Algorithm and Data Structures
Week 5

Endang Wahyu Pamungkas, P.hD.

Linear Structure

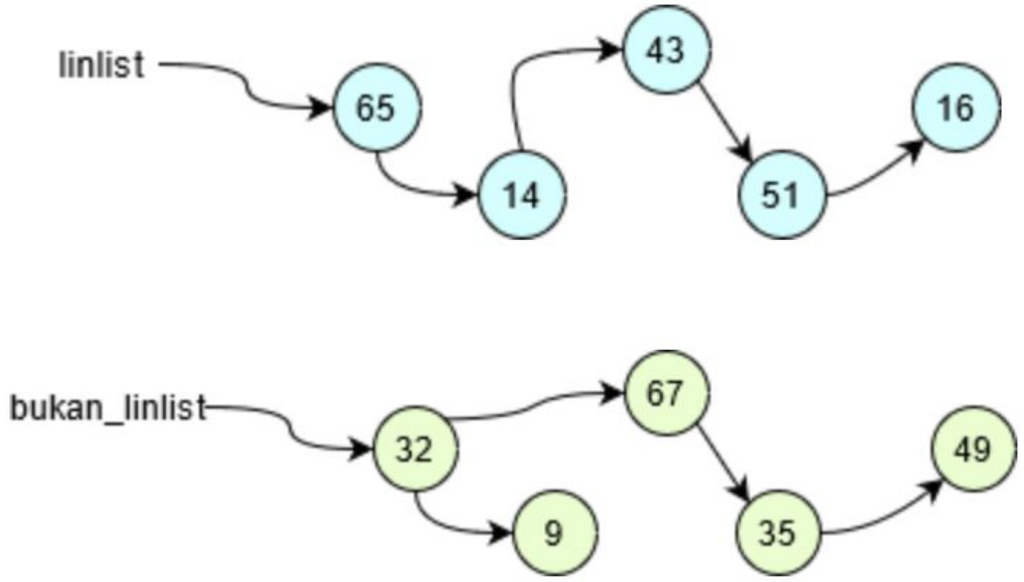
- Not all data can be stored using tables, or collections such as **arrays**, **lists**, or **tuples**. For example, the organizational structure should be saved using a typical shaped chart which is commonly called an organization chart.
- Data such as an organizational structure is not an ordinary data set that can be arranged in a row like an array.
- There are many examples of data that is not easy to store using ordinary arrays and collections, and is easier to store in a linked structure.

Linked Structure

- **Linked structure** is a data structure in which every data is stored at a node, and all nodes are linked by means of pointer/reference.
- There are three types of linked structures, namely:
  - Linear lists / linked lists
  - Tree
  - Graph

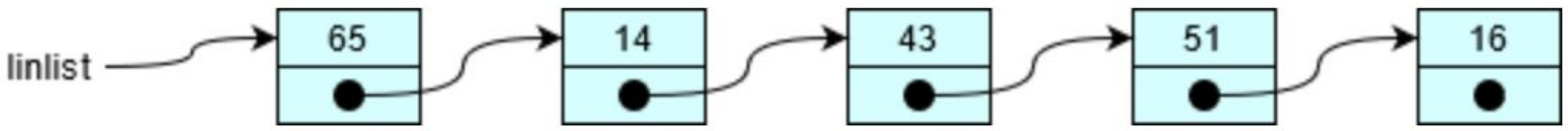
Linear List / Linked List

- Linear list or linked list is a data structure where data is stored in a node and each node can be linked to one and another node. All vertices if traced from start to finish can form a linear line.



Linked List

- Each node of a linked list has a data (for the example an integer value). In addition, every node (except the tail node) has a pointer to another node. So a node in a linked list consists of two components, namely data and links. If described in the form of data storage blocks, the linked list above can be redrawn as follows.



Linked List

- Each node can be represented by two data storage boxes. One contains data, and the other contains links. Such a repository can be realized in Python with a datastore class. A class has two attributes, namely data and pointer.

```
class Node():
    def __init__(self, data, pointer=None):
        self.data = data
        self.pointer = pointer
```

Linked List

- So now how to make a Node whose data is 82 and doesn’t have a pointer yet?

```
node = Node(82)
```

- How to link each node?

```
node1 = Node(23)
node2 = Node(76)
node1.pointer = node2

linlist = Node(65, Node(14, Node(43, Node(51,
Node(16)))))
```

Linear List / Linked List

- Traverse a linked list means visiting every node from the head node to the tail node. To move from one node to the next, we take advantage of the pointers that exist in the node. During the tracing process, we need a variable to store the link indicating which node we are at.

```
tt = linlist
while tt is not None:
    print(tt.data)
    tt = tt.pointer
```

