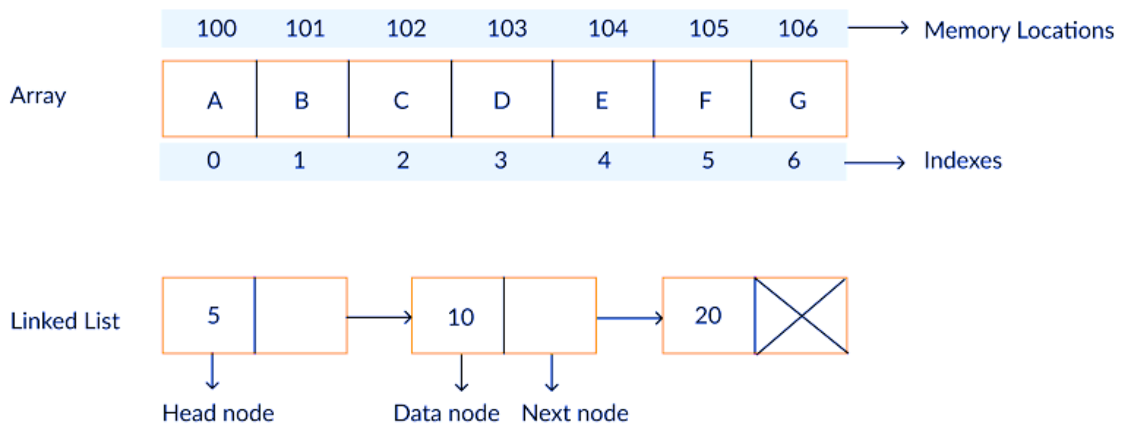
LINKED LIST

WHAT IS A 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 form of a pointer. Python does not have linked lists in its standard library.

LINKED LIST VERSUS ARRAY



(https://encrypted-tbn0.gstatic.com/images?q=tbn%3AANd9GcRcYFunn7Kqb5ssOme497qdg0LT-83MDCjxhsZsiQU7nnXJz0QA)

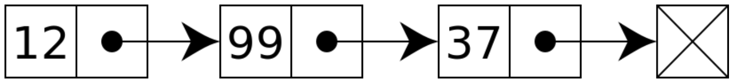
Basically, an **array** is a set of similar data objects stored in sequential memory locations under a common heading or a variable name.

While a **linked list** is a data structure which contains a sequence of the elements where each element is linked to its next element.

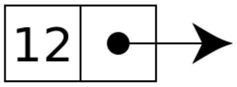
There are two fields in an element of linked list. One is **data** field, and other is **link** field, Data field contains the actual value to be stored and processed. Furthermore, the link field holds the address of the next data item in the linked list. The address used to access a particular node is known as a pointer.

Another significant difference between an array and linked list is that Array has a fixed size and required to be declared prior, but Linked List is not restricted to size and expand and contract during execution.

Here’s an image of a linked list. Two key things to note:



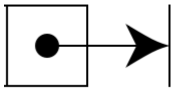
1. A linked list consists of **nodes**.



A node

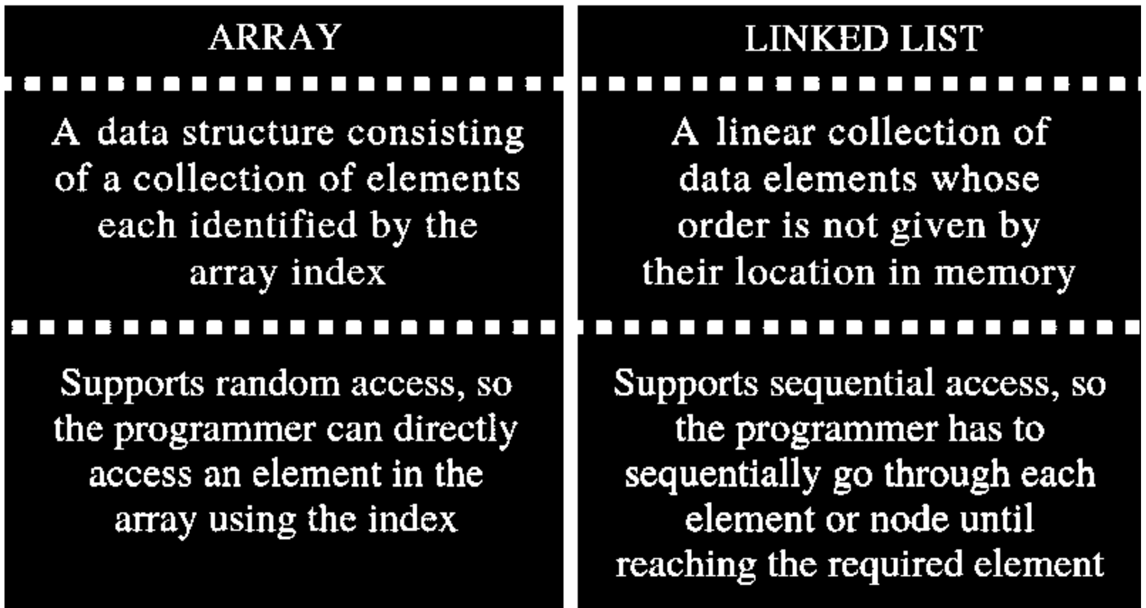
2. Each node consists of a **value** and a **pointer** to another node.

