval
      middle_node.nextval = NewNode

# 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.Inbetween(list.headval.nextval,"Fri")

list.listprint()
```

16

Removing a Node – the Head Node



```
Algorithm remove_first(L):
    if L.head is None then
        Indicate an error: the list is empty.
    L.head = L.head.next            {make head point to next node (or None)}
    L.size = L.size − 1                          {decrement the node count}
```

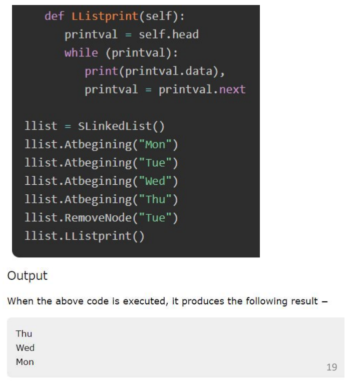
Code Fragment 7.3: Removing the node at the beginning of a singly linked list.

Removing a Node in the Middle

- 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.

Removing a Node in the Middle

```
def RemoveNode(self, Removekey):
    current = self.head #Let current variable point to head

    if (current is not None): #if there is only one node
        if (current.data == Removekey):
            self.head = current.next
            return
    while (current is not None):
        if current.data == Removekey:
            break
        prev = current #keep track of previous node
        current = current.next
    #let the previous node point to the node after next node.
    prev.next = current.next
```



```
    def LListprint(self):
        printval = self.head
        while (printval):
            print(printval.data),
            printval = printval.next

llist = SLinkedList()
llist.Atbegining("Mon")
llist.Atbegining("Tue")
llist.Atbegining("Wed")
llist.Atbegining("Thu")
llist.RemoveNode("Tue")
llist.LListprint()
```

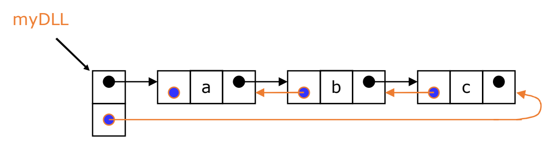
Output

When the above code is executed, it produces the following result –

```
Thu
Wed
Mon
```

Doubly-linked lists

- Here is a doubly-linked list (DLL):



- Each node contains a value, a link to its successor (if any), *and* a link to its predecessor (if any)
- The header points to the first node in the list *and* to the last node in the list (or contains null links if the list is empty)

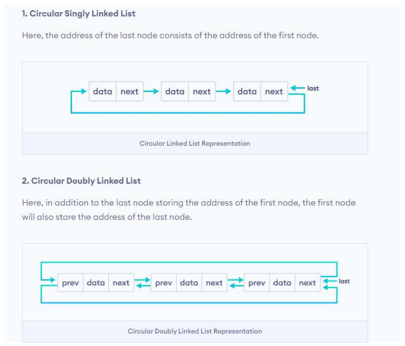
DLLs compared to SLLs

- Advantages:
  - Can be traversed in either direction (may be essential for some programs)
  - Some operations, such as deletion and inserting before a node, become easier.
    - In singly linked list, to delete a node, pointer to the previous node is needed.
    - In DLL, we can get the previous node using previous pointer.

DLLs Compared to SLLs

- Disadvantages:
  - Requires more space
  - List manipulations are slower (because more links must be changed)
  - Greater chance of having bugs (because more links must be manipulated)

Circular Linked List



1. Circular Singly Linked List

Here, the address of the last node consists of the address of the first node.

Circular Linked List Representation

2. Circular Doubly Linked List

Here, in addition to the last node storing the address of the first node, the first node will also store the address of the last node.

Circular Doubly Linked List Representation

Advantages of Circular Linked List

- We can go to any node from any node in the circular linked list which was not possible in the singly linked list if we reached the last node. It is easy go to head from the last node
- In a circular list, any node can be starting point means we can traverse each node from any point.

Advantages of Linked Lists

- Size of linked lists is not fixed, they can expand and shrink during run time.
- Insertion and deletion operations are fast and easier in linked lists.
- Memory allocation is done during run-time (no need to allocate any fixed memory).
- Data Structures like stacks, queues, and trees can be easily implemented using Linked list.

Singly Linked List vs Array

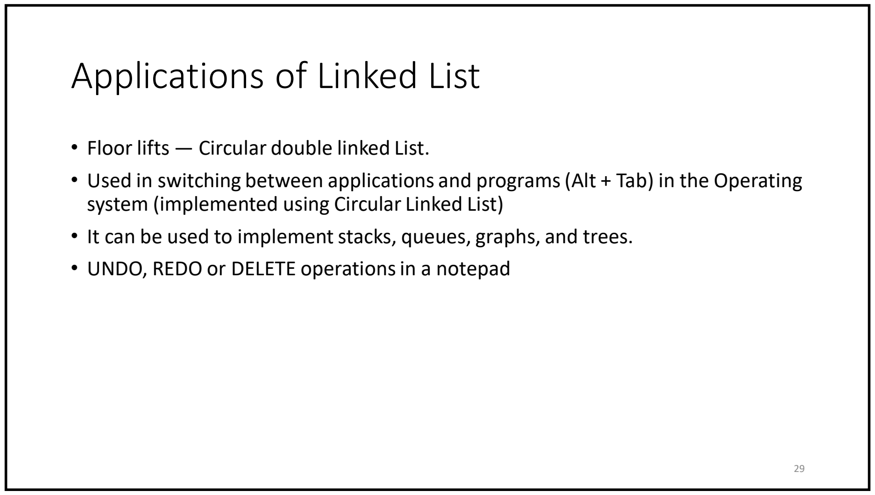
Singly linked list	**Array**
Elements are stored in linear order, accessible with links.	Elements are stored in linear order, accessible with an index.
Do not have a fixed size.	Have a fixed size.

Singly Linked List vs Array

Arrays / Python List	Linked Lists
An array is a collection of elements of a similar data type. List can have different data types.	Linked List is an ordered collection of elements not necessarily of same type in which each element is connected to the next using pointers.
Array elements can be accessed randomly using the array index for both array and list.	Random accessing is not possible in linked lists. The elements will have to be accessed sequentially.
Data elements are stored in contiguous locations in memory for both array and list	New elements can be stored anywhere and a reference is created for the new element using pointers.
Insertion and Deletion operations are costlier since the memory locations are consecutive and fixed for both array and list	Insertion and Deletion operations are fast and easy in a linked list.
Memory is allocated during the compile time (Static memory allocation for arrays). BUT lists are dynamic arrays hence can grow or shrink.	Memory is allocated during the run-time (Dynamic memory allocation).
Size of the array must be specified at the time of array declaration/initialization.	Size of a Linked list grows/shrinks as and when new elements are inserted/deleted.

Applications of Linked List

- Images are linked with each other. So, an image viewer software uses a linked list to view the previous and the next images using the previous and next buttons.
- Web pages can be accessed using the previous and the next URL links which are linked using a linked list.
- The music players also use the same technique to switch between music.



Applications of Linked List

- Floor lifts — Circular double linked List.
- Used in switching between applications and programs (Alt + Tab) in the Operating system (implemented using Circular Linked List)
- It can be used to implement stacks, queues, graphs, and trees.
- UNDO, REDO or DELETE operations in a notepad

29