Linear List and Object

```python
class Mahasiswa():
    def __init__(self, data):
        self.nama = data[0]
        self.nim = data[1]
        self.umur = data[2]

class Node():
    def __init__(self, data, pointer=None):
        self.data = data
        self.pointer = pointer

mylist = Node(Mahasiswa(["Badu", "Z123", 18]),
              Node(Mahasiswa(["Joko", "Z124", 20]),
                   Node(Mahasiswa(["Dodi", "Z125", 21]))))
```

```python
p = mylist
while p is not None:
    print(p.data.nama)
    p = p.pointer
```

Linked List Operation

- There are several operation that we can do for manipulating a linked list data structure in Python.
- These operations are insert at the beginning, insert at the end, insert in between the node, and remove node.
- To do these operations, we need to design such algorithms.

Build Class for Node and LinkedList

```
class Node:
    def __init__(self, data=None):
        self.data = data
        self.pointer = None

class LinkedList:
    def __init__(self):
        self.head = None
```

Traverse the LinkedList

```
def PrintLinkedList(self):
    printval = self.head
    while printval is not None:
        print (printval.data)
        printval = printval.pointer
```

Insert at the Beginning

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

Insert at the Last

```
def AtEnd(self, newdata):
      NewNode = Node(newdata)
      laste = self.head
      while(laste.pointer != None):
         laste = laste.pointer
      laste.pointer=NewNode
```

Insert at the Middle

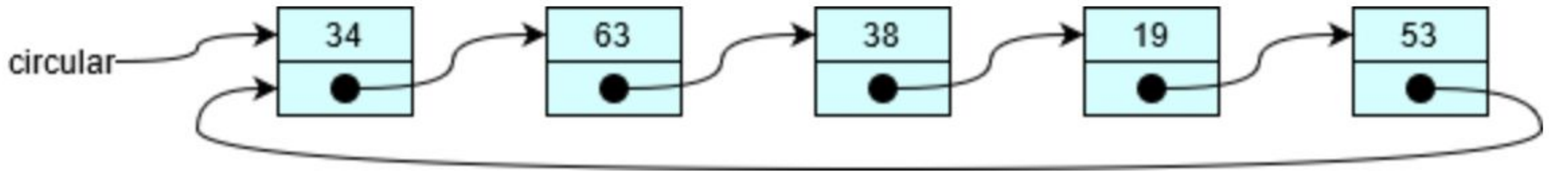
```
def Inbetween(self,middle_node,newdata):
        NewNode = Node(newdata)
        NewNode.pointer = middle_node.pointer
        middle_node.pointer = NewNode
```

Remove Items

```
def RemoveNode(self, Removekey):
      HeadVal = self.head

      if (HeadVal is not None):
         if (HeadVal.data == Removekey):
            self.head = HeadVal.pointer
            HeadVal = None
            return
      while (HeadVal is not None):
         if HeadVal.data == Removekey:
            break
         prev = HeadVal
         HeadVal = HeadVal.pointer

      if (HeadVal == None):
         return

      prev.pointer = HeadVal.pointer
         HeadVal = None
```

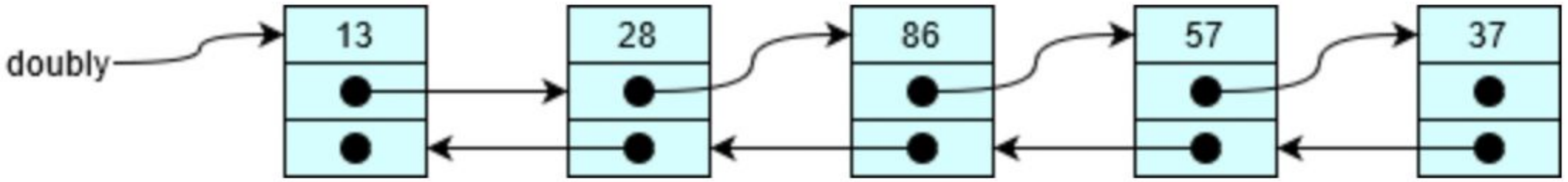
Circular LinkedList

- In a circular linked list, the pointer at the tail node has no value of None, while the pointer refers to the head node.
- The existence of a pointer from the tail node to the head node affects the work of the process of tracing, searching, adding and deleting nodes.



Doubly LinkedList

- A doubly linked list is a linked list that has two links at each node. One link to the next node and another one to the previous node. Doubly linked lists help the search process run in two directions, right and left (or up and down).



Latihan

- Try the previous source code to make a linkedlist structure and use traverse, insert at the beginning, insert at last, insert at the middle and remove nodes method.
- Build a single linked list which contains 6 nodes, where each node has an integer value. The value of the integers are 3,5,2,6,9,7. Then, sum all of these integers by traversing the linked list from the head node until the tail node.