A value

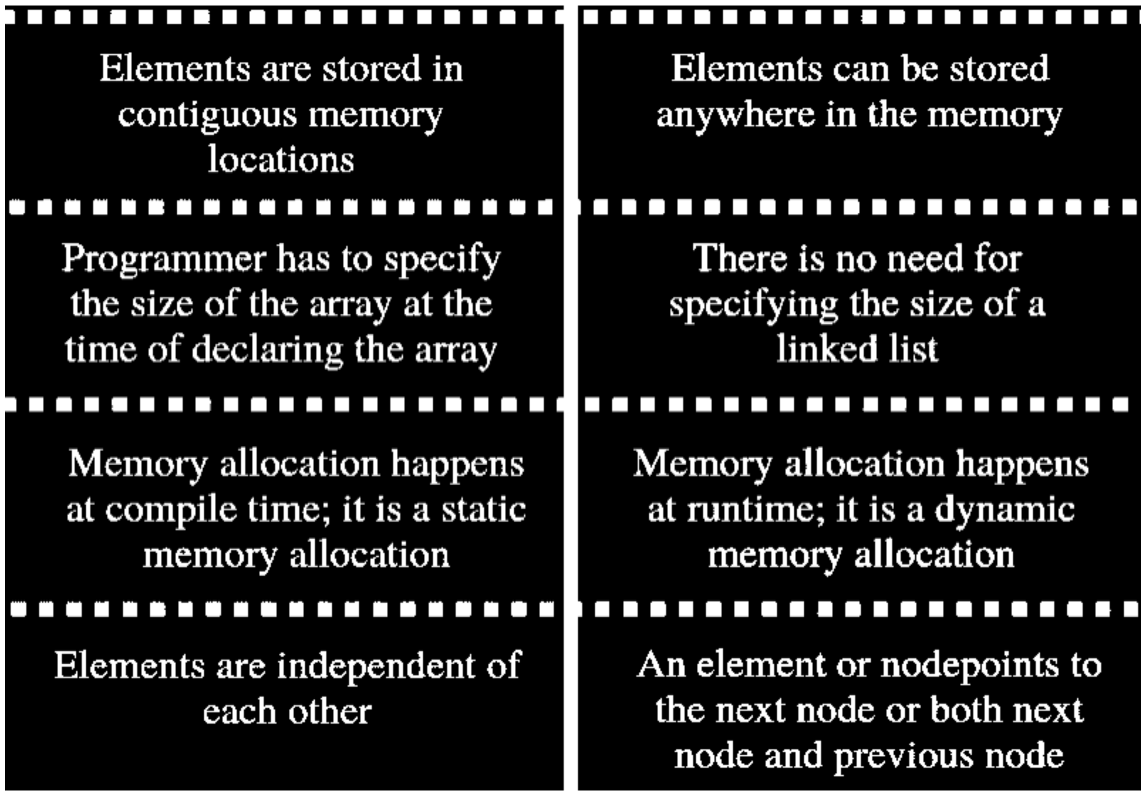

A pointer

The starting node of a linked list is referred to as the **header**. Essentially, linked list is a chain of values connected with pointers.

Below is a more comprehensive differences of array and linked list.



ARRAY	LINKED LIST
A data structure consisting of a collection of elements each identified by the array index	A linear collection of data elements whose order is not given by their location in memory
Supports random access, so the programmer can directly access an element in the array using the index	Supports sequential access, so the programmer has to sequentially go through each element or node until reaching the required element



Elements are stored in contiguous memory locations	Elements can be stored anywhere in the memory
Programmer has to specify the size of the array at the time of declaring the array	There is no need for specifying the size of a linked list
Memory allocation happens at compile time; it is a static memory allocation	Memory allocation happens at runtime; it is a dynamic memory allocation
Elements are independent of each other	An element or nodepoints to the next node or both next node and previous node

http://pediaa.com/wp-content/uploads/2018/12/Difference-Between-Array-and-Linked-List-Comparison-Summary.jpg

WHY USE LINKED LIST

Linked list is often compared to arrays. Whereas an array is a fixed size of sequence, a linked list can have its elements to be dynamically allocated. What are the pros and cons of these characteristics? Here are some major ones:

Advantages

A linked list saves memory. It only allocates the memory required for values to be stored. In arrays, you have to set an array size before filling it with values, which can potentially waste memory.

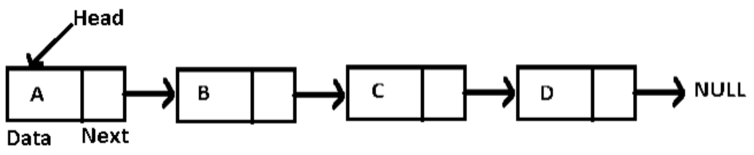
Linked list nodes can live anywhere in the memory. Whereas an array requires a sequence of memory to be initiated, as long as the references are updated, each linked list node can be flexibly moved to a different address.

Disadvantages

Linear look up time. When looking for a value in a linked list, you have to start from the beginning of chain, and check one element at a time for a value you're looking for. If the linked list is n elements long, this can take up to n time. On the contrary many languages allow constant lookups in arrays.

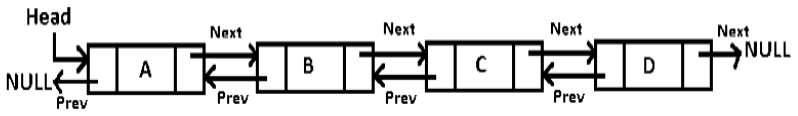
SINGLY LINKED LIST (SLL) AND DOUBLY LINKED LIST (DLL)

Singly linked list : A singly linked list is a set of nodes where each node has two fields ‘data’ and ‘link’. The ‘data’ field stores actual piece of information and ‘link’ field is used to point to next node. Basically ‘link’ field is nothing but address only.



https://www.geeksforgeeks.org/difference-between-singly-linked-list-and-doubly-linked-list/

Doubly linked list : A doubly linked list contains an extra pointer, typically called ***previous pointer***, together with next pointer and data which are there in singly linked list.



https://www.geeksforgeeks.org/difference-between-singly-linked-list-and-doubly-linked-list/

SINGLY LINKED LIST (SLL)	DOUBLY LINKED LIST (DLL)
SLL has nodes with only a data field and next link field.	DLL has nodes with a data field, a previous link field and a next link field.
In SLL, the traversal can be done using the next node link only.	In DLL, the traversal can be done using the previous node link or the next node link.
The SLL occupies less memory than DLL as it has only 2 fields.	The DLL occupies more memory than SLL as it has 3 fields.
Less efficient access to elements.	More efficient access to elements.

LINKED LIST IMPLEMEMTATION USING PYTHON

- **Creation of Linked List**
  A linked list is created by using the node class we studied in the last chapter. We create a Node object and create another class to use this ode object. We pass the appropriate values through the node object to point the to the next data elements. The below program creates the linked list with three data elements. In the next section we will see how to traverse the linked list.

```
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 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.

```
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()
```

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

```
Mon
Tue
Wed
```

- **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 of the 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.

```
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

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

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

list.AtBegining("Sun")

list.listprint()
```

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

```
Sun
Mon
Tue
Wed
```

- **Inserting at the End of the Linked List**
  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.

```
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
        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()
```

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

```
Mon
Tue
Wed
Thu
```

- **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 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.

```
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,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()
```

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

```
Mon
Tue
Fri
Thu
```

- **Removing an Item form a Liked List**
  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.

```
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):

        HeadVal = self.head

        if (HeadVal is not None):
            if (HeadVal.data == Removekey):
                self.head = HeadVal.next
                HeadVal = None
                return

        while (HeadVal is not None):
            if HeadVal.data == Removekey:
                break
            prev = HeadVal
            HeadVal = HeadVal.next

        if (HeadVal == None):
            return

        prev.next = HeadVal.next

        HeadVal = None

    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()
```

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

```
Thu
Wed
Mon
```

References:

https://www.tutorialspoint.com/python_data_structure/python_linked_lists.htm
https://techdifferences.com/difference-between-array-and-linked-list.html
https://www.geeksforgeeks.org/difference-between-singly-linked-list-and-doubly-linked-list/
https://www.tutorialspoint.com/python_data_structure/python_linked_lists.htm

To learn more, you may watch a tutorial on:
https://www.youtube.com/watch?v=njTh_OwMljA
https://www.youtube.com/watch?v=WwfhLC16